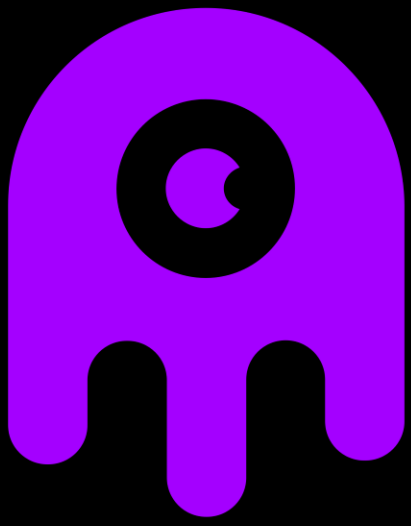


HashiCorp Terraform

Certified Associate Preparation Guide

Ned Bellavance
Adin Ermie

HashiCorp Terraform Certified Associate Preparation Guide

Ned Bellavance and Adin Ermie

This book is for sale at http://leanpub.com/terraform-certified

This version was published on 2020-05-09

Contents

About the Authors

Ned Bellavance

Ned Bellavance

Ned is an IT professional with almost 20 years of experience in the field. He has been a helpdesk operator, systems administrator, cloud architect, and product manager. In his newest incarnation, he is the Founder of Ned in the Cloud LLC. As a one-man-tech-juggernaut, he develops courses for Pluralsight, runs a podcast for Packet Pushers, runs a podcast for himself, and creates original content for technology vendors.

Ned has been a Microsoft MVP since 2017 and holds a bunch of industry certifications that have no bearing on anything beyond his exceptional ability to take exams and pass them.

Ned has three guiding principles:

- Embrace discomfort
- Fail often
- Be nice

You can learn more about Ned on his website https://nedinthecloud.com[1].

Adin Ermie

Adin Ermie

[1] https://nedinthecloud.com

Adin is an experienced Microsoft Azure Subject Matter Expert (SME) with over 15 years of experience in the IT field. He brings passion and enthusiasm to everything he does.

He has worked for a number of Microsoft Partners, engaging on projects to help and support customers with Azure infrastructure, management, and governance.

Prior to joining Microsoft, Adin was a Microsoft MVP for 5 consecutive years in the Cloud and Datacenter Management (CDM) award category. Now that he has joined the mothership, he continues in the role of Cloud Solution Architect (CSA) - Azure Infrasutrcture and supports many of the Canada-based Microsoft Partners.

For further information about him, check out his blog https://adinermie.com[2].

[2] https://adinermie.com

Tech Reviewers

Steve Buchanan

Steve Buchanan

Steve Buchanan (@buchatech) is a Director, & Midwest Containers Services Lead on the Cloud Transformation/DevOps team with Avanade, the Microsoft arm of Accenture.

He is an eight-time Microsoft MVP, and the author of six technical books. He has presented at tech events, including Midwest Management Summit (MMS), Microsoft Ignite, BITCon, Experts Live Europe, OSCON, and user groups.

Steve is active in the technical community and enjoys blogging about his adventures in the world of IT on his blog at buchatech.com[3].

[3] http://www.buchatech.com/

Exam Overview

Hello! Welcome to our guide for attaining the Terraform Associate certification from HashiCorp. You've likely picked up this guide because you dabbled in Terraform as a hobbyist or in a development environment, and you're excited about the possibilities of deploying Infrastructure as Code. Now you want to sharpen your skills and prove your knowledge with a certification from HashiCorp. At least that's one possible path. Whatever has brought you to our guide, we're glad you're here! We think Terraform is an amazing tool for deploying Infrastructure as Code in a cloud agnostic way, and we're excited to share our accumulated knowledge for you.

Setting Expectations

Before we get into the nitty-gritty, let's talk about the exam itself. After all, it's going to be hard to prepare for an exam, if you don't know what's in it. The Terraform Associate certification is targeted at professionals in IT, DevOps, and operations roles, who have a basic level of knowledge about Infrastructure as Code and HashiCorp Terraform[4]. You are NOT expected to be a Terraform expert, this is an associate level exam after all. In addition, there are some paid versions of Terraform that have features that do not exist in the open source offering. You are expected to have read the documentation on these paid offerings, but not necessarily have first hand knowledge in using them. HashiCorp is not expecting you to spend hundreds or thousands of dollars gaining the necessary knowledge to pass this exam.

The exam for the HashiCorp Certified Terraform Associate certification is based on a list of exam objectives published by HashiCorp. Each terminal objective is broken into a set of enabling objectives that will test you knowledge and experience. We have structured this guide so each chapter is mapped to a terminal objective, and the sections of chapter deal with each enabling objective.

While we believe this will be a useful form of preparation, we also believe that it will not be enough to pass the exam. There is no substitute for hands-on practical experience. Reading this book will certainly help, but even more important is to open up your code editor of choice and start hacking away. We have provided some examples in the book to help you on your way, and we encourage you to experiment on your own. Break things and try to fix them. Open up the Terraform state file and poke around - in a non-production scenario of course. Not only is playing with Terraform fun, it will also improve your knowledge and understanding faster than studying a book alone.

Prerequisites

Before you embark on your Terraform journey there are some prerequisites we believe will be helpful.

[4] https://www.terraform.io/

- Basic skills in a terminal environment
- Basic understanding of infrastructure components on-premises and in the cloud
- Familiarity with a scripting language
- Familiarity with basic development concepts
- Access to a cloud provider

You can use whatever code editor and operating system works best for you. We use a mix of Linux, Windows, and macOS for our operating systems. Terraform is written in Go and compiled as a binary for each OS, so it doesn't really make a difference which one you choose. We all use Visual Studio Code as our editor of choice because of its Terraform extensions and integrated terminal, but we encourage you to use whatever is comfortable for you.

Product Version Tested

The images and scripts included with this booked were created and tested using Terraform version `0.12`. There are significant changes in version `0.12` that will be reflected in the exam. If you have only used older versions of Terraform, then we highly recommend that you review the **What's New**[5] section of the release notes for `0.12`.

Chapter Summary

In the next chapter, we will take a look at the list of exam objectives and provide some background for each one.

---

[5] https://github.com/hashicorp/terraform/blob/v0.12.0/CHANGELOG.md#new-features

Exam Objectives

HashiCorp has identified a number of terminal objectives that a practitioner should understand before attempting the exam, and the enabling objectives that compliment each terminal objective. The remainder of the chapters go into more depth on each of these objectives. Here is the current list, as of publishing. The most current version can always be found on the official exam objectives page[6].

High-Level Objectives

1. **Understand Infrastructure-as-Code (IaC) Concepts**: IaC is composed of core deployment patterns that provide benefits over traditional approaches.
2. **Understand Terraform's Purpose (vs Other IaC)**: Terraform can be used to manage multi-cloud environments using their provider model and store state to continue maintenance of infrastructure beyond the initial deployment.
3. **Understand Terraform Basics**: Terraform is built upon some basic concepts like installation, plug-ins, providers, and provisioners.
4. **Use the Terraform CLI (Outside of Core Workflow)**: Terraform is largely CLI driven, and in this chapter we will look at some of the commands that sit outside of the `init, plan, apply` workflow.
5. **Interact with Terraform Modules**: Modules provide a way to abstract common deployment patterns into reusable code blocks.
6. **Navigate Terraform Workflow**: The deployment of infrastructure follows a core workflow of write->plan->create to manage the lifecycle of infrastructure deployments.
7. **Implement and Maintain State**: The state data managed by Terraform enables the application to manage the lifecycle of infrastructure you deploy.
8. **Read, Generate, and Modify Configuration**: The configuration of a Terraform deployment is composed of things like variables, resources, data sources, and secrets. We'll dig into these objects and learn how to use them.
9. **Understand Terraform Enterprise Capabilities**: Terraform Enterprise and Terraform Cloud build on the functionality of Terraform OSS. We'll review the enhancements that you might want to take advantage of as your use of Terraform matures.

Chapter Layout

Each chapter deals with one of the above terminal objectives, examining its enabling objectives in greater detail. At the end of each enabling objective section are **Supplemental Information**

[6] https://www.hashicorp.com/certification#terraform-associate

and **Key Takeaway** items. The Supplemental Information includes links to HashiCorp's Terraform documentation relevant to the sub-category and any external links that may be helpful. The Key Takeaway is the main idea or concept to remember from each sub-category. The enabling objectives and their key takeaways are repeated as a list at the end of the chapter in a **Chapter Summary** for quick reference when studying.

Chapter Summary

In this chapter you learned about the terminal objectives and teh structure of the book. Now we are ready to get started with the objective **Understand Infrastructure-as-Code (IaC) Concepts**.

Objective 1: Understand Infrastructure-as-Code (IaC) Concepts

Terraform is an example of Infrastructure-as-Code (IaC), and therefore HashiCorp expects that you have a fundamental understanding of what IaC is and some of its concepts and patterns. This exam objective is not focused specifically around Terraform itself but covers general topics of Infrastructure-as-Code.

1A: Explain What IaC Is

Infrastructure can be thought of as the resources an application runs on. Traditionally this was composed of servers, storage, and networking. With the advent of virtualization, servers were now split into both physical resources and virtual machines. Cloud providers created additional abstractions, starting with Infrastructure-as-a-Service (IaaS) and moving up to Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). While traditional infrastructure was deployed using manual and cumbersome processes fraught with potential for mistakes and inconsistencies, the various services offered by cloud providers and other vendors introduce a software-driven model of creation and configuration.

Infrastructure-as-Code (IaC) is the practice of defining and provisioning infrastructure resources through a machine-readable format. The infrastructure provisioned and managed using IaC can include bare-metal servers, virtual machines, networking resources, storage volumes, or even databases provided as a service. While IaC is often used in a cloud environment, it is not necessary to use the cloud for IaC. As long as your infrastructure management system has a programmatic interface, it can participate in IaC.

The format can be declarative (defining the desired outcome) or imperative (defining the process of provisioning.) Declarative provisioning solutions, such as Terraform, are focused on the desired end state of a deployment and rely on an interpretation engine to create and configure the actual resources. Imperative provisioning solutions, such as a batch script, focus on the actual provisioning process and may reference a file containing a list of settings and configuration values. The code created for IaC may be stored in a Version Control System (VCS) to enable change tracking and team collaboration. IaC can also be combined with other common software development practices like automated testing, deployment pipelines, and reusable components.

The key point behind Iac is that infrastructure is defined and provisioned through machine-readable code and is not provisioned by hand using a portal, command line, or API. If the commands or process is performed manually by a person, it is **not** IaC.

Supplemental Information

- https://www.terraform.io/intro/index.html#infrastructure-as-code[7]
- https://en.wikipedia.org/wiki/Infrastructure_as_code[8]

Key Takeaways: Infrastructure as Code is the practice of defining infrastructure deployments using machine-readable files that can be used to provision infrastructure in an automated fashion.

1B: Describe Advantages of IaC Patterns

The advantages of using IaC can be broken down into a few key categories: consistency, repeatability, and efficiency.

By defining infrastructure as code, the deployment of that infrastructure should be consistent across multiple builds and environments. When the same code is used to stand up infrastructure for staging and production, the operations team no longer has to wonder if the two environments are at parity. In a cloud context, using the same code to deploy infrastructure in multiple regions assures that each region is set up consistently.

There are many common components and deployment patterns in infrastructure. Using IaC allows these patterns to be defined once and used multiple times. For example, it may be a common pattern to deploy web servers for a new application. By defining the web server deployment in code, it can be reused by each new deployment that requires a set of web servers. Additionally, if something changes about the desired configuration of those web servers, an update of the web server deployment code can be created and pushed to each application environment.

Defining infrastructure as code allows operations teams to more tightly integrate with their developer counterparts. The provisioning process of infrastructure can be incorporated into the existing application development workflow, creating a consistent and repeatable process that flows through development, QA, staging, and production environments. By automating and integrating with the development team, greater levels of efficiency can be achieved.

Supplemental Information

- https://en.wikipedia.org/wiki/Infrastructure_as_code[9]

[7] https://www.terraform.io/intro/index.html#infrastructure-as-code
[8] https://en.wikipedia.org/wiki/Infrastructure_as_code
[9] https://en.wikipedia.org/wiki/Infrastructure_as_code

Key Takeaways: Infrastructure as Code improves consistency, repeatability, and efficiency while lowering risk and costs.

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **1A**: **Explain What IaC Is**: Infrastructure as Code is the practice of defining infrastructure deployments using machine-readable files that can be used to provision infrastructure in an automated fashion.
- **1B**: **Describe Advantages of IaC Patterns**: Infrastructure as Code improves consistency, repeatability, and efficiency while lowering risk and costs.

In the next chapter we will dig into the terminal objective **Understand Terraform's Purpose (vs Other IaC)**, showing how Terraform applies the concepts of IaC.

Objective 2: Understand Terraform's Purpose (vs Other IaC)

Terraform is not the only player in town when it comes to working with Infrastructure-as-Code (IaC.) There are public cloud provider solutions, like CloudFormation from AWS or Azure Resource Manager (ARM) Templates from Microsoft. Various IaC solutions all have different features and approaches to achieve the goal of automated infrastructure deployment. HashiCorp wants you to know what makes Terraform different from these other solutions.

2A: Explain Multi-Cloud and Provider-Agnostic Benefits

As alluded to in the introduction, there are many different clouds out there, including Amazon Web Services (AWS), Microsoft Azure, and Google's Cloud Platform (GCP), not to mention others such as Alibaba Cloud (AliCloud), Oracle Public Cloud (OPC), etc. Many of these public cloud providers have some type of proprietary IaC solution specific to their cloud.

Terraform is agnostic when it comes to the public cloud. It leverages providers for each different cloud, as well as other solutions including VMware, Kubernetes, and MySQL. Rather than focusing on a specific cloud or service, Terraform is able to provide a common tool, process, and language (HashiCorp Configuration Language) to be used across multiple clouds and services.

Supplemental Information

- Comparison to other solutions[10]

Key Takeaways: Terraform has a multi-cloud and provider-agnostic approach that makes it ideal for use across multiple clouds and services.

2B: Explain the Benefits of State

The creation and manipulation of resources managed by Terraform needs to be tracked in some manner. Terraform achieves this through the implementation of *state*. When a resource is created,

[10] https://www.terraform.io/intro/vs/index.html

such as an EC2 instance in AWS, Terraform creates an entry in state that maps the metadata about the resource (such as `instance-id`) to key/value pairs in the entry. The tracking of resource metadata serves multiple functions.

Idempotence

Each time a Terraform configuration is planned or applied, Terraform checks to see if there are any changes required to the actual environment to match the desired configuration. Only those resources that require changes will be updated, and all other resources will be left alone. If there are no changes to the configuration, then Terraform will not make any changes to the environment. State is what allows Terraform to map the resources defined in the configuration to the resources that exist in the real world.

Dependencies

Imagine a scenario where you have removed a subnet and EC2 instance from a configuration. Terraform will attempt to destroy those resources the next time you apply the configuration. You know intuitively that the EC2 instance must be removed before the subnet, since the instance is dependent on the subnet. How can Terraform figure this out? When the resources were created, the EC2 instance referenced the subnet and so Terraform found the dependency when building its graph. With both resources removed from the configuration, Terraform cannot deduce the dependency by building a graph. The answer? State!

Terraform maintains a list of dependencies in the state file so that it can properly deal with dependencies that no longer exist in the current configuration.

Performance

The state is a representation of the current state of the world as Terraform understands it. When planning a change, Terraform needs to look at the resources and their attributes to determine if changes are necessary. Do the EC2 instances need a new metadata tag? Terraform can check the `tags` attribute in state for the EC2 instance and make a quick decision. While Terraform could query resources directly for each planning run, the performance of planning would rapidly degrade as the infrastructure grows in scale.

By default Terraform will refresh the state before each planning run, but to improve performance Terraform can be told to skip the refresh with the `--refresh=false` argument. The `-target` argument can be used to specify a particular resource to refresh, without triggering a full refresh of the state. The state can be periodically refreshed from the actual world when required in order to keep up with changes, while not requiring a full scan during each plan.

It's important to note that choosing not to refresh the state means that the reality of your infrastructure deployment may not match what is in the state file. This can lead to inconsistent results when you apply the plan, or an outright failure. The risk of not refreshing state should be balanced against any performance improvements.

Collaboration

State keeps track of the version of an applied configuration, and it supports the locking of state during updates. Combined with the storage of state in a remote, shared location, teams are able to collaborate on deployments without overwriting each other's work.

Supplemental Information

- https://www.terraform.io/docs/state/purpose.html[11]

Key Takeaways: Terraform state enables the mapping of real world instances to resources in a configuration, improved performance of the planning engine, and collaboration of teams.

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **2A: Explain Multi-Cloud and Provider-Agnostic Benefits**: Terraform has a multi-cloud and provider-agnostic approach that makes it ideal for use across multiple clouds and services.
- **2B: Explain the Benefits of State**: Terraform state enables the mapping of real world instances to resources in a configuration, improved performance of the planning engine, and collaboration of teams.

In the next chapter we will get into the terminal objective of **Understand Terraform Basics** which deals with the installation of Terraform, using providers, and using provisioners.

---

[11] https://www.terraform.io/docs/state/purpose.html

Objective 3: Understand Terraform Basics

Before you can get started with Terraform, you'll need to install it somewhere. You also can't do a whole lot without using the various and sundry providers that enable Terraform to talk to cloud providers, platform services, datacenter applications, and more. We'll also touch on provisioners, and when to use them. You might find the answer counterintuitive!

3A: Handle Terraform and Provider Installation and Versioning

Terraform

Terraform is written in Golang and compiled as a single binary for multiple operating systems. There is no complicated setup script or laundry lists of dynamic link libraries. You don't install it using an MSI, rpm, or dpm. The executable binary comes inside a zip file. Terraform is supported on Windows, macOS, and Linux, FreeBSD, OpenBSD, and Solaris.

To install Terraform on any a supported operating system:

1. Find the Terraform package[12] for your operating system type and download it. Terraform is distributed as a single .zip file.
2. After downloading Terraform, unzip the package to a directory of your choosing. Terraform runs as a single binary named terraform. Any other files in the package can be safely removed and Terraform will still function.
3. To enable running Terraform from any path, you can modify the PATH environment variable to include the directory that contains the Terraform binary.
    1. How to set the $PATH on Linux and Mac[13]
    2. How to set the PATH on Windows[14]

[12] https://www.terraform.io/downloads.html
[13] https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux
[14] https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows

Providers

The resources and data sources used by Terraform are all enabled through the provider system. A provider is an executable plug-in that contains the code necessary to interact with the API of the service it was written for. Typically this includes a way to authenticate to a service, manage resources, and access data sources.

Providers can be explicitly defined within a configuration, or implied by the presence of a resource or data source that uses the provider. The actual arguments in a provider block vary depending on the provider, but all providers support the meta-arguments of `version` and `alias`. More on those in a moment.

An example Azure provider block looks like this:

```
provider azurerm {
    version = "=1.41.0"
    tenant_id = var.tenant_id
    subscription = var.subscription_id
}
```

In the case of the Azure provider, the authentication information could be supplied through an environment variable or cached credentials from the Azure CLI. The general best practice is to avoid hard-coding secret information, like credentials, into the Terraform configuration.

The `version` argument is used to constrain the provider to a specific version or a range of versions in order to prevent downloading a new provider that may possibly contain breaking changes. If the version isn't specified, Terraform will automatically download the most recent provider during initialization. While you can specify the version of the provider in the provider block, HashiCorp recommends that you create a special `required_providers` block for Terraform configuration as follows:

```
terraform {
    required_providers {
        azurerm = "=1.41.0"
    }
}
```

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.

There are multiple arguments for specifying the version number. It's probably a good idea to know these:

- `>= 1.41.0` is greater than or equal to the version.

- `<= 1.41.0` is less than or equal to the version.
- `~> 1.41.0` this one is funky. It means any version in the `1.41.X` range.
- `>= 1.20, <= 1.41` is any version between `1.20` and `1.41` inclusive.

One of the more common arguments is `~>` which is meant to keep you on the same major version, while still allowing for minor version updates. For instance, let's say there's 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.

Supplemental Information

- (AWS) https://learn.hashicorp.com/terraform/getting-started/install[15]
- (Azure) https://learn.hashicorp.com/terraform/azure/install_az[16]
- (GCP) https://learn.hashicorp.com/terraform/gcp/install[17]

Key Takeaways: Terraform is installed as a single binary. It uses providers to interact with other services, and you can specify the version of the provider.

3B: Describe Plugin Based Architecture

Terraform is provided as a single binary that includes core components required to parse and deploy Terraform configurations. What is does not include is the necessary code to interact with various providers and provisioners. That code is supplied via plugins. Each plugin is executed as a separate process communicating with the core Terraform binary using an RPC interface.

> *Fun fact*: The code for many of the providers used to be bundled into the Terraform binary. This caused the binary to bloat as more providers were added, and forced a new version of Terraform each time a provider wanted to add and update or feature. HashiCorp shifted to a plugin architecture to lean out the Terraform binary and allow the providers to update at their own cadence.

Plugins keep the Terraform binary relatively small, lowers the potential attack surface, and simplifies debugging of core Terraform code. Allowing providers and provisioners to develop their plugins separately creates a firm delineation between what Terraform is meant to do and the plugins.

[15] https://learn.hashicorp.com/terraform/getting-started/install
[16] https://learn.hashicorp.com/terraform/azure/install_az
[17] https://learn.hashicorp.com/terraform/gcp/install

Supplemental Information

- https://www.terraform.io/docs/plugins/basics.html[18]

Key Takeaways: Terraform makes use of plugins for its providers and provisioners instead of bundling them in the core binary.

3C: Demonstrate Using Multiple Providers

This objective can be taken in two different ways. It could mean using multiple different providers in a configuration, or using multiple instances of the same provider. Both are fairly common scenarios that you need to be aware of.

Multiple different providers

Even the most basic configurations will likely use multiple different providers. Let's take the following configuration as an example:

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

resource "random_integer" "rand" {
  min = 10000
  max = 99999
}

resource "aws_s3_bucket" "bucket" {
  name = "unique-name-${random_integer.rand.result}"
  ...
}
```

In this configuration we have already used two separate providers to create a single S3 bucket. The *Random* provider gives us a random integer for naming the bucket and the *AWS* provider gives us the S3 bucket resource. When using a provider it can be explicitly defined in a provider block, or implied by the presence of a resource that uses the provider.

[18] https://www.terraform.io/docs/plugins/basics.html

Multiple instances of a provider

There may be times when you need to specify multiple instances of the same provider. For instance, the AWS provider is region specific. If you want to create AWS resources in more than one region, you're going to need multiple instances of the provider. Other examples might be the use of multiple subscriptions in Azure or multiple master hosts with Kubernetes.

All providers support the use of the `alias` meta-argument, which enables you to give a provider an alias for reference. When creating resources using that provider, you can specify the alias of the provider instance to use. Consider the following example:

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

provider "aws" {
    region = "us-west-2"
    alias  = "west"
}

resource "aws_ec2_instance" "example-west" {
  name     = "instance2"
  provider = aws.west
  ...
}
```

The EC2 instance in the example will be created in the `us-west-2` region because we have specified the provider to use in the resource block. The generalized format for provider aliases is `<PROVIDER_NAME>.<ALIAS_NAME>` or `aws.west` in our case.

Now what would happen if we add this entry to the configuration?

```
resource "aws_ec2_instance" "example-east" {
  name = "instance1"
  ...
}
```

We haven't specified which provider instance to use. Terraform will look for a provider with no alias defined, and use that provider. In our case, the `example-east` EC2 instance would be created in the `us-east-1` region.

Provider aliases can also be used with modules, where the expression would be in this format:

```
module "vpc" {
  source    = "terraform-aws-modules/vpc/aws"
  providers = {
      aws = aws.west
  }
}
```

This is because modules can contain multiple providers, so we provide the module with a map containing the provider name as the key and the full alias as the value. It's actually a bit more nuanced with that, and we will cover it in more detail in Objective 5: Interact with Terraform Modules.

Supplemental Information

- https://www.terraform.io/docs/configuration/providers.html#alias-multiple-provider-instances[19]

Key Takeaways: You can have multiple different providers or multiple instances of the same provider in a configuration. It's important to understand how to use the `alias` argument and the use cases for multiple provider instances.

3D: Describe How Terraform Finds and Fetches Providers

Providers fall into two broad categories, HashiCorp distributed and third-party. HashiCorp distributed providers are available for download automatically during Terraform initialization. Third-party providers must be placed in a local plug-ins directory located at either `%APPDATA%\terraform.d\plugins` for Windows or `~/.terraform.d/plugins` for other operating systems.

The provider executable can be downloaded directly from the internet, or can be located on a local directory or file share. When a Terraform configuration is initialized, it looks at the providers being used by the configuration and retrieves the provider plug-ins, storing them in a `.terraform` subdirectory in the current configuration folder.

By default, each configuration will download its own copy of the provider plug-ins. To reduce download times and improve performance, plugin caching can be enabled, either through the `plugin_cache_dir` setting in the Terraform CLI configuration file or using the `TF_PLUGIN_CACHE_DIR`

[19] https://www.terraform.io/docs/configuration/providers.html#alias-multiple-provider-instances

environment variable. When caching is enabled, Terraform will check the cache for a viable plugin before downloading it from the internet. If no suitable plugin is found, Terraform will download the plugin to both the cache and the `.terraform` folder.

There may be scenarios where you don't want Terraform to download plugins, such as a disconnected or highly-regulated environment. In those cases, you can download the proper plug-ins ahead of time and store them in a plugin directory.

The location of pre-downloaded plugins can be configured by setting the environment variable `TF_PLUGIN_DIR` or by using the `--plugin-dir` argument during initialization. Multiple plugin directories can be specified by using the argument multiple times. If the plugin directory is set, Terraform will not automatically download plugins from the internet or check the third-party plugin directory.

We'll cover the `init` subcommand in more detail in Objective 6: Navigate Terraform Workflow, where we review the core workflow for Terraform.

Supplemental Information

- https://www.terraform.io/docs/configuration/providers.html#initialization[20]

Key Takeaways: Terraform will download HashiCorp distributed plugins automatically when a configuration is initialized. Third-party plugin must be downloaded manually and placed in a special folder. You can alter the default behavior by using plugin caching or specifying directories that already contain plugins.

3E: Explain When to Use and Not Use Provisioners and When to Use Local-Exec or Remote-Exec

When a resource is created, you may have some scripts or operations you would like to be performed locally or on the remote resource. Terraform provisioners are used to accomplish this goal. To a certain degree, they break the declarative and idempotent model of Terraform. The execution of provisioners is not necessarily atomic or idempotent, since it is executing an arbitrary script or instruction. Terraform is not able to track the results and status of provisioners in the same way it can for other resources. For this reason, HashiCorp recommends using provisioners as a **last resort only**.

That probably bears repeating, because HashiCorp has changed their stance over time on using provisioners. We can't guarantee any particular question will be on the exam, but HashiCorp seems to feel passionately about this. To repeat:

[20] https://www.terraform.io/docs/configuration/providers.html#initialization

> Don't use provisioners unless there is absolutely no other way to accomplish your goal.

Use cases

What are the use cases for a provisioner anyway? There's probably three that you've already encountered, or will in the near future.

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

The remote-exec provisioner allows you to connect to a remote machine via WinRM or SSH and run a script remotely. Note that this requires the machine to accept a remote connection. Instead of using remote-exec to pass data to a VM, use the tools in your cloud provider of choice to pass data. That could be the `user_data` argument in AWS or `custom_data` in Azure. All of the public cloud support some type of data exchange that doesn't require remote access to the machine.

In addition to the remote-exec provisioner, there are also provisioners for Chef, Puppet, Salt, and other configuration managers. They allow you to bootstrap the VM to use your config manager of choice. A better alternative is to create a custom image with the config manager software already installed and register it with your config management server at boot up using one of the data loading options mentioned in the previous paragraph.

You may wish to run a script locally as part of your Terraform configuration using the `local-exec` provisioner. In some cases there is a provider that already has the functionality you're looking for. For instance the the `Local` provider can interact with files on your local system. Still there may be situations where a local script is the only option.

Provisioner details

Despite your best efforts, you've decided that you need to use a provisioner after all. A provisioner is part of a resource configuration, and it can be fired off when a resource is created or destroyed. It cannot be fired off when a resource is altered.

When a creation-time provisioner fails, it sets the resource as tainted because it has no way of knowing how to remediate the issue outside of deleting the resource and trying again. This behavior can be altered using the `on_failure` argument. When a destroy-time provisioner fails, Terraform does not destroy the resource and tries to run the provisioner again during the next apply.

A single resource can have multiple provisioners described within its configuration block. The provisioners will be run in the order they appear.

Of course you can avoid all this nonsense by not using provisioners in the first place. We're just sayin'.

Supplemental Information

- https://www.terraform.io/docs/provisioners/index.html[21]
- https://www.terraform.io/docs/provisioners/local-exec.html[22]
- https://www.terraform.io/docs/provisioners/remote-exec.html[23]

Key Takeaways: Provisioners are a measure of last resort. `Remote-exec` will run a script on the remote machine through WinRM or ssh, and `local-exec` will run a script on your local machine.

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **3A: Handle Terraform and Provider Installation and Versioning:** Terraform is installed as a single binary. It uses providers to interact with other services, and you can specify the version of the provider.
- **3B: Describe Plugin Based Architecture:** Terraform makes use of plugins for its providers and provisioners instead of bundling them in the core binary.
- **3C: Demonstrate Using Multiple Providers:** You can have multiple different providers or multiple instances of the same provider in a configuration. It's important to understand how to use the `alias` argument and the use cases for multiple provider instances.
- **3D: Describe How Terraform Finds and Fetches Providers:** Terraform will download HashiCorp distributed plugins automatically when a configuration is initialized. Third-party plugin must be downloaded manually and placed in a special folder. You can alter the default behavior by using plugin caching or specifying directories that already contain plugins.
- **3E: Explain When to Use and Not Use Provisioners and When to Use Local-Exec or Remote-Exec:** Provisioners are a measure of last resort. `Remote-exec` will run a script on the remote machine through WinRM or ssh, and `local-exec` will run a script on your local machine.

[21] https://www.terraform.io/docs/provisioners/index.html
[22] https://www.terraform.io/docs/provisioners/local-exec.html
[23] https://www.terraform.io/docs/provisioners/remote-exec.html

In the next chapter we will dive into the terminal objective **Use the Terraform CLI (Outside of Core Workflow)**. That's the next item that is in the official list from HashiCorp. If you'd rather look at the core workflow first, check out Objective 6: Navigate Terraform Workflow that deals with the objective **Navigate Terraform Workflow**.

Objective 4: Use the Terraform CLI (Outside of Core Workflow)

This section will discuss the use of the Terraform Command-Line Interface (CLI) separate from the standard core workflow process (which is discussed in Objective 6: Navigate Terraform Workflow.

While we find it a little strange to focus on the larger CLI prior to delving into the core workflow, that is the order in which HashiCorp arranged the terminal objectives. You may want to review Objective 6 first and then come back to this chapter if you have never used Terraform before. The utility of the non-core CLI commands will be more obvious once you understand the core workflow.

4A: Understand the Help Command (terraform help)

As is common in many Command-Line Interfaces (CLIs), there is a help system in Terraform. When you first start out with Terraform, you learn the primary commands of `init`, `plan`, `apply`, and `destroy`.

If you come across a command that you're not familiar with, or want to learn more about, of course you can browse the web for it, and more than likely, you'll end up on the official Terraform Commands (CLI)[24] page.

But we can also accomplish the same thing using the command-line.

The full list of commands can be enumerated by running either `terraform` without arguments or `terraform help`.

```
$> terraform help
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
```

[24] https://www.terraform.io/docs/commands/index.html

```
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the configuration
    graph              Create a visual graph of Terraform resources
    import             Import existing infrastructure into Terraform
    init               Initialize a Terraform working directory
    output             Read an output from a state file
    plan               Generate and show an execution plan
    providers          Prints a tree of the providers used in the configuration
    refresh            Update local state file against real resources
    show               Inspect Terraform state or plan
    taint              Manually mark a resource for recreation
    untaint            Manually unmark a resource as tainted
    validate           Validates the Terraform files
    version            Prints the Terraform version
    workspace          Workspace management

All other commands:
    0.12upgrade        Rewrites pre-0.12 module source code for v0.12
    debug              Debug output management (experimental)
    force-unlock       Manually unlock the terraform state
    push               Obsolete command for Terraform Enterprise legacy (v1)
    state              Advanced state management
```

To use the help system to find out more about any of the commands, all we need to do us enter `terraform commandName -help`

For example, if we were to look at the help for the `terraform plan` command (using `terraform plan -help`), it would output a description and options for the command:

```
$> terraform plan -help
Usage: terraform plan [options] [DIR]

  Generates an execution plan for Terraform.

  This execution plan can be reviewed prior to running apply to get a
  sense for what Terraform will do. Optionally, the plan can be saved to
  a Terraform plan file, and apply can take this plan file to execute
  this plan exactly.

Options:

```

```
  -destroy            If set, a plan will be generated to destroy all resources
                      managed by the given configuration and state.

  -detailed-exitcode  Return detailed exit codes when the command exits. This
                      will change the meaning of exit codes to:
                      0 - Succeeded, diff is empty (no changes)
                      1 - Errored
                      2 - Succeeded, there is a diff

  -input=true         Ask for input for variables if not directly set.

  -lock=true          Lock the state file when locking is supported.

  -lock-timeout=0s    Duration to retry a state lock.

                      resource and its dependencies. This flag can be used
                      multiple times.

  -var 'foo=bar'      Set a variable in the Terraform configuration. This
                      flag can be set multiple times.

  -var-file=foo       Set variables in the Terraform configuration from
                      a file. If "terraform.tfvars" or any ".auto.tfvars"
                      files are present, they will be automatically loaded.
```

This can be a helpful way to discover additional options for a specific command, which could provide greater flexibility and automation options.

Supplemental Information

- https://www.terraform.io/docs/commands/index.html[25]

Key Takeaways: Terraform has a built-in help system that can be accessed from the command line. Adding the `-help` flag to the end of a command will print the instructions for use to `stdout`.

[25] https://www.terraform.io/docs/commands/index.html

4B: Given a Scenario: Choose When to Use Terraform FMT to Format Code

The `terraform fmt` command is used to rewrite Terraform configuration files to match the canonical format and style outlined by the Terraform language style conventions[26]. Note that the canonical format is constantly evolving to match the features and syntax in HashiCorp Configuration Language and Terraform. As a consequence, we recommend running `terraform fmt` after updating to a newer version of Terraform.

The `fmt` command has the syntax `terraform fmt [options] [DIR]`.

The `fmt` command scans the current directory for configuration files ending in `tf` and `tfvars` unless a DIR argument is specified. There are several options available for the `fmt` command including: list, write, diff, check, and recursive. All of these are optional, and further information about each can be found by running `terraform fmt -help`.

Scenario

With that in mind, let's consider a scenario. Let's assume that you're working as a team (though this can apply even if you're not). You write a Terraform template that looks like this:

Terraform configuration without formatting

```
provider "azurerm" {
  environment = "public"
}

module "vnet" {
    source = "Azure/vnet/azurerm"
    resource_group_name = "tacos"
    location = "westus"
    address_space = "10.0.0.0/16"
    subnet_prefixes = ["10.0.1.0/24", "10.0.2.0/24"]
    subnet_names = ["cheese","beans"]
}
```

It's readable and organized well. But, compare the difference after running `terraform fmt`.

[26] https://www.terraform.io/docs/configuration/style.html

Terraform configuration with formatting

```
provider "azurerm" {
  environment = "public"
}

module "vnet" {
  source              = "Azure/vnet/azurerm"
  resource_group_name = "tacos"
  location            = "westus"
  address_space       = "10.0.0.0/16"
  subnet_prefixes     = ["10.0.1.0/24", "10.0.2.0/24"]
  subnet_names        = ["cheese", "beans"]
}
```

Notice how the indentation, equal signs, etc. are all aligned? Although both code examples are readable and will run successfully, the `fmt` version makes it more uniform.

Especially when working as a team, it's a good practice to run `terraform fmt` as part of a code commit, so that everyone's code is formatted the same way.

> One of the `terraform fmt` options is `-recursive`. If you would like to update a collection of configurations, you can simply run the `fmt` command on the parent directory and it will process the contents of all subdirectories!

Supplemental Information

- https://www.terraform.io/docs/commands/fmt.html[27]

Key Takeaways: Terraform `fmt` applies Terraform canonical format to any `tf` or `tfvars` files. Having consistent formatting of code helps when sharing or collaborating with others.

[27] https://www.terraform.io/docs/commands/fmt.html

4C: Given a Scenario: Choose When to Use Terraform TAINT to Taint Terraform Resources

The `terraform taint` command does exactly what you might expect, it taints a resource in the state. Terraform will destroy and recreate the resource the next time the configuration is applied.

Running `terraform taint` on its own will not destroy the resource immediately, it simply edits the state. By running `terraform plan` on the root module, you will see that a resource has been tainted and will be recreated on the next apply. Running `terraform apply` will execute the plan, destroying then recreating the resource.

The `taint` command has the syntax `terraform taint [options] <address>`.

The `<address>` argument refers to the address of a resource using the same syntax you would use inside a configuration. For example, let's say we have an aws_instance resource with a name label of nginx. Running `terraform taint aws_instance.nginx` will result in the following output.

```
Resource instance aws_instance.nginx has been marked as tainted.
```

If you view the state file using the command `terraform show` you will see the resource `aws_instance.nginx` flagged as such.

```
# aws_instance.nginx: (tainted)
resource "aws_instance" "nginx"
```

Recreating a resource is useful for situations where a change is not obvious to Terraform, or the resource's internal state is known to be bad. Terraform does not have visibility into the inner state of some resources, for instance a virtual machine. As long as the virtual machine's resource attributes seem valid, Terraform will not flag it for recreation. As an example, you may make a charge to the startup script for a virtual machine that will not take effect until the machine is recreated. Terraform does not track the startup script as a resource attribute, so it has no knowledge that something about the resource has changed. By tainting the virtual machine resource, Terraform will know that the resource must be replaced.

Other resources in the configuration may have properties that are dependent on the resource being recreated, therefore a tainted resource may result in additional changes in the remainder of the configuration. For instance, a load balancer connected to the virtual machine will need to be updated with the new virtual machine's network interface to direct traffic properly.

Scenario

Let's say we've modified some existing code, but when we execute `terraform apply` the plan fails. Terraform does not roll-back any resources that it's deployed, even if it was deployed partially. Using

taint is a way of saying, "This resource may not be right." That way, when you re-run `terraform apply` (assuming you corrected whatever may have caused the partial apply failure), Terraform will attempt to destroy and recreate any tainted resources.

Running with our resource example of `aws_instance.nginx`, once we have determined that the instance needs to be recreated and have tainted our resource accordingly, we can run `terraform plan` and receive the following output.

```
  # aws_instance.nginx is tainted, so must be replaced
-/+ resource "aws_instance" "nginx"

#output truncated...

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

It is also possible to remove the taint on an object by using the command `terraform untaint`. You might use this to remove a taint that was applied by accident, or to clear a taint applied by Terraform on a failed resource provisioning.

Supplemental Information

- https://www.terraform.io/docs/commands/taint.html[28]

Key Takeaways: Terraform taint is used to mark resources for destruction and recreation. Terraform may not know that a resource is no longer valid or incorrectly configured. Taint is a way to let Terraform know.

4D: Given a Scenario: Choose When to Use Terraform IMPORT to Import Existing Infrastructure into your Terraform State

The terraform import command is used to import existing resources into Terraform. This enables you to take existing resources and bring them under Terraform's management.

Most environments are not going to be greenfield deployments with no existing infrastructure. The `import` command is a way to bring that existing infrastructure under Terraform's management without altering the underlying resources. The `import` command can also be useful in situations

[28] https://www.terraform.io/docs/commands/taint.html

where an administrator manually created a resource outside of Terraform's management (whoops!) and that resource needs to be brought into the fold. Ideally, once you've adopted Terraform, all new resources will be created and managed through Terraform.

As of this writing, the current version of Terraform does not generate the configuration for an imported resource, it simply adds the resource to the state. It is up to you to write a configuration that matches the existing resources prior to running the `import` command. Admittedly, this is an inefficient process, but there are benefits. Writing a configuration for existing resources may give you additional insight into how those resources were created, and highlight any deviations from best practices. HashiCorp has indicated that a future version of Terraform will also generate configuration.

> HashiCorp has been indicating this since at least version 0.9. We wouldn't recommend waiting for it.

Scenario

It's only been in the last little while that the whole 'DevOps' thing has caught on (from an IT Ops perspective). Up until now, most organizations that are/have been using the cloud, have been deploying resources either by hand (via the GUI), or through scripts (like PowerShell, Bash, etc.). True, some may have been using APIs or templates (which is closer to what Terraform is/does), but even these are a run-once method.

With Terraform's state files (covered in Objective 2B: Explain the Benefits of State, and Objective 7: Implement and Maintain State), we have a way for on-going resource management.

When an organization starts using Terraform, they are more likely to already have at least some resources deployed in the cloud.

In this case, let's assume that your organization has multiple VMs, perhaps load balanced for high availability. You want to manage them with Terraform so that you can make a simple code change (ie. nodeCount = 3) and quickly grow/shrink your cluster.

At this point you have 3 options:

1. Delete the existing infrastructure and redeploy using a Terraform configuration (not really an option if it's in Production).
2. Create a new set of infrastructure using a Terraform configuration and migrate the data/content to the new environment (which will cause an increase in expense due to doubling the infrastructure, albeit temporarily).
3. Create a Terraform configuration (as if you are deploying a new set of identical infrastructure), and use `terraform import` to bring the existing deployments under Terraform management.

By using `terraform import` you do not have to re-create any infrastructure (thus removing the issue of downtime), and you can manage your entire cloud environment using Infrastructure as Code (IaC).

> If you'd like an easy way to play with the import command, simply use the `terraform state rm` command to remove a resource from the state file. The physical resource will continue to exist. Then use the `terraform import` command to import it back into the state file.

Supplemental Information

- https://www.terraform.io/docs/commands/import.html[29]
- https://www.terraform.io/docs/import/index.html[30]

Key Takeaways: Terraform import can bring existing resources into management by Terraform. Import alone does not create the necessary configuration to import a resource, that must be done manually.

4E: Given a Scenario: Choose When to Use Terraform WORKSPACE to Create Workspaces

A workspace in Terraform Cloud and Terraform open-source mean two slightly different things. For the purposes of the certification, we are focusing on the Terraform open-source option. We will cover Terraform Cloud workspaces and the difference in Objective 9: Understand Terraform Enterprise Capabilities. Workspaces in Terraform open-source are simply independently managed state files that share a common configuration.

> In the olden days of Terraform, there was a similar construct called an environment which was managed using the `terraform env` command. That command still exists, but it is basically an alias for `terraform workspace` and may be deprecated in a future release.

[29] https://www.terraform.io/docs/commands/import.html
[30] https://www.terraform.io/docs/import/index.html

Many organizations have multiple environments, including development, testing, staging, and production. Each Terraform workspace is meant to represent the deployment of a configuration to one of these separate working environments. This enables you to use a single configuration to manage the same deployment in all environments, adding a high level of consistency as applications roll through the lower environments to production.

HashiCorp recommends that each configuration represent a separate architecture component in an environment. For instance, the networking for an environment might be expressed as a configuration, while one of the web apps running on the network could be expressed as a separate configuration.

Each workspace will have persistent data stored in the backend, most often as a state file although it could be stored as JSON in a database as well. Terraform starts with a *default* workspace that is special in two regards, it is selected by default and cannot be deleted. All other workspaces can be created, selected, and deleted.

A backend refers to the location where the Terraform state is stored, and not all backends support the use of workspaces to store multiple states. The local file backend and Amazon Simple Storage Service (AWS S3) both support workspaces, etcd does not. The full list of supported backends can be found in the referenced links below.

Workspaces are managed with the `terraform workspace` set of commands. Below is a list of subcommands available for use.

- `new` - Create a new workspace and select it
- `select` - Select an existing workspace
- `list` - List the existing workspaces
- `show` - Show the name of the current workspace
- `delete` - Delete an empty workspace

It should be noted that in order to run `terraform workspace delete`, the workspace must not currently be selected and the state file must be empty. The `-force` flag can be specified if the state file is not empty, and the resources created by the state file will be abandoned.

Workflow

To create a new workspace, you can simply run `terraform workspace new development` which will result in the following output.

```
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.
```

If you are using a local file backend, Terraform will create the folder `terraform.state.d` inside the current working directory and create an additional subfolder for each workspace as shown below.

```
├──.terraform
│   └──plugins
│       └──windows_amd64
└──terraform.tfstate.d
    └──development
    └──staging
    └──production
```

Each subfolder will be empty until resources are created in the workspace.

Running `terraform plan` will result in a plan within the context of the current workspace. Any resources that were created on other workspaces **still exist**, but are not part of the state for the current workspace. Terraform does not know about these other resources, so be sure to differentiate the resources deployed by each workspace to avoid one workspace colliding with another.

Terraform workspaces are commonly used in large organizations to delegate responsibilities and roles to different team members. Each team would be granted permissions to perform updates on workspaces related to their role. Additionally, production and staging type environments may have different permissions applied to their workspaces to further restrict who has access to deploy updates to these sensitive environments.

The "current workspace" name is stored locally in the `.terraform` directory. This allows multiple team members to work on different workspaces concurrently.

Scenario

Let's assume you're working in a large organization. In this organization there is a Networking team, a Governance team, and an Application team. Also, in this organization, you have multiple environment types, like Production, Staging, and Development.

Each team is responsible for their own Terraform configurations that create and manage their respective components. In order to provide separation of roles and permissions across all the environments and configurations, you could create the following workspaces:

- networking-dev

- networking-stage
- networking-prod
- governance-dev
- governance-stage
- governance-prod
- application-dev
- application-stage
- application-prod

This will allow each respective team to keep their work separate (even if it's contained in the same repository), and also allow for independent progression through the development and staging of changes.

Supplemental Information

- https://www.terraform.io/docs/state/workspaces.html[31]
- https://www.terraform.io/docs/cloud/guides/recommended-practices/part1.html#the-recommended-terraform-workspace-structure[32]

Key Takeaways: Terraform workspaces are a combination of a state file and configuration. They are used to create multiple instances of a configuration in separate environments. You cannot delete the *default* workspace.

4F: Given a Scenario: Choose When to Use Terraform STATE to View Terraform State

The state maintained by Terraform is a JSON document stored either locally or on a remote backend. Terraform users are **highly** discouraged from making direct edits to the file. *Seriously, don't do that.* There are some occasions where you will need to directly interact with the Terraform state, and in those cases you will use the `terraform state` command and its subcommands to assist you.

`terraform state` has a set of subcommand commands. Below is a list of subcommands available for use.

- `list` - List resources in the state
- `mv` - Move an item in the state

[31] https://www.terraform.io/docs/state/workspaces.html
[32] https://www.terraform.io/docs/cloud/guides/recommended-practices/part1.html#the-recommended-terraform-workspace-structure

- `pull` - Pull current state and output to stdout
- `push` - Update remote state from a local state file
- `rm` - Remove instances from the state
- `show` - Show a resource in the state

Resource Addressing

When using the `terraform state` command you are probably going to want to interact with specific resource instances defined in the state. They can be accessed by what HashiCorp calls *standard address syntax*, or what you might think of as "the way I reference resources in my configuration file."

References to resources in the state includes individual resource instances, like an `aws_instance` or groups of resources created by `count` or `for_each` resource arguments or a module.

The basic syntax is `[module path][resource spec]`. Technically, all resource instances are contained within a module. Instances defined directly in a configuration are part of the root module, which is implicit when using *standard address syntax*.

The `resource_spec` component is the path to the instance within a given module. For example, a subnet defined by the Azure vnet module named *my-vnet* could be referred to by the path `module.my-vnet.azurerm_subnets.subnet[0]`.

More generally the form could be expressed as `module.module_name_label.resource_type.resource_name_label.resource_attribute[element]`. Nested child modules would simply extend the form until the proper level is reached to refer to the resource instance.

List

The `terraform state list` command is used to list resources within a Terraform state.

The `list` subcommand has the following syntax: `terraform state list [options] [address...]`

The options allow you to specify a particular state file using `-state=statefile` or a specific id using `-id=ID`. The address refers to a particular resource within the state file as described in the preceding *Resource Addressing* section. If no address is given, then all resources will be returned by the command.

Scenario

Going back to the aws_instance example in the `taint` section, let's say we have deployed an aws_instance in the default VPC and we need to see the list of all resources that were created by the configuration. Running `terraform state list` results in the following output.

```
$> terraform state list
data.aws_ami.aws-linux
aws_default_vpc.default
aws_instance.nginx
aws_security_group.allow_ssh
```

The `list` subcommand is an excellent way to see all the resources in your deployment, and then choose one to perform further operations on.

Move

The `terraform state mv` command is used to move items in a Terraform state.

The `mv` subcommand has the following syntax: `terraform state mv [options] SOURCE DESTINATION`

Within the options there are a few important flags to highlight.

- `dry-run` - Think of this as a what-if. It will tell you what it will do, without making any changes.
- `state` - This is the path of the source state file, which needs to be specified if you're not using the `terraform.tfstate` file or a remote backend.
- `state-out` - This is the path of the destination file. Terraform uses the source state file unless this is specified.

You can use this command to move a single resource, single instance of a resource, entire modules, and more. By specifying a destination path, you can move those resources to an entirely new state file.

> The changes made by `mv` are kind of a big deal. That's why Terraform creates a backup of the source and destination state files. If something goes awry, you can always revert to the previous state file.

There are a few reasons to use the `mv` command. If an existing resource needs to be renamed, but you don't want to destroy and recreate it, you can update the name in the configuration and use the `mv` command to "move" the resource to its new name. Running `terraform plan` will result in no necessary changes.

Scenario

Let's say that we want to change out the name label for the resource instance `aws_instance.nginx` to `aws_instance.web`. First we would update the relevant code from this:

```
resource "aws_instance" "nginx"
```

to this:

```
resource "aws_instance" "web"
```

If we run `terraform plan` without updating the state, it will result in the following output.

```
  # aws_instance.nginx will be destroyed
  - resource "aws_instance" "nginx" {
[...]
  # aws_instance.web will be created
  + resource "aws_instance" "web" {
[...]
Plan: 1 to add, 0 to change, 1 to destroy.
```

Now let's run the `mv` command.

```
$> terraform state mv aws_instance.nginx aws_instance.web
Move "aws_instance.nginx" to "aws_instance.web"
Successfully moved 1 object(s).
```

If we run `terraform plan` again, we get the following output.

```
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

You can also use the `mv` command to move resources to a module within the configuration. Let's say you are replacing the existing resource definitions with a module, but do not want to destroy and recreate those resources. By using the `mv` command, you can put those resources inside the child module before running `terraform plan`.

There may also be cases where a configuration is too big or unwieldy and needs to be broken into smaller parts. Using the `mv` command, you can move a collection of resources to a new state file, and then copy the configuration for those resources to a new directory.

Pull

The `terraform state pull` command is used to manually download and output the state from a remote backend. This command also works with the local backend.

The `pull` subcommand has the following syntax: `terraform state pull [options]`

This command is equivalent to viewing the contents of the Terraform state directly, in a read-only mode. The output of the command is sent in JSON format to *stdout* and can be piped to another command - like `jq` - to extract data.

The primary use for the command is to read the contents of remote state, since if you were using the local backend you could just as easily use `cat` or `jq` on the local file.

Push

The `terraform state push` command is used to manually upload a local state file to a remote backend. This command also works with the local backend.

> You probably won't need to use this command. If you *think* you do, take a moment and reconsider. You are taking a local file and replacing the remote state with the contents of that local file. Here there be dragons.

The `push` subcommand has the following syntax: `terraform state push [options] PATH`

The PATH argument defines the local path of the file that will be pushed to the remote state. You can also specify `stdin` as the PATH by using the value "-".

Since this is such a risky manuever, Terraform checks a couple things to prevent you from doing something dumb. If the *lineage* value in the state is different or the *serial* value of the destination state is higher that the local state, Terraform will prevent the push. Lineage is specific to a state and workspace, so if the lineage is different the configuration is likely to be different. A higher serial number on the destination state suggests that it is a newer state and the state being pushed has stale data.

Despite all of these safety checks, if you are certain about your action, you can specify the `-force` flag and Terraform will overwrite the remote state anyway.

Remove

The `terraform state rm` command is used to remove items from the Terraform state.

The `rm` subcommand has the following syntax: `terraform state rm [options] ADDRESS`

You can use this command to remove a single resource, single instance of a resource, entire modules, and more. The resources removed by the command will be removed from the state file, but will not be destroyed in the target environment.

The most common use for the `rm` command is to remove a resource from Terraform management. Perhaps the resource will be added to another configuration and imported. Perhaps Terraform is being replaced by something else. In either case, the goal is to remove the reference to the resource without deleting the actual resource.

Scenario

Let's say that we are no longer going to manage our aws_instance with Terraform. First we can run the `rm` command.

```
$> terraform state rm aws_instance.web
Removed aws_instance.web
Successfully removed 1 resource instance(s).
```

Removing the resource from the state file also does not remove it from the configuration. The resource and all references should be removed from the configuration before running another `plan` or `apply`. Otherwise Terraform may attempt to create the resource again or throw errors due to invalid references in the configuration file.

If we run `terraform plan` before updating the configuration, we will see the following output.

```
Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
[...]

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

Terraform is attempting to recreate the missing aws_instance specified in the configuration. If we comment out the resource, remove any references, and run `terraform plan` again, the following output is generated.

```
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

Show

The `terraform state show` command is used to show the attributes of a single resource in the Terraform state.

The `show` subcommand has the following syntax: `terraform state show [options] ADDRESS`

The ADDRESS specified by the show command must be an instance of a resource and not a grouping of resources created by `count` or `for_each`. The attributes of the resource are listed in alphabetical order and are sent to `stdout` for consumption by some type of parsing engine.

Scenario

Let's say we want to view the properties of our aws_instance resource. The console output below shows the results of running `terraform state show aws_instance.web`.

```
# aws_instance.web:
resource "aws_instance" "web" {
    ami                          = "ami-00eb20669e0990cb4"
    arn                          = "arn:aws:ec2:us-east-1:123456789123:instance/i-07\
8e3d0f003ac07bf"
    associate_public_ip_address  = true
    availability_zone            = "us-east-1d"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
[...]
}
```

The most common use for this command is to inspect the values stored in particular attributes. The output could be used by a secondary process to initiate post-configuration of an instance or be fed into a Configuration Management Database (CMDB).

Supplemental Information

- https://www.terraform.io/docs/commands/state/index.html[33]

Key Takeaways: Terraform state commands are meant to view and manipulate the content of the state. Commands like `mv`, `rm`, and `push` edit the existing state file directly, something to be done with caution. The commands `list`, `pull`, and `show` are for viewing the contents of the state file in different formats.

[33] https://www.terraform.io/docs/commands/state/index.html

4G: Given a Scenario: Choose When to Enable Verbose Logging and What the Outcome/Value Is

Terraform can generate detailed logs by setting the environment variable `TF_LOG` to a particular level of verbosity. There are five log levels. From most verbose to least they are TRACE, DEBUG, INFO, WARN and ERROR. The output of the logs will be sent to `stderr`. If the `TF_LOG` variable is set to anything other than one of the listed log levels (ie. TF_LOG=HAMSTER), Terraform will set the log verbosity to TRACE.

In addition to logs being sent to `stderr`, Terraform will also generate a log file if the executable crashes. The log file contains the debug logs, panic message, and backtrace saved to the crash.log file.

The crash.log is unlikely to be of much use to regular Terraform users. It is meant to be submitted along with a bug report to Terraform support, so that they can troubleshoot the issue and provide a resolution or mitigation. Additional guidance can be found on the Terraform website at the link found in the supplemental information below.

Scenario

You're trying to deploy a new configuration, but `terraform apply` keeps failing. The standard error output (stderr) is not providing enough information to lead to a root cause. In that situation, you can set the `TF_LOG` environment variable to INFO to get detailed logging of everything the Terraform executable is attempting to do. If the level of detail is not sufficient, then you can adjust logging to DEBUG or TRACE to see even more information.

This level of logging can be especially useful in the context of automation, where a human is not sitting and watching the executable run.

Supplemental Information

- https://www.terraform.io/docs/internals/debugging.html[34]

Key Takeaways: Terraform logging can be configured by setting the `TF_LOG` environment variable. If Terraform crashes, it creates a crash.log file with the debug logs, backtrace, and panic report.

[34] https://www.terraform.io/docs/internals/debugging.html

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **4A: Understand the Help Command:** Terraform has a built-in help system that can be accessed from the command line. Adding the `-help` flag to the end of a command will print the instructions for use to `stdout`.
- **4B: Given a Scenario: Choose When to Use Terraform FMT to Format Code:** Terraform `fmt` applies Terraform canonical format to any `tf` or `tfvars` files. Having consistent formatting of code helps when sharing or collaborating with others.
- **4C: Given a Scenario: Choose When to Use Terraform TAINT to Taint Terraform Resources:** Terraform taint is used to mark resources for destruction and recreation. Terraform may not know that a resource is no longer valid or incorrectly configured. Taint is a way to let Terraform know.
- **4D: Given a Scenario: Choose When to Use Terraform IMPORT to Import Existing Infrastructure into your Terraform State:** Terraform import can bring existing resources into management by Terraform. Import alone does not create the necessary configuration to import a resource, that must be done manually.
- **4E: Given a Scenario: Choose When to Use Terraform WORKSPACE to Create Workspaces:** Terraform workspaces are a combination of a state file and configuration. They are used to create multiple instances of a configuration in separate environments. You cannot delete the *default* workspace.
- **4F: Given a Scenario: Choose When to Use Terraform STATE to View Terraform State:** Terraform state commands are meant to view and manipulate the content of the state. Commands like `mv`, `rm`, and `push` edit the existing state file directly, something to be done with caution. The commands `list`, `pull`, and `show` are for viewing the contents of the state file in different formats.
- **4G: Given a Scenario: Choose When to Enable Verbose Logging and What the Outcome/-Value Is:** Terraform logging can be configured by setting the `TF_LOG` environment variable. If Terraform crashes, it creates a crash.log file with the debug logs, backtrace, and panic report.

In the next chapter we will explore how modules are used in Terraform with the terminal objective **Interact with Terraform Modules**.

Objective 5: Interact with Terraform Modules

Whether or not you realize it, you've been using Terraform modules all along. What is a Terraform module? It is simply a configuration that defines inputs, resources, and outputs. And all of those are optional! When you create a set of `tf` or `tf.json` files in a directory, that is a module. The main configuration you are working with is known as the *root module*, and it can invoke other modules to create resources.

There is a hierarchal structure to modules where the root module invokes a child module, which in turn can invoke another child module. Consider a root module that invokes a child module (*lb-vms*) to create a load balanced cluster of Azure VMs. The lb-vms module may invoke a *vnet* module to create the necessary vnet and subnets for the load balanced cluster. If you were to use `terraform state list` (discussed in Objective 4: Use the Terraform CLI), you would see a resource address for the subnet like `module.lb-vms.module.vnet.azurerm_subnet.subnet[0]`.

Modules are invoked by using the `module` script block keyword and applying a name label.

```
module <name_label> {}
```

In Terraform parlance, the module invoking the `module` code block is referred to as the *calling module* and the module being invoked is referred to as the *child module*.

5A: Contrast Module Source Options

In order to use a module, Terraform needs to know where to find the files that make up the module. The `source` argument in the `module` block defines where the module files can be found.

The actual files can be stored in a number of locations:

- Local paths[35]
- Terraform Registry[36]
- GitHub[37]
- Bitbucket[38]
- Generic Git[39], Mercurial[40] repositories

[35] https://www.terraform.io/docs/modules/sources.html#local-paths
[36] https://www.terraform.io/docs/modules/sources.html#terraform-registry
[37] https://www.terraform.io/docs/modules/sources.html#github
[38] https://www.terraform.io/docs/modules/sources.html#bitbucket
[39] https://www.terraform.io/docs/modules/sources.html#generic-git-repository
[40] https://www.terraform.io/docs/modules/sources.html#generic-mercurial-repository

- HTTP URLs[41]
- S3 buckets[42]
- GCS buckets[43]

> The Terraform Registry in particular is important to highlight. On the publicly available Terraform Registry, found at registry.terraform.io[a], there are verified modules maintained by third-party vendors. There are verified modules for common cloud components like an AWS VPC or an Azure AKS cluster. Since they are maintained and verified by the vendors, there is a high degree of likelihood that the module will function as expected.
>
> [a] https://registry.terraform.io/

If the source is a remote destination, Terraform will copy the files to the local `.terraform` directory of the root module when `terraform init` is run. You can also use `terraform get` to retrieve the modules separately from the initialization routine.

When trying to determine where to store modules you are writing yourself, the recommendation from HashiCorp is to store modules that are tightly coupled with your configuration locally. If a module is a reusable component that other teams might leverage, then it should be stored on a shared location, either public or private.

Supplemental Information

- https://www.terraform.io/docs/modules/sources.html[44]

Key Takeaways: Modules are reusable configurations that can be called by a root module or other child modules. The source for the module can be the local file system or a remote repository, including the Terraform Registry that includes verified modules from third-party vendors.

5B: Interact with Module Inputs and Outputs

Inputs

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared

[41] https://www.terraform.io/docs/modules/sources.html#http-urls
[42] https://www.terraform.io/docs/modules/sources.html#s3-bucket
[43] https://www.terraform.io/docs/modules/sources.html#gcs-bucket
[44] https://www.terraform.io/docs/modules/sources.html

between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the `module` block.

The example shown below passes inputs for the official *vpc* module on the Terraform registry. The inputs can be variables from the root module, hard-coded values in the configuration, or values from other resources in the root module or child modules.

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "${var.prefix}-vpc"
  cidr = var.cidr_address

  azs             = var.vpc_azs
  private_subnets = var.vpc_private_subnets
  public_subnets  = var.vpc_private_subnets

  enable_nat_gateway = true
  enable_vpn_gateway = true

}
```

In the same way that you define variables for the root module, each input variable accepted by a child module must be declared using a `variable` block. The example below shows some of the inputs declared in the `variables.tf` file of the official *vpc* module.

```
variable "name" {
  description = "Name to be used on all the resources as identifier"
  type        = string
  default     = ""
}

variable "cidr" {
  description = "The CIDR block for the VPC. Default value is a valid CIDR, but not \
acceptable by AWS and should be overridden"
  type        = string
  default     = "0.0.0.0/0"
}
```

The label after the `variable` keyword is a name for the variable, which **must** be unique among all variables in the same module. This name is used to assign a value to the variable from the calling module and to reference the variable's value from within the module.

The variable declaration can optionally include a `type` argument to specify what value type is accepted for the variable. Modules can accept simple and complex types, for instance a valid type could be `string` or `list(int)`. This also means that entire resources can be passed as an input, since resources are a complex object type. A module could accept an existing *azurerm_virtual_network* as an input, and use the attributes of the resource in the configuration.

The variable declaration can also include a `default` argument. If present, the variable is considered by Terraform to be optional, and the default value will be used if no value is set when calling the module. The `default` argument requires a literal value and cannot reference other objects in the configuration.

The variable declaration can also include a `description` argument. When creating a module for use by other teams or people, it is highly recommended to use the description field to provide guidance for those consuming the module.

More information about declaring variables can be found in Objective 8: Read, Generate, and Modify Configuration.

Outputs

Output values expose the attributes stored in local resources for external consumption. That might sound a bit confusing. Let's look a few uses for outputs.

- A child module can use outputs to expose a subset of its resource attributes to a calling module.
- A root module can use outputs to print certain values in the CLI output after running `terraform apply`.
- When using remote state, root module outputs can be accessed by other configurations via a `terraform_remote_state` data source.

When you create a resource in a module, the attributes of that resource can be referenced throughout the rest of that module. But those attributes are not directly available for consumption by other modules or as a data source when using remote state. Output values are a way to name, transform, and expose the information stored in attributes to the consumer of your module. If an attribute is not exposed as an output, it will not be available to a calling module.

An output value exported by a module is declared using an `output` block. The example below shows one of the outputs of the official *vpc* module.

```
output "vpc_id" {
  description = "The ID of the VPC"
  value       = concat(aws_vpc.this.*.id, [""])[0]
}
```

The label immediately after the `output` keyword is the name, which must be a valid identifier. When an output is defined in a root module, the output name and value is shown when the configuration is run. When an output is defined in a child module, the name is used by the calling module to reference the exposed value in the format: `module.module_name_label.output_name_label`.

The `description` argument is used to inform the module user what is contained in the output. When creating a module for use by other teams or people, it is highly recommended to use the description field to provide guidance for those consuming the module.

The value argument takes an expression whose result is to be returned to the user. Any valid expression is allowed as an output value. Previous versions of Terraform could only return a `string` as a value back to the calling module. Terraform 0.12 added the ability to return complex values, including entire resources. Rather than returning each attribute of a VPC as a string, a module could return the entire VPC resource and the calling module could reference the attributes within that resource.

Supplemental Information

- https://www.terraform.io/docs/configuration/variables.html[45]
- https://www.terraform.io/docs/configuration/outputs.html[46]

Key Takeaways: Variables defined in a module can be used as inputs for that module. The attributes of resources within a module are only available to the calling module if exposed as outputs. Inputs and outputs can be simple or complex types.

5C: Describe Variable Scope Within Modules/Child Modules

You can configure inputs and outputs for modules: an API interface to your modules. This allows you to customize them for specific requirements, while your code remains as DRY[47] and reusable as possible.

To Terraform, every directory containing configuration is automatically a module. Using modules just means referencing that configuration explicitly. References to modules are created with the module block, and resources created by the child module are exposed through outputs defined in the child module.

[45] https://www.terraform.io/docs/configuration/variables.html
[46] https://www.terraform.io/docs/configuration/outputs.html
[47] https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

A local value defined in a module is only available in the context of that module. The calling module and any other child modules invoked by that calling module will not have access to those local values unless they are exposed as output values. For instance, let's say you have a root module that invokes two child modules A and B. Any local values created in module A will not be available to the root module or module B.

Variables are contained within the module itself, however, if you are calling a module, then you will also need to pass values for any variables that do not have a default value as part of that module call. Those are the inputs for the module. The child module does not have access to variables defined in the calling module, unless they are explicitly passed as inputs. For instance, let's say you have a root module with a variable called `naming_prefix`.

```
variable "naming_prefix {
    type        = string
    description = "Variable to be used as a prefix for naming resources"
    default     = "tfm"
}
```

In the root module you invoke a child module to create a set of virtual machines.

```
module "virtual_machine_set" {
    source   = "../modules/virtual_machine_set"

    name     = "vms"
    num_vms  = 3
}
```

The child module cannot access the `naming_prefix` variable in the calling module by doing any of the following:

```
# This would refer to a variable naming_prefix in the child module
var.naming_prefix

# This is not a valid object reference
root.var.naming_prefix

# Neither is this!
parent.var.naming_prefix
```

None of those expressions would work because child modules do not have access the the values of resources in the calling module. The inverse is also true. Let's say you have a local variable defined in the child module:

```
locals {
    vm_name = "${var.prefix}-vms"
}
```

The calling module cannot refer to the local variable in the child module by doing any of the following:

```
# This would refer to a local variable vm_name in the calling module
local.vm_name

# This is not a valid object reference
module.virtual_machine_set.local.vm_name

# Neither is this!
module.virtual_machine_set.vm_name
```

The only way to expose the `local.vm_name` variable is to create an output in the child module.

It is possible to use the outputs of one child module and pass them to another child module. Look at this example. In it, we are calling a module to create some Network Security Groups (NSGs). The NSG module itself is expecting a bunch of variables like Tag values, Resource Group Name, and Network Security Group names. Notice that we're also populating the variable values that we're passing into the module, from output received from a different module (the 'VNets-Core' module), which was called prior to this one.

```
module "nsgs-Core" {
    source     = "../../nsgs/core"

    ResourceID = var.ResourceID
    CostCenter = var.CostCenter

    CoreNetworking-ResourceGroup-Name = azurerm_resource_group.CoreNetworking.name

    CoreProduction-NSG-Name = module.vnets-Core.CoreProduction-NSG-Name
}
```

Supplemental Information

- https://www.terraform.io/docs/modules/index.html[48]

[48] https://www.terraform.io/docs/modules/index.html

Key Takeaways: Calling modules cannot access the values stored in the variables and resources of the child modules, and child modules cannot access the variables and resources of the calling module. Sharing of information has to be explicit through input variables and output values.

5D: Discover Modules from the Public Module Registry

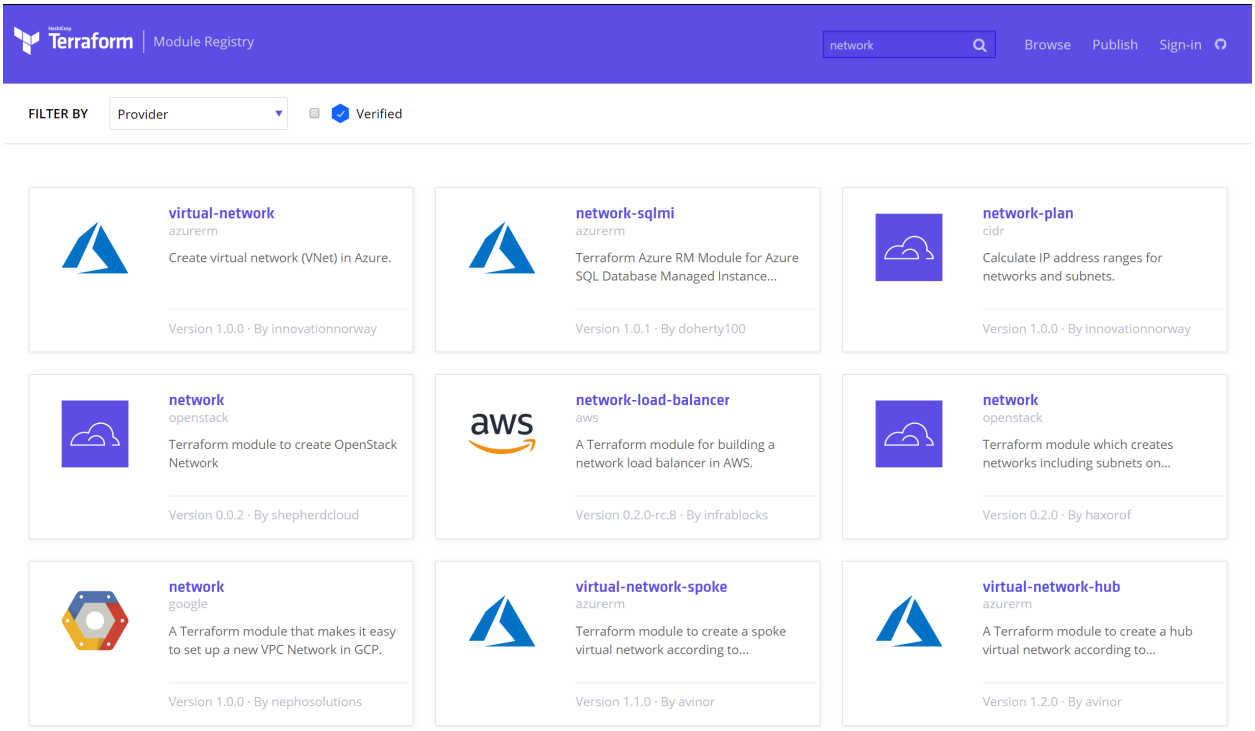
If you navigate to https://registry.terraform.io/[49] you will see that there are public modules made available for many of the popular public cloud providers, including Amazon Web Services (AWS), Microsoft Azure, and Google Compute Platform (GCP); and even modules for Alibaba Cloud (AliCloud), and Oracle Public Cloud (OPC).

All modules published on the public registry are open-source, with the code available on GitHub for review. The is no charge for using any of the modules published on the Terraform registry, aside from the cost of any resources created on a public cloud provider when using a module.

When you select an existing public module, you will see all the related information, including if there are multiple versions, examples of how to use it, what inputs it requires (or are optional), the outputs it produces; even any dependencies it has or the specific resource types that it creates. There will also be a link to the GitHub repository with the code for the module.

For example, if you were to search for 'network', you would see results for multiple cloud providers, and even multiple examples including Hub-and-Spoke, Load Balancers, and SQL Managed Instance.

[49] https://registry.terraform.io/



Terraform Module Registry Example

When consuming a publicly available module, it is best to specify a version in the module configuration block, like in the example below.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.20.0"

}
```

Specifying a particular minor or major version number will reduce the chance that untested changes will impact your existing infrastructure.

Supplemental Information

- https://www.terraform.io/docs/modules/sources.html#terraform-registry[50]

Key Takeaways: HashiCorp hosts a public registry for Terraform modules with many official modules from major cloud providers.

[50] https://www.terraform.io/docs/modules/sources.html#terraform-registry

5E: Defining Module Version

As mentioned in the previous section, we recommend explicitly setting the acceptable version number for any child modules being invoked in order to prevent unexpected and unwanted changes. This is especially true if you are not the one maintaining the module.

The version number for a module is defined by using the `version` argument in the module block:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.20.0"

}
```

The `version` argument value may be a single explicit version or a version constraint expression. It is possible to specify an exact version as shown in the example with a simple = operator. It is also possible to use additional operators such as the following:

- `>= 2.20.0`: version 2.20.0 or newer
- `<= 2.20.0`: version 2.20.0 or older
- `~> 2.20.0`: any version in the 2.20.x family, version 2.21.x would not work
- `>= 2.0.0, <= 2.20.0`: any version between 2.0.0 and 2.20.0 inclusive

The operator `~>` is particularly useful to stay on the same major version, but include minor version updates.

The version constraint will only work if version numbers are available for the module. Locally defined modules do not have a concept of version number, therefore the version constraint can only be used on modules sourced from a module registry. It is possible to host a private registry using Terraform Cloud[51] for use by those internal to your organization.

Supplemental Information

- https://www.terraform.io/docs/registry/modules/publish.html#releasing-new-versions[52]

Key Takeaways When using a module registry, it is best to specify a particular version of a child module to avoid unexpected and unwanted updates.

[51] https://www.terraform.io/docs/cloud/registry/publish.html
[52] https://www.terraform.io/docs/registry/modules/publish.html#releasing-new-versions

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **5A: Contrast Module Source Options:** Modules are reusable configurations that can be called by a root module or other child modules. The source for the module can be the local file system or a remote repository, including the Terraform Registry that includes verified modules from third-party vendors.
- **5B: Interact with Module Inputs and Outputs:** Variables defined in a module can be used as inputs for that module. The attributes of resources within a module are only available to the calling module if exposed as outputs. Inputs and outputs can be simple or complex types.
- **5C: Describe Variable Scope Within Modules/Child Modules:** Calling modules cannot access the values stored in the variables and resources of the child modules, and child modules cannot access the variables and resources of the calling module. Sharing of information has to be explicit through input variables and output values.
- **5D: Discover Modules from the Public Module Registry:** HashiCorp hosts a public registry for Terraform modules with many official modules from major cloud providers.
- **5E: Defining Module Version:** When using a module registry, it is best to specify a particular version of a child module to avoid unexpected and unwanted updates.

In the next chapter it's time to get to the core of of using Terraform with the terminal objective **Navigate Terraform Workflow**.

Objective 6: Navigate Terraform Workflow

In this chapter we will take a look at the core workflow you'll follow in Terraform. This is loosely the process of creating a configuration (Write), previewing the anticipated changes (Plan), and committing those changes to the target environment (Create). While that is the usual lifecycle for deployments, it does leave out the key step of removing a deployment once it is no longer necessary. That could be called the Delete portion of the workflow.

First we'll take a look at the core workflow, then we'll talk about the commands to achieve the workflow, and finally how to delete your deployment if you desire to do so.

6A: Describe Terraform workflow (Write -> Plan -> Create)

As mentioned in the introduction, HashiCorp has identified a core workflow that is composed of three parts. HashiCorp defines them as follows:

- **Write** - Author infrastructure as code
- **Plan** - Preview changes before applying
- **Apply** - Provision reproducible infrastructure

Write

Writing Infrastructure as Code (IaC) is just like writing any other piece of code. Typically you will do your writing in some type of Integrated Development Environment (IDE) or advanced text editor. The purpose of those applications is to provide you with syntax checking, source control integration, and perhaps even shortcuts for creating common Terraform constructs.

The code you are writing for Terraform will either be in HashiCorp Configuration Language (HCL) or in Javascript Object Notation (JSON). It is far more common to write in HCL, since it is simpler for a human to write and read.

The code you write will often be checked into some type of Source Control Management (SCM) repository where it can be versioned and changes are tracked. This is not a hard requirement, but is a best practice. Whenever we start a new project in Terraform, we first create a source code repository to commit our code to. This serves the purpose of tracking changes, backing the code up outside of our computers, and versioning so we can roll-back when we mess something up. Using

source control becomes even more important when working on a project as a team. Different team members may be working on different parts of the same configuration, and they can use branching and pull requests to prevent the main branch from getting corrupted.

Terraform code can be validated for syntax and formatted properly before reaching the Plan stage. Doing so ensures that simple syntax and logic errors are caught before the Plan stage, and that your code is formatted consistently with other existing code. It sounds like a little thing, but consistent formatting makes a huge difference in readability.

Plan

During the plan process, you check to see what changes your code will make in the target environment. You might know what you want the end result to be, but very few of us get it right the first time. The planning process allows you to test your changes and make sure that the changes you had in your mind match up with the reality of what Terraform will do.

During your writing process, you will often run `terraform plan` to preview changes and make sure you are on the right track. By doing so iteratively, you can ensure your configuration remains valid through each small change. All of us have tried to write a big change into a configuration before, and then spent hours trying to figure out where we went wrong. You can save yourself a lot of time by keeping your changes small, and making sure that the plan is valid on a rolling basis.

Each time you run a plan you have the option to save the proposed plan to a file. We will review that option when we delve deeper into the `terraform plan` command. When you are sure that your configuration is ready to deploy, you can run `terraform apply` which will show the proposed changes once more and ask for your approval to implement the change.

Apply

The last step in the workflow is to apply your updated configuration to the target environment and make you dreams a reality. That might be overselling it a bit, but we think the apply stage is pretty cool.

With a successful apply of the new configuration, it's also a good time to commit your changes to source control to make sure they are not lost in the kerfuffle.

Then it's time to start the whole process over again! Infrastructure requirements are constantly changing, and you'll need to keep altering the configuration, planning its changes, and applying them to the target environment. It's a cycle of continuous improvement, just like any other software development lifecycle.

In the following sections we will get into the Terraform commands that are used to manage the core workflow.

Supplemental Information

- https://www.terraform.io/guides/core-workflow.html[53]

Key Takeaways: Terraform uses a workflow of Write->Plan->Apply to deploy IaC and keep it up to date with requirements.

6B: Initialize a Terraform Working Directory (terraform init)

The command `terraform init` is used to initialize the directory containing a Terraform configuration. Once you have written a configuration or cloned one from source control, this is the first command you will run. That probably bears repeating since it's the sort of thing that might be asked in the exam:

> `terraform init` is the first command you can and must run on a Terraform configuration.

The `init` command performs operations to prepare the directory holding the configuration for use. Roughly, they are as follows:

- **Prepare the state storage**: Whether you're using a local or remote backend, `init` is what prepares that backend for use by Terraform's state.
- **Retrieve modules**: The `init` process will check for any modules being used, retrieve the modules from their source location, and place them in the config directory.
- **Retrieve plugins**: The `init` process looks for all references, direct and indirect, to providers and provisioners and then retrieves the plugins from the desired location.

Initialization is safe to run multiple times against a configuration. If nothing has changed about the configuration, Terraform will not make any changes. There are several reasons to run initialization again:

- You've added a new module, provider, or provisioner.
- You've updated the required version of a module, provider, or provisioner.
- You want to change the configured backend.

[53] https://www.terraform.io/guides/core-workflow.html

The `terraform init` command has several arguments. The most commonly used ones are listed below:

- **`-backend-config`**: Used along with backend information in the configuration file to prepare the backend for state storage.
- **`-input`**: This can be set to false so `init` will not prompt the user for missing information. Useful for automation scenarios where there is no user to provide the required input.
- **`-plugin-dir`**: Sets a directory containing the required plugins for the configuration. Prevents the automatic installation of plugins, and is useful for regulated and highly secure environments.
- **`-upgrade`**: Setting this to true will force the update of any plugins and modules to the newest version allowed by the constraints.
- **`[DIR]`**: The final argument of the command is the directory that contains the configuration to be initialized. If no directory is given, Terraform will use the current working directory.

As discussed in the enabling objective **Describe How Terraform Finds and Fetches Providers**, for providers distributed by HashiCorp, `init` will automatically download and install plugins if necessary. Third party plugins can 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. Terraform will not automatically download third-party plugins.

Examples

Let's take a look at a Terraform configuration example to demonstrate how `init` performs its actions. In the following example, we are using the AWS provider and VPC module.

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

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.24.0"
}
```

There is no backend defined, so init will use a local state file. The current working directory looks like this:

```
.
└── terraform_aws_vpc_init.tf
```

By running `terraform init` in the working directory, the following output is produced.

```
Initializing modules...
Downloading terraform-aws-modules/vpc/aws 2.24.0 for vpc...
- vpc in .terraform\modules\vpc

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.46.0...
```

The updated working directory structure now looks like this:

```
.
├── .terraform
│   ├── modules
│   │   ├── modules.json
│   │   └── vpc
│   └── plugins
│       └── windows_amd64
└── terraform_aws_vpc_init.tf
```

We've limited the directory level to three deep, since vpc module directory has a lot of files and subdirectories. You can see that the subdirectory `.terraform` was created, and within that directory are the `modules` and `plugins` directories. What you don't see is a state file. That's weird right? Didn't Terraform just say that it was "Initializing the backend..." in its output? During initialization, Terraform is simply verifying that it can access the backend location where state data will be written. The actual state data won't be written until there is state to write.

> There is actually a temporary state file which is created during `terraform plan` during the first run against the configuration. It's necessary for an execution plan to be generated, but since there is no state data yet the temporary file is discarded. The permanent state data will be written once `terraform apply` is run.

The `init` process downloaded the modules and plugins we are using in our configuration and validated that it had access to the local file system. We'll leave off a deeper discussion of state and backends until Objective 7: Implement and Maintain State.

Supplemental Information

- https://www.terraform.io/docs/commands/init.html[54]

[54] https://www.terraform.io/docs/commands/init.html

Key Takeaways Before you can use your Terraform configuration, you must initialize it using `terraform init`. Initialization verifies the state backend and downloads modules, plugins, and providers.

6C: Validate a Terraform Configuration (terraform validate)

The `terraform validate` command is used to validate the syntax of Terraform files. Terraform performs a syntax check on all the Terraform files in the directory specified, and will display warnings and errors if any of the files contain invalid syntax.

This command **does not** check formatting (e.g. tabs vs spaces, newlines, comments etc.). For information on formatting, check out Objective 4B: Given a Scenario: Choose When to Use Terraform FMT to Format Code.

Validation can be run explicitly by using the `terraform validate` command or implicitly during the `plan` or `apply` commands. By default, `terraform plan` will validate the configuration before generating an execution plan. If `terraform apply` is run without a saved execution plan, Terraform will run an implicit validation as well.

The configuration must be initialized before validation can be run. Terraform is trying to validate the syntax of a configuration, including any modules, providers, and provisioners. The plugins for each of those resources are downloaded during initialization, and Terraform needs those plugins to understand and validate the syntax of a configuration. That sounds a little confusing, so here's a quick example:

```
provider "azurerm" {
  version = "~> 1.0"
}

resource "azurerm_resource_group" "main" {
  name     = "test"
  location = "eastus"
  colors   = ["blue"]
}

module "network" {
    source              = "Azure/network/azurerm"
    version             = "~> 1.1.1"
    location            = "eastus"
    allow_rdp_traffic   = "true"
    allow_ssh_traffic   = "maybe"
```

```
    resource_group_name = azurerm_resource_group.main.group_name
  }
```

We are using the `azurerm_resource_group` resource from the `azurerm` provider and the `Azure/network/azurerm` module from the public Terraform registry. Each configuration block has arguments defined, such as `name` and `allow_rdp_traffic`. Are those valid arguments for the resource or module? Are the values given for each argument of the correct data type? Terraform will look at the downloaded provider plugin and module to determine which arguments and values are valid and which are not.

If we do run `terraform validate` against this configuration, we will get the following error:

```
Error: Unsupported argument

  on terraform_validate.tf line 9, in resource "azurerm_resource_group" "main":
   9:   colors = ["blue"]

An argument named "colors" is not expected here.
```

Turns out that `colors` is not a valid argument for the `azurerm_resource_group` resource type. If we remove the offending argument and try again, we'll get the following:

```
Error: Unsupported attribute

  on terraform_validate.tf line 18, in module "network":
  18:     resource_group_name = azurerm_resource_group.main.group_name

This object has no argument, nested block, or exported attribute named
"group_name".
```

Terraform was able to validate our resource group, but then realized that `group_name` is not an argument or exported attribute of the `azurerm_resource_group` resource type. Let's say we fix the configuration to reference `azurerm_resource_group.main.name` instead.

```
provider "azurerm" {
  version = "~> 1.0"
}

resource "azurerm_resource_group" "main" {
  name     = "test"
  location = "eastus"
}

module "network" {
```

```
11      source              = "Azure/network/azurerm"
12      version             = "~> 1.1.1"
13      location            = "eastus"
14      allow_rdp_traffic   = "true"
15      allow_ssh_traffic   = "maybe"
16      resource_group_name = azurerm_resource_group.main.name #Fixed!
17    }
```

Now our validation will come back free of errors.

```
1  Success! The configuration is valid, but there were some validation warnings as show\
2  n above.
```

The warnings are from the network module, which is still using some older 0.11 style Terraform syntax. It's not invalid, but will be deprecated at some point.

It's important to note that just because a configuration is valid, that doesn't mean it will be successfully deployed. There's a mistake in our configuration of the network module.

```
1      allow_ssh_traffic   = "maybe"
```

The value of "maybe" is not valid for the argument, it should be "true" or "false". Because the type constraint of the module specifies a string as input, Terraform doesn't see any problem with the value "maybe". It is a string, and that's what the network module wants for the `allow_ssh_traffic` argument.

Supplemental Information

- https://www.terraform.io/docs/commands/validate.html[55]

Key Takeaways Terraform validate checks the syntax of your configuration, including providers and modules. It can be run explicitly or implicitly. The configuration must be initialized before running `terraform validate`.

[55] https://www.terraform.io/docs/commands/validate.html

6D: Generate and Review an Execution Plan for Terraform (terraform plan)

The `terraform plan` command is used to create an execution plan. Terraform performs a syntax validation of the current configuration, a refresh of state based on the actual environment, and finally a comparison of the state against the contents of the configuration.

Running the plan command does not alter the actual environment. It may alter the state during the refresh process, if it finds that one of the managed resources has changed since the previous refresh. Running `plan` will show you whether changes are necessary to align the actual environment with the configuration, and what changes will be made.

The execution plan generated by Terraform can be saved to a file by using the `-out` argument and giving a file name as a destination. The execution plan is aligned with the current version of the state, and if the state changes Terraform will no longer accept the execution plan as valid. The saved execution plan can be used by `terraform apply` to execute the planned changes against the actual environment.

You might use `plan` for several reasons:

- As a check before merging code in source control
- As a check to validate the current configuration
- As a preparation step to execute changes to the actual environment

The `terraform plan` command has several arguments. The most commonly used ones are listed below:

- **`-input`**: determines whether or not to prompt for input, set to false in automation
- **`-out`**: specify a destination file where the execution plan will be saved
- **`-refresh`**: whether or not a refresh of state should be run, defaults to true
- **`-var`**: set a value for a variable in the configuration, can be used multiple times
- **`-var-file`**: specify a file that contains key/value pairs for variable values

The `var` and `var-file` arguments are especially common, as that is one of the ways to submit values for variables. There are other ways to submit values as well, which is covered in detail in Objective 8: Read, Generate, and Modify Configuration.

The `plan` command also takes a directory containing a Terraform configuration as optional input. If no directory is specified, Terraform will use the configuration found in the current working directory.

There is a special argument, `destroy`, that will generate a plan to destroy all resources managed by the current configuration and state. We'll take a look at why you would use this in the section dealing with `terraform destroy`.

Supplemental Information

- https://www.terraform.io/docs/commands/plan.html[56]

Key Takeaways Terraform plan compares the current configuration against the state data and generates an execution plan of actions to align the state with the configuration. The execution plan can be saved to a file and used by `terraform apply`.

6E: Execute Changes to Infrastructure with Terraform (terraform apply)

The `terraform apply` command executes changes to the actual environment to align it with the desired state expressed by the configuration or from the execution plan generated by `terraform plan`.

By default, `apply` will look for a Terraform configuration in the current working directory and create an execution plan based on the desired state and the actual environment. The execution plan is presented to the user for approval. Once the user approves the execution plan, the changes are applied to the actual environment and the state data is updated.

If a saved execution plan is submitted to `terraform apply`, it will skip the execution plan generation and approval steps, and move directly to making the changes to the actual environment and state data.

Strictly speaking, you do not have to run `terraform plan` before `terraform apply`. When running Terraform manually outside of a team setting, skipping the plan stage might be justified. If you start to automate Terraform runs, or collaborate with a team, then it is advised to always run `terraform plan` first and save the execution plan to a file.

The `terraform apply` command has several arguments. The most commonly used ones are listed below:

- **`-auto-approve`**: skips the approval step and automatically moves to making changes
- **`-input`**: determines whether or not to prompt for input, set to false in automation
- **`-refresh`**: whether or not a refresh of state should be run, defaults to true
- **`-var`**: set a value for a variable in the configuration, can be used multiple times
- **`-var-file`**: specify a file that contains key/value pairs for variable values

You might notice that several of the arguments are the same for `plan` and `apply`. That is because `apply` can be run without first running plan, although we don't recommend doing so. If you've saved your execution plan in a file like `terraform.tfplan`, then you can submit the plan by running:

[56] https://www.terraform.io/docs/commands/plan.html

```
$> terraform apply terraform.tfplan
```

Doing so will remove the need to specify variables, and the apply will be automatically approved.

Having an execution plan is no guarantee that the changes will be applied successfully. Terraform will do its best to apply all the changes, but things do occasionally go awry. It might be a problem with the cloud provider or an unanticipated problem with your configuration. When a error is encountered, Terraform does not roll back to the previous configuration. It will leave the resources as they are so you can attempt to troubleshoot the problem. Resources that were not successfully configured will be marked as tainted, and Terraform will try to recreate them on the next run.

Supplemental Information

- https://www.terraform.io/docs/commands/apply.html[57]

Key Takeaways Terraform apply makes the necessary changes to an actual environment to align it with desired state expressed by a configuration. The `apply` command can generate an execution plan on its own and prompt for approval or take a saved execution plan and apply it.

6F: Destroy Terraform Managed Infrastructure (terraform destroy)

Destroying what you deployed may not seem like an obvious step in the lifecycle of infrastructure, but there are many cases where this might be useful. You might be deploying development environments for a project and you want to clean up the resources when the project concludes. You might be building an environment for testing in a CI/CD pipeline, and want to tear down the environment when the tests are complete. The `terraform destroy` command is used to destroy Terraform-managed infrastructure.

The `destroy` command is obviously a powerful command. It will delete all resources under management by Terraform, and this action **cannot** be undone. For that reason, Terraform will present you with an execution plan for all the resources that will be destroyed and prompt to confirm before performing the destroy operation.

In many respects, the `destroy` command is a mirror of the `apply` command, except that you cannot hand it an execution plan. This command accepts all the arguments and flags that the apply command accepts, with the exception of a plan file argument. You can preview what the command

[57] https://www.terraform.io/docs/commands/apply.html

will do by running `terraform plan -destroy` along with any other necessary arguments. You cannot save the execution plan to a file and submit it to the `destroy` command.

There are two arguments that are useful to know:

- **`-auto-approve`**: will not prompt the user for confirmation, used to be called `-force` which has been deprecated
- **`-target`**: you can specify a target and only the target and its dependencies will be destroyed. The flag can be used multiple times.

Supplemental Information

- https://www.terraform.io/docs/commands/destroy.html[58]

Key Takeaways Terraform destroy permanently deletes all resources managed by the current configuration and state. The command will prompt for confirmation, and select resources can be targeted with the `-target` argument.

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **6A: Describe Terraform workflow**: Terraform uses a workflow of Write->Plan->Apply to deploy IaC and keep it up to date with requirements.
- **6B: Initialize a Terraform Working Directory**: Before you can use your Terraform configuration, you must initialize it using `terraform init`. Initialization verifies the state backend and downloads modules, plugins, and providers.
- **6C: Validate a Terraform Configuration**: Terraform validate checks the syntax of your configuration, including providers and modules. It can be run explicitly or implicitly. The configuration must be initialized before running `terraform validate`.
- **6D: Generate and Review an Execution Plan for Terraform**: Terraform plan compares the current configuration against the state data and generates an execution plan to align the state with the configuration. The execution plan can be saved to a file and used by `terraform apply`.

[58] https://www.terraform.io/docs/commands/destroy.html

- **6E: Execute Changes to Infrastructure with Terraform**: Terraform apply makes the necessary changes to an actual environment to align it with desired state expressed by a configuration. The `apply` command can generate an execution plan on its own and prompt for approval or take a saved execution plan and apply it.
- **6F: Destroy Terraform Managed Infrastructure:** Terraform destroy permanently deletes all resources managed by the current configuration and state. The command will prompt for confirmation, and select resources can be targeted with the `-target` argument.

In the next chapter we are going to talk about Terraform state data and how it's managed with the terminal objective **Implement and Maintain State**.

Objective 7: Implement and Maintain State

In this chapter we are going to be talking about Terraform state. 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 (aka the real world) and what is expressed in the declarative code. Terraform tracks this mapping through a JSON-formatted data structure called the *state*.

We are going to look at where state can be stored (backends), how it is configured and accessed, and some considerations when it comes to collaboration and secrets management. If you're looking for an examination of the `terraform state` subcommands, those can be found in Objective 4: Use the Terraform CLI.

7A: Describe Default Local Backend

Terraform needs somewhere to store its state data, this is called a backend, and in the absence of an alternate configuration, it will store state on the local file system where the configuration is stored. Let's examine the typical folder structure for an example configuration:

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

This Terraform configuration has been initialized and run through a plan and apply. As a result, the `.terraform` directory holds the plug-ins used by the configuration, and the `terraform.tfstate` file holds the state data about the configuration.

The location of the local state file can be altered using the command line flag `-state=statefile` for commands like `plan` and `apply`.

The directory structure will be slightly different if you are using Terraform workspaces. One of the key features of workspaces is that every workspace maintains a separate state, allowing multiple environments to use the same Terraform configuration. Let's say we created a workspace called `development` in our configuration.

```
├── .terraform
├── main.tf
├── terraform.tfstate
└── terraform.tfstate.d
    └── development
        └── terraform.tfstate
```

Terraform creates a directory `terraform.state.d` and a subdirectory for each workspace. In each workspace directory is the `terraform.tfstate` file for that workspace. The default workspace continues to use the `terraform.tfstate` file in the root directory.

Supplemental Information

- https://www.terraform.io/docs/backends/types/local.html[59]

Key Takeaways Terraform will use the root module directory to store state data if no alternative is specified.

7B: Outline State Locking

State data is pretty important, and transactions that involve state should be atomic in nature. To prevent multiple processes from editing the state at the same time, Terraform will lock the state.

Any operation that could edit the state will first check to make sure the state is not locked. If the state is not locked, Terraform will attempt to create a lock on the state before making changes. Terraform will not continue if it cannot attain a lock. You can override the locking behavior for several commands by using the `-lock=false` flag. This would be an exceptional event and should be approached with caution. Corrupted state is disastrous at best.

There is also a `-lock-timeout` flag for most commands that determines how long Terraform should wait to attain a lock on state. This is useful for a state backend that takes longer to create a lock for Terraform.

The majority of state backends support locking, but there are a few that do not. You should check the documentation for any given backend to determine what features it supports.

Supplemental Information

- https://www.terraform.io/docs/state/locking.html[60]

[59] https://www.terraform.io/docs/backends/types/local.html
[60] https://www.terraform.io/docs/state/locking.html

Key Takeaways Terraform can lock the state file on supported backends to prevent simultaneous write operations to the state data.

7C: Handle Backend Authentication Methods

Some backends require authentication and authorization to access state data stored in the backend. For instance, Azure Storage requires access keys and the MySQL backend requires database credentials. The documentation for each backend type will list out the exact format and type of authentication required.

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

```
terraform {
  backend "azurerm" {
    storage_account_name = "arthurdent42"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
    access_key           = "qwertyuiop12345678..."
  }
}
```

Azure storage accounts have an access key that allows full access to the contents of a storage account. In the code snippet above, the access key value is stored directly in the configuration. Generally speaking, this is **not** a recommended practice.

Statically defining credentials in a configuration has a couple issues. You may want to update the credentials on a regular basis, which means updating the configuration each time. It's also not a good idea to store important credentials in clear text on your local workstation or in source control. That's a major potential security hole.

> Hypothetically, someone might define AWS keys in a Terraform configuration and then accidentally check that configuration into a public GitHub repository, exposing their AWS keys to the entire world. That's a hypothetical scenario and definitely not something that each of us has done at least once.

You cannot define the credentials for a backend using Terraform variables. The backend is evaluated during initialization, before variables are loaded or evaluated. Because the variables have not been evaluated, Terraform has no idea they exist, let alone what value is stored in them. For that reason, Terraform cannot use the values defined in variables for your backend configuration.

The solution is to use a partial backend configuration in the root module and provide the rest of the information at runtime. We'll discuss partial configurations in more detail further on in the chapter when we delve into the `backend` configuration block.

Supplemental Information

- https://www.terraform.io/docs/backends/types/index.html[61]

Key Takeaways Remote backends usually require authentication. You'll need to supply that information in the backend config or at runtime.

7D: Describe Remote State Storage Mechanisms and Supported Standard Backends

As we mentioned earlier in the chapter, Terraform uses a local file backend by default. While that might be good enough when working on a small project by yourself, it doesn't provide data protection or allow for collaboration. The solution is to use a remote state backend, which stores the state data in a remote, shared location.

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 backends.

- 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, covered in Objective 8: Read, Generate, and Modify Configuration. Both of those services allow you to run your Terraform operations - like `plan` and `apply` - on the remote service instead of locally.

There are fourteen standard backends at the time of writing, and going into each would be beyond the scope of this guide. We recommend checking out the documentation for each one to learn details about what features are supported and what values are required.

When state data is stored in a remote backend, it is not written to disk on the local system. The state data is loaded into memory during Terraform operations and then flushed when no longer in use. Sensitive data within the state file is never stored on the local disk, providing an additional level of security if your local system was lost or compromised.

[61] https://www.terraform.io/docs/backends/types/index.html

In addition to storing the state in a secure location, using remote state also enables other team members access to alter the state or use it as a data source. Let's say that you are collaborating with Joan the network admin on a network configuration. Both of you can be update your local copy of the configuration and then push it to version control. When it is time to update the actual environment, you run the plan and apply phases. The remote state is locked during the update, so Joan can't make any changes. When Joan runs her next plan, it will be using the updated remote state data from your most recent apply.

Any `outputs` defined in a configuration can be accessed using the state file as a data source. For instance, let's say that an application team needs to query information about a network configuration. You could give them read-only access to the network state, and make sure the information they need is exposed through outputs. The application team would then define the network state as a data source in their configuration and reference the outputs directly.

Supplemental Information

- https://www.terraform.io/docs/state/remote.html[62]
- https://www.terraform.io/docs/backends[63]

Key Takeaways Terraform can store state data in a remote location that supports collaboration and state as a data source.

7E: Describe Effect of Terraform Refresh on State

Data stored in state needs to reflect the actual state of infrastructure being managed by Terraform. A refresh action examines the attributes of the resources in the target environment and compares them to the values stored in state. Terraform will not alter the target infrastructure on a refresh action, but it will update the state to accurately reflect the target environment.

The refresh action can be triggered manually, by running `terraform refresh` and specifying any options. The refresh action is also automatically run as part of the plan stage, whether from an explicit `terraform plan` or implicitly when running `terraform apply` without an execution plan file. Refreshing the state is critical to Terraform determining what changes are necessary to make the desired-state express in the configuration match the managed infrastructure.

[62] https://www.terraform.io/docs/state/remote.html
[63] https://www.terraform.io/docs/backends

> Terraform will **not** discover and import new resources created outside of the configuration. For instance, if you manually add new resources to an Azure Resource group or EC2 instances to a VPC, Terraform is not going to find those resources and bring them under management. You would need to update the configuration and use the `import` command to make that happen.

Supplemental Information

- https://www.terraform.io/docs/commands/refresh.html[64]

Key Takeaways Terraform can refresh the contents of the state file from the actual state of the managed infrastructure.

7F: Describe Backend Block in Configuration and Best Practices for Partial Configurations

The configuration of Terraform backends is defined inside the `terraform` block of a root module. The nested `backend` block defines the type of backend being used and the required and optional properties of that backend. Only one backend can be specified per configuration. You cannot have your state stored in two different backends simultaneously, or split your state across two different backends. That type of situation can be solved by splitting the configuration itself into two interdependent configurations.

As we saw earlier in the chapter, a basic backend configuration block using Azure storage might look like this:

[64] https://www.terraform.io/docs/commands/refresh.html

```
terraform {
  backend "azurerm" {
    storage_account_name = "arthurdent42"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
    access_key           = "qwertyuiop12345678..."
  }
}
```

Storing the storage account name and the access key directly in the configuration is probably not a good idea. But Terraform does not allow the use of variables or any other interpolation in a backend configuration block. The alternative is omit the settings you'd like to dynamically configure, and instead supply them at runtime. The resulting block is called a partial configuration.

Our example above would look like this:

```
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:

1. Interactively at the command line when you run `terraform init`
2. Through the `-backend-config` flag with a set of key/value pairs
3. Through the `-backend-config` flag with a path to a file with key/value pairs

Some arguments in a backend can also be sourced from environment variables. For instance, using Managed Security Identity (MSI) for Azure storage can be configured by setting `ARM_USE_MSI` to `true`. Check the backend documentation to see which settings are supported by environment variables.

After the first time you run `terraform init` with your backend values, you do not need to supply the values again for other commands. The backend information and other initialization data is stored in a local file in the `.terraform` subdirectory of your configuration. This directory should be exempted from source control since your backend configuration may have sensitive data in it, like authentication credentials.

> When you create a new repository on GitHub and create a `.gitignore` file, you can select Terraform as one of the options. It will automatically exclude the `.terraform` directory, as well as files like `terraform.tfvars` and `terraform.tfstate`. We recommend using `.gitignore` on any new project

with those exemptions.

Supplemental Information

- https://www.terraform.io/docs/backends/config.html[65]

Key Takeaways Terraform backends are defined using a `backend` configuration block. Partial configurations should be used to omit sensitive and dynamic values, which will be submitted at runtime.

7G: Understand Secret Management in State Files

State data holds all of the information about your configuration, including any sensitive data like passwords, API keys, and access credentials. The state data is not encrypted by Terraform, but it can sit on an encrypted storage platform. By using a backend that provides data encryption, you can add a level of protection for any sensitive data living in state.

When Terraform is using a remote state backend, the contents of the state are copied to memory only. Unless you explicitly ask Terraform to do so, the state data will not be persisted to disk.

The best practice when dealing with sensitive data is to not store it in state at all. If you have a secrets management platform, like HashiCorp Vault, then have your applications retrieve sensitive data from there. If you must have sensitive data in your configuration, then it should be stored on a remote backend that provides encryption of data at rest and in transit.

Supplemental Information

- https://www.terraform.io/docs/state/sensitive-data.html[66]

Key Takeaways Use a remote state backend to keep sensitive data off your machine.

[65] https://www.terraform.io/docs/backends/config.html
[66] https://www.terraform.io/docs/state/sensitive-data.html

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- 7A: **Describe Default Local Backend**: Terraform will use the root module directory to store state data if no alternative is specified.
- 7B: **Outline State Locking**: Terraform can lock the state file on supported backends to prevent simultaneous write operations to the state data.
- 7C: **Handle Backend Authentication Methods**: Remote backends usually require authentication. You'll need to supply that information in the backend config or at runtime.
- 7D: **Describe Remote State Storage Mechanisms and Supported Standard Backends**: Terraform can store state data in a remote location that supports collaboration and state as a data source.
- 7E: **Describe Effect of Terraform Refresh on State**: Terraform can refresh the contents of the state file from the actual state of the managed infrastructure.
- 7F: **Describe Backend Block in Configuration and Best Practices for Partial Configurations**: Terraform backends are defined using a `backend` configuration block. Partial configurations should be used to omit sensitive and dynamic values, which will be submitted at runtime.
- 7G: **Understand Secret Management in State Files**: Use a remote state backend to keep sensitive data off your machine.

In the next chapter we will dig into the terminal objective **Understand Terraform's Purpose (vs Other IaC)**, showing how Terraform applies the concepts of IaC.

Objective 8: Read, Generate, and Modify Configuration

In this chapter, it's time to get down into the nitty-gritty of Terraform configuration. A typical configuration in Terraform is going to be made up of variables, local values, resources, data sources, modules, and outputs. We've already covered the use of modules in Objective 5: Interact with Terraform Modules. The rest of the object types in Terraform have only been mentioned in passing. Now we're going to see how we inject information into a configuration and retrieve output. Then we'll look at the various data structures supported by Terraform, and how you can manipulate those data structures using a suite of built-in functions and constructs to create loops and iterations. Finally, we'll take a look at the dependency engine in Terraform and how it examines a configuration to determine an order of operations.

8A: Demonstrate Use of Variables and Outputs

Input Variables

Variables are defined in a Terraform configuration. You can supply values for those variables at runtime, or set a default value for the variable to use. If you do not set a default value for a variable, Terraform will require that you supply one at runtime. Failing to supply a value will cause the relevant command to error out.

At it's most basic, a variable can be defined with a name label and without any arguments:

```
variable "aws_region" {}
```

That's pretty boring. Let's define some more information about our `aws_region`:

```
variable "aws_region" {
    type        = string
    default     = "us-east-1"
    description = "The AWS region to use for this configuration"
}
```

That's much better. We defined what data type to expect (`string`), what value to default to, and gave a description for ourselves and others who might use this configuration. We'll cover data types a little later in the chapter. Providing a data type for the variable means that Terraform can do some

validation to make sure the correct data type is being provided. It's not full blown data validation, but it's better than nothing!

How do values get submitted for these variables? Good question! The values are submitted at runtime through several possible means. Here's a handy list of your options:

1. Setting environment variables with the prefix `TF_VAR_` followed by the variable name
2. Placing a `terraform.tfvars` or `terraform.tfvars.json` file in the same directory as the configuration
3. Placing a file ending with `.auto.tfvars` or `.auto.tfvars.json` in the same directory as the configuration
4. Using the `-var-file` flag with the path to a file containing the key/value pairs
5. Using the `-var` flag with a key/value pair (`aws_region=us-west-1`)
6. Typing them in when prompted at the command line

That's a LOT of options. And you can combine all of these options together, just to make things extra confusing. Terraform does have an order of precedence for evaluating the submitted values, which just happens to match the list above. The later options take precedence over earlier options, so if you defined a variable value in an environment variable and submitted a different value for the same variable using the `-var` flag, the value submitted with the `-var` flag would win. At the command line level, the last value submitted for a value wins.

With values applied to our variables, we can now use those variables by referencing them in the rest of the configuration. We'll dig into that process more when we get to resource addressing.

Outputs Values

Terraform can make outputs available after a configuration has been successfully deployed. The outputs created by a root module (that's your main configuration) are printed as part of the console output and they can be queried using the `terraform output` command. That may not seem especially useful, but there are two ways in which output becomes incredibly useful, in child modules and terraform state data sources.

When you invoke a child module in Terraform, you are provided access to some of the attributes or resources created by that module. Outputs in the child module determine what information is available to the calling module. This means that the child module can create variables and resources however it sees fit, as long as it produces a consistent set of outputs for use by the calling module.

Prior to Terraform 0.12, outputs were limited to strings. You could not pass a complex data type back as an output. This often meant using a built-in function to transform that string output to a data type like a list or map. Complex data types are now supported, meaning you can pass an entire resource, along with all of its attributes, back as output.

Terraform state can be consumed as a data source by another configuration. The attributes available from that data source are limited to what is exposed through outputs. Let's imagine

your network team created networking environment for your application team using Terraform. The application team can use the networking state file as a data source in their configuration and query for information about the target network environment. The network team can determine what information is available by creating proper outputs in their configuration.

Creating an output in a configuration follows a simple format. The output needs a name and value.

```
output "vpc_id" {
    value       = module.vpc.vpc_id
    description = "The vpc id for your application"
}
```

The output block can have two additional arguments: `sensitive` and `depends_on`. If an output has `sensitive` set to `true`, Terraform will not print the value of the output at the CLI. The `depends_on` argument takes a list of resources the output is dependent on before being rendered. Usually Terraform can determine dependence on its own, but there are exceptional cases where the chain of dependence is not obvious.

Outputs are rendered when `terraform apply` is run. If you change your existing outputs, or add new ones, you'll need to run `apply` again even if nothing else has changed in the configuration.

Supplemental Information

- https://www.terraform.io/docs/configuration/variables.html[67]
- https://www.terraform.io/docs/configuration/outputs.html[68]

Key Takeaways Variables are how values are submitted to a Terraform configuration. There are many way to submit a value, and there is an order of evaluation, with last value winning. Outputs are primarily used to expose information from a child module to a calling module or expose information through remote state. Outputs are rendered during apply and can be of a complex data type.

8B: Describe Secure Secret Injection Best Practice

The information you submit to Terraform may be of a sensitive nature. You may include things like application passwords, API tokens, or usernames and passwords to authenticate with providers and configure the resources being provisioned by Terraform. This type of sensitive data should be treated with care. Here are some recommendations around dealing with secret data.

[67] https://www.terraform.io/docs/configuration/variables.html
[68] https://www.terraform.io/docs/configuration/outputs.html

Don't store sensitive data in a `tfvars` file. There are no protections provided by Terraform to encrypt or secure variables files, and there is a good chance that the variables files are being checked into your source control. Don't let your AWS access and secret keys get exposed to the world because you checked your variables file into source control.

You can submit your variable values at the command line using the `-var` flag, but even then your sensitive data will be stored in your command history unobscured.

The easiest alternative is to load your sensitive data into environment variables, using the naming convention `TF_VAR_` and the variable name. You can configure these environment variables to pull your sensitive data without ever exposing it to the command line, and Terraform will not show the sensitive data in its output.

It's important to note that sensitive data submitted as part of a resource configuration is stored in the Terraform state. Ideally, your state data should be stored in a secure backend that supports encryption at rest. If you are using the local file backend for state storage, make sure that it is exempted from source control.

Supplemental Information

- https://www.terraform.io/docs/providers/vault/index.html[69]

Key Takeaways Secret data doesn't belong in a `.tfvars` file or submitted at the command line. Keep secret data secret, and use environment variables.

8C: Understand the Use of Collection and Structural Types

Terraform introduced well-defined data structures in 0.12. A full discussion of data structures would probably take up a whole chapter - or book - by itself, but we'll try to simplify it down to the essentials.

There are three primitive data types in Terraform:

- **string**: A sequence of Unicode characters.
- **number**: A number value including both integers and decimals.
- **boolean**: A *true* or *false* value.

When you are specifying the type for a variable, you can use any of these primitive data types.

[69] https://www.terraform.io/docs/providers/vault/index.html

```
variable "my_bool" {
    type    = boolean
    default = true
}
```

Primitive types have a single value. Complex types have multiple values and can be broken into two categories: *collection* and *structural*. This is where things can get a bit confusing, so stay with us.

Collection Types

Collection types are the multi-value types you'll probably encounter most, especially at the Associate level. There are three collection types:

- **list**: A sequential list of values identified by their position in the list starting with `0`.
- **map**: A collection of values each identified by a unique key of type string.
- **set**: A collection of unique values with no identifiers or ordering.

The *list* and *map* collection types are much more prevalent than the *set* type. All the values in a collection **must** be of the same primitive type. You can define that primitive type within a set of parenthesis.

```
variable "my_list" {
    type = list(string) # A list of strings
}

variable "my_map" {
    type = map(number) # A map with number values
}
```

There is a special primitive type called *any* that allows any type to be used in a collection. Terraform tries to figure out what the intended type is based on what value is submitted. Generally, it's better to be proscriptive when declaring input variables to make sure you get the desired structure.

Here are a few examples of lists and maps.

```
# Lists!
[1,2,3,4] # A list of four numbers
["a",2,false] # Not a valid list

var.list_var[1] # Returns the second element in the list

# Maps!
{ # A map with two keys and number values
    "one" = 1
    "two" = 2
}

{ # A map with two keys and list values
    "one" = [3,1,4]
    "two" = ["pi","is","delicious"]
}

var.map_var["one"] # Returns the value with the key "one"
```

Structural types

The structural types allow you to create a more complex object, where the values can be of multiple types. This is distinctly different from collection types, where each value in the collection must be of the same type. There are two structural types:

- **object**: Similar in nature to a map, it is a set of key/value pairs, but each value can be of a different type.
- **tuple**: Similar in nature to a list, it is an indexed set of values, where each value can be of a different type.

Maps can be converted into objects with little issue, but the reverse it not always true. The same goes for tuples and lists. When you're deciding which one to use, ask yourself if you need multiple types defined within the variable. If you do, then go with structural types. Here are some examples of objects and tuples.

```
variable "network_info" {
    type = object( # An object with multiple types
        {
            network_name = string
            cidr_ranges  = list(string)
            subnet_count = number
            subnet_mask  = number
            nat_enabled  = boolean
        }
      )
}

# Example of the object
{
    network_name = "my_net"
    cidr_ranges  = ["10.0.0.0/24","10.0.1.0/24"]
    subnet_count = 2
    subnet_mask  = 25
    nat_enabled  = false
}

variable "pi_tuple" {
    type = tuple( # A tuple with four types
              [ string, number, boolean, list(number) ]
            )
}

#Example of the tuple

[ "Pi", 3.14, true, [3,1,4,1,5]]
```

The differences between primitive, collection, and structural type can be a bit confusing at first. We recommend reviewing this a few times, and probably trying to create your own variables to test out the various types.

For the most part, Terraform treats maps/objects and list/tuples as the same data structure. While the actual difference can be important when you are writing modules and providers for Terraform, in general you can also treat them as equivalent.

Supplemental Information

- https://www.terraform.io/docs/configuration/types.html#complex-types[70]

Key Takeaways String, number, and boolean are the primitive types. Map, list, and set are the collection types. Object and tuple are the structural types. Collection types must have values of the same type, structural types can have values of different types.

8D: Create and Differentiate Resource and Data Configuration

Most providers in Terraform are composed of resources and data sources. Data sources are existing resources or information available from the provider for use within the configuration.

Data Sources

Data sources are simply sources of information that you can retrieve from a provider. For example, maybe you want to retrieve an AMI ID for a specific version of Ubuntu in the AWS us-east-1 region. Or you need to get the vnet ID where you will be deploying some Azure VMs. Those are two examples of a data source.

Most data sources require that you provide some configuration data to get the relevant information you want. They also assume that you have correctly configured the provider for your data source, as we reviewed in Objective 3: Understand Terraform Basics. Let's take a look at the AWS AMI example:

```
data "aws_ami" "ubuntu" {
    most_recent = true

    filter {
      name   = "name"
      values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
    }

    filter {
      name   = "virtualization-type"
```

[70] https://www.terraform.io/docs/configuration/types.html#complex-types

```
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```

The object type label for a data source is `data`, followed by the data source type label (`aws_ami`) and a name label for the data source (`ubuntu`). We are supplying arguments to get the Ubuntu image for 16.04. Each data source has arguments required to retrieve the data, and then a set of attributes that are available once the data is retrieved. The `aws_ami` data source has an attribute called `image_id`, which can be used to create an EC2 instance using that AMI ID. We'll go over data source and resource addressing in the next section.

Resources

Resources are the central reason Terraform exists. You want to provision infrastructure, and that infrastructure is represented by resources available from a providers. Resources take arguments to define the configuration of the physical resource. Arguments can be required or optional, and can include simple key/value pairs and nested blocks.

The resource also has attributes available as a reference for other portions of your configuration. Attributes are exported based on the creation of the physical resource. Many of the attributes are unknown until the creation of a resource is complete; for example, the ID of an EC2 instance.

Let's take a look at a sample resource:

```
resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location            = var.location
  resource_group_name = azurerm_resource_group.net_group.name
  address_space       = ["10.0.0.0/16"]

  subnet {
    name           = "subnet1"
    address_prefix = "10.0.1.0/24"
  }

}
```

The object type label in this case is `resource`, followed by a resource type (`azurerm_virtual_network`), and a name label (`vnet`). Within the resource configuration we are providing values to arguments, such as the name and location of the virtual network. Notice that some arguments require a list type value, such as the `address_space` argument, while others take a string value. We are also

using a nested block to define the properties of a subnet within the virtual network. Since a single virtual network can have multiple subnets, we could use a dynamic block to define all of the subnets. We'll cover dynamic blocks in more detail later.

Another thing to note is that we can use variables and attributes of other resources as values for the arguments. The value type exposed by the attribute should match the type expected by the argument. Terraform will do some implicit conversion of similar types, e.g. converting a number to a string. More complex conversions will need to be handled through syntax or built-in functions.

Supplemental Information

- https://www.terraform.io/docs/configuration/resources.html[71]
- https://www.terraform.io/docs/configuration/data-sources.html[72]

Key Takeaways Data sources provide access to information stored in a provider. Resources allow you to create a physical resource on a provider. Both data sources and resources take arguments for their configuration and expose attributes for use in the rest of your Terraform configuration.

8E: Use Resource Addressing and Resource Parameters to Connect Resources Together

The resources you create in your Terraform configuration will often be related. For instance, creating a web application running on virtual machines might involve a virtual network, virtual machines, network interfaces, firewall rules, and a load balancer. All of these resources will refer to each other for their configuration. The load balancer will need to reference the network interfaces on the virtual machines, and the network interfaces will need to reference a subnet on the virtual network. Resource addressing is the way that you can reference the attributes of a resource within your Terraform configuration.

We have already seen several examples of references to both named values and indices. The general format is different depending on the object type in question. When referring to the value inside a variable the format is `var.<NAME>`. Resources use the format `resource_type.<NAME>`, followed by the attribute whose value you would like to retrieve. If you have used a `for_each` or `count` argument to create multiple instances of a resource, then the value returned will be a `map` or `list` respectively.

When referencing a resource, you have access to any arguments you define in the configuration and any attributes that are exported by the resource. A simple example would be getting the name of an Azure resource group to be used by another resource.

[71] https://www.terraform.io/docs/configuration/resources.html
[72] https://www.terraform.io/docs/configuration/data-sources.html

```
resource "azurerm_resource_group" "net_group" {
    name     = "net-group"
    location = "eastus"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location            = var.location
  resource_group_name = azurerm_resource_group.net_group.name
  address_space       = ["10.0.0.0/16"]

  subnet {
    name           = "subnet1"
    address_prefix = "10.0.1.0/24"
  }

  subnet {
    name           = "subnet2"
    address_prefix = "10.0.2.0/24"
  }

}
```

The `resource_group_name` argument for the virtual network is referencing the `name` attribute of the `net_group` resource group we are creating. The `azurerm_resource_group` resource also exports the attribute `id`, which could be accessed through the expression `azurerm_resource_group.net_group.id`.

Some of the arguments and attributes can be complex value types, which means you may need to use special syntax to extract the value you want. In the arguments of the virtual network above, we are defining two subnets. The virtual network resource exports the `subnet` blocks with an `id` attribute for each subnet defined. The follow expressions will retrieve different aspects of the subnets:

- `azurerm_virtual_network.vnet.subnet`: retrieves a set of subnets, each with a map of properties defining the subnet
- `azurerm_virtual_network.vnet.subnet[*].id`: retrieves a list of the id attribute for each subnet by using the splat operator `*`

Since the `subnet` attribute returns a set and not a list or map, you cannot refer to a specific subnet by index or key. You could use a `for` expression to iterate over the set and select the `name` key from each subnet `[for subnet in azurerm_virtual_network.vnet.subnet : subnet["name"] ]`. That expression would return a list of the subnet names. That is probably more advanced than what you

might see in the exam, but the important point is to understand what data type is returned by an attribute or argument.

Supplemental Information

- https://www.terraform.io/docs/configuration/expressions.html[73]

Key Takeaways Resources and data sources include arguments and attributes that can be referenced within a configuration. Be sure to pay attention to the data type exported by each attribute.

8F: Use Terraform Built-In Functions to Write Configuration

The format of data returned by data sources and resources may not be in the format you need to pass to another resource. You may also want to transform outputs from a module, combine variables, or read in file data. Terraform offers built-in functions to perform many of these tasks. There are nine different function categories:

- Numeric
- String
- Collection
- Encoding
- Filesystem
- Date and Time
- Hash and Crypto
- IP Network
- Type Conversion

We aren't going to go into all the different functions in each category. If you really need to know, we suggest reading the official Terraform docs. That being said, the important thing here is to understand how to invoke a function in your Terraform configuration, and how to test a function using the Terraform console.

Functions have the basic format of `function(arguments,...)`. If you've done any programming, this should feel pretty familiar. The data type of the argument is governed by the function. Let's look at a few examples:

[73] https://www.terraform.io/docs/configuration/expressions.html

- **`max(number,number,...)`** - max returns a number that is the largest value of the given set
- **`lower(string)`** - lower returns a string in all lowercase based on the submitted string
- **`keys(map)`** - keys returns a list of the keys in the given map

Each function takes arguments of specific types and returns a value of a specific type. You can also combine functions together, for instance `abs(min(1,-10, 5))` would first find the smallest value and then return the absolute value.

You can test the output of functions by using the Terraform console command. You can launch the console from the Terraform command-line by simply running `terraform console`. If you are in a directory that has a Terraform configuration, it will load the current state so you have access to the state data to test functions and other interpolations. The console will never alter the current state. If you launch it in a directory without a Terraform configuration, then you can simply run functions within the console, but it is not nearly as useful.

Let's take a look at an example:

```
provider "azurerm" {
  version = "~>1.0"
}

variable "location" {
  type    = string
  default = "eastus"
}

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

resource "azurerm_resource_group" "net_group" {
    name     = "net-group"
    location = var.location
}

resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.net_group.name
  address_space       = var.address_space

  subnet {
    name           = "subnet1"
```

```
    address_prefix = "10.0.1.0/24"
  }

}
```

We've defined the address space for the virtual network using a variable. Sweet! But we haven't defined the subnet using a variable. That's no good. We think we can use the `cidrsubnet` function to compute a subnet range, but we want to test it first. Let's use Terraform console to find out.

```
$> 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, we have access to the variables we've defined in the configuration. Then we can use that variable with the `cidrsubnet` function to test how our subnet address is generated. Once we're happy with the result, we can update our existing configuration to use the function.

```
  subnet {
    name           = "subnet1"
    address_prefix = cidrsubnet(var.address_space[0],8,1)
  }
```

Supplemental Information

- https://www.terraform.io/docs/configuration/functions.html[74]

Key Takeaways Built-in functions enable you to manipulate information to match what you need. Functions take typed arguments and return typed data. Terraform console can help you test and validate functions to use in your configuration.

8G: Configure Resource Using a Dynamic Block

In the previous section we saw an example of a virtual network with a single subnet defined as a nested block.

[74] https://www.terraform.io/docs/configuration/functions.html

```
  subnet {
    name           = "subnet1"
    address_prefix = cidrsubnet(var.address_space[0],8,1)
  }
```

There's a good chance you are going to have more than one subnet in your virtual network. You could define a separate nested block for each subnet, but that is not a scalable or dynamic approach. It makes more sense to dynamically generate the nested blocks based off the values in variables. The mechanism for creating the nested blocks is called dynamic blocks.

Dynamic blocks can be used inside resources, data sources, providers, and provisioners. It works in a similar way to the `for` expression. The `for` expression produces a collection of either list or map type. A dynamic block expression produces one or more nested blocks. The syntax can be a little confusing, so let's first look at an example based on our subnet configuration.

```
resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.net_group.name
  address_space       = var.address_space

  dynamic "subnet" {
      for_each = var.subnet_data
      content {
          name           = subnet.value["name"]
          address_prefix = subnet.value["address_prefix"]
      }
  }

}
```

The keyword `dynamic` establishes that this is a dynamic block expression. The name label of `subnet` lets Terraform know that this will be a set of nested blocks each of type `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 a variable called `subnet_data`.

The `content` section defines the actual content of each generated nested block. Within the content section we need to be able to refer to the information in the `subnet_data` variable. Terraform creates a temporary iterator with the same name as the name label of our dynamic block, `subnet` in this case. If you want to specify a different iterator name, there is an optional argument called `iterator` that takes a string as a value.

The iterator has two attributes, `key` and `value`. The key is either the map key if the data type is a map, or a list index if the data type is a list. If the data type is a set, then the key is set the same as the value and is not used in the configuration.

The data stored in `subnet_data` needs to be either a map or set, with each element holding the necessary values for a single nested block. Let's take a look at how we might define the variable for two subnets.

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

The `subnet_data` variable is a map. The map has two keys, `subnet1` and `subnet2`. The values for those keys are maps containing the information for each subnet. On each loop, the iterator (`subnet`) loads the the key and value for an element of the map. Of course, now we're back to a situation where we are statically defining our subnets in a variable. We'll leave that as an exercise to you.

If you find dynamic blocks a bit confusing, that's not uncommon. There are a powerful and necessary abstraction, but they also reduce the readability of Terraform configurations. They should be used where required and expedient, but also approached with caution.

Supplemental Information

- https://www.terraform.io/docs/configuration/expressions.html[75]

Key Takeaways Dynamic blocks allow you to define multiple nested blocks by supplying a set of values and a structure for the blocks. The use of dynamic blocks should be balanced between readability and efficiency.

8H: Describe Built-In Dependency Management (order of execution based)

When Terraform is analyzing a configuration, it creates a resource graph that lists out all the resources in your configuration and their dependencies. This allows Terraform to create the physical

[75] https://www.terraform.io/docs/configuration/expressions.html

resources in the proper order, and when possible create resources in parallel if they are not interdependent. Let's consider an example configuration where we have defined the following resources:

- Virtual network
- Virtual machine
- Virtual NIC
- Database as a service
- DNS Record

The virtual NIC will be attached to the virtual network. One of the arguments in configuration block of the virtual NIC will reference the virtual network, and thus Terraform knows that the virtual NIC is dependent on the virtual network. Following the same logic, the virtual machine will reference the virtual NIC. The DNS record will reference the public IP address of the VM, creating a new dependency. Terraform will evaluate this and realize that it must create resources in this order:

Virtual network -> Virtual NIC -> VM -> DNS Record

What about that database? Let's say that the application running on our VM needs the database to be available, but there's nothing in the configuration block that directly references the database. Terraform has no way of knowing that you need the database to be created first. In those situations you can let Terraform know by adding a `depends_on` meta-argument to the configuration block of the VM. The `depends_on` argument takes a list of resources in the configuration that this resource is dependent on.

In most cases you can rely on Terraform determining the correct order of resource creation. An explicit dependency will be the exception, rather than the rule. If you think you need to add a `depends_on` to your configuration, that's a signal that you might want to rethink the design of your architecture.

The virtual network and database much be created

Supplemental Information

- https://learn.hashicorp.com/terraform/getting-started/dependencies.html[76]

Key Takeaways Terraform builds a resource graph to automatically determine dependencies and resource creation order. It is possible to create an explicit dependency using `depends_on`, but this should be used sparingly.

[76] https://learn.hashicorp.com/terraform/getting-started/dependencies.html

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **8A: Demonstrate Use of Variables and Outputs:** Variables are how values are submitted to a Terraform configuration. There are many way to submit a value, and there is an order of evaluation, with last value winning. Outputs are primarily used to expose information from a child module to a parent module or expose information through remote state. Outputs are rendered during apply and can be of a complex data type.
- **8B: Describe Secure Secret Injection Best Practice:** Secret data doesn't belong in a `.tfvars` file or submitted at the command line. Keep secret data secret, and use environment variables.
- **8C: Understand the Use of Collection and Structural Types:** String, number, and boolean are the primitive types. Map, list, and set are the collection types. Object and tuple are the structural types. Collection types must have values of the same type, structural types can have values of different types.
- **8D: Create and Differentiate Resource and Data Configuration:** Data sources provide access to information stored in a provider. Resources allow you to create a physical resource on a provider. Both data sources and resources take arguments for their configuration and expose attributes for use in the rest of your Terraform configuration.
- **8E: Use Resource Addressing and Resource Parameters to Connect Resources Together:** Resources and data sources include arguments and attributes that can be referenced within a configuration. Be sure to pay attention to the data type exported by each attribute.
- **8F: Use Terraform Built-In Functions to Write Configuration:** Built-in functions enable you to manipulate information to match what you need. Functions take typed arguments and return typed data. Terraform console can help you test and validate functions to use in your configuration.
- **8G: Configure Resource Using a Dynamic Block:** Dynamic blocks allow you to define multiple nested blocks by supplying a set of values and a structure for the blocks. The use of dynamic blocks should be balanced between readability and efficiency.
- **8H: Describe Built-In Dependency Management:** Terraform builds a resource graph to automatically determine dependencies and resource creation order. It is possible to create an explicit dependency using `depends_on`, but this should be used sparingly.

In the next chapter we are going to explore the features of HashiCorp's paid offerings Terraform Cloud and Terraform Enterprise with the terminal objective **Understand Terraform Enterprise Capabilities**.

Objective 9: Understand Terraform Enterprise Capabilities

Terraform Enterprise is an on-premises distribution of Terraform Cloud. It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on. Terraform Cloud is a cloud-hosted version of Terraform Enterprise, with many of the same enterprise-grade architectural features. For most intents and purposes, Terraform Enterprise and Terraform Cloud can be treated as the same product. Although the terminal objective specifically calls out Terraform Enterprise, Terraform Cloud is part of this objective.

9A: Describe the Benefits of Sentinel, Registry, and Workspaces

Sentinel

Sentinel is a language and framework for policy, built to be embedded in existing software enabling fine-grained, logic-based policy decisions. A policy describes under what circumstances certain behaviors are allowed. Sentinel is an enterprise-only feature of HashiCorp Consul, Nomad, Terraform, and Vault.

Terraform Enterprise uses Sentinel to enforce policy on Terraform configurations, states, and plans.

The Sentinel integration with Terraform runs within Terraform Cloud after a `terraform plan` and before a `terraform apply`. The policies have access to the created plan, the state at the time of the plan, and the configuration at the time of the plan.

Sentinel Policies are rules enforced on Terraform runs to validate that the plan and corresponding resources follow company policies. Once a policy is added to an organization it is enforced on all runs.

The policy check will occur immediately after a plan is successfully executed in the run. If the plan fails, the policy check will not be performed. The policy check uses the generated `tfplan` file, simulated apply objects, state and configuration to verify the rules in each of the policies.

In short, Sentinel Policies can validate that the `terraform plan` results will align to corporate requirements (as defined within a Sentinel Policy), and if it doesn't, the creation of those resources (through `terraform apply`) will be prevented.

Registry

As we reviewed in Objective 5: Interact with Terraform Modules, Terraform has a public registry available for Terraform modules located at https://registry.terraform.io[77]. Some organizations may not want to use the public registry for their modules. Terraform Cloud offers a private registry feature that allows teams across an organization to share modules privately, including support for versioning, configuration design, and search functionality.

Organizations can further privatize their registry by choosing to go with Terraform Enterprise (a locally hosted version of Terraform Cloud). In that case, the private registry is hosted on company owned hardware, and is not exposed to the internet.

Workspaces

Okay, this is going to be a bit confusing, because you might be thinking to yourself that we already covered workspaces in Objective 4E: Given a Scenario: Choose When to Use Terraform WORKSPACE to Create Workspaces. You're not going crazy, we totally did. As a quick refresher, workspaces in Terraform running on your local machine create a separate state file for each workspace using the same configuration stored in a directory.

Terraform Cloud uses the term *Workspaces* in a slightly different capacity. Workspaces are how Terraform Cloud organizes infrastructure configurations, including the configuration, variables, state, and logs. We'll touch on that distinction more in the next section.

Supplemental Information

- https://www.terraform.io/docs/cloud/sentinel/index.html[78]
- https://www.terraform.io/docs/registry/index.html[79]
- https://www.terraform.io/docs/cloud/workspaces/index.html[80]

Key Takeaways Sentinel is a policy engine that can be used with multiple HashiCorp products including Terraform. Terraform Cloud includes both a private registry for modules and Workspaces for infrastructure configurations.

[77] https://registry.terraform.io
[78] https://www.terraform.io/docs/cloud/sentinel/index.html
[79] https://www.terraform.io/docs/registry/index.html
[80] https://www.terraform.io/docs/cloud/workspaces/index.html

9B: Differentiate OSS and TFE Workspaces

Before we dive into the differences between workspaces in Terraform Cloud and workspaces in the Terraform CLI, let's spend a few moments touching on the difference between Terraform Cloud and OSS. Terraform Open-Source Software is the free, open-source CLI that you can download and run on a local system.

Prior to the announcement of Terraform Cloud in 2019, HashiCorp had a hosted offering called Terraform Enterprise. It was built on top of Terraform OSS, while adding several enhancements and additional features. Terraform Enterprise could also run on servers hosted on-premises, and that was called Private Terraform Enterprise.

The hosted version of Terraform Enterprise was renamed Terraform Cloud, along with several licensing updates and a free-tier for teams of five or less. Private Terraform Enterprise was renamed to simply Terraform Enterprise and continued to run on-premises. Naturally, this has led to a fair amount on confusion when people refer to Terraform Enterprise, so here's a helpful reminder:

- Terraform OSS - Free and open-source CLI
- Terraform Cloud - hosted Terraform for Teams (formerly Terraform Enterprise)
- Terraform Enterprise - on-premises Terraform for Teams (formerly Private Terraform Enterprise)

Alright, with that out of the way, we can talk about the workspaces in Terraform OSS vs. the workspaces in Terraform Cloud.

Open Source Software (OSS)

In Objective 4: Use the Terraform CLI we talked about workspaces; how to create them and when to use them. The central point of a workspace in Terraform OSS is to use the same configuration for multiple environments, but have a separate state file for each environment managed by Terraform.

> Fun fact, *workspaces* used to be called *environments*, and if you look at the available subcommands for `terraform` one of them is `env` - short for environment. The subcommand is still there for backwards compatibility, but it is essentially an alias for the `workspace` subcommand.

Workspaces in Terraform OSS deal exclusively with state management, and none of the other aspects of a configuration, such as the variable values submitted or the logs of previous actions. If you'd like to submit different values for each workspace, you'll need to work that into the logic of your script or pipeline.

Terraform Cloud

Terraform Cloud takes a different approach to using workspaces. The state management component is still there, but some extras have been added on top. The Terraform Cloud offering integrates with Version Control Systems (VCS) like GitHub and GitLab. The workspace construct in Terraform Cloud points to a VCS repository for its configuration. The variable values, state, secrets, and credentials are also stored as part of the workspace settings.

A Terraform Cloud workspace contains:

- A Terraform configuration (usually retrieved from a VCS repo, but sometimes uploaded directly)
- Values for variables used by the configuration
- Credentials and secrets stored as sensitive variables
- Persistent stored state for managed resources
- Historical state and run logs

You can still use the same repository for multiple workspaces in the way you used the same directory for multiple workspaces in Terraform OSS. The big improvements are really around managing values for variables, credentials, and secrets. It's also simpler to automate updates using webhooks into the VCS you are using.

Supplemental Information

- OSS workspaces: https://www.terraform.io/docs/state/workspaces.html[81]
- TFE workspaces: https://www.terraform.io/docs/cloud/workspaces/index.html[82]
- Terraform Enterprise Workspaces Walkthrough[83]
- Staying DRY Using Terraform Workspaces with GitLab-CI[84]
- Why Consider Terraform Enterprise Over Open Source?[85]

Key Takeaways Workspaces in Terraform OSS provide managed state using the same configuration. Terraform Cloud adds functionality to workspaces with value management, VCS integration, and historical logs.

[81] https://www.terraform.io/docs/state/workspaces.html
[82] https://www.terraform.io/docs/cloud/workspaces/index.html
[83] https://www.youtube.com/watch?v=atBRAG_3yNQ
[84] https://www.youtube.com/watch?v=PtxtGPxCaQ8
[85] https://www.hashicorp.com/resources/why-consider-terraform-enterprise-over-open-source

9C: Summarize Features of Terraform Cloud

Terraform Cloud (formerly known as Terraform Enterprise) is a hosted service from HashiCorp that simplifies the process of collaboration when using Terraform. Rather than running Terraform locally, Terraform Cloud runs in a hosted environment that is consistent, reliable, and collaborative. It's more than just running Terraform remotely, Terraform Cloud includes additional features to enable shared state, secret data, access controls, a private registry, and more.

Here's a quick list of the features included in Terraform Cloud for your reference:

- **Workspaces** - Workspaces are linked to a VCS and include values, persistent state, and historical logs.
- **Version Control System (VCS) integration** - Integration with several popular VCS solutions allows automatic runs of Terraform, planning runs based on pull requests, and publishing modules to a private registry.
- **Private Module Registry** - Used to share modules across an organization privately and securely.
- **Remote Operations** - Terraform Cloud uses managed, disposable virtual machines to run Terraform commands remotely and provide advanced features like Sentinel policy enforcement and cost estimation.
- **Organizations** - Access control is managed through users, groups, and organizations.
- **Sentinel** - A language and framework for policy to enable fine-grained, logic-based policy decisions.
- **Cost Estimation** - Estimate the cost of resources being provisioned by a configuration.
- **API** - A subset of Terraform Cloud's features are available through their API.
- **ServiceNow** - An integration with ServiceNow is available for Terraform Enterprise customers.

Supplemental Information

- https://www.terraform.io/docs/cloud/index.html[86]

Key Takeaways Terraform Cloud is a hosted service that expands and enhances the Terraform OSS offering. Some of the most important additions are workspaces, VCS integration, private module registry, and Sentinel.

[86] https://www.terraform.io/docs/cloud/index.html

Chapter Summary

Let's quickly summarize the enabling objectives for this chapter and the key takeaways from each objective.

- **9A: Describe the Benefits of Sentinel, Registry, and Workspaces:** Sentinel is a policy engine that can be used with multiple HashiCorp products including Terraform. Terraform Cloud includes both a private registry for modules and Workspaces for infrastructure configurations.
- **9B: Differentiate OSS and TFE Workspaces:** Workspaces in Terraform OSS provide managed state using the same configuration. Terraform Cloud adds functionality to workspaces with value management, VCS integration, and historical logs.
- **9C: Summarize Features of Terraform Cloud:** Terraform Cloud is a hosted service that expands and enhances the Terraform OSS offering. Some of the most important additions are workspaces, VCS integration, private module registry, and Sentinel.

That's it! You've finished all the objectives. Give yourself a pat on the back and go schedule that exam.

Conclusion

We hope that this guide has proven useful in your preparation for the HashiCorp Certified Terraform Associate exam.

We covered a lot of material, but only scratched the surface. Over the course of writing this guide, we learned some things about Terraform that we didn't know! Don't worry that you are not yet a Terraform expert, the exam is for someone at the Associate level.

At the end of each chapter we provided a summary of Key Takeaways to help you cram before you take the exam. We recommend reading through them again before going in to take the exam.

Speaking of the exam, if you decide to sit the exam we'd love to know how you did. Pass or fail, we want to hear about it! Hit either of us up on LinkedIn or Twitter and share your experience. We want to make this guide the best it can be and understanding your experience will help us tremendously.

In the Appendix, you will find additional resources to help you on your Terraform learning journey.

Appendix – Additional Resources

The authors have spent considerable time reading, watching, and learning about Terraform. And through all this effort, we wanted to provide a list of additional resources that we feel are particularly useful in getting up to speed with Terraform.

Articles

A collection of various articles that have clarified key concepts, and demonstrated real-world environments leveraging Terraform.

- A Comprehensive Guide to Terraform[87]
- 7 Tips to Start Your Terraform Project the Right Way[88]
- Terraform Best Practices[89]
- Terraform Recommended Practices[90]
- How to create reusable infrastructure with Terraform modules[91]
- Running Terraform in Automation[92]
- Terraform Enterprise – Repository Structure[93]
- Using Terraform with Azure – What's the benefit?[94]

Books

These are some the books that we've found most useful/helpful in learning about Terraform.

- Terraform Up & Running[95] book
  - NOTE: There is a 2nd edition[96] of this publication that was released in October 2019.
- The Terraform Book[97]

[87] https://blog.gruntwork.io/a-comprehensive-guide-to-terraform-b3d32832baca
[88] https://medium.com/@simon.so/7-tips-to-start-your-terraform-project-the-right-way-93d9b890721a
[89] https://jamesdld.github.io/terraform/Best-Practice/
[90] https://www.terraform.io/docs/enterprise/guides/recommended-practices/index.html
[91] https://blog.gruntwork.io/how-to-create-reusable-infrastructure-with-terraform-modules-25526d65f73d
[92] https://www.terraform.io/guides/running-terraform-in-automation.html?utm_content=bufferd6ac3&utm_medium=social&utm_source=linkedin.com&utm_campaign=buffer&lipi=urn%3Ali%3Apage%3Ad_flagship3_feed%3BsnbY7kQITMCjj4ViGG2WDw%3D%3D
[93] https://www.terraform.io/docs/enterprise/workspaces/repo-structure.html
[94] https://samcogan.com/terraform-and-azure-whats-the-benefit/?utm_content=buffer7563f&utm_medium=social&utm_source=linkedin.com&utm_campaign=buffer
[95] https://www.amazon.com/Terraform-Running-Writing-Infrastructure-Code/dp/1491977086
[96] https://www.amazon.com/Terraform-Running-Writing-Infrastructure-Code/dp/1492046906/ref=dp_ob_title_bk
[97] https://terraformbook.com/

Pluralsight Courses

These are the courses created by Ned Bellavance (one of the authors of this publication), that are also referred to throughout this guide.

- Terraform – Getting Started[98]
- Deep Dive – Terraform[99]
- Implementing Terraform on Microsoft Azure[100]

Hands-On Labs

In learning anything new, there is only so much you can learn/pick up from reading alone. That's why we also appreciate it when we find a good hands-on lab guide to help re-enforce our understanding and learning.

One thing that we especially appreciate about the Azure Citadel site, is that the workshops are not just a step-by-step hold-your-hand type of guide. They start out that way to get you started, but very quickly, they change to "this is what you need to do, go figure out how to do it". We like this approach, because it forces you to research and learn, and not just copy/paste.

- Terraform on Azure (Hands-On Lab)[101]

HashiCorp has also developed an learning platform that includes many common scenarios with Terraform. We highly recommed checking it out as well.

- HashiCorp - Learn Terraform[102]

Videos

This is in addition to the Pluralsight courses themselves (as not everyone may have access, or a subscription, to Pluralsight).

- Evolving Your Infrastructure with Terraform[103]
- Happy Terraforming! Real-world experience and proven best practices[104]

[98] https://app.pluralsight.com/library/courses/terraform-getting-started/
[99] https://app.pluralsight.com/library/courses/terraform-getting-started/
[100] https://app.pluralsight.com/library/courses/implementing-terraform-microsoft-azure/
[101] https://azurecitadel.com/automation/terraform/
[102] https://learn.hashicorp.com/terraform
[103] https://www.youtube.com/watch?v=wgzgVm7Sqlk&t=6s
[104] https://www.hashicorp.com/resources/terraforming-real-world-experience-best-practices

- HashiCorp Terraform adoption: A typical journey[105]
- HashiConf 2018 Day Two Keynote: Terraform is Changing the World[106]
- Scaling with Terraform: The journey from startup to enterprise[107]
- 5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code[108]

---

[105] https://www.hashicorp.com/resources/terraform-adoption-journey
[106] https://www.hashicorp.com/resources/hashiconf-day-two-keynote-terraform-way
[107] https://www.hashicorp.com/resources/scaling-with-terraform-startup-enterprise
[108] https://www.youtube.com/watch?v=RTEgE2lcyk4