CELEBRATING 13 YEARS

Quality Thought®

TERRAFORM®

BY KHAJA

TABLE OF CONTENTS

TERRAFORM – INTRODUCTION

GETTING TO KNOW INFRASTRUCTURE AS CODE (IAC)

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

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

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

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

ADVANTAGES OF IAC

SIMPLE AND SPEEDY

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

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

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

CONFIGURATION CONSISTENCY

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

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

RISK MINIMIZATION

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

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

INCREASED EFFICIENCY IN SOFTWARE DEVELOPMENT

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

infrastructure code can be done together following **Continuous Integration and Continuous Deployment** (**CI/CD**) techniques.

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

COST SAVINGS

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

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

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

INTRODUCTION TO TERRAFORM

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

WHAT IS TERRAFORM?

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

Terraform mainly uses Terraform files ending with **.tf** or **.tf.json** that hold detailed information about what infrastructure components are required in order to run a single application or your entire data

center. Terraform generates an execution plan, which describes what it is going to do to reach the desired state, and then executes it to build the infrastructure described. If there is any change in the configuration file, Terraform is able to determine what has been changed and create incremental execution plans that can be applied.

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

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

FEATURES OF TERRAFORM

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

INFRASTRUCTURE AS CODE

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

EXECUTION PLANS

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

RESOURCE GRAPH

Terraform builds a *graph* of all your resources and parallelizes the creation and modification of any non-dependent resources. Because of this resource graph, Terraform manages to build infrastructure as efficiently as possible that is sufficiently intelligent to understand dependencies in its infrastructure.

CHANGING AUTOMATION

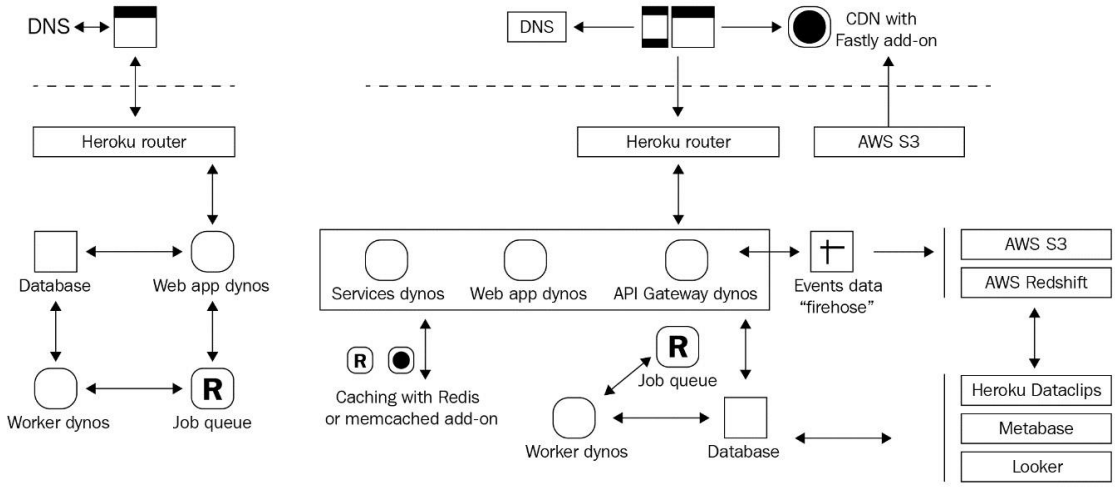
Complex changes to your defined infrastructure can be applied with minimal *human interaction*. With the aforementioned execution plan and resource graph, you know exactly what Terraform will change and in what order, thereby avoiding multiple possible human errors.

TERRAFORM USE CASES

As we have got to know what Terraform is, let's now learn about some of the *use cases* of Terraform in the enterprise world. A few of them have been discussed as follows.

HEROKU APP SETUP

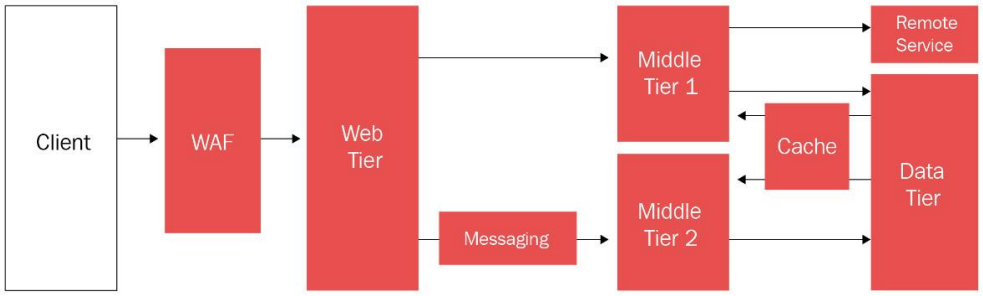
Heroku is one of the most popular **Platforms as a Service** (**PaaS**) for hosting *web apps*. Developers use it to create an app and then attach other services, such as a database or email provider. One of the best features of the Heroku app is its ability to elastically scale the number of dynos or workers. However, most non-trivial applications quickly require many add-ons and external services:



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

MULTI-TIER APPLICATIONS

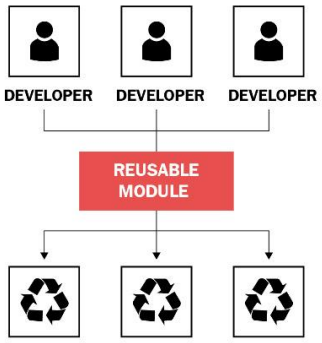
N-tier architecture deployment is quite common across the industry when thinking about the required infrastructure for an application. Generally, two-tier architecture is more in demand. This is a pool of web servers and a database tier. As per the application requirements, additional tiers can be added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



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

SELF-SERVICE CLUSTERS

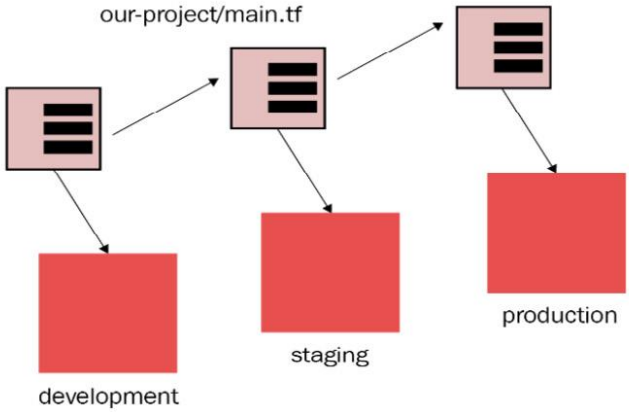
In a large organization, it's quite challenging for the central operation team to provide infrastructure to the product team as and when needed. The product team should be able to create and maintain their infrastructure using tooling provided by the central operations team:



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

DISPOSABLE ENVIRONMENTS

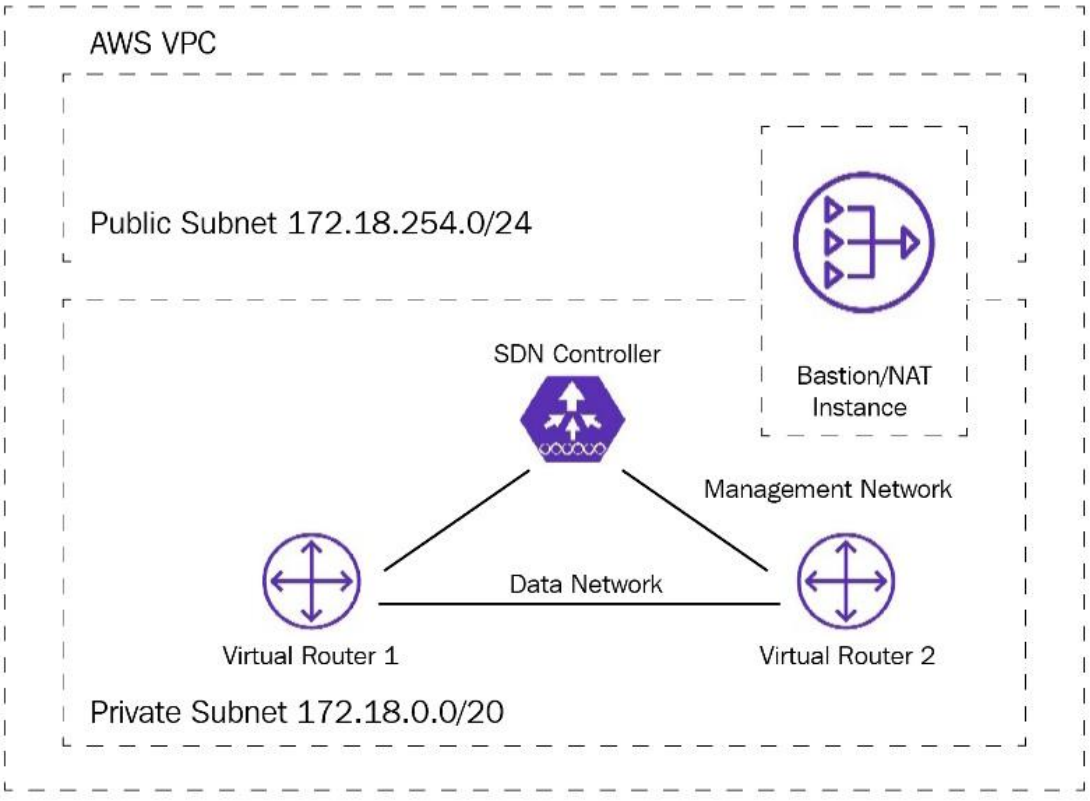
In the industry, it is quite common to have multiple landscapes, including production, staging, or development environments. These environments are generally designed as a subset of the production environment, so as and when any application needs to be deployed and tested, it can easily be done in the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



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

SOFTWARE-DEFINED NETWORKING

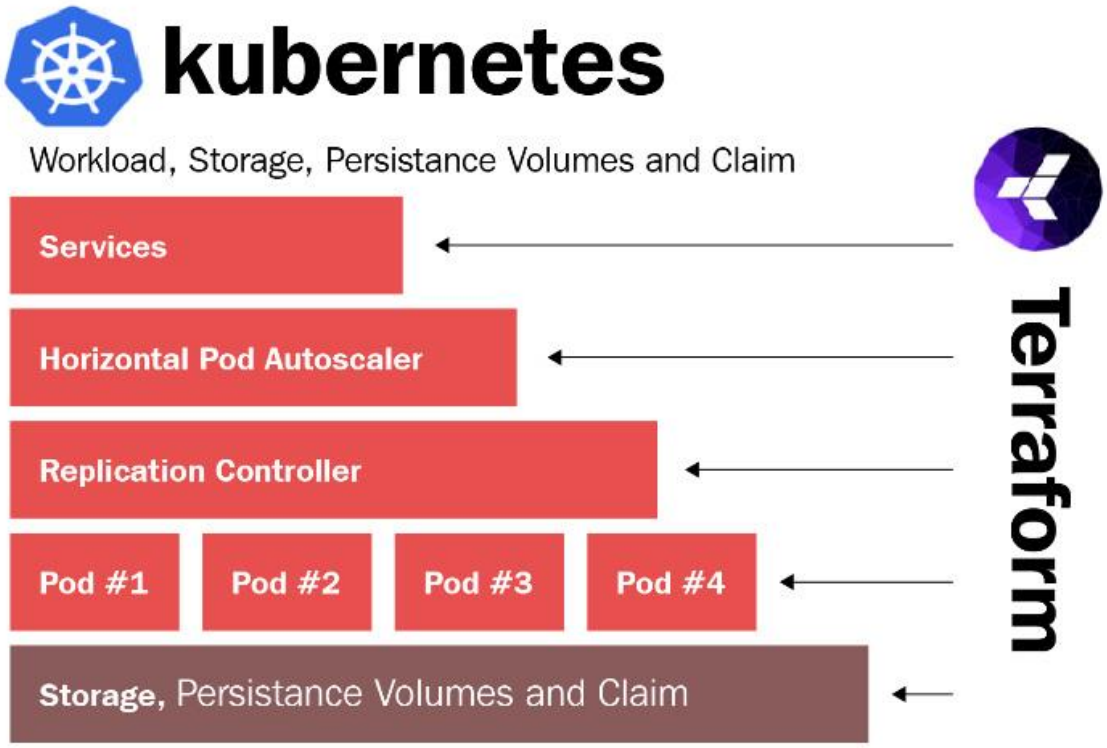
Software-Defined Networking (**SDN**) is quite famous in data centers, as it allows operators to operate a software-defined network very smoothly and developers are able to develop their applications, which can easily be run on top of the network infrastructure provided. The control layer and infrastructure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



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

RESOURCE SCHEDULERS

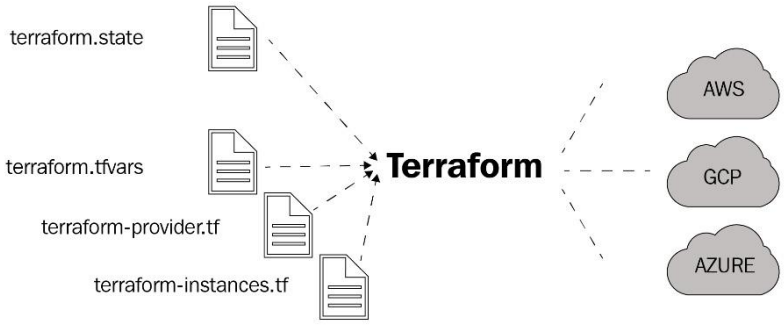
In large-scale infrastructures, the static assignment of applications to machines is very challenging. In terms of Terraform, there are many schedulers available, such as **Borg**, **Mesos**, **YARN**, and **Kubernetes**, that can be used to overcome this problem. These can be used to dynamically schedule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



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

MULTI-CLOUD DEPLOYMENT

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

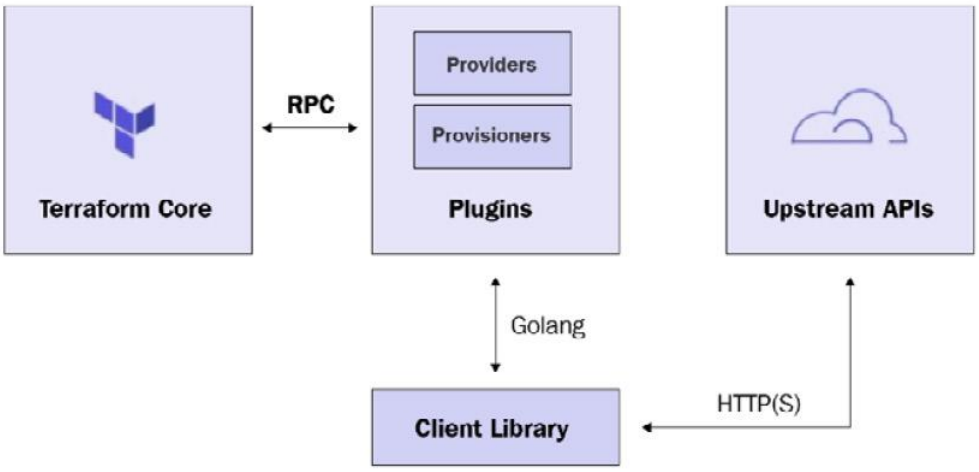


Realizing the complexity of multi-cloud infrastructure deployments using already-existing tools that are very specific to each cloud provider, HashiCorp came up with an approach known as Terraform. Terraform is cloud-agnostic. A single configuration can be used to manage multiple providers, and can even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

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

AN UNDERSTANDING OF TERRAFORM ARCHITECTURE

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



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

TERRAFORM CORE

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

The responsibilities of Terraform Core are as follows:

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

TERRAFORM PLUGINS

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

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

The responsibilities of provider plugins are as follows:

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

The responsibilities of provisioner plugins are as follows:

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

PLUGIN LOCATIONS

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

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

You can visit the following link for more information on plugin locations: https://developer.hashicorp.com/terraform/plugin/how-terraform-works#plugin-locations

SELECTING PLUGINS

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

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

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

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

UPGRADING PLUGINS

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

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

TERRAFORM INSTALLATION GUIDE

Navigate to here and follow the docs https://developer.hashicorp.com/terraform/downloads

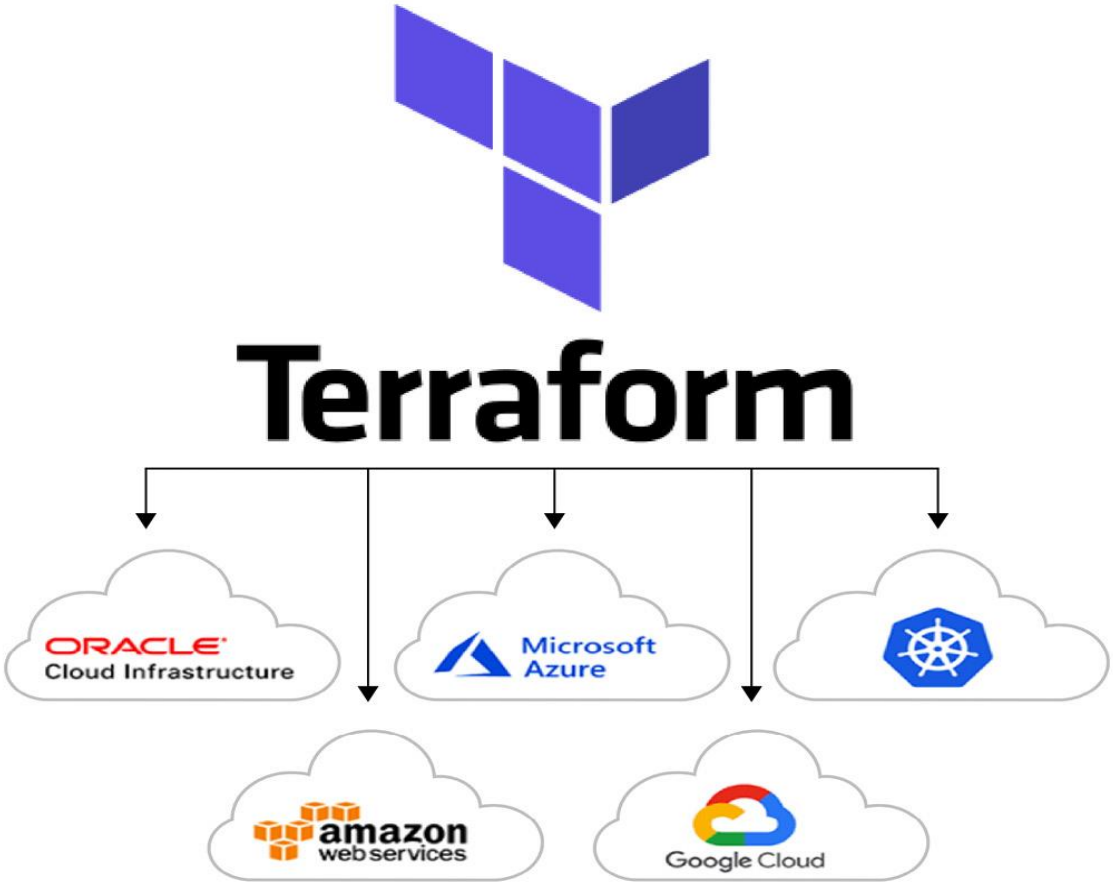
GETTING STARTED WITH TERRAFORM

INTRODUCING TERRAFORM PROVIDERS

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

TERRAFORM PROVIDERS

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



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

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

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

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

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

AZURERM TERRAFORM PROVIDER

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

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.49.0"
    }
  }
}

provider "azurerm" {
  # Configuration options

  features {}

  subscription_id   = "...."
  client_id         = "...."
  client_secret     = "...."
  tenant_id         = "...."

}
```

Arguments are the inputs and Attributes are outputs in the context of terraform.

Let's try to understand what are the different arguments supported:

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

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

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

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

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

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "3.49.0"
    }
  }

}
```

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

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

- **>= 3.49.0**: Greater than or equal to the version.
- **= 3.49.0**: Equal to the version.
- **!= 3.49.0**: Not equal to the version.
- **<= 3.49.0**: Less than or equal to the version.
- **~> 3.49.0**: This one is funky. It means any version in the **3.49.X** range. It will always look for the rightmost version increment.

- **>= 3.46, <= 3.49** : Any version between **2.46** and **3.49**, inclusive.

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

- key_vault
- template_deployment
- virtual_machine
- virtual_machine_scale_set
- log_analytics_workspace

The **key_vault** block supports the following arguments:

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

The **template_deployment** block supports the following argument:

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

The **virtual_machine** block supports the following argument:

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

The **virtual_machine_scale_set** block supports the following argument:

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

The **log_analytics_workspace** block supports the following arguments:

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

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

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

Here is an example of the code:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "3.49.0"
    }
  }
}
# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
}
provider "azurerm" {
  features {}
  alias = "nonprod_01_subscription"
}
# To Create Resource Group in specific subscription
resource "azurerm_resource_group" "example" {
  provider = azurerm.nonprod_01_subscription
  name     = "example-resources"
  location = "West Europe"
}
```

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

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "3.49.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
```

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

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

AWS TERRAFORM PROVIDER

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

The following is a code snippet:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.60.0"
    }
  }
}

provider "aws" {
  # Configuration options
}
```

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

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

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

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

GOOGLE TERRAFORM PROVIDER

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

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version = "4.58.0"
    }
  }
}

provider "google" {
  # Configuration options
}
```

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

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

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

KNOWING ABOUT TERRAFORM RESOURCES

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

TERRAFORM RESOURCES

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

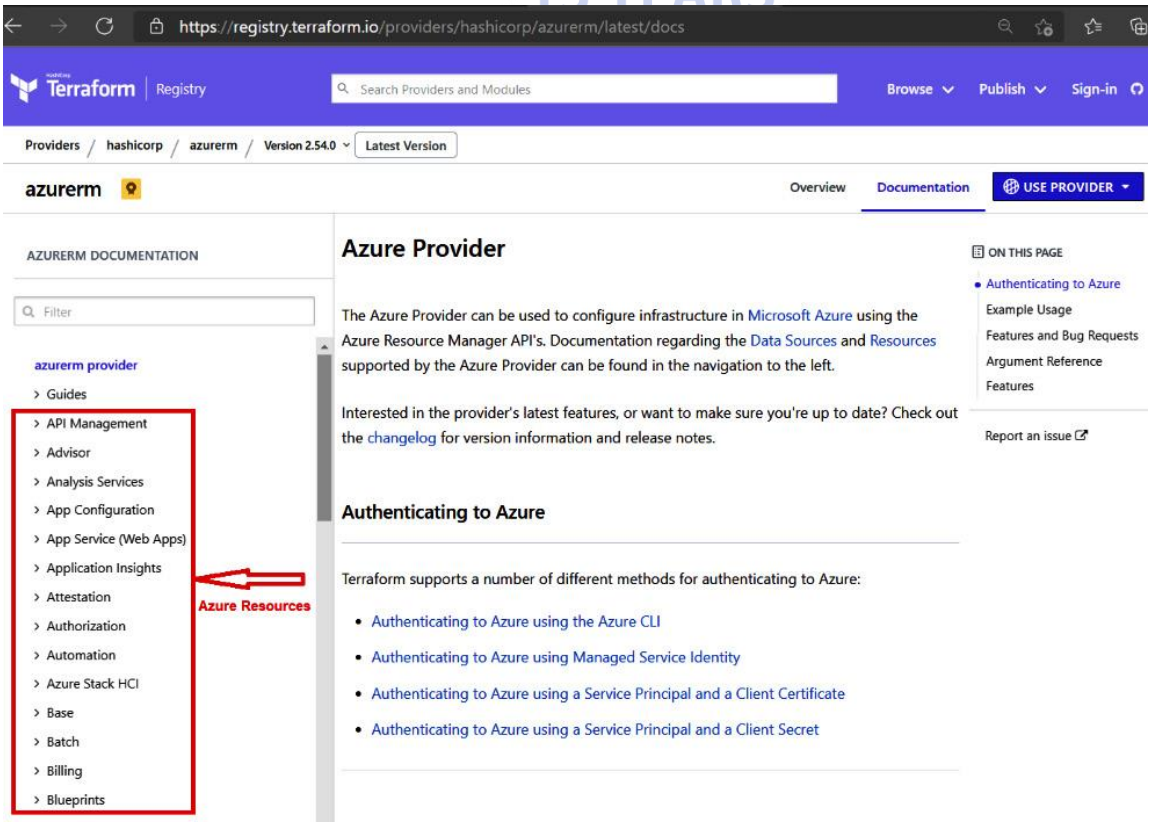
AZURE TERRAFORM RESOURCE

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

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

  tags = {
    environment = "test"
  }
}
```

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



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

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

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

AWS TERRAFORM RESOURCE

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

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

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

GOOGLE TERRAFORM RESOURCE

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

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

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

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

UNDERSTANDING TERRAFORM VARIABLES

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

TERRAFORM VARIABLES

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

AZURE TERRAFORM INPUT VARIABLES

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

```
# To Create Resource Group
resource "azurerm_resource_group" "example" {
  name     = var.rgname
  location = var.rglocation
}
# To Create Azure Public IP Address
resource "azurerm_public_ip" "azure-pip" {
  name                = var.public_ip_name
  location            = azurerm_resource_group.example.location
```

```
  resource_group_name     = azurerm_resource_group.example.name
  allocation_method       = var.allocation_method
  idle_timeout_in_minutes = var.idle_timeout_in_minutes
  tags                    = var.tags
}
```

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

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

```
  default = {
    environment = "Test"
    Owner       = "Azure-Terraform"
  }
}
```

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

$ export **TF_VAR_rgname**=example-rg

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

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

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

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

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

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

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

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

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

Here, the question is which one should be preferred; if you are defining variable values in a Terraform environment variables, defining in **terraform.tfvars**, defining default values, or providing values during the runtime. Hence, things happen in the following sequence: *Environment variable values* | *values during runtime* | **terraform.tfvars** | *default values*.

AWS TERRAFORM INPUT VARIABLES

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

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

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

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

We have to define the **variables.tf** file for the **ec2 instance** resource code. Next, we need to have the values of those defined variables. So, let's define the values in the **terraform.tfvars** file:

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

GCP TERRAFORM INPUT VARIABLES

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

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

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

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

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

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

We have covered Terraform variables and acquired an understanding of what the best practices of defining Terraform variables are and how we can take input from the users by passing variable values in Terraform environment variables, **terraform.tfvars**, default values, and runtime from the CLI. The take-away from this entire topic is how effectively you can write a Terraform configuration file with the help of Terraform variables.

UNDERSTANDING TERRAFORM OUTPUT

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

TERRAFORM OUTPUT

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

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

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

AZURE TERRAFORM OUTPUT

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

```
resource "azurerm_resource_group" "example" {
  name     = "resourceGroup1"
  location = "West US"
}
resource "azurerm_public_ip" "example" {
  name                = "acceptanceTestPublicIp1"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  allocation_method   = "Static"
  tags = {
    environment = "Production"
  }
}
```

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

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

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

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

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

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

```
# To Create Azure Resource Group
resource "azurerm_resource_group" "example" {
  name     = "Terraform-rg"
  location = "West Europe"
}
# To Create Azure Public IP Address
resource "azurerm_public_ip" "example" {
  name                    = "Terraform-pip"
  location                = azurerm_resource_group.example.location
  resource_group_name     = azurerm_resource_group.example.name
  allocation_method       = "Static"
  idle_timeout_in_minutes = 10
}
# To Create Azure Load Balancer
resource "azurerm_lb" "example" {
  name                = "Terraform-LoadBalancer"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  frontend_ip_configuration {
    name                 = azurerm_public_ip.example.name
    public_ip_address_id = azurerm_public_ip.example.id
  }
}
```

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

AWS TERRAFORM OUTPUT

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

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

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

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

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

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

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

```
resource "aws_subnet" "terraform-subnet" {
  vpc_id     = aws_vpc.terraform-vpc.id
  cidr_block = "10.0.0.0/24"
  tags = {
    Environment = "Terraform-lab"
  }
}
```


GCP TERRAFORM OUTPUT

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

```
resource "google_project" "my_project" {
  name       = "My Project"
  project_id = "your-project-id"
  org_id     = "1234567"
}
resource "google_app_engine_application" "app" {
  project     = google_project.my_project.project_id
  location_id = "us-central"
}
```

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

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

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

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

TERRAFORM OUTPUT OPTIONAL ARGUMENTS

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

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

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

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

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

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

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

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

```
output "instance_ip_addr" {
  value       = aws_instance.server.private_ip
  description = "The private IP address of the main server instance."
  depends_on = [
    # Security group rule must be created before this IP address could
    # actually be used, otherwise the services will be unreachable.
    aws_security_group_rule.local_access,
  ]
}
```

UNDERSTANDING TERRAFORM DATA

n this section, we are going to discuss how you can define Terraform data sources. You can also refer to https://www.terraform.io/docs/configuration/data-sources.html

to see under which circumstances you would be able to use Terraform data sources. Just as with Terraform output, resources, providers, and variables, we will be concentrating on Terraform data sources for AWS, GCP, and Azure.

TERRAFORM DATA SOURCES

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

AZURE TERRAFORM DATA SOURCES

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

```
data "azurerm_virtual_network" "example" {
  name                = "production-vnet"
  resource_group_name = "Terraform-rg"
}
output "virtual_network_id" {
  value = data.azurerm_virtual_network.example.id
}
```

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

```
data "azurerm_resource_group" "example" {
  name = "Terraform-rg"
}
data "azurerm_virtual_network" "example" {
  name                = "production-vnet"
  resource_group_name = data.azurerm_resource_group.example.name
}
resource "azurerm_subnet" "example" {
  name                 = "terraform-subnet"
```

```
  resource_group_name  = data.azurerm_resource_group.example.name
  virtual_network_name = data.azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.1.0/24"]
}
```

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

AWS TERRAFORM DATA SOURCES

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

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

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

GCP TERRAFORM DATA SOURCES

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

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

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

INTRODUCING THE TERRAFORM BACKEND

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

TERRAFORM STATE

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

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

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

THE PURPOSE OF THE TERRAFORM STATE FILE

As you know, Terraform generates a state file. You might be thinking, why does Terraform **Infrastructure as Code** (**IaC**) need a state file, and what will happen if Terraform doesn't have a state file? To answer that question, a state file in Terraform stores current knowledge of the state of the configured

infrastructure, which reduces the complexity of the resource deployment when you are handling large Enterprise infrastructures. We are going to discuss what the main purpose of having a Terraform state file is and how it benefits us:

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

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

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

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

TERRAFORM BACKEND TYPES

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

LOCAL BACKEND

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

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

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

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

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

}
```

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

terraform.tfstate.d directory and a subdirectory for each workspace. In each workspace directory, a new **terraform.tfstate** file got created:

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

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

REMOTE BACKEND

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

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

and you will find that the Terraform operation will keep on running remotely until it is completed. This remote operation is only supported by the *SaaS* product of HashiCorp, that is *Terraform Cloud*.

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

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

The enhanced remote backends are either Terraform Cloud or Terraform Enterprise

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

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

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

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

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

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

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

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

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

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

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

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

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

```
  }
}
```

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

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

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

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

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

```
data "terraform_remote_state" "example" {
  backend = "gcs"
  config = {
    bucket  = "terraform-state"
    prefix  = "prod"
  }
}
resource "template_file" "terraform" {
  template = "${greeting}"
  vars {
    greeting = "${data.terraform_remote_state.example.greeting}"
  }
}
```

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

UNDERSTANDING TERRAFORM PROVISIONERS

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

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

TERRAFORM PROVISIONER USE CASES

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

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

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

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

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

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

TERRAFORM PROVISIONER TYPES

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

THE LOCAL-EXEC PROVISIONER

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

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

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

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

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

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


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

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

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

```
resource "google_compute_instance" "example" {
  # ...
  provisioner "local-exec" {
    when    = destroy
    command = "echo We are discussing Destroy-time provisioner"
  }
}
```

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

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

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

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


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

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

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

Suppose you are willing to export some output of the resource to the local file. Then, you can write the following code block. As you can see from this code block, we would be able to get the AWS

instance's private IP address to our local **private_ips.txt** filename and that will be created in the local directory:

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

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

THE FILE PROVISIONER

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

```
resource "aws_instance" "example" {
  # ...
  # Copies the app.conf file to /etc/app.conf
  provisioner "file" {
    source      = "conf/app.conf"
    destination = "/etc/app.conf"
  }
  # Copies the string in content into /tmp/amifile.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/amifile.log"
  }
  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }
  # Copies all files and folders in apps/apptest to D:/IIS/webapp
  provisioner "file" {
    source      = "apps/apptest/"
    destination = "D:/IIS/webapp"
  }
}
```

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

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

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

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


THE REMOTE-EXEC PROVISIONER

The **remote-exec** provisioner helps you to run a script on the remote resource once it is created. You can use the **remote-exec** provisioner to run a configuration management tool, bootstrap into a

cluster, and so on. It also supports both **ssh** and **winrm** type connections. The following code block will give you an idea of how you can define the **remote-exec** provisioner:

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

The previously defined code block has the following arguments:

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

UNDERSTANDING TERRAFORM LOOPS

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

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

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

THE COUNT EXPRESSION

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

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

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

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


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

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

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

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

We can give them our own defined names, rather than take them through **count.index**. We can perform this customization by defining a variable code block:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
------------------------------------------------------------------------
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy
-/+ destroy and then create replacement
Terraform will perform the following actions:
  # azurerm_resource_group.example[1] must be replaced
-/+ resource "azurerm_resource_group" "example" {
      ~ id       = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/AWS-rg" ->
(known after apply)
        location = "westeurope"
      ~ name     = "AWS-rg" -> "Google-rg" # forces replacement
      - tags     = {} -> null
    }
  # azurerm_resource_group.example[2] will be destroyed
  - resource "azurerm_resource_group" "example" {
      - id       = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Google-rg" ->
null
      - location = "westeurope" -> null
      - name     = "Google-rg" -> null
      - tags     = {} -> null
    }
Plan: 1 to add, 0 to change, 2 to destroy.
```

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

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

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

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

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

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

THE FOR_EACH EXPRESSION

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

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

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

We have used the **toset** Terraform function to convert the **var.rg_names** list into a set of strings. **for_each** only supports a set of strings and maps in the resource code block. If we use a **for_each** loop

over this set, then **each.value** or **each.key** will provide the names of the resource groups. Generally, **each.key** is used for maps where you have key-value pairs.

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

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

THE FOR EXPRESSION

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

```
variable "cloud" {
  description = "A list of cloud"
  type        = list(string)
  default     = ["azure", "aws", "gcp"]
}
output "cloud_names" {
  value = [for cloud_name in var.cloud : upper(cloud_name)]
}
```

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

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

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

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

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

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


UNDERSTANDING TERRAFORM FUNCTIONS

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

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

It is most important for you to understand how to use functions and how to test the Terraform functions using the Terraform console.I f you are already familiar with any programming language, then defining a Terraform function works the same. The basic format for writing a Terraform function is **function(arguments,...)**.

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

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

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

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

Let's try to understand this with an example:

```
variable "address_space" {
  type    = list(string)
  default = ["10.0.0.0/16"]
}
resource "azurerm_resource_group" "azure-rg" {
  name     = "Terraform-rg"
  location = "eastus"
}
resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location            = azurerm_resource_group.azure-rg.location
  resource_group_name = azurerm_resource_group.azure-rg.name
  address_space       = var.address_space
  subnet {
    name          = "subnet1"
    address_prefix = cidrsubnet(var.address_space[0], 8, 1)
  }
}
```

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

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

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

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

UNDERSTANDING TERRAFORM DEBUGGING

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

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

```
export TF_LOG=TRACE
```

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

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

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

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

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

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

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

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

INTRODUCTION TO THE TERRAFORM CLI

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

```
$ terraform
Usage: terraform [global options] <subcommand> [args]
The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure
All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management
Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

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

```
$ terraform graph -h
Usage: terraform graph [options] [DIR]
  Outputs the visual execution graph of Terraform resources according to configuration files in DIR (or
the current directory if omitted).
  The graph is outputted in DOT format. The typical program that can read this format is GraphViz,
but many web services are also available to read this format.
```

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

Options:

-draw-cycles Highlight any cycles **in** the graph with colored edges. This helps when diagnosing cycle errors.

-type=plan Type of graph **to** output. Can be: plan, plan-destroy, apply, validate, input, refresh.

-module-depth=n (deprecated) **In** prior versions of Terraform, specified the depth of modules **to** show **in** the output.

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

terraform -install-autocomplete

UNDERSTANDING THE TERRAFORM CLI COMMANDS

We will now discuss some of the basic the Terraform CLI commands, as follows:

- **terraform console**: You can run this command on the CLI to open a Terraform console, where you can test or get the output of the code of certain Terraform functions. The following example shows how you can use this command:


```
$ terraform console
> max(5,10,-5)
10
>
```

- **terraform fmt**: You can run this command to rewrite configuration files to a canonical format and style. This command performs some sort of adjustment so that your configuration code is in a readable format, and even helps you make some changes to follow Terraform's language-style conventions. To know more about Terraform language-style conventions, you can read https://www.terraform.io/docs/configuration/style.html. The following subcommand flags are supported by the **terraform fmt** command:

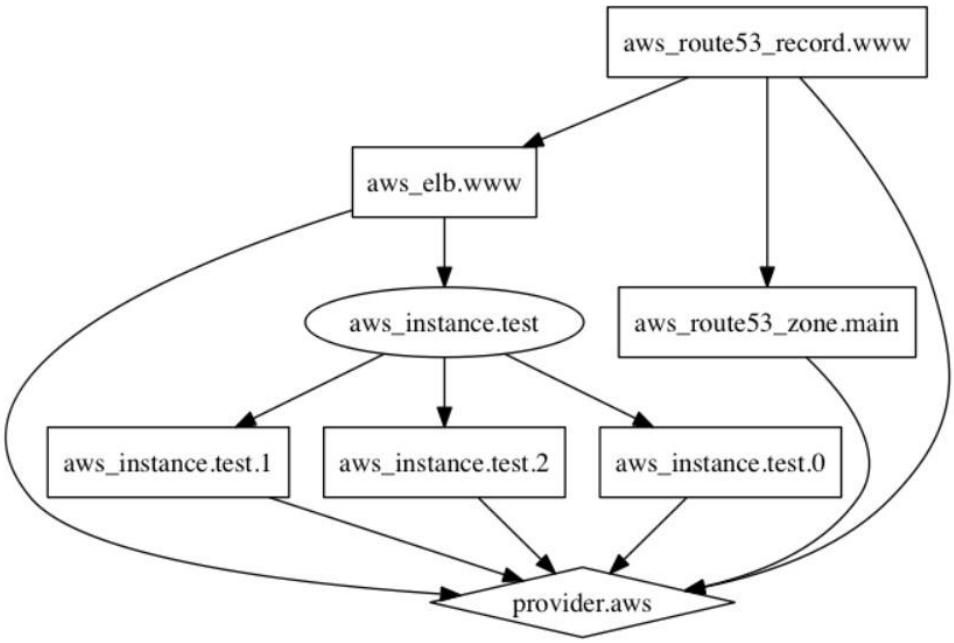
```
$ terraform fmt -h
Usage: terraform fmt [options] [DIR]
    Rewrites all Terraform configuration files to a canonical format. Both configuration files (.tf) and
variables files (.tfvars) are updated. JSON files (.tf.json or .tfvars.json) are not modified.
    If DIR is not specified then the current working directory will be used.
    If DIR is "-" then content will be read from STDIN. The given content must be in the Terraform
language native syntax; JSON is not supported.
Options:
  -list=false    Don't list files whose formatting differs
```

(always disabled **if** using STDIN)
-write=false Don't write **to** source files
(always disabled **if** using STDIN **or** -check)
-diff Display diffs **of** formatting changes
-check Check **if** the input **is** formatted. **Exit** status will be 0 **if all** input **is** properly formatted **and** non-zero otherwise.
-no-color **If** specified, output won't contain any color.
-recursive Also **process** files **in** subdirectories. By **default**, only the given directory (**or** current directory) **is** processed.

- **terraform graph**: This command helps you to generate a visual representation of the Terraform configuration or execution plan. You can expect output in a DOT format. You can use Graphviz (http://www.graphviz.org/) to generate charts. This helps you to have a visual dependency graph of Terraform resources as per the defined configuration file in the **DIR** (or in the current directory). You can have the following subcommand flags with **terraform graph**:

```
$ terraform graph | dot -Tsvg > graph.svg
```

Here is an example graph output:



- **terraform output**: This command yields the output
- **terraform refresh**: This command helps you to update the Terraform state file, comparing it with the real-world infrastructure. It doesn't make changes to the infrastructure directly, but your infrastructure may expect changes when you run the **terraform plan** and **apply** commands after **terraform refresh**. This command also helps to figure out whether there is any kind of drift

from the last-known state and updates the state file accordingly. For a better understanding of this command, refer to the following link: https://www.terraform.io/docs/commands/refresh.html.

- **terraform show**: This command can help you see outputs on the CLI itself from the state file, and these will be in a human-readable format. If you are looking for specific information from your state file, you can use this command.
- **terraform taint**: Using this command, you can mark specific resources to get destroyed and recreated. Many times, if you encounter some sort of deployment error, then this command can be very useful. This command doesn't modify the infrastructure but it does perform changes to the state file, which means that if you are going to run **terraform apply**, it will destroy the specific resource that has been marked as tainted and recreate it. Let's see how we can taint a specific resource, as follows:

```
$ terraform taint aws_security_group.rdp_allow
```

The **aws_security_group.rdp_allow** resource has been marked as tainted, so this specific resource will get destroyed and recreated when we perform **terraform apply**. You can read more about the **terraform taint** command at https://www.terraform.io/docs/commands/taint.html.

- **terraform workspace**: Terraform supports a command named **workspace** that helps you to create multiple landscapes. Suppose you want to use the same Terraform configuration in multiple landscapes. In that case, you can use this **workspace** command. For example, if you have three environments such as *dev*, *test*, and *production*, then by using this **workspace** command, you can create these virtual environments for Terraform's reference, and Terraform will maintain the state file and plugins for the respective environment accordingly. For more information about **terraform workspace**, you can read https://www.terraform.io/docs/state/workspaces.html and https://www.terraform.io/docs/commands/workspace/index.html.
- **terraform force-unlock**: Suppose your state file got locked by some other user in the remote backend provided. In that case, you could use **terraform force-unlock LOCK_ID[DIR]**. This command would help you to unlock your state file. This is only possible when you have stored your state file in a remote backend; if you have kept the state file in your local machine, then it can't be unlocked by any other process. One more thing: this command doesn't make any changes to your environment
- **terraform validate**: If you have some sort of syntax error in your configuration file, then, by using the **terraform validate** command, you can get detailed information about the error. This command should not be run individually because when you run the **terraform plan** or **apply** commands, Terraform automatically runs the **terraform validate** command in the backend to check any sort of syntax error within your configuration code block.
- **terraform import**: If you want to import some already existing resources or services into your Terraform state file, you can do this by referring to the resource ID of that specific resource. Let's try to understand how you can import a resource in the state file, with the following example: **terraform import** aws_instance.abc i-acdf10234

UNDERSTANDING THE TERRAFORM LIFE CYCLE

As you have become familiar with how to use the Terraform CLI and run the respective Terraform commands, you must have also been wondering how Terraform creates or updates an infrastructure? Terraform follows a sequence of commands that are defined under the Terraform life cycle. Let's try to understand how the Terraform life cycle works with **terraform init**, **terraform plan**, **terraform apply**, and **terraform destroy**. A Terraform workflow starts with writing the Terraform code file, downloading all the providers and plugins, displaying in preview which actions Terraform is going to perform, and then—finally—whether you wish to deploy the resources that have been defined in the Terraform configuration code file. After creating or updating this infrastructure, let's suppose you wish to have these resources destroyed—how would you do this? All this can be clarified by following a Terraform workflow, which is depicted in the following diagram:

TERRAFORM INIT

terraform init is the first and foremost Terraform command that you generally run to initialize Terraform in the working directory. There could be several reasons to run the **terraform init** command, such as the following:

- When you have added a new module, provider, or provisioner
- When you have updated the required version of a module, provider, or provisioner
- When you want to change or migrate the configured backend

Terraform modules and these downloaded files get stored in the current working directory that you have provided. The **terraform init** command supports many subcommands or arguments.

Let's try to understand what exactly happens when we run the **terraform init** command with some simple example code. In this example, we are taking an **Azure Resource Manager** (**AzureRM**) provider. In the following code snippet, we have placed it in the **providers.tf** file:

TERRAFORM VALIDATE

If you have some Terraform syntax errors in the configuration code, then you must be thinking: *Does Terraform provide any way to get these errors checked?* The answer to this is yes—Terraform has an built-in command, **terraform validate**, which will let you know if there are any syntax errors in the Terraform configuration code in the specified directory.

You can run the **terraform validate** command explicitly, otherwise validation of the configuration file will be done implicitly during the execution of the **terraform plan** or **terraform apply** commands, so it is not mandatory to run this command. Terraform is knowledgeable enough to run validation during other Terraform workflows such as **terraform plan** or **terraform apply** workflows.

The only thing needed before Terraform performs validation of the configuration code is the **terraform init** command, which downloads all plugins and providers by default. Here is a quick example, to give a walkthrough of some possible syntax validation errors:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "2.55.0"
    }
  }
}
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "example" {
  name             = "Terraform-lab-RG"
  location         = "eastus"
  terraform_location = "eastus"
}
```

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

```
PS C:\provider> terraform validate
Error: Unsupported argument
  on resourcegroup.tf line 16, in resource "azurerm_resource_group" "example":
   16:   terraform_location = "eastus"
An argument named "terraform_location" is not expected here
```

Remove the **terraform_location** argument from the configuration code and then execute **terraform validate**. You can see the result in the following code snippet:

```
PS C:\provider> terraform validate
Success! The configuration is valid.
```

The **terraform validate** command does not check Terraform configuration file formatting (for example, tabs versus spaces, newlines, comments, and so on). For formatting, you can use the **terraform fmt** command.

TERRAFORM PLAN

After executing **terraform init**, you are supposed to run the **terraform plan** command, which would generate an execution plan. When the **terraform plan** command is being run, Terraform performs a refresh in the backend (unless you have explicitly disabled it), and Terraform then determines which

actions it needs to perform to meet the desired state you have defined in the configuration files. If there is no change to the configuration files, then **terraform plan** will let you know that it is not performing any change to the infrastructure. Some of the subcommands/arguments that we can run with **terraform plan** are listed here—a complete list of the subcommands/arguments supported by the **terraform plan** phase can be seen by running the **terraform plan -h** command:

- **-destroy**: If set, a plan will be generated to destroy all resources managed by the given configuration and state.
- **-input=true**: Asks for input for variables if not directly set.
- **-out=path**: Writes a plan file to the given path. This can be used as input to the **apply** command.
- **-state=statefile**: Provides a path to a Terraform state file to use to look up Terraform-managed resources. By default, it will use the **terraform.tfstate** state file if it exists.
- **-target=resource**: Provides a resource to target. The operation will be limited to this resource and its dependencies. This flag can be used multiple times.
- **-var 'foo=bar'**: Sets a variable in the Terraform configuration. This flag can be set multiple times.
- **-var-file=foo**: Sets variables in the Terraform configuration from a file. If a **terraform.tfvars** files or any **.auto.tfvars** files are present, they will be automatically loaded.

In order to understand the **terraform plan** command, here is a line of code that's been defined in the **resourcegroup.tf** file:

```
resource "azurerm_resource_group" "example" {
  name     = "Terraform-lab-RG"
  location = "east us"
}
```

When we run the **terraform plan** command, we can expect the following output:

```
PS C:\terraform> terraform plan
Acquiring state lock. This may take a few moments...
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.
------------------------------------------------------------------------
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # azurerm_resource_group.example will be created
  + resource "azurerm_resource_group" "example" {
      + id       = (known after apply)
      + location = "eastus"
      + name     = "Terraform-lab-RG"
```

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

Note: You didn't specify an "-out" parameter **to** save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.
Releasing **state** lock. This may take a few moments...

From the previously defined output from the **terraform plan** command, we got to know all the resources Terraform is going to create or update. In our example, it is showing that it is going to create a resource group with the name **Terraform-lab-RG**.

Let's try to understand how to store the **terraform plan** output into any file. In order to store the **terraform plan** output, we need to run the **terraform plan -out <filename>** command, as follows:

PS C:\provider> terraform plan -out plan.txt
Acquiring state lock. This may take a few moments...
Refreshing Terraform state in-memory prior **to** plan...
The refreshed state will be used **to** calculate this plan, but will **not** be persisted **to** local **or** remote state storage.

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

This plan was saved **to**: plan.txt
To perform exactly these actions, run the following command **to** apply:
terraform apply "plan.txt"
Releasing state lock. This may take a few moments...

The **terraform plan -out plan.txt** command has created a **plan.txt** file in the local present working directory and will have a binary file of the **terraform plan** output, which you can see here in the expanded directory:

PS C:\provider> ls
Directory: C:\provider
Mode LastWriteTime Length Name

```
----            ----------      ------ ----
d-----      09-12-2020  03:53          .terraform
-a----      09-12-2020  16:41        2040 plan.txt
-a----      09-12-2020  03:53         316 providers.tf
-a----      09-12-2020  16:21         105 resourcegroup.tf
```

TERRAFORM APPLY

After executing **terraform init** and **terraform plan**, if you find things are changing as per your expectations, you can then run **terraform apply** to help you provision or update the infrastructure. This will update the Terraform state file and will get it stored in the local or remote backend

TERRAFORM DESTROY

You might be wondering why we need to have **destroy** in the life cycle of the infrastructure. There could be cases where you want to get rid of the resources that you have provisioned using **terraform apply**, and in such cases you can run the **terraform destroy** command. This will delete all the resources or services you defined in the configuration file and update the state file accordingly. The **terraform destroy** command is a very powerful command, which is why when you execute it, it will present you with an execution plan for all resources it is going to delete and later ask for confirmation, because once the command gets executed it cannot be undone.

UNDERSTANDING TERRAFORM MODULES

So far, we have written Terraform configuration code, but have you ever thought about how effectively we can make it reusable? Just consider that you have many product teams with whom you are working, and they wish to use Terraform configuration code for their infrastructure deployment. Now, you may be wondering if it's possible to write code once and share it with these teams so that they don't need to write the same code again and again. This is where **Terraform modules** come in. Let's try to get an idea of what these are.

A Terraform module is basically a simple configuration file ending with **.tf** or **tf.json**, mainly consisting of resources, inputs, and outputs. There is a main root module that has the main configuration file you would be working with, which can consume multiple other modules. We can define the hierarchical structure of modules like this: the root module can ingest a child module, and that child module can invoke multiple other child modules. We can reference or read submodules in this way:

module.<rootmodulename>.**module**.<childmodulename>

However, it is recommended to keep a module tree flat and have only one level of child module. Let's try to understand about **module composition**, with an example of a **virtual private cloud** (**VPC**) and a **subnet** of AWS. Here is the resource code block:

```
resource "aws_vpc" "vpc" {
  cidr_block = var.cidr_block
}
resource "aws_subnet" "subnet" {
  vpc_id            = aws_vpc.vpc.id
  availability_zone = "us-east-1a"
  cidr_block        = cidrsubnet(aws_vpc.vpc.cidr_block, 4, 1)
}
variable "cidr_block" {
  default     = "10.0.0.0/16"
  type        = string
  description = "provide cidr block range"
}
```

When we try to write module code blocks, the configuration appears more hierarchical than flat. Each of the modules would have its own set of resources and child modules that may go deeper and create a complex tree of resource configurations.

Here is a code example, to explain how a module can be referenced in another module:

```
module "aws_network" {
  source     = "./modules/network"
  cidr_block = "10.0.0.0/8"
}
module "aws_instance" {
  source     = "./modules/instance"
  vpc_id     = module.aws_network.vpc_id
  subnet_ids = module.aws_network.subnet_ids
}
```

In our preceding example, you can see how we referenced one module in another module. It is always suggested to have a one-level module reference rather than referencing complex modules and submodules.

In simple words, we can define a module as a container with multiple resources that can be consumed together.

The syntax for defining Terraform modules, whereby we need to start with the module and provide its local name, is shown here:

```
module "name" {}
```

Terraform modules support some key arguments such as **source**, **version**, and **input variable**, and loops such as **for_each** and **count**. We can define a module code block in this way:

```
module "terraform-module" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.55.0"
}
```

n the previously defined code block, we have defined a module with a local name of **terraform-module**. We can give any name to a module—it's totally up to us, because it's just a local name. You may be wondering what these **source** and **version** arguments are doing inside the module code block. An explanation for this follows.

SOURCE

This argument is mandatory, referring to either the local path where the module configuration file is located or the remote reference **Uniform Resource Locator** (**URL**) from where it should be downloaded. The same source path can be defined in multiple modules or in the same file and can be called multiple times, making modules more efficient for reuse. The next code example will give you an idea of how a module can be defined.

For local referencing, you can define a module in the following way:

```
module "terraform-module" {
   source = "../terraform-module"
}
```

In the previously defined code block, we are referencing the local path where the module configuration file is located. We have two options for referencing a local path: **./** and **../**. We have used **../**, which takes the **terraform-module** directory (that is, the parent directory). In this case, when we run **terraform init** and **terraform plan**, the module file doesn't get downloaded as it already exists on the local disk. It is also recommended to use a local file path for closely related modules that are used for the factoring of repeated code. This referencing won't work if you are working in a team and they want to consume a different version of a module. In that case, a remote file path would help because the published version of the code would be there in the remote location, and anyone could easily reference it by providing the respective file URL.

The next aspect you need to know about is the different remote ways of referencing supported by Terraform that can be defined in the source block of the Terraform module. For in-depth learning about remote referencing, you can refer to https://www.terraform.io/docs/language/modules/sources.html. Here are a few ways in which you can reference remotely:

- Terraform Registry
- GitHub
- Bitbucket
- Generic Git; Mercurial repositories
- **HyperText Transfer Protocol** (**HTTP**) URLs

- **Simple Storage Service** (**S3**) buckets
- **Google Cloud Storage** (**GCS**) buckets

TERRAFORM REGISTRY

Terraform Registry has a list of published modules written by community members. This is a public registry, and it's very easy for others to start consuming it directly in their code. Terraform Registry supports versioning of published modules. It can be referenced using the specified syntax **<NAMESPACE>/<NAME>/<PROVIDER>**—for example, **hashicorp/consul/aws**, as illustrated in the following code snippet:

```
module "Terraform-consul" {
  source = "hashicorp/consul/aws"
  version = "0.8.0"
}
```

For a detailed understanding about usage of the **consul** module for AWS, you can read more at https://registry.terraform.io/modules/hashicorp/consul/aws/latest.

If you want to use modules that are hosted in a private registry similar to Terraform Cloud, then you can reference them in code by providing a source path with the syntax **<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>**. For a detailed understanding of private registry referencing, you can read more at https://www.terraform.io/docs/cloud/registry/using.html.

Have a look at the following code example:

```
module "aws-vpc" {
  source = "app.terraform.io/aws_vpc/vpc/aws"
  version = "1.0.0"
}
```

In this example, we have shown the **Software-as-a-Service** (**SaaS**) version of Terraform Cloud, which has a private registry hostname of **app.terraform.io**. If you are going to use any other private registry, you can replace this hostname with your private registry hostname.

GITHUB

Terraform will able to recognize github.com URLs and understand them as Git repository sources. The following code snippet shows cloning over **HTTP Secure** (**HTTPS**):

```
module "terraform-module" {
  source = "github.com/hashicorp/terraform-module"
}
```

To clone over **Secure Shell** (**SSH**), you can use the following code:

```
module "terraform-module" {
  source = "git@github.com:hashicorp/terraform-module.git"
}
```

GitHub supports a **ref** argument that helps select a specific version of a module. You would be required to pass credentials to get authenticated to a private repository.

GENERIC GIT REPOSITORY

You can define any valid Git repository by prefixing the address with **git::**. Both SSH and HTTPS can be defined in the following ways:

```
module "terraform-module" {
  source = "git::https://example.com/terraform-module.git"
}
module "terraform-module" {
  source = "git::ssh://username@example.com/terraform-module.git"

}
```

Terraform downloads modules from the Git repository by executing **git clone**. If you are trying to access a private repository, you would then be required to provide an SSH key or credentials so that it can be accessed.

If you are looking for a specific version of a module from the Git repository, then you can pass that version value into the **ref** argument, as follows:

```
module "terraform-module" {
  source = "git::https://example.com/terraform-module.git?ref=v1.2.0"
}
```

The Terraform modules support some meta-arguments, along with **source** and **version**. These are outlined as follows:

- **count**—You can define this argument within a module code block if you willing to create multiple instances from a single module block. For a greater understanding of the **count** expression. you can go to the **count** expression page of the Terraform documentation, at https://www.terraform.io/docs/configuration/meta-arguments/count.html.
- **for_each**—As with the **count** expression, the **for_each** expression is also used for creating multiple instances of a module from a single module block. If you want to read more about the **for_each** expression, you can visit https://www.terraform.io/docs/configuration/meta-arguments/for_each.html.
- **providers**—This is an optional meta-argument that you can define within a module code block. You may be wondering why we need to define a **providers** argument inside a module code block. Basically, you need to define this when you want to have multiple configurations

in a module of the same provider. You can then call them by placing **alias** for the provider in the following way:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
provider "aws" {
  region = "us-east-1"
}
# An alternate configuration is also defined for a different
# region, using the alias "terraform_aws_west".
provider "aws" {
  alias  = "terraform_aws_west"
  region = "us-west-1"
}
module "terraform-module" {
  source    = "./terraform-module"
  providers = {
    aws = aws.terraform_aws_west
  }
}
```


In this example, we defined the same provider multiple times, one with an alias and another without an alias. If we hadn't defined a **providers** arguments in the child module code block, it would have taken the default provider that we defined without an alias. So, in order to use a specific provider within the module, we defined **aws=aws.terraform_aws_west**. From this expression, you can easily understand that the **providers** argument inside the child module is a map that will be supporting only the key and the value, without placing them in double quotes.

- **depends_on**—This meta-argument is an optional one, whereby if we need to define an explicit dependency then we can define a **depends_on** argument within a module code block. This should be used only as a very last resort because Terraform is smart enough to follow an implicit dependency of resources, but in some cases we need to forcefully tell Terraform that there is a dependency so that Terraform can follow that sequence for the creation or deletion of resources. Let's try to understand how we can define the **depends_on** argument within a module, as follows:

```
resource "aws_iam_role" "terraform-role" {
  name = "terraform-vpc-role"
```

```
}
# We need to create AWS IAM role specific to VPC then we are supposed to create
module "terraform-module" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~>2.5.0"
  depends_on = [
    aws_iam_role.terraform-role,
    ]
}
```

- In this example, we mentioned explicit dependency in the module by defining **depends_on**, which means that Terraform will deploy the **terraform-module** module only when we already have the AWS **Identity and Access Management** (**IAM**) role in place. Similar to this example, we can even define explicit dependency of one module to other child modules.
- There are many more aspects related to Terraform modules and we will try to discuss them in our upcoming sections, providing examples so that you will gain a better understanding. For now, let's try to understand how we can taint a resource within a module. To taint a specific resource within a module, we can use Terraform's **taint** command, as illustrated in the following code snippet:

```
$ terraform taint module.terraform_module.aws_instance.terraform_instance
```

This will tell Terraform to destroy and recreate a specific resource of modules that's been tainted during the next **apply** operation. Remember—you won't able to taint a complete module; only specific resources within a module can be tainted.

TERRAFORM GLOSSARY

In this glossary, we are going to list all the acronyms and key terms used throughout this book. This will help to clarify all the technical terms. These terms should assist in helping you to get to grips with the different conversations currently taking place in the Terraform community:

- **API**: An application programming interface. This is an interface that has been designed to manipulate some functionality of an application. Terraform uses cloud or on-premises API software to manage infrastructure. Terraform resources are responsible for mapping to each API call that is required to create, update, or delete the infrastructure.

Terraform Cloud also has APIs that can be used to easily manage policies and workspaces.

- **Apply**: This is one of the phases in the Terraform life cycle that helps Terraform to perform implementation by accepting planned changes, or to make real changes to the Terraform providers using their resource APIs.
- **Azure**: This is a public cloud provided by Microsoft.
- **AzureRM**: This is one of the Terraform providers specific to Azure.

- **AWS**: This is a public cloud provided by Amazon.
- **Arguments**: When we write a Terraform configuration file, arguments take the form **<IDENTIFIER> = <EXPRESSION>** and are defined by a block of resources and data sources. Generally, they hold the properties of the Terraform providers.
- **Attribute**: An attribute is a property of an object that can be exported. For example, we can get an attribute value in a defined way: **aws_instance.eg.id**.
- **Backend**: A backend is required to store the Terraform state files, so it could be remote or local. The choice of backend determines how the state is stored and where Terraform will execute.
- **Block**: The HashiCorp configuration syntax used to create a container that holds all the defined objects like a resource. A block contains one or more labels and a body containing name and value pairs:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
```

- **CI/CD**: Continuous Integration/Continuous Delivery. CI/CD involves operating principles and practices that enable a reliable sequence in terms of the development and deployment of cloud resources, typically an application. This helps as regards complete automation. In our course, we used Azure DevOps Pipeline to explain CI/CD concepts using Infrastructure as Code using Terraform.
- **CLI**: The command-line interface. We can run Terraform commands on any terminal, such as a Unix shell or the Windows CLI. The Terraform CLI is an open source binary. In other words, **terraform.exe** can be downloaded and placed in local system environment variables and Terraform commands can start to be used.
- **Clone**: Cloning means copying files to the local system so that it will be easy for us to make changes and push them back to the repository. It is basically a **git clone** command that helps to perform cloning from the source control repository to the local system.
- **Configuration language**: Terraform files used to have code written in declarative format describing the desired state of the infrastructure. It is mainly referred to as **HashiCorp Configuration Language** (**HCL**).
- **Data source**: Like a Terraform resource code block, we have a data source that generally helps you to fetch information regarding the already existing infrastructure that has been deployed by some other means, for example, manually, or using any other Terraform configuration code. In places, you may also find it referred to as a data resource.
- **Fork**: This is like a clone where one copy of the repository gets cloned into the VCS server. It might be the case that you are copying content and history from one repository to another. For example, suppose you need to work on the Microsoft repository. What you can do is that you can clone it to your local VCS so that it will be easy for you to work on it. You would be getting the full option of copying content to your local VCS with all its branches. The main goal of a fork is to copy a source repository to another repository.
- **Git**: A distributed version control system that will hold all the change history that has been performed on any file or folder defined in the source code. This is beneficial when you are working with many developers, as they would easily be able to coordinate with one another.

- **HCL**: HashiCorp Configuration Language. This is a structured configuration syntax that is used to write Terraform code. It uses specific block types, such as a resource, variable, provider, and built-in function. It is written as a named value pair.
- **IaC**: Infrastructure as Code is a way of writing infrastructure in code format and keeping it in a file so that it will be easy to manage infrastructure, for example, Terraform configuration files.
- **Input variables**: This is a method of declaring a variable in Terraform and taking a value from the user.
- **Locking**: A Terraform state file is used to go into a locked state so that it can avoid another Terraform process, such as **apply**, when a Terraform operation is already running.
- **Log**: This is a text-based output that gets printed to **stderr** following the execution of any Terraform operations, such as **plan** and **apply**.
- **Module**: A Terraform module is more like a code function. You have input and a module can provide output, but all processes are internal to the module. It is like a self-defined container where it has all the Terraform configuration code that can be used to manage the infrastructure. Other Terraform configurations can call the module that tells Terraform to manage a resource that is defined in the module configuration code.
- **Output values**: Suppose we want to see what Terraform operation had been performed, irrespective of whether a defined resource was created. We can see them during the runtime itself by defining them in the output file as an output value.
- **Plan**: This is one of the Terraform workflow operations where Terraform compares the infrastructure's real state to the configuration, and is shown to the user in a readable format, about the changes it is going to perform to match the desired state.
- **Policy**: This is basically a rule that is enforced by Terraform to validate the plan, irrespective of whether the resources comply with company policy.
- **Policy set**: A list of policies defined to be enforced globally or in relation to a specific workspace.
- **Provider**: This is a plugin for Terraform that holds collections of the available resources. There are many Terraform providers, including AWS, Azure, and GCP. Terraform has already published a list of available providers that can be found at https://www.terraform.io/docs/providers/index.html.
- **Remote backend**: Terraform state files can be stored in defined storage, such as S3 and Google Cloud Storage. If we want to store the state file securely, then it's desirable to configure the remote backend.
- **Repository**: This is a place where code files can be kept and managed. Primarily, this will be any version control system that can hold the entire history of changes to the file. A repository is mainly a Git repository that will retain all Terraform configuration files except secrets.
- **Resource**: In the Terraform configuration file, we defined a resource code block that describes one or more infrastructure objects. A resource block instructs Terraform to manage the defined resource. Terraform uses cloud provider APIs to create, edit, and destroy resources.
- **State**: Terraform generates a state file in a JSON format where it holds all the information about the defined infrastructure and maps those resources to the real world. Without state, Terraform can't know which resources got provisioned during the previous run. So, it is very important to know what action has been performed during the previous run if you are working with many people. That is why Terraform generates state files and records all the defined infrastructure resources in the state file.

- **VCS**: A version control system, such as Git. This is a software application that tracks changes to collections of files and helps you to monitor changes, undo changes, or combine them.
- **Workspace**: In the Terraform CLI, a workspace is an isolated instance that can be used to deploy multiple environments using a single Terraform configuration file.