**Terraform**

IAC (Infrastructure as a code): IAC is a widespread terminology among DevOps professionals.

It is the process of managing and provisioning the complete IT infrastructure (comprising both physical and virtual machines) using machine-readable definition files.

It is a software engineering approach toward operation.

It helps in automating the complete data entry by using programming scripts.

Infrastructure as a code has multiple challenges

Need to learn to code.

Don’t know the change impact

Need to revert the change.

Cannot track changes.

Cannot automate a resource

Multiple environments for infrastructure.

Terraform has been created to solve these challenges.

What is terraform?

Terraform is an open-source infrastructure as a code software tool developed by HashiCorp. It is used to define and provision the complete infrastructure using easy-to-learn declarative languages.

It is an infrastructure provisioning tool where you can store your cloud infrastructure set up as code. It is very similar to tools such as cloud formation, which would use to automate your aws infrastructure, but you can only use that on AWS. With Terraform you can use it on another cloud platform as well.

Benefits of using terraform:

* + - 1. Does orchestration, not just configuration management.
      2. Supports multiple providers such as AWS, Azure, GCP, digital motions and etc.
      3. Provides immutable infrastructure where configuration changes smoothly
      4. Uses easy-to-understand language HCL (HashiCorp configuration language)
      5. Easily portable to any other provider.
      6. Supports client-only architecture. So, no need of additional configuration management on a server.

**Terraform core concepts:**

Below are the core concepts/terminologies used in Terraform:

**Variables:** Also used as input variables, it is key-value pair used by Terraform modules to allow customization.

**Provider:** It is a plugin to interact with APIs of service and access its related resources.

**Module:** It is a folder with Terraform templates where all the configurations are defined

**State:** It consists of cached information about the infrastructure managed by Terraform and the related configurations.

**Resources:** It refers to a block of one or more infrastructure objects (compute instances, virtual networks, etc.), which are used in configuring and managing the infrastructure.

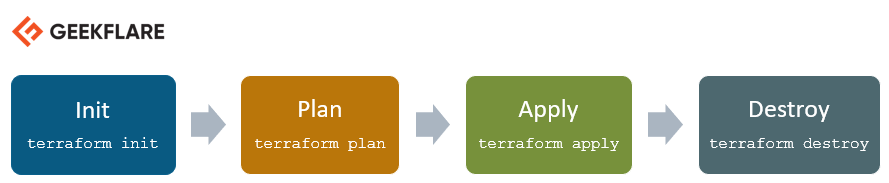
**Data Source:** It is implemented by providers to return information on external objects to terraform.

**Output Values:** These are return values of a terraform module that can be used by other configurations.

**Plan:** It is one of the stages where it determines what needs to be created, updated, or destroyed to move from the real/current state of the infrastructure to the desired state.

**Apply:** It is one of the stages where it applies the changes real/current state of the infrastructure to move to the desired state.

**Terraform Life Cycle:**

****

**Terraform init** initializes the working directory which consists of all the configuration files.

**Terraform Plan** is used to create and execute the plan to reach the desired state of the infrastructure. changes in the configuration files are done to achieve the desired state.

**Terraform Apply** then makes the changes in the infrastructure as defined in the plan, and the infrastructure comes to the desired state.

**Terraform destroy** is used to delete all the old infrastructure resources which are marked tainted after the application face.

**What is Terraform Used For?**

Terraform can relate to different infrastructure hosts and achieve **complex management scenarios and compliance across multiple clouds**. Its configuration can be easily packaged, shared, and reused in the form of [Terraform modules](https://spacelift.io/blog/what-are-terraform-modules-and-how-do-they-work).

In this post, we will go over some basic elements and features of Terraform that can help you get started. Before we move forward, however, let’s discuss Infrastructure as Code (IaC) a bit more to develop a good understanding of Terraform.

**Infrastructure as Code (IaC)**

Traditionally, the process of preparing a Windows or Linux server for your lab or data center consisted of several steps involving mouse clicks, running scripts, and manually configuring different aspects such as storage and networking. Doing this at scale for hundreds or thousands of servers becomes increasingly **complex and error-prone**.

[IaC](https://www.redhat.com/cms/managed-files/pa-terraform-and-ansible-overview-f14774wg-201811-en.pdf) makes this process much easier by allowing you to define and provision your infrastructure with the help of **declarative language**. With this approach, you can gain consistency, repeatability, and enhanced reliability.

In addition to day one infrastructure setup, this approach also helps with the management of any changes that happen to your infrastructure along the way, commonly referred to as drift management. (If you’re interested in how to manage Terraform workflows at scale, see: [How to Manage Terraform at Scale](https://spacelift.io/blog/5-ways-to-manage-terraform-at-scale))

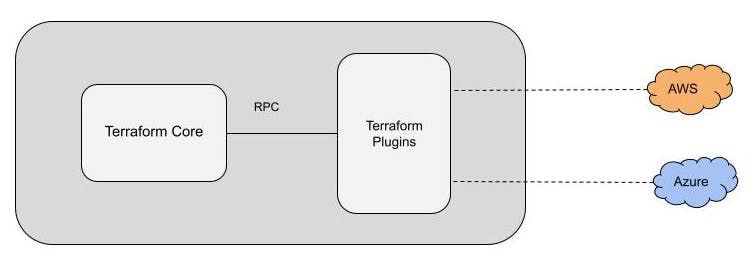
Other common IaC tools include Cloud Formation Templates (CFT) for AWS, Azure Resource Manager (ARM) templates for Microsoft Azure, and Deployment Manager for Google Cloud.

**How Terraform Works?**

At a high level, terraform can be thought of as consisting of **two main parts**: Terraform Core and Plugins. Core is responsible for life cycle management of infrastructure. It is the open-source binary that you download and use from the command line.

**Terraform Core** takes into consideration the current state and evaluates it against your desired configuration. It then proposes a plan to add or remove infrastructure components as needed. Next, it takes care of provisioning or decommissioning any resources if you choose to apply the plan.

**Terraform Plugins** provide a mechanism for Terraform Core to communicate with your infrastructure host or SaaS providers. Terraform Providers and Provisioners are examples of plugins as mentioned above. Terraform Core communicates with the plugins via Remote Procedure Call (RPC).



**Terraform Workflow**

A typical Terraform workflow will consist of the following steps:

**Step 1. Write**

Declare your infrastructure resources as code in Hashicorp Configuration Language (HCL).

**Step 2. Review the Plan**

Terraform will display a plan to add or remove resources based on the comparison of your declared infrastructure and the current state of any existing resources.

**Step 3. Apply**

Accept the planned changes to add or remove any infrastructure resources. This workflow works well for a single author as Terraform State is stored locally. As more contributors join the team, it becomes important to ensure everyone has a **correct view of the current infrastructure state** and that changes are applied sequentially without overlaps.

Terraform offers Enterprise and Cloud versions for **teams to work together**. These store the state remotely so everyone can have a consistent view. Terraform Cloud is available as a hosted service, whereas Terraform Enterprise enables you to deploy and self-host the Terraform Cloud in your private instance. Authors contribute their code to a central version control system (e.g. Github) and Terraform Cloud uses the latest code versions from the central repository.

**Core Elements of Terraform**

**Terraform CLI**

Terraform is an [open-source](https://github.com/hashicorp/terraform) tool that compiles to a single binary file that can be downloaded on your system and used interactively from the command prompt. The latest version of Terraform can be downloaded [here](https://www.terraform.io/downloads.html). A list of available Terraform commands can be retrieved from the command line.

~ terraform --help

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

The available commands for execution are listed below.

The primary workflow commands are given first, followed by

less common or more advanced commands.

Main commands:

init Prepare your working directory for other commands

validate Check whether the configuration is valid

plan Show changes required by the current configuration

apply Create or update infrastructure

destroy Destroy previously-created infrastructure

All other commands:

console Try Terraform expressions at an interactive command prompt

fmt Reformat your configuration in the standard style

force-unlock Release a stuck lock on the current workspace

get Install or upgrade remote Terraform modules

graph Generate a Graphviz graph of the steps in an operation

import Associate existing infrastructure with a Terraform resource

login Obtain and save credentials for a remote host

logout Remove locally-stored credentials for a remote host

output Show output values from your root module

providers Show the providers required for this configuration

refresh Update the state to match remote systems

show Show the current state or a saved plan

state Advanced state management

taint Mark a resource instance as not fully functional

test Experimental support for module integration testing

untaint Remove the 'tainted' state from a resource instance

version Show the current Terraform version

workspace Workspace management

Global options (use these before the subcommand, if any):

-chdir=DIR Switch to a different working directory before executing the

given subcommand.

-help Show this help output, or the help for a specified subcommand.

-version An alias for the "version" subcommand.

**Terraform Language**

Infrastructure is defined in the [Hashicorp Configuration Language (HCL)](https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/products/nsx/vmware-nsx-t-data-center-and-terraform.pdf) when working with Terraform. HCL is a **declarative language** targeted towards DevOps tools and servers. It provides a nice balance between machine-readable formats such as JSON and configuration formats included in high-level languages such as Ruby.

Infrastructure components managed by Terraform are called Resources. **Examples of resources** include virtual machines, database tables, AWS S3 buckets, VPCs, and more. Each resource block in HCL helps define the resource and configure its properties. An example VPC resource looks like this:

resource "aws\_vpc" "default\_vpc" {

cidr\_block = "172.31.0.0/16"

tags = {

Name = "example\_vpc"

} }

**Terraform Providers**

Terraform needs to connect to your infrastructure host to effectively manage your infrastructure. Infrastructure hosts generally have a published API that facilitates operations to create, update and delete infrastructure elements. [Terraform Providers](https://spacelift.io/blog/terraform-providers) supply an **abstraction to these APIs**.

Terraform configuration must include providers for it to connect with required cloud or SaaS services. Providers are published by infrastructure hosts and made available to Terraform as plugins. The plugin-based architecture allows publishers to independently publish and maintain their providers. The [Terraform registry](https://registry.terraform.io/browse/providers) is the main directory of publicly available providers (you can also find the [Spacelift Terraform Provider](https://registry.terraform.io/providers/spacelift-io/spacelift/latest/docs) in this registry).

Here is an example of configuring AWS Terraform Provider in HCL:

provider "aws" {

version = "~> 3.0"

region = "us-east-1"

}

**Terraform Provisioners**

[Provisioners](https://spacelift.io/blog/terraform-provisioners) are another **type of plugin**available to use with Terraform as a means of configuring the infrastructure and making it ready for service. For example, you might need to copy certain files to your virtual machine necessary for your application to function, or you might need to run configuration management software (e.g. Chef or Puppet, etc.) that you normally use to configure your environment.

Provisioners add complexity to overall usage of Terraform. Their behavior cannot be easily modeled with Terraform for planning as they can take a wide range of actions and might access your critical infrastructure services with high privileges. Hashicorp recommends using Provisioners as a **last resort** for your needs, only after searching for options available with other Terraform constructs.

**Terraform Modules**

Modules provide a way to **package resources** that provide relevant functionality together as a group. Each Terraform configuration has at least one module called the root module and can call other modules referred to as child modules.

It is easy to write modules and share them with your team or others. The [Terraform registry](https://registry.terraform.io/browse/modules) hosts a large collection of Terraform modules for commonly used configuration scenarios. Terraform can automatically download these modules if you include them in your configuration.

**Terraform State**

Terraform stores a representation of your infrastructure objects, their interdependencies and bindings in a configuration file named *terraform.tfstate*. Storing the state information helps Terraform immensely in making planning and execution decisions. State can be stored locally as well as remotely in a team environment.

Read more about the [elements of Terraform architecture](https://spacelift.io/blog/terraform-architecture).

**Example Use Cases**

Let’s take a look at some common scenarios where Terraform can be used:

**Example 1. Rapidly Build and Dispose Environments**

It is common practice to have multiple environments in a typical IT organization such as Development, Testing, Staging, and Production. Terraform allows us to **save all the infrastructure resources and their interdependencies** specific to an environment as code in Hashicorp Configuration Language (HCL). We can use this code to quickly create a new environment as needed.

For example, when we need to test a new feature and want to make sure our Testing environment is an exact copy of Production, we can quickly create a Test environment with the help of code that is used to create the Production, ensuring our changes will run smoothly when moved to Production after testing.

Similarly, when developing a new feature, we might need to create a new development environment for a short period of time. We can easily create and tear down one with the help of Terraform automation. The same concept applies to building recovery environments in case a disaster strikes and makes Production unavailable.

**Example 2. Use Multiple Clouds**

Using more than one public cloud provider can be very beneficial if we want to have some fault tolerance for our application. In case cloud services for a particular region fail for a cloud provider, the application can continue to work if it is configured to use multiple cloud providers. It is also common to keep some resources on-prem and use a hybrid environment, combining private and public cloud resources.

Terraform is cloud-agnostic. It can **deploy applications across multiple clouds** quickly and consistently. With Terraform, we don’t have to worry about using deployment tools specific to cloud providers. Instead, we can have a uniform experience working with different clouds.

**Example 3. Deploy N-Tier Applications**

N-Tier applications provide separation of concerns by having different tiers targeted to specific purposes. For example, a typical web application will have a tier dedicated to databases, and another tier for web servers. The web servers will depend on databases to function properly and will be behind a load balancer that will distribute requests among these web servers.

Terraform can help ensure dependencies in this environment are implemented correctly, as databases will have to be deployed and be available before web servers. It is also easy to scale multi-tiered applications by adding more resources to a specific tier.

For example, if you feel your application can be more robust with the help of an additional database server, it can be a matter of a simple configuration change and deployment with Terraform to add another server to the database tier.

**Example 4. Work with Popular PaaS tools**

Infrastructure provisioning is an important part of application deployment. However, it is not the only one required for successful deployment of applications. Modern applications use several tools to accomplish critical functions such as scheduling of resources, caching, and routing, etc. **Kubernetes** is a popular orchestration platform used to productively deploy containers across multiple hardware resources.

Terraform Providers not only work as abstractions for infrastructure hosts, but they can essentially interact with any API based platform such as Kubernetes. This allows us to use Terraform as a single tool to deploy infrastructure and then configure any required platforms like Kubernetes.

**Terraform vs Other Infrastructure Tools**

Terraform solves several important infrastructure problems and naturally has some overlapping features with other infrastructure tools. Let’s look at some frequently asked questions about other tools in relation to Terraform.

**Should I use Terraform or other popular configuration management tools like Chef or Puppet?**

* There is a basic difference between Terraform and configuration management tools. Terraform focuses on making it easy to provision and manage your infrastructure, whereas configuration management software is used to prepare the VMs and Servers to fit your application and environment needs after provisioning. Furthermore, terraform can be used in conjunction with configuration management tools as they address separate concerns.

**What about cloud specific tools like Cloud Formation and ARM Templates?**

* While cloud specific tools can be excellent at managing resources for a particular cloud provider, terraform strives to be cloud agnostic. It allows you to use the same language to define your resources across different clouds and makes it easy to work with multi-cloud and hybrid cloud scenarios.

**What about common orchestration tools such as Kubernetes?**

* Similarly, to configuration management tools, container orchestration platforms focus on a separate concern and can be easily used in collaboration with Terraform. They usually have a robust API that helps develop Terraform Providers that in turn can be used effectively with Terraform Core, for example. [Kubernetes Terraform Provider](https://registry.terraform.io/providers/hashicorp/kubernetes/latest) can be found in the Terraform Provider registry.

**Why use Terraform instead of our home-grown scripts?**

* There are several reasons for moving away from custom ad-hoc scripts. A common issue encountered is their limitation to scale and adapt to new infrastructure needs. A custom script typically addresses a specific environment and needs updating and refactoring as new applications and infrastructure are added to the IT organization. This creates an overhead in terms of time, effort, and budget to achieve business-critical goals.