Terraform: Automated deployment and management of cloud resources

Table of Contents

Introduction

Infrastructure as Code

Before the advent of cloud computing, most infrastructure environments were built manually. All of the creation and maintenance tasks were done by infrastructure engineers. The infrastructure would generally not change, there was not much need to scale the infrastructure, and neither were there disruptive changes to it apart from failing hardware. This manual maintenance would work fine in most cases.

Things have changed rapidly over the years, especially after the cloud gained significant traction. The monolithic application architecture has changed and now the applications are getting broken into smaller microservices, with each having its development and deployment lifecycle, and each having its own infrastructure requirements in terms of size, performance, scalability, disaster recovery, and availability.

With this increase in size, complexity, and number of infrastructure deployments, it was becoming increasingly difficult to continue with the traditional manual ways to administer, manage, and configure infrastructure. Moreover, the manual steps to create the infrastructure environment led to delays due to lower consistency, predictability, and standardization.

All this led to the emergence of a new paradigm called infrastructure as code (IaC). IaC is a process of converting infrastructure to code and then treating that code similar to application code and taking it through the same lifecycle, steps, and process that an application code would undergo. This includes authoring infrastructure code, version control, different levels and types of testing (unit, integration, acceptance), linting, etc.

To convert infrastructure into code, we can write bash or PowerShell scripts, which is imperative for managing an environment. Here we have to explicitly specify instructions about provisioning, deleting, or updating an environment and also have to provide configuration data associated with each environment. Or we can use tools like Terraform which allow us to write Declarative code, where we just need to provide the configuration data while the how part (like instructions about provisioning, deleting, or updating an environment) is being managed by the tool.

Benefits of IaC

Faster Release Cycles and High-Quality Delivery: Since we use code for the creation of infrastructure components, it leads to consistent and predictable deployments, reducing human error and time spent in fixing those errors.

More Institutional knowledge: Removing manual deployments, executing scripts, and using tools like Terraform help in bringing upon higher reliance on process rather than individual skill and availability.

Predictability and Idempotency: Given a set of inputs, the outputs generated will always remain the same (Predictability) and the outputs will not change unless there are changes in the inputs (Idempotency).

Better Collaboration between Dev and Operations: While the initial configuration is authored by developers, it is eventually used by operations to update and manage the resources. Hence authoring an IaC configuration is a joint responsibility of both developers and operations.

Compliance with Organization Policies: It is possible to lay down policies that should be adhered to while provisioning new or updating existing resources.

Enables Effective DevOps: Infrastructure-related code goes through the same engineering process as an application, the DevOps continuous integration pipelines are being used for code reviews, linting, and testing. Similarly, continuous delivery and deployment pipelines are used for validating the code by deploying it early in a sandbox environment. These practices ensure that the infrastructure code is validated and verified before deployment to production.

Terraform

Hashicorp's Terraform is one of the leading open-source, cross-platform, multi-cloud, hybrid IaC tools available to provision and manage IT and development resources on any cloud. It was built using the Golang language and also provides its own declarative scripting language known as the Hashicorp Configuration Language (HCL). Terraform provides a CLI based executable with the help of which all Terraform commands can be executed. It provides detailed logs and auditing features and using the locking feature, we can also use terraform in a multiteam environment.

Every HCL configuration has two main elements: the configuration script and the configuration values. The script is generally considered static and does not change from one environment to another, whereas the configuration values are environment specific. This feature helps developers write generic Terraform scripts that can be used in deployment pipelines to provision multiple environments each with unique configuration values.

Terraform architecture

Each cloud platform provides multiple APIs, also known as upstream APIs for managing its resources. We can consume these APIs directly using request-response objects however, a better approach is to write a library that encapsulates the low-level HTTP request-response communication and exposes a well-defined client that can be used by anyone to talk to the upstream APIs. Terraform is based on extensible architecture and uses a plugin model. Terraform supports two types of plugins.

- Providers
- Provisioners

Now the client library that encapsulates the low-level HTTP request-response communication becomes a provider library that is used to consume its remote upstream APIs. The client libraries

are not used by Terraform core directly, instead, we make remote procedure calls to providers which in turn use its

associated client library. This architecture enables us to work with multiple providers at the same time and use multiple client libraries.
The Terraform core comprises the important component implementation like Terraform CLI, its commands, state management, backend, and workspace management. The core itself does not contain any cloud resources.

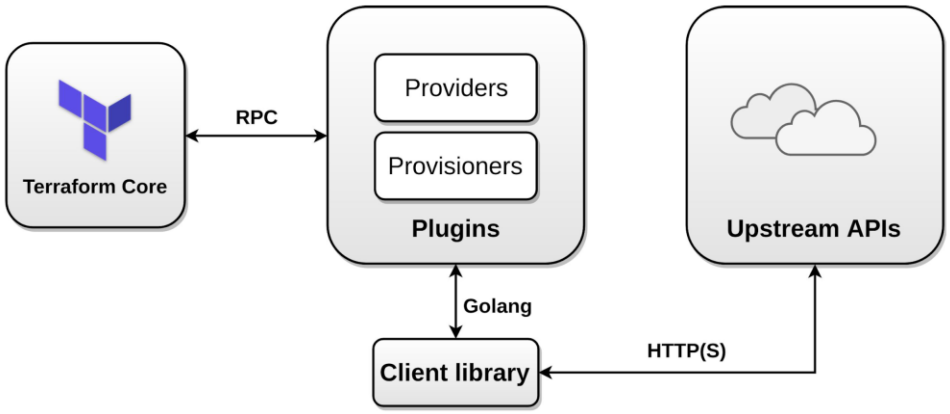

Fig 1) Generic Terraform architecture

The Azure cloud platform already has made its APIs publicly available for almost all its resources. There is a provider called terraform-provider-azurerm available along with its associated client library to enable Terraform to work with Azure resources and manage them.

State File

A state file is a Terraform owned, text-based (JSON), dynamically generated file containing information about the resource configuration. The state file is a replica of the current state of resources in the cloud platform, basically, it is Terraform's view of the remote real-world infrastructure. Terraform compares the configuration in an HCL script with the state file to determine whether it should create a new resource, update an existing resource, or delete a resource.

Terraform configuration script

Terraform provides the Hashicorp Configuration Language (HCL) to write Terraform configurations. Configuration scripts have the following generic structure:

```
terraform {
   #This block is used to configure terraform behavior.

   #required_version mentions Terraform's binary version. All providers
   #and modules should be able to work with the version provided in this
   #block.
   required_version = "v1.1.6"

   #required_providers specify the providers used within the configuration
   #along with their versions. It is responsible for setting up a local
   #name for the provider.
   required_providers {
       azurerm = {
       source  = "hashicorp/azurerm"
       version = "~>2.90"
       }
   }

   #This block is also used to configure the backend for the configuration
   #and for authentication when using remote state
}
```

```
provider "azurerm" {
        /*
    To manage Azure resources using terraform we need to get terraform
    authenticated and authorized to azure so that it can perform actions
    on our behalf.
    There are multiple ways to get terraform authenticated to azure like
    using azure cli with user credentials, or using service principal and
    client secret, or using service principal and client certificate, or
    using managed identities.
    So this block is used to configure those credentials.
    */
    features {
        /*
        The Azure Provider allows the behavior of certain resources to be
        configured using the features block.
        This allows different users to select the behavior they require,
        for example, some users may wish for the OS Disks for a Virtual
        Machine to be removed automatically when the Virtual Machine is
        destroyed whereas other users may wish for these OS Disks to be
        detached but not deleted.
        */
    }
}
variable "variableName" {
    /*
    Variables are placeholders in the configuration that accept
```

```
   external values as parameters during execution time. Variables
   help in authoring generic configurations and help in avoiding
   hard-coding within the script
   */
   type        = variableType
   description = "short description about variable"
   sensitive   = set this to true for sensitive variables like password,
                 this hides the value of the variable at console
}
resource "resourceType" "resourceIdentifier" {
   /*This block describes one or more infrastructure objects, such as
   virtual networks, compute instances, containers, etc. It contains
   the configuration of these objects*/
}
data "resourceType" "resourceIdentifier" {
   /*Data sources allow Terraform to use the information defined outside of
   Terraform, defined by another separate Terraform configuration, or
   modified by functions. A data block requests that Terraform read from
   a given data source ("resourceType") and export the result under the
   given local name ("resourceIdentifier"). The name and the data source
   act as an identifier in the Terraform module.
   Data sources are different from resources as resources(also known as
   managed resources) can cause Terraform to create, update, and delete
   infrastructure objects, while data resources cause Terraform only to
   read objects.
   */
}
```

```
module "moduleName" {

   /*

   Modules are containers for multiple resources that are used together.

   A module consists of a collection of .tf and/or .tf.JSON files kept

   together in a directory. It accepts values for variables and generates

   output.

   */

}

output "name" {

   #this blocks help generate and return output values after evaluation

   #of the expressions associated with them

}
```

We can see that HCL configuration scripts are flat in nature, and there is no nesting of blocks in them.

Applying a Terraform configuration to Azure

The general workflow to apply a Terraform configuration is as follows:

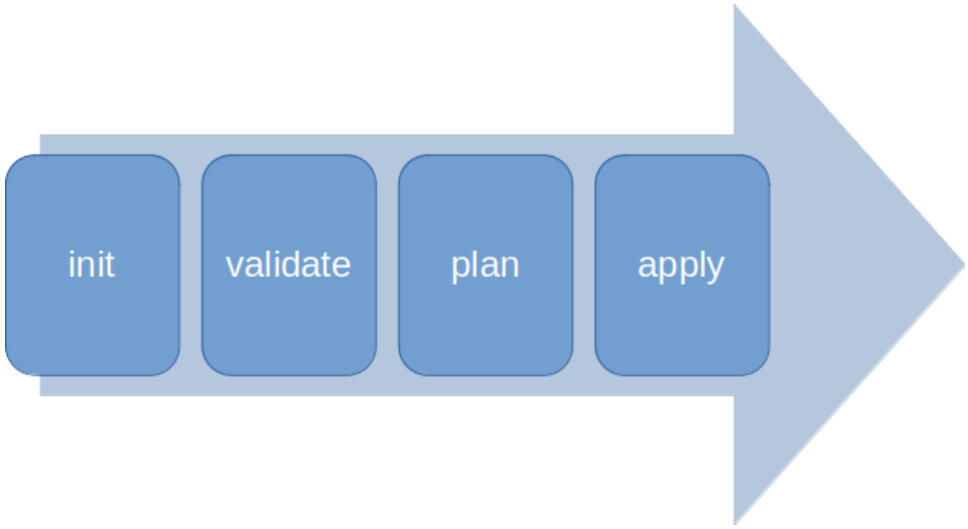

Fig 2) Terraform activities in a workflow

1. Initialize the Terraform working environment in a folder containing the configuration script.

   "init" is the first command that should be executed after authoring the Terraform script. init is the short form for "initialization," which means this command is used for initializing the Terraform context and environment. This command creates a backend that is used to manage context and

   workspace. init ensures that Terraform downloads all the dependencies with valid versions in the local working directory, initializes the state file, and helps in the management of the environment by updating and upgrading the modules and providers.

2. Validate the configuration files for syntactical validity and consistency.

   The command "terraform validate" helps in validating configuration files in a directory. It validates and checks whether the configuration files within a working directory are syntactically correct, well-formed, and valid.

3. Generate the execution plan to check what will change in the target environment.

   The purpose of the plan command is to generate an execution plan that shows the drift between the configuration in the configuration files and the current state file. For this, it first reconciles the state

   file with the configuration of resources in the remote cloud infrastructure and then further compares the state file with the configurations in the working directory. It provides feedback

about whether a resource will be provisioned, deleted, replaced, updated in place, or have no changes at all. It does

not make any actual changes to the remote cloud infrastructure, the state file, or the configuration files, instead, it is a way to evaluate the changes that might occur as a result of the actual application of configuration files to the target cloud infrastructure.

4. Apply the configuration to the target environment.

   To apply the configuration to the target environment we use the "apply" command. It internally executes the "validate" and "plan" command before applying the configuration. It then generates an execution plan, asks for user confirmation and then applies the configuration to the remote cloud infrastructure, and updates the state file simultaneously.

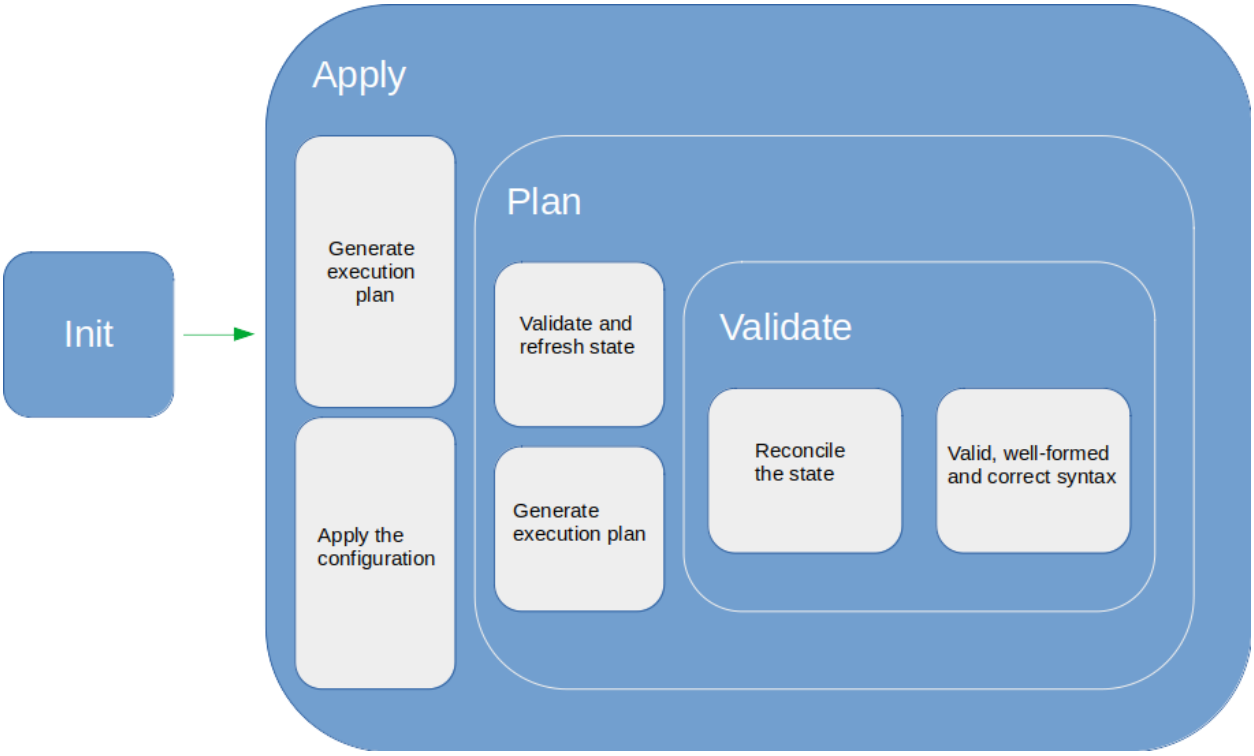

Fig 3) Terraform commands and their relationships

Terraform Datatypes

Terraform variables helps in writing generic scripts and help to customize the scripts without changing the code. There are 4 types of data types available for variables in terraform:

1. Primitive types: It is a simple type that isn't made from any other types. The available primitive types are:
    i. string: a sequence of Unicode characters representing some text, such as "hello".
    ii. number: a numeric value. The number type can represent both whole numbers like 15 and fractional values such as 6.283185.
    iii. bool: either true or false. bool values can be used in conditional logic.

2. Collection types: It allows multiple values of one type to be grouped as a single value. The three kinds of collection types in the Terraform language are:
    i. list(<type>): a sequence of values identified by an index starting with zero.
    ii. map(<type>): a collection of values where each value is identified by a string label.
    iii. set(<type>): a collection of unique values that do not have any secondary identifiers or ordering.

3. Structural types: A *structural* type allows multiple values of *several distinct types* to be grouped as a single value. Structural types require a *schema* as an argument, to specify which types are allowed for which elements. The two kinds of structural types in the Terraform language are:
    i. object({ <key> = <type>, <key> = <type>,..... }): a collection of named attributes with each attribute having their own type.
    ii. tuple([<type>, <type>,.....]): a sequence of elements identified by an index starting with zero, where each element has its own type.
4. Dynamic types: The keyword “any” is a special construct that serves as a placeholder for a type yet to be decided. “any” is not itself a type: when interpreting a value against a type constraint containing “any”, Terraform will attempt to find a single actual type that could replace the “any” keyword to produce a valid result.

Remote state

By default, Terraform stores the state locally in a file named terraform.tfstate. When working with Terraform in a team, the use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time. The other problem with storing state file on a local machine is that if the machine is unavailable due to some reason that can stop work progress.

To overcome these problems Terraform allows storing state file at a central location, which can then be shared between all members of a team. This central location can be Azure/Google/Amazon storage accounts, Etcdv2/v3, Kubernetes, Consul, etc. Since state file contains sensitive information, we must

choose a secure location for remote state. Terraform can securely authenticate and connect to Azure storage accounts using a Service Principal, Managed identity, SAS tokens, or storage keys.

Whichever central location and method of authentication we choose for the remote state, we need to provide that information to Terraform and this can be done by configuring the backend attribute of terraform block in the script file. It looks something like this:

```
# let's assume we are using an azure storage account for storing remote state,
#then we can provide that information to terraform using backend attribute
terraform {
    backend "azurerm" {
        resource_group_name = "resource group that contains storage account"
        storage_account_name = "name of Azure storage account"
        container_name = "the name of the blob container"
        key = "the name of the remote state file to be created"
        sas_token = "the security token needed to access storage account"
        #value of sas token should not be hardcoded since it is a sensitive
        #information and it will introduce security risks and vulnerabilities
        #therefore it must be provided at runtime
  }
}
```

Terraform modules

Terraform CLI commands generally uses a folder as a scope consisting of Terraform files for deployment. That folder is the root configuration and is the starting point of script execution. The root configuration can declare provider specific resources directly or instead they can use modules. A module is a container for multiple resources that are used together. It accepts values for variables and generates output. The difference between a module and a Terraform script is that a Terraform module cannot be executed on its own. They need a main Terraform script or root configuration to be executed. They do not have their own configuration related to the backend state. Terraform scripts are used to provision multiple types of environments like development, staging, and production environment, and deploying them using modules provides benefits in terms of uniformity, reusability, and governance.

1. **Uniformity**: With multiple environments for each solution, Terraform developers have to author resources for each environment. Even multiple resources of the same type are sometimes needed for an environment. Terraform developers in a team will often configure resources of the same type in different ways. This could lead to non-uniformity within the code. With modules in place, instead of authoring resources directly, the developers refer to modules for the creation of resources. The modules remain the same for all the developers and solutions. This ensures that each resource is defined uniformly across teams.
2. **Reusability**: A module contains resources that can be applied across projects, solutions, and teams. This ensures that developers need not spend time on configuring resources directly

for Terraform configurations; rather, they can reuse modules and configure them as part of the solution.

3. **Governance**: Modules can be authored with constraints and rules that best suit the organization and solution. For example, adding organization-level tags to resources, constraining the resource deployment region, etc., can be implemented as part of the module. Using the module will ensure that those tags are automatically added to the resource in case the developer misses using them. Similarly, other custom policies can be developed within the module, and all resources will adhere to them during deployment.

Structure of module

A module essentially is a folder or directory comprised of Terraform scripts. The structure of a module is identical to the root folder. The files contained within each of the module's folders are:

1. **main.tf**: File that contains the important resource definitions and configurations.
2. **variables.tf** and **outputs.tf**: Input and output blocks are defined within these files.
3. **versions.tf**: The required version for providers is declared within this file.
4. **dependencies.tf:** This file contains blocks related to data resources and local variables.

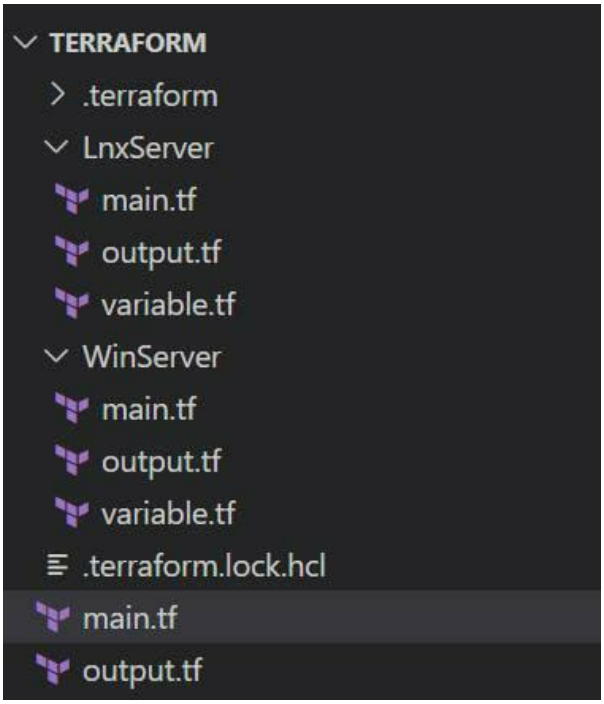

Fig 4) Structure of modules

Looping constructs in terraform

By default, a resource block configures one real infrastructure object. However, sometimes we want to manage several similar objects without writing a separate block for each one. Terraform has two ways to do this: count and for_each.

1. **Count:** It is a meta-argument defined by the Terraform, that can be used with modules and with every resource type. The count meta-argument accepts a whole number, and creates that many instances of the resource or module. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.
2. **For_each:** It is a meta-argument that accepts a map or a set, and creates an instance for each item in that map or set. It provides two meta-elements: each.key and each.value which are used to retrieve individual key and values from a map during iterations. One of the primary differences between for_each and count is that for_each generates a map of resource instances, while count generates an array of resources.

References

[1]. Azure Fundamentals part 1: Describe core Azure concepts (AZ-900) | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Cloud computing

[2]. Docs overview | hashicorp/azurerm | Terraform registry | registry.terraform.io. Accessed on: April 21, 2022. [Online]. Available: Terraform registry

[3]. Overview of Terraform on Azure – What is Terraform? | Microsoft docs | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Terraform on Azure

[4]. Infrastructure as code – HashiCorp Terraform | Microsoft Azure | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Infrastructure as Code

[5]. R. Modi, Deep-Dive Terraform on Azure: Automated Delivery and Deployment of Azure Solutions, 1st ed. Berkeley, CA: Apress, 2021

[6]. Overview – Configuration Language | Terraform by HashiCorp | terraform.io. Accessed on: April 21, 2022. [Online]. Available: HashiCorp Configuration Language