**TERRAFORM**

**Infrastructure as Code (IAC):**

* It is a process of managing and provisioning infra through configuration files.
* Either scripts or declarative definitions can be used for creating the infra.
* It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle
* Terraform plugins called providers let Terraform interact with cloud platforms and other services via their application programming interfaces (APIs)

Ex: Terraform, Ansible

**Terraform:**

An open source IAC tool developed by Hashicorp, used to plan, apply and destroy infrastructure using HCL - Hashicorp Configuration Language

The extension of Terraform Scripts is .tf

**Advantages:**

1. **Plannability** --> we can plan the changes that we are going to make to the infrastructure before actually doing or applying changes
2. **Reusability** --> same code can be re used to multiple environments and multiple teams with different setups by making minor changes.
3. **Maintainability** --> create and maintain all the infra in the cloud provider.

We generally save the terraform files in versioning tools like GitHub.

**Requirements –** Amazon CLI, aws access key, secret key

In AWS user – copy access key, Secret Key and add it in Terraform

**To deploy infrastructure with Terraform:**

Scope - Identify the infrastructure for your project.

Author - Write the configuration for your infrastructure.

Initialize - Install the plugins Terraform needs to manage the infrastructure.

Plan - Preview the changes Terraform will make to match your configuration.

Apply - Make the planned changes.

**Installation:**

Step1: Download the executable file from terraform website

https://www.terraform.io/downloads

Step2: Extract the zipped file and copy it to some folder

Step3: Add the path of the extracted file to the environment variables

**Example: To Create EC2-Instance**

provider "aws" {

region = "ap-south-1"

}

resource "aws\_instance" "my-instance" {

ami = "ami-0851b76e8b1bce90b"

instance\_type = "t2.micro"

key\_name = "terraform"

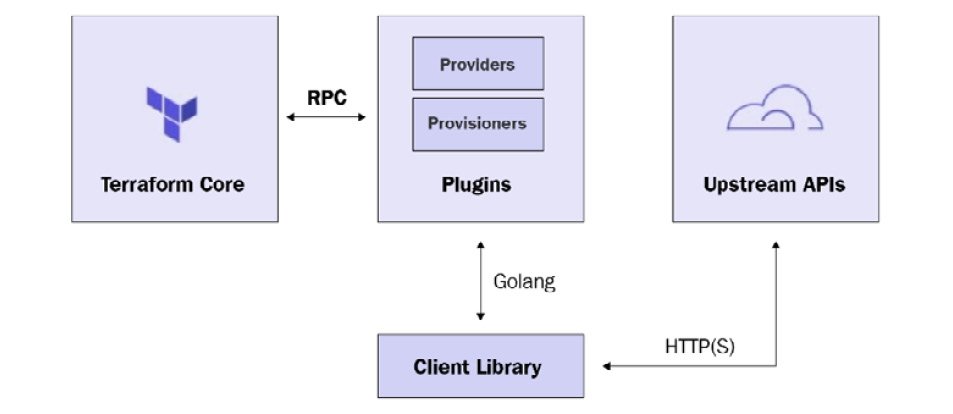
tags = {

Name = "terraform-instance"

}

}

**Terraform Architecture:**



Terraform has two important components: **Terraform Core and Terraform Plugins**. Terraform Core oversees the reading and interpolation of resource plan executions, resource graphs, state management features and configuration files. Core is composed of compiled binaries written in the Go programming language.

**Terraform Life Cycle/Execution Process:**

1. terraform init
2. terraform plan
3. terraform apply
4. terraform destroy

**Terraform Init:**

• This command is used to initialise a working directory containing terraform configuration files.

• Terraform init operations:

a. Initialises the provider plugin (downloads provider plugin of AWS or any other cloud plugin to make use of all the operations to be performed)

b. Initialises the backend

c. Initialises the child module

The folder we are working is root module/parent module. If we create more sub folder for ease of operations, they are child modules.

**Providers:**

• Terraform uses a plugin-based architecture called providers to support hundreds of infrastructure and service providers (AWS, Azure, Google cloud platform, Kubernetes etc.)

• Terraform init command downloads and installs providers used within the configuration file

Syntax:

provider “aws” {

region = “<region>”

access\_key = “<access\_key\_here>”

secret\_key = “<secret\_key\_here>”

}

**2. Terraform Plan:**

• This command is used to create an execution plan which is useful to check whether a set of changes matches your expectations without making any changes to real infra.

Commands:

* Terraform plan
* Terraform plan –out <file\_name> 🡪 to save terraform plan to a file

**3. Terraform Apply:**

* It is used to apply the changes to the infra to reach the desired state

Commands:

• Terraform apply

• Terraform apply --auto-approve

• Terraform apply <plan\_file\_name> 🡪 to apply a particular plan file

**4. Terraform Destroy:**

* It is used to destroy the terraform managed infrastructure

Commands:

• Terraform destroy

• Terraform destroy - -auto-approve

• Terraform plan –destroy 🡪 to create destroy plan

• Terraform destroy –target=<resource-type>.<resource\_name>

**Terraform validate:**

* It is used to check the syntax of terraform configuration files

**Input variables**

* They serve as parameters for terraform configuration files.
* They allow the .tf files to be customised without altering the source code and therefore allows configuration files to be shared between different setups or projects

Syntax:

variable “<variable\_name>” {

type = “<string, integer, Boolean, list>”

description = “<this is to let other people understand about this variable>”

default = “<value>” This is to set the default value for variable

}

**Output Values:**

* Used to extract outputs of terraform managed infra.
* If we ever want to feed the values of one child module to another child module, we can use output values

Syntax:

output “<output\_name>” {

value = <resource\_type>.<resource\_name>.<value>

}

**Terraform settings:**

* If terraform versions are different, terraform settings can be used to limit in what versions terraform could work.
* It is used to configure the version of terraform, provider version and also to define the backend.

**syntax:**

terraform {

required\_version = ">= 1"

required\_providers {

aws = {

source = "hashicorp/aws'

version = ">= 2.6.0"

}

}

backend "s3" {

bucket = "<bucket\_name>"

key = "<path>"

region = "<region>"

dynamodb\_table = "<table\_name>"

encrypt = true

}

**count and for\_each:**

* By default, the terraform resource block can only create one infrastructure object
* If we ever want to create multiple objects, then we can use count and for\_each.

**count example:**

provider "aws" {

region = "ap-south-1"

}

resource "aws\_instance" "my-instance" {

ami = "ami-0851b76e8b1bce90b"

instance\_type = "t2.micro"

key\_name = "terraform"

count = 4

tags = {

Name = “Terraform-Instance ${count.index}”

}

}

**Alias:**

* It is a way of defining multiple configurations for the same provider and select which one to use
* Main use case of alias is to support multiple regions for same cloud provider
* when we want different AWS regions for resources then we can give alias block

provider "aws" {

region = "ap-south-1"

}

provider "aws" {

alias = "<ohio>"

region = "us-east-2"

}

resource "aws\_instance" "my-instance" {

provider = aws.ohio

.

.

.

.

**Assignment: for\_each**

If your instances are almost identical, count is appropriate. If some of their arguments need distinct values that can’t be directly derived from an integer, it’s safer to use for\_each

provider "aws" {

region = "ap-south-1"

}

resource "aws\_instance" "my-instance" {

ami = var.ami

instance\_type = var.instance-type

key\_name = var.key

for\_each = var. instance-names

tags = {

Name = "Terraform ${each.key}"

}

}

variable "instance-names" {

type = set(string)

default = ["a", "b", "c", "d"]

}

**Terraform Provisioners:**

* Used to execute scripts or shell commands on the local machine or on a remote machine during the process of resource creation. For the provisioner to work, it requires a connection block to be able to run commands on a remote machine.
* Provisioner need connection block which contains - ssh, private key, user, host (public ip address)
* **Connection Block Syntax:**

connection {

type = "ssh"

user = "<user\_name>"

private\_key = file("pem\_file")

host = <resource\_type>.<resource\_name>.public\_ip

}

* **\*provisioner "file"** 🡪 **for copy a file to remote then it is used, else not necessary.**

**Three types:**

**1. File**

The file provisioner is used to copy files or directories from the local machine to the newly created resource

**Syntax:**

****

**2. Local-exec**

It is used to run a script or command on a local machine, where the terraform is running.

**syntax:**

resource "aws\_instance" "my-instance" {

.

.

.

# no need of connection block

provisioner "local-exec" {

command = "<command>" (ex: echo hi > test.txt)

}

**3. Remote-exec**

It is used to run scripts or commands on remote resource after it is created. types :-

a. inline - execute a list of commands

b. script - copy local script and execute it.

c. scripts - To copy List of scripts and execute them

**remote-exec** provisioner helps invoke a script on the remote resource once it is created. We can provide a list of command strings which are executed in the order they are provided. We can also provide scripts with a local path which is copied remotely and then executed on the remote resource. File provisioner is used to copy files or directories to a remote resource. We can’t provide any arguments to script in remote-exec provisioner. We can achieve this by copying script from file provisioner and then execute a script using a list of commands.

Provisioner which execute commands on a resource (like running a script or copying file)needs to connect to the resource which can be done through SSH(Connection Block).

<https://medium.com/@mitesh_shamra/terraform-provisioner-fa0911d65ce9>

**Syntax:**

connection {

.

}

provisioner "remote-exec" {

inline = [

"sudo sh /home/ec2-user/script.sh"

]

}

**Null Resourcce:**

* Null resource implements the standard resource life cycle but takes no action by itself on the infrastructure (i.e., doesn't create any resource).
* It can be used as a container for other actions
* Ex: To run provisioners on already existing resources. (ex: already existing EC2 instance)

**Syntax:**

resource "null\_resource" "provisioner" {

#connection\_block

#provisioner block

}

**\*Note**:

* provisioners come under null resource or any other resource (Ex: instance).
* But we can’t separate provisioner as a different <.tf> file. It should always be under a resource\_block and that entire resource\_block can be put in a separate <.tf> file

**Data Source:**

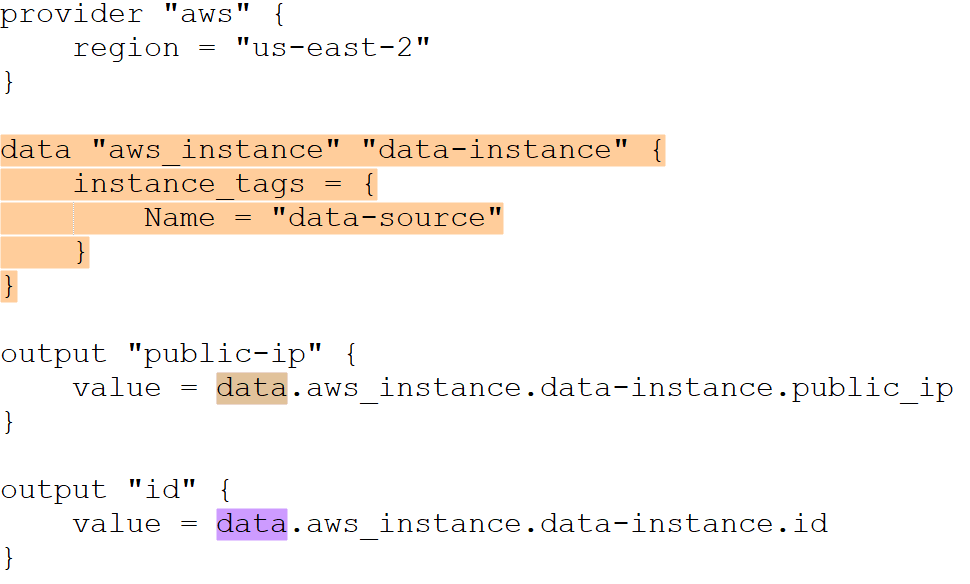
* Allows terraform to use the information about infrastructure that is already provisioned outside of terraform or by another separate terraform configuration.
* Suppose we have different folders for VPC, EC2 etc, the terraform infra of VPC cannot be given to EC2 folder, in such a scenario, data source can be used.

**Syntax:**

data "<resource\_type>" "<name>" {

#identifier

}

****

**Terraform State:**

* Terraform maintains the state of infrastructure and always try to match the state with the cloud infrastructure.
* Whenever we apply the terraform configuration files, a state file is created in the root module. (Root module is where we initialise the terraform files. Child module is modules within root module for creating further infra)
* State file is created in root module by the name "<terraform.tfstate>". which contains all the changes applied to the infra in "json" format (java script object notation)
* if something is manually changed on the cloud provider, terraform has the capability to identify these changes and revert back to original state by comparing it to the state file.

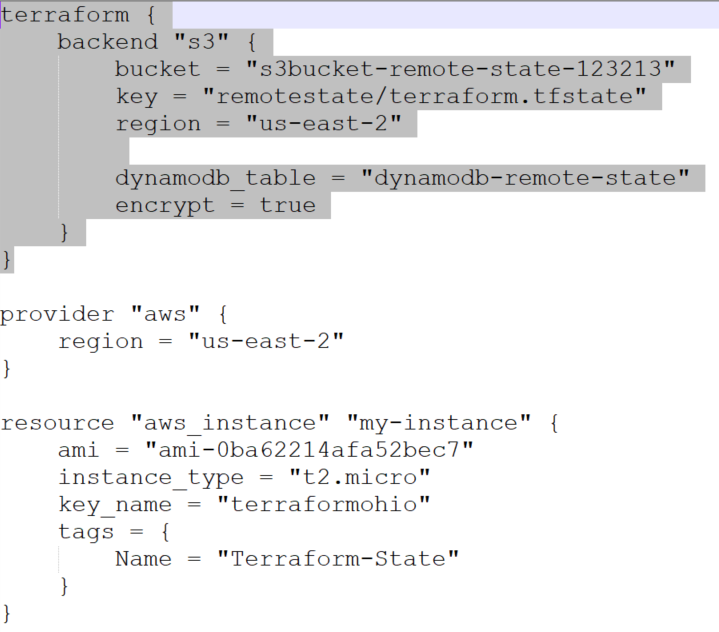
**Remote state/Backend:**

* By default, terraform uses local backend. that is, the terraform.tfstate file is created in the root module
* With remote state, terraform writes the state data to a remote location which can then be shared between all the members of a team.
* we generally use s3 as backend for terraform state file.

**State Lock:**

* It is applied on remote backend to avoid changes for multiple users at the same time to the state file
* DynamoDB table can be used to lock remote state file.





**Advantages of remote state:**

* common state file for the entire team
* versioning option 🡪 allows rollback of mistakes
* safer storage
* avoid exposure of sensitive data present in state file.

**Terraform Taints:**

* It is used to manually mark a terraform managed resource as tainted.
* Once the resource is marked as tainted, it will be destroyed and recreated on the next terraform apply.

🡪 to list all the resources managed by the terraform.

terraform state list

command:

terraform taint <resource\_type>.<resource\_name> to taint a resource

terraform untaint <resource\_type>.<resource\_name> to untaint a resource

**Assignment:**

1. **Terraform Template**

* It is a market place where we can get readymade terraform scripts for using in projects.

1. **tfvars file vs vars.tf**

* A <variables.tf> file is used to define the variables type and optionally set a default value.
* A terraform <.tfvars> file is used to set the actual values or declare the variables.

1. **Terraform import**

* The terraform import command is used to **import existing infrastructure** from console.
* The command currently can only import one resource at a time. This means you can't yet point Terraform import to an entire collection of resources such as an AWS VPC and import all of it.

1. **Terragrunt**

* thin wrapper around Terraform that provides extra tools for:
  + Reducing repetition
  + Working with multiple Terraform modules
  + Managing remote state

1. **Terraform architecture**

* Terraform Core which oversees the reading and interpolation of resource plan executions, resource graphs, state management features and configuration files. Core is composed of compiled binaries written in the Go programming language.
* Terraform Plugins are responsible for defining resources for specific services. This includes authenticating infrastructure providers and initializing the libraries used to make API calls.

**Ignoring Changes in Terraform:**

* **ignore\_changes** (list of attribute names) - By default, Terraform detects any difference in the current settings of a real infrastructure object and plans to update the remote object to match configuration.
* The **ignore\_changes** feature is intended to be used when a resource is created with references to data that may change in the future, but should not affect said resource after its creation. In some rare cases, settings of a remote object are modified by processes outside of Terraform, which Terraform would then attempt to "fix" on the next run. In order to make Terraform share management responsibilities of a single object with a separate process, the ignore\_changes meta-argument specifies resource attributes that Terraform should ignore when planning updates to the associated remote object.

### What are the use cases of Terraform?

 Following are the use cases of Terraform:

* **Setting Up a Heroku App:**
  + Heroku is a prominent platform as a service (PaaS) for hosting web applications. Developers build an app first, then add add-ons like a database or an email service. The ability to elastically scale the number of dynos or workers is one of the nicest features. Most non-trivial applications, on the other hand, quickly require a large number of add-ons and external services.
  + Terraform may be used to codify the setup required for a Heroku application, ensuring that all essential add-ons are present, but it can also go beyond, such as configuring DNSimple to set a CNAME or configuring Cloudflare as the app's CDN. Best of all, terraform can achieve all of this without using a web interface in about 30 seconds.
* **Clusters of Self-Service:**
  + A centralised operations team managing a huge and growing infrastructure becomes extremely difficult at a given organisational level. Making "self-serve" infrastructure, which allows product teams to manage their own infrastructure using tooling given by the central operations team, becomes more appealing.
  + Terraform configuration may be used to document knowledge about how to construct and scale a service. You may then publish these configurations across your business, allowing client teams to administer their services using Terraform.
* **Quick Creation of Environments:**
  + It is usual to have both a production and staging or quality assurance environment. These environments are smaller clones of their production counterparts, and they're used to test new apps before they're released to the public. Maintaining an up-to-date staging environment gets increasingly difficult as the production environment grows larger and more complicated.
  + Terraform may be used to codify the production environment, which can then be shared with staging, QA, and development. These settings can be used to quickly create new environments in which to test and then readily discarded. Terraform can help to reduce the challenge of sustaining parallel environments by making it possible to create and destroy them on the fly.
* **Deployment of Multiple Clouds:**
  + To boost fault tolerance, it's common to disperse infrastructure across different clouds. When only one region or cloud provider is used, fault tolerance is restricted by that provider's availability. When a region or an entire provider goes down, a multi-cloud strategy provides for more gentle recovery.
  + Because many existing infrastructure management solutions are cloud-specific, implementing multi-cloud installations can be difficult. Terraform is cloud-agnostic, allowing you to manage numerous providers and even cross-cloud dependencies with a single configuration. This helps operators develop large-scale multi-cloud infrastructures by simplifying management and orchestration.
* **Applications with Multiple Tier Architecture:**
  + The N-tier architecture is a relatively popular structure. A pool of web servers with a database tier is the most popular 2-tier system. API servers, cache servers, routing meshes, and more levels are added. Because the stages can be scaled individually and provide a separation of concerns, this structure is used.
  + Terraform is a great tool for creating and managing these types of infrastructures. Terraform will automatically handle the dependencies between each layer if you arrange resources in each tier together. Before provisioning the web servers, Terraform will check that the database layer is up and running, as well as that the load balancers are linked to the web nodes. The count configuration value can then be changed in Terraform to quickly scale each tier. Because resource creation and provisioning are codified and automated, elastic scaling in response to load is a breeze.
* **Schedulers of Resources:**
  + The static assignment of applications to computers in large-scale infrastructures becomes increasingly difficult. There are a variety of schedulers available to handle this problem, including Borg, Mesos, YARN, and Kubernetes. These may be used to schedule Docker containers, Hadoop, Spark, and a variety of other software tools on a dynamic basis.
  + Terraform isn't just for physical providers like Amazon Web Services. Terraform can request resources from resource schedulers because they can be treated as providers. This allows Terraform to work in layers, such as setting up the physical infrastructure that runs the schedulers and provisioning into the scheduled grid.
* **Software-Defined Networking:**
  + SDN (Software Defined Networking) is becoming more common in data centers as it gives operators and developers more control over the network and allows the network to better support the applications running on top. A control layer and an infrastructure layer are common in SDN systems.
  + Terraform may be used to codify software-defined network setup. By interacting with the control layer, Terraform may use this configuration to automatically set up and adjust settings. This makes it possible to version the settings and automate modifications. Terraform, for example, may be used to set up an AWS VPC.
* **Demonstrations of software:**
  + Modern software is becoming increasingly networked and distributed. Although solutions such as Vagrant exist to create virtualized environments for demos, demonstrating software on real infrastructure that more closely resembles production environments remains difficult.
  + On cloud providers like AWS, software authors can give a Terraform configuration to develop, provision, and bootstrap a demo. End customers may simply demo the programme on their own infrastructure, and settings like cluster size can be tweaked to more thoroughly test tools at any scale.