**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.
4. 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 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:

1. Initialises the provider plugin (downloads provider plugin of AWS or any other cloud plugin to make use of all the operations to be performed)
2. Initialises the backend
3. 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 infra 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 = “<acess\_key\_here>”

secret\_key = “<secret\_key\_here>”

}

**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

**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

**Terraform Destroy:**

* It is used to destroy the terraform managed infra

**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, terrafom 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 object, 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

}

**Alias:**

it is a way of defining multiple configuration 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-type

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 provision to work, it requires a connection block to be able to

run commands on remote machine.

Help us execute shell commands in local machine(oure system) also on remote server.

Provisioner need connection block which contains - ssh, private key, user, host (public ip address)

**Three types:**

**1. File**

the file provisioner is used to copy files or directories from the local machine to the

newly created resource

**Syntax:**

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

.

.

.

.

provisioner "file" {

source = "<path\_of\_the\_file>"

destination = "<destination\_path>"

}

**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" {

.

.

.

.

provisioner "local-exec" {

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

**3. Remote-exec**

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.

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

**Syntax:**

connection {

provisioner "remote-exec" {

inline = [

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

]

}

**Syntax:**

connection {

type = "ssh"

user = "<user\_name>"

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

host = aws\_instance.<resource\_name>.public\_ip

}

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

**---------------------------------------------------------------------------------------------**

**---------------------------------------------------------------------------------------------**

**Null Resourcce:**

It implements a standard resource life cycle but takes no action by itself on the infra.

it can be used as a container for other actions.

**Ex:** we can run provisions on existing resources.

To run provisioners on already existing resources. (ex: already existing EC2 instance)

Null resource implements standard resource life cycle but doesn't create any resource.

**Syntax:**

resource "null\_resource" "provisioner" {

#connection\_block

#provisioner block

}

**\*Note**: provisioners come under null resource or any other resource for example instance.

But we cant seperate provisioner as a different <.tf> file.

**---------------------------------------------------------------------------------------------**

**---------------------------------------------------------------------------------------------**

**Data Source:**

Allows terraform to use the information about infrastructure that is already provisioned outside

terraform or by another seperate terraform configuration.

(suppose we have different folders for VPC, EC2 etc. the terraform infra of vpc cannot be given

to EC2 folder. in such 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 futher 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:**

1. common state file within team
2. safer storage
3. 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.

**command:**

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

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

terraform state list --> to list all the resources managed by the terraform.

**Assignment:**

**1. Terraform Template**

**2. 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.

**3. Terraform import**

The terraform import command is used to **import existing infrastructure**. 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