9/10

Date: / /

# (Terraform)

## Infrastructure as Code (IAC)

**Infrastructure :-** The servers (Physical/Virtual) placed in a datacentre for running apps, we call it as Infrastructure.

**Infrastructure as Code :-** If we create infrastructure with a script then it is IaC.

### Terraform. (Iac Tool)

Terraform is a tool helps to build/create infra automated with scripts with HashiCorp Conf. lang. (HCL) It is an IaC software used for provisioning infrastructure safely & efficiently.

**Provisioning :-** it involves providing IT infra to host services for an org's business, users, emp, & customers.

**Terraform features :-** IaC, Execution Plan, Resource graph, change Automation,

It runs in Parallel all the resources creation

Q Can we use Terraform to implement changes inside a server?

Yes we can but only limited changes whereas Ansible is specially design for this purpose. So, we can 1st use Terraform to create the server & then we ask TF to execute ansible Playbooks / roles.

Date: / /

| Terraform | Ansible |
|---|---|
| (1) Mainly a infrastructure Provision tool | (1) Mainly a infrastructure configuration tool |
| (2) Advanced Orchestration option. | (2) Relatively poor Orchestrator performance (creating a infrastructure is possible but distroying it is a real challenge) |
| (3) Chances of configuration drift less, means it keeps tracks of changes & and can be easily undo if needed. | (3) Higher chances of configuration drift |

Why terraform :-

Terraform Supports :- (1) One workflow for all types of clouds.

(2) The hosted infrastructure on all publics clouds like AWS/Azure/GCP.

(3) On-prem ie private clouds such as OpenStack / VMwareVsphere / Cloud Stack

(4) Over 100 Publicly available infrastructure intg.

(5) Managing S/w & services, including Dbs like Mysql Source Control systems like Github, Configuration management tools like Ansible, puppet, chef.

(6) Need to learn only 1 lang HCL lang. means terraform itself.

MATRIKAS

* Terraform uses API as code approach to communicate with the Cloud Providers, { HCL converts to respective API understandable format } API as code

AWS Provider → HCL → AWS API
Azure Provider → HCL → Azure API

## Installation on linux

* Install via script or manually
* it is available on terraform site.
* it is a CLI
  (i) # terraform version (Version of TF)

### login to Cloud

#1st ly we need to login to cloud. So we are using here AWS, So doing the same we need to add users to IAM Service *

1) So select IAM then add user
2) Select AWS Access Type. ie ☑ Access key. (Programatically access)
3) Attach Existing Policies
   ☑ Administrator Access.
4) Tags (learn it)
5) Create user.

So after creating it will give you. Access key ID & Secret key. download it (User ID) (PW) b'coz if you loose it you cant recover it.

### Connect with Tf using IAM

# Aws configure & install AWS CLI in your machine
provide above access key id
secret key

# AWS S3 list

# Connected with AWS → check using

MATRIKAS

## LIFE CYCLE OF TERRAFORM

**Terraform init → Terraform Plan → Terraform Apply → Terraform Destroy**

Date / /

1) make directory

```
# mkdir terraformcode.
```

for writing Terraform script one should know Hashicorp conf. lang (HCL).

· The terraform files are plain text file with extension .tf file extension.

* Create new empty file under terraformcode directory

```
# cd terraformcode.
# touch access.tf
```

* Script some code inside it

```
# vi access.tf.
```

1) under file we need to write this

```
Provider "aws" {
  access_key = " accesskey "
  Secret_key = " secretkey "
} region = " us-east-2 "
```

:wq! (Save file & exit)

INIT

now under terraformcode dir. write command initialize

```
# terraform init
```

it will download all the AWS plugins b'coz the provider name is aws in access.tf file

* once init completed on the dir. then .terraform file will get created inside the directory.

* we can write multiple provider inside a single .tf file. but we need to initialize each time if any changes made into .tf file. * interview Question

Date / /

* create new file for launching via Terraform script.

```
# vi ec2.tf
```

ami = Amazon machine Image.

```
resource aws_instance "dev instance" {
  ami = " amicode from Aws Console"
  instance_type = "t2.micro"
}
```

:wq!

Note: In AWS console go to EC2 instance & then AMI there is a code beside the Name of AMI which we need to create. It is pre-defined code from AWS for programatically access the AMI

APPLY

To run terraform script (to apply changes which were written in EC2.tf

```
# terraform apply. then yes
```

* Create n no. of EC2 instance via script

```
# cat ec2.tf
resource aws_instance "dev instance" {
  count = 10
  ami =
  instance_type =
}
```

**PLAN**

Before applying any changes to infrastructure/cloud it is a good practice to run below command

Date

```
# terraform Plan.
```

Plan is like a diagram/Preview of the changes which we are going to implement through any .tf file.

**DESTROY**

* to destroy resource from AWS via script

```
# terraform destroy
```
: yes.

{ before destroying it will create .tfstate backup file & emptied the .tfstate file }

**STATE FILE**

* Terraform creates the state of the instance once it is get created/once the terraform apply run.

```
# ls -l
```
terrafo.tfstate { autogenerated file } on local system

terrafo.tfstate file will record all the changes & the states which were done by terraform on cloud.

"After destroying the .tfstate file will be available but the file is emptied."

**STATE**

```
# terraform state list
```
gives you the list of services which were created through .tf file.

**STATESHOW**

```
# terraform state show "serviceid"
```
instead of list & all resource information it will give info. about the particular resource which passed in command.

MATRIKAS

Date: / /

## # Terraform Terminology

* **Provider** :- is something on which we are about to construct our environment.

↳ provider "AWS"

means we are telling terraform to use AWS cloud Platform to construct the environment. So, terraform will go & dl/w all the plugins related to AWS.

* **Resources** :- On cloud if we create any service is known as Resource like

Compute - EC2
Storage - S3
Networking - VPC etc.

↳ resource aws_instance "name" {
diff Parameter related to EC2
} Passed when we write block.

(i) **Arguments** :- it is simple a key = Pair value, ex: eg- ami = "amicode from AWS Console"

(ii) **Blocks** :- is a piece of code/args. enclosed inside { }
after resource { } → Block

MATRIKAS

* 100% of time key & pair pre defined
90% of time values are ~~also defined~~ pre/to write according
10% of time values are also pre-defined.

Block types :- (1) Provider block
(2) Resource block
(3) Provisioner block

(3) Attributes :- are known after the apply. Info about the resource

## Steps of Terraform {workflow}

(1) terraform init

(2) terraform plan. → to write o/p of plan in a file (interview/certification ques)
terraform plan -out <filename> (after terraform init)

(3) terraform apply
(after terraform init)

(4) terraform state
(after terraform init)

(5) terraform destroy (after terraform init)

* Commands :- terraform -version — version of terraform.

* Whenever we apply terraform apply from a directory that become root directory root module.

important Commands.

* terraform fmt :- It is used to do the correct formatting of your tf file.
This cmd is the only command which runs before terraform init. (interview/exam Question)

* terraform validate :- It is used to validate the tf file for any syntatical error. This runs after terraform init

Destroy target

# terraform destroy -target=resource id.

# terraform destroy -auto-approve

This will destroy the specific id not the whole infra will get destroyed/all resources will not get destroyed. Only targetted resource instance will get deleted.

# Variables

var.instType

var → use to define variable
instType → variable name

I/P & O/P variables in TF are essentials for Parameter & sharing values within the TF config. & modules. They allow you to make your configuration more dynamic reusable & flexible.

Eg :-
```
resource "aws_instance" "test" {
  count = 4
  ami = "amiid"
  instance_type = var.instype
  key_name = "edumar28"
}
```

So instead of value to instance type we are passing variable named as "instType".

2 types Variable :- Input Variable
Output Variable

MATRIKAS

O/P Variable :- O/P variable is used to capture the o/p value of any resources. It is started with o/p keyword. Create New file

Syntax - Output.tf

```
Output "PublicIp" {
  Value = aws_instance.example.PublicIp
}
```

How to pass value to variable in ec2.tf file

1st we need to create a new .tf file say name as vars.tf in the same folder/directory.

then

vi vars.tf

Now write variable declaration under vars.tf file So, basically this file starts with Variable

I/P syntax

```
Variable "variablename" { type = String
                          default = "t2.micro" }
```

:wq → Primitive variable

Eg.

```
Variable "instType" {
}
```

:wq then run plan so it will ask for value.

We can pass value via 3 types listed below 1st, 2nd, 3rd

1st we can give default value to the variable by putting value under the block of variable

{ default = "t2.micro"

2nd we can provide value while executing the plan command

terraform plan

Enter a value: instType = t2.micro



* Tfvars.file is a special file in which we are passing values to our variables. We can modify the default

Date: / /

Default file :- terraform.tfvars file name but during apply you have to the Pass the modified file name

3rd we can provide into the single line while executing plan command

terraform plan -var="instype=t2.micro"

## #tags

Tags are used to provide names to resources such as while we are initiating count=4 for EC2 instance but while launching it the EC2 instances does not have name to it so giving name to resource we are using tags

Eg.

ec2.tf file

after resource aws_instance {

} block.

we can write tags in the same file ec2.tf

```
tags = {
  Name = "test ${count.index + 1}"
  Training = "devops"
}
```

after count 4 it will add & append to test say

Ole Variable = ole

Date: / /

## # Provisioner Block (PB)

PB is about implementing a change once the resource is created.

PB always goes inside the resource block

Eg. resource {

syntax.

provisioner {

}

}

it is mandatory to write Provisioner code inside the resource block.

Eg. Resource {

Provisioner "file" {
source = " some path from terraform server"
destination = " deploy changes to newly created resource path"

}

MATRIKAS

Date: / /

## # Modules.

Any terraform file inside a directory with .tf is a module.

A Terraform module allows you to create a logical Abstraction on the top of some resource set. it is used to group resources in single file & then we can reuse it like functions etc.

How to call modules from main.tf of calling project.
in main.tf

Provider "aws" {

}.

module " Name of module " {
Soura = ". // module1 "
↳ Path of module1
}

module " Name of module" {
Soura = ". // module2"
}
↳ Path of module2

MATRIKAS

Date: / /

## Multi Region Tf file.

If you want multiple region to be a part a single tf file then we have to write multiple Provider block. eg:-

```
Provider "aws" {
    alias = "abc"
    region = "us-east-1"
}

Provider "aws" {
    alias = "xyz"
    region = "us-west-2"
}

resource "aws_instances" "example" {
    ami = "ami-0123456789ORbc"
    instance-type = "t2.micro"
    provider = "aws.abc"
}

resource "aws_instances" "Example2" {
    ami = "ami-09876543210cba"
    instance-type = "t4.micro"
    provider = "aws.xyz"
}
```



Date: / /

## Multi Cloud Tf file

If you want multiple cloud to be a part of single tf file then we have to multiple provider block eg:-

```
Provider "aws" {
    region = "us-east-1"
}

Provider "azurerm" {
    subscription_id = "cio-staging1"
    client_id = "abcdef1234"
    client_secret = "12345edcba"
    tenant_id = "X42134987"
}

resource ".EC2" {

}

resource "aws_instance" {

}
```

Date: / /

## Modules.

Continuing module from last page.

The advantages of using TF modules in our Iac projects lies in improved organisation, reusability, maintainability.:

(1) Modularity :- TF modules allows you to break down your infra. configuration into smaller, self contained components. This modularity keeps & handle a specific piece of functionality eg. EC2, DB, VPC etc seperately & makes it easier to manage.

(2) Reusability :- With modules, you can create reusable templates for common infrastructure components. Instead of writing similar configuration for multiple projects, you can reuse modules across different TF projects. Its promotes consistency in your infrastructure

(3) Simplified Collaboration :- modules makes it easier for teams to collaborate on infrastructure projects. Different members can work on diffent modules & then these modules can be combined to build complex infrastructure deployments.



Date: / /

(4) Versioning & Maintenance :- Modules can have their own versioning making it easier to manage updates & changes.

(5) Abstraction :- Modules can abstract away the complexity of underlying resources. Eg. EC2 instances module can hide info of security groups, subnets etc, allowing users to focus on high level parameters like instance type & image id.

(6) Testing & Validation :- Modules can be individually tested, ensuring that they work correctly before being used in Multiple proj.

(7) Documentation :- Modules promotes self-documen

(8) Scalability :- As your infrastructure grows, modules provides a scalable approach to managing complexity.

(9) Security & Compliance :- Modules can encapsulate security & compliance best practices.

MATRIKAS

Date / /

## Module Structure

* Module itself is a self/independent project which has its own

Main.tf , variable.tf , output.tf etc

In modules the source is the mandate when calling it from Parent module

Once your module is ready with the logic you can call the module in your project via Projects Main.tf

*abc project

Module folder → EC2 instance → Main.tf
→ O/P.tf
→ variable.tf.

} Child module

(1) EC2 instance → Main.tf

```
provider "aws" {
    region = "us-east-1"
}

resource "aws_instance" "Example" {
    ami = var.ami_value
    instance_type = var.instance_type_value
    subnet_id = var.subnet_id_value
}
```



* In O/P.tf file we write those parameters which we need to capture once the execution of tf is complete in below case we are capturing ip address of EC2 instance

Date

once the deployment is done

(2) EC2 instance → output.tf

```
Output "Public-ip-address" {
    value = aws_instance.example.public_ip
}
```

(3) EC2 instance → variable.tf

```
variable "ami_value" {
    description = "value for ami"
    default = "ami-0123456abc"
}
variable "instance_type_value" {
    description = "Value for instance_type"
}
variable "subnet_id_value" {
    description = "value for subnetid"
}
```

* Once the module is created it is necessary to initialize the module so that we can refer it from another directory/repo

⇒ terraform init

Date: / /

## * How to use/call modules from another project

main.tf

* Test project

main.tf, .tfvars, variable.tf

(1) Main.tf

```
provider "aws" {
        region = "us-east-1"
}
module "Ec2instance" {
  source = "Path of module"
          ./modules/ec2instance
  ami-value = "ami-050cm123"
  instance-type-value = "t2.micro"
  subnet-id-value = "subnet-0195219abc2"
}
```

} Parent module

Note: If a project is using any module then it should have info about the module, parameters & most importantly Path of the module. Then only you will successfully call a module in your project.

Once the module is called from main.tf lets pass ami-value dynamically from .tfvars file.



Date: / /

(2) Terraform.tfvars

```
instance_type_value = "t2.small"
ami_value = "abcdef123890"
```

(3) Variable.tf

```
Variable "instance-type_value" {
        type = string
}
Variable "ami-value" {
        type = string
}
```

How to pass above value from .tfvars file in main.tf file of test folder/ root directory

(4) main.tf

```
module "myweb1ser" {
source = "Path of module"
ami-value = "${var.ami-value}"

instance-type_value = "${var.instance-type-value}"
}
```

"${var.parameter}"

Date / /

How to catch o/p value of ip from main.tf of root directory which is declared in the o/p file of module. So again in main.tf

(i) main.tf

```
module same code as previous{
    }
output mypublicIP{
 value = module.myweb.Public_ip_address
}
```

This name is case sensitive make sure you are well aware about the o/p value of module in this case o/p.tf file consist of

```
output {"Public_ip_address"}
    }
```

If name is differ then it will not allow you to catch the o/p error will occurs.

MATRIKAS

Date / /

Source of modules can be from diff2 types

The module installer supports installation from a number of different source types

(1) local Paths
(2) Different Project / Sub directory
(3) Terraform Registry
(4) Github / Bitbucket
(5) Generic Git, Mercurial repo.
(6) HTTP URL's
(7) S3 buckets
(8) GCS buckets

MATRIKAS

(certificate Prospective

{file, number, string}
lookup, Elements

Date: / /

## ✗ functions ✗

Terraform includes a no. of built-in functions that you can call from within expressions to transform & combine values.

Syntax :- { Tf does not allow user defined fun^n as of now only built fun^n can be used }

```
max(5,12,9)
join(';', 'a,b,c')
```

Eg. Terrafor console is used to test functions

⇒ output:tf.

```
Output Print {
    value = "${join("-->", vars.users)}"
}
```

function name

How to call function in a tf file

⇒ Variable.tf

```
variable users {
    type = list
    default = ["a", "b", "c"]
}
```

o/p of the above is

Print = "a --> b --> c"

MATRIKAS

(heart of TF)

Date: / /

## Terraform State (terraform.tfstate)

Terraform State file is a crucial component of TF that helps it keep track of resources it manages & their current state. Tf must store state about your managed infra and Configuration. It is also used to improve the performance of large infra by keeping the track of metadata of resources.

Terraform.tfstate file is a Json or YCL formatted file

Advantages :-
- → Resource Tracking
- → Concurrency Control (statefilelock)
- → Plan Calculation
- → Resources Metadata

Disadvantages :-
- → Version Complexity
- → Security Risks

* Terraform State file is autogenerated file which creats when the 1st time terraform apply comman runs.

Command Terraform Show → To see available state file

MATRIKAS

Date: / /

State file is the heart of TF & it holds very sensitive info about secrets, PW etc so to make this secure we use Remote Backend To overcome the disadvantages

A Remote backend stores the terraform state file outside of your local file system & Version control, (i) Terraform cloud (2) Azure blob etc (3) S3, etc are used as Remote Backend.

* S3 as a Remote backend is the popular choice due to its reliability & scalability

⇒ for this we need S3 bucket in AWS with appropiate IAM Permission

⇒ Configure Remote backend logic in Terraform project

Test Project

main.tf, backend.tf, variab.tf, .tfvars, O/P.tf

(i) Backend.tf

```
terraform {
backend "s3" {
bucket = "Path/Name of bucket"
key = "Path to you .tfstate"
region = "S3 region"
encrypt = true
dynamodb_table = "DB Table name"
}
}
```

Date: / /

(2) Main.tf

```
Provider "Aws {
    region = " "
}

resource "aws_instance" "Example" {
    ami
    instance
}
```

After applying terraform apply State file will get created in S3 bucket.

(ii) => Create a dynamoDB table (for state locking) so lock the state file in order to prevent concurrent access issues when multiple users/Processes run Terraform.

(i) Create Dynamo Table :- you can create by using CLI/UI make sure it has lockid as a column name. (Partition key)

(2) Configure DynamoDB Table in Terraform backend configuration :- Give your reference of dynamo db table to backend.tf file

MATRIKAS

Terraform state list :- How many resources available as per terraform state file.

Is :- If any file created using tf so system file name present

By following these steps, you can securely store your Terraform state in S3 with state locking using DynamoDB, this will overcome the disadvantages of Tf like storing sensitive info in VCS. & ensuring safe concurrent access to your infra.

* How to migrate existing statefile from Remote location to another location/local.

⇒ S3 → local/Azure

To do so 1st comment the backend.tf file code then run.

→ terraform init

Once initialization is done so it will remove the backend config. then run

→ terraform init -migrate-state

Enter value - Yes.

Once this is done your remote backed get changed & the State file get copied to the local dir or if you add new backed config So it will relocate to that remote location.

MATRIKAS

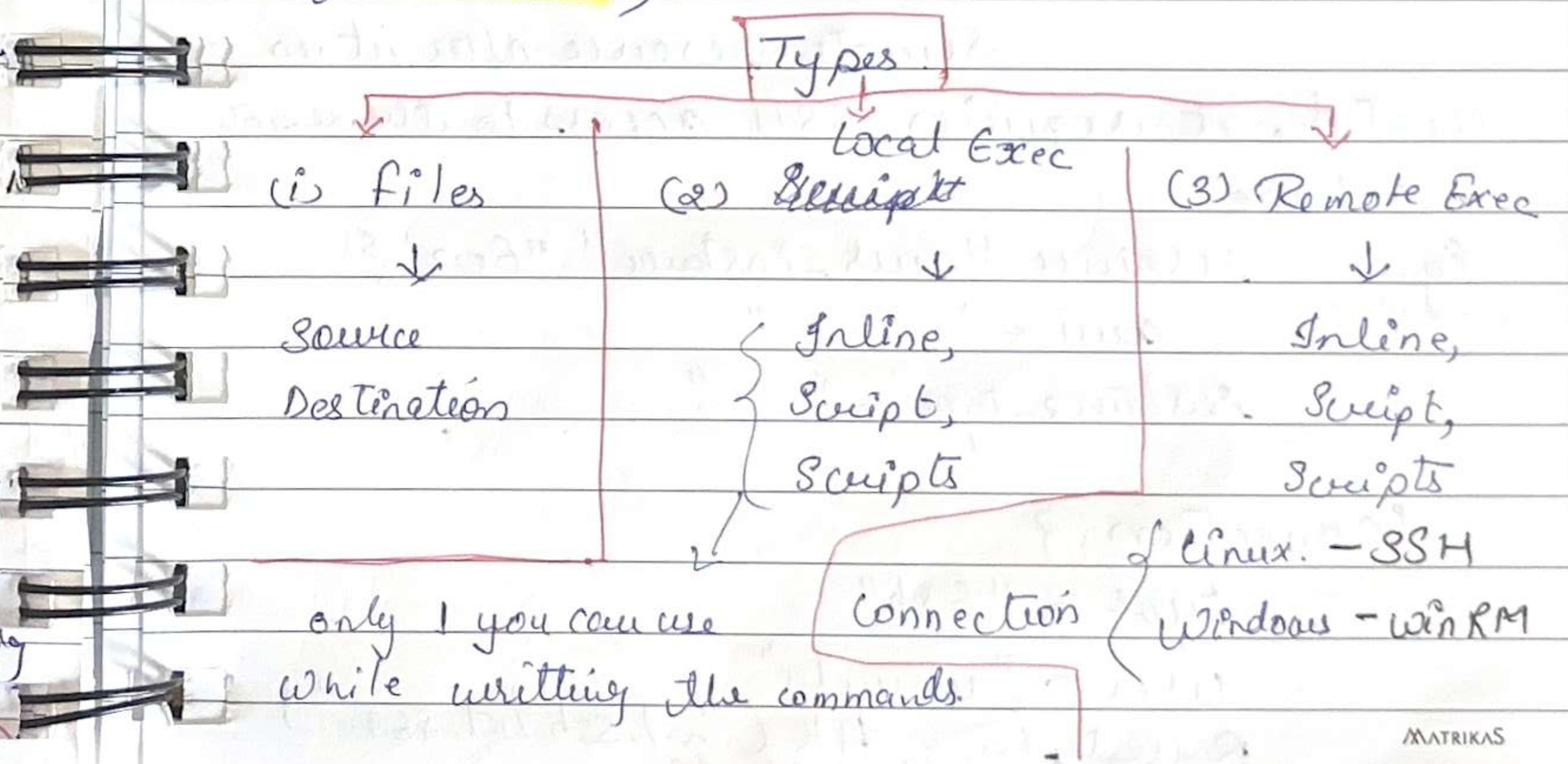
Date: / /

## Terraform Provisioners

Provisioner is a concept in Tf that will let you copied or execute particular actions during the creation/destruction of resources.

By default the Provisioners will run during the creation time of you are explicitly not declaring in the provider block.

Provisioner is basically used to update some packages at the target machine (Basically this is done by Ansible but small changes we can do it by State Provisioners).




MATRIKAS

Date: / /

1) Local Exec :- Executes command locally where tf is running. useful for interacting with local tools or processes.

Eg.
```
resource "aws_instance" "EC2" {
    ami = "ami-1234.."
    instance_type = "t2.micro"

    provisioner "local-exec" {
        command = "echo ${self.public_ip} >> ip_address.txt"
        when = "create"
    }
}
```

(2) Remote-exec :- executes commands on the remote resource after it is created. it requires SSH access to the remote machine

Eg.
```
resource "aws_instance" "EC2" {
    ami = "     "
    instance_type = "     "

    connection {
        type = "ssh"
        user = "ubuntu"
        private_key = file("~/.ssh/id_rsa")
        host = self.PublicIP
```

Date: / /

```
    provisioner "remote-exec" {
        inline = [
            "sudo apt-get update",
            "sudo apt-get install -y nginx"]
    } on_failure = "continue"
}
```

"fail" is default setting

(3) file :- Copies file from local to Remote resources. often combine with remote-exec for configuration.

Eg.
```
resource "aws_instance" "EC2" {
    ami = "     "
    instance_type = "     "

    connection {
        .
    }

    provisioner "file" {
        source = "app.config.conf"
        destination = "/etc/app/config.conf"
    }
}
```

Date: / /

When to use :- (1) upload files
(2) cleanup tasks
(3) Perform local actions
(4) update a resource after creation.

NOTE

Provisioners must be used carefully because it does not change state. their changes may lead to discrepancies b/w tf & actual infra in cloud. Always use TF_log = Debug to troubleshoot issues during provisioners.
Handle your connection block securely.



Date: / /

## Local Variables

It's scope is within the current file only. why we are using local variables. because the same info / tags has to be passed multiple resources so we can write all tags in local variable block & then you can access it through out the file.

```
Provider "aws" { region = "us-east-1" }

locals {
  Mytags = {
    Name = "Beethika-local"
    Team = "DevOps"
  }
}

resource "aws_instance" "Example" {
  ami = " "
  instance_type = "t2.micro"
  tags = local.Mytags
}
```

This is how it is used to refer in resource block local → without s

This is how it is used to declare

local(s) → (s) is needed.

Date / /

## (Output Data Block)

O/P block retrive information about your infrastructure available on the CLI. & expose info for other terraform config's to use. Output values are very similar to return value in any prog. language

Reuse the info which was given by TF after resource creation.

```
output "ec2_public_ip" {
    value = aws_instance.myinstance.public_ip
}
```

O/P Block

Eg.

```
output "ec_private_ip" {
    value = aws_instance.myinstance.private_ip
                                    [*]
}
```

If your have multiple EC2 instances

Eg.

```
output "ec_public_ip" {
value = { for key, instance in aws_instance.
          myinstance : key => instance.public_ip
}
```

If you have used local variables then you can use this code for more than 1 instance in AWS.



* To retrive any info from anywhere in TF we use data block.
* To create any resource anywhere in TF resource block

Date / /

## Secret Managment / Vault Integration.

In TF if we want to secure the PW, certificates or any sensitive information we can use the ~~Hcl~~ Hashicorp vault to achieve the same.

There are few ways to manage sensitive info. in terraform files.

i) Configure Vault Provider :- it helps to fetch secrets dynamically during TF execution

Eg.

```
provider "vault" {
address = "https://vault.eg.com"
token = var.vault_token }
```

ii) fetch secrets using the vault_generic_secret Data source :- secrets are not stored in state file as long as they are not refenced in O/P or resource arguments

Eg.

```
data "vault_generic_secret" "sg" {
  path = "secret/my-app"
}
resource "aws_db_instance" "db" {
```

Date: / /

username = "admin"
password = data.vault_generic_secret.db
.data["db-password"]
}.

3) Use Envir variables :- Store sensitive info (vault token) in env vars and retrieve them dynamically.

Eg.

export vault_token = "your token"
export vault_addr = "https://vault.com"

refer in TF

provider "vault" {}

4) Store vault token/secret in TF vars. with encryption

(5) Avoid storing secrets in TF statefile

(6) Use vault agent with auto-Auth

(7) Rotate secrets regularly.

MATRIKAS

Date: / /

## Lifecycle block.

It comes under resource blocks to controls its behaviour during apply. lifecycle behaviour in Terraform

Default :- create before destroy before/not

Rest :- Prevent destroy
:- ignore-changes. {array}
:- replace-triggered-by.
:- create before destroy.

Eg.

resource "aws_s3_bucket" "s3" {
bucket = "My bucket"
acl = "private"

lifecycle {
prevent_destroy = true
}
}

Types of LCB Actions

(1) Prevent_Destroy :- Protecting a resource from deletion

(2) Create_before_destroy :- Zero downtime replacement

(3) ignore_changes :- Ignore specific changes

MATRIKAS

Date: / /

NOTE:- { You can import whole infra at once one by one resource id/ name has to be given in terraform import command }

Date: / /

individually

## Terraform import { works with resources }

Existing infra. has to be managed by terraform.

for importing we need to create a main.tf file with some mandate details.

* EC2 instance should be available in AWS before import so that we can manage it by terraform.

* for EC2 instance

```
resource "aws_instance" "test" {
    instance type :-
    name :-
    ami :- " it should match the existing ami of EC2 instance".
    tags :-
    Name :- " Name of EC2 "
}
```

* Plan will not create a tfstate so for that you have to do terraform import

terraform import aws_instance.test <resource id>

(id of EC2 instance)

* Now tfstate will be available now for double sure see the details in tfstate & AWS console resource.

* terraform Plan

* Terafaform apply -auto-approve.

MATRIKAS

MATRIKAS

Date: / /

## Verbose Logging. {Set logs to Env variable}

Create tf file

```
resource "localfile" "Eg" {
    file name = "${Path.module}/hello.txt"
    content = "Hello"
}
```

Set TF_log environment variable to enable logs in terraform

export TF_Log = Debug/TRACE/INFO/ERROR

enable TF_log_Path to redirect this to a file

export TF_LOG_PATH = "./terrafor.log"

* terrafor init logs get created
  terrafom init

ls -la

* Your terrafom log file created
  terrafor apply
  to write changes in logs.
  ~~unset to unlog changes in log file~~

* See log file.
  less terrafom.log
  @ to come out from log file.

MATRIKAS

Date: / /

* To not log any changes in log file then remove the environment variables using unset

unset TF_LOG
unset TF_LOG_PATH

} Capital letter the env. variable

MATRIKAS

Date: / /

## (Refresh-only) STATE DRIFT

When AWS Config. is not in sync with statefile. To keep your AWS configuration in sync with the statefile file then -refresh-only used to overcome the situation of State Drift.

Eg :-

You have created EC2 instance using TF but someone from console have stopped it so there will be mismatch in the configuration of state file so, to update statefile as per the aws configuration, we use -refresh-only command

⇒ terraform apply -refresh-only

Enter value : yes.

This command will show only changes to the file but it will apply only when you say yes, then it will be in sync.



Date: / /

## Data Block :-

Data block is used to retrieve information about existing resources in your AWS, particularly useful when you need to reference or use attributes of existing resources (S3, EC2AMI, VPC) in TF configuration

Imp Notes.

- It's a Read only block. only fetch info, they don't, create, update or delete resources
- They are useful for dynamically fetching resources attributes that are not hardcoded
- We can use data block o/p as i/p for other TF resources and ensuring consistency in complex deployments

Data block provides info which is already pre-defined in cloud Provider. It acts like a filter when we don't want to go to console for the search.

Date: / /

To fetch latest ami value for ubuntu OS. with available state

```
data "aws_ami" "OSname" {
    most_recent = true  owners = ["Au"]
    filter {                    for ubuntu x64 Archi
    name = "name"
    values = ["ubuntu/images/hvm-ssd/*amd64*"]
    }   values = ["amzn2-ami-*"] → for Amazon Linux machine
    filter {
      name = "state"
      values = ["available"]
    }
}
```

Declare the data block here

```
resource "aws_ec2_instance" "My instance" {
    ami = data.aws_ami.OSname.id
    type =
    count = 1
}
```

Referenced here in Rs block



Date: / /

## Meta Arguments

These are special arguments that can be used within resource & module block to control how TF manages resources/modules.

→ lifecycle :- manages lifecycle of resource.

→ depends on :- it ensures that the resources creation is done until the dependent resource is created.

Eg. resource "aws_s3-bucket" "S3" {
    bucket = "MyB." }

```
resource "aws_s3_bucket_object" "obj" {
    bucket = aws_s3-bucket.S3.bucket
    Key = " "
    Source = " "
depends_on = [aws_s3-bucket.S3] }
```

→ Count :- it ensures that how many instances of a resource/ module created.

Eg. resource "aws_instance" "EC2" {
    Count = 3
    ami = " "
    instance type = " "
}

| count | for_each |
|---|---|
| use count for sequential indexing | use for_each for mapping specific keys & values. |

Date: / /

→ for_each :- allows you to iterate over a map or set of strings, creating resources with unique conf. for each item.

Eg. resource "aws_s3_bucket" "Eg" {

for_each = {

bucket1 = "Eg-buc-1"

bucket2 = "Eg-buc-2"

}

bucket = each.value

tags {

name = each.key

}

}

→ Provider :- It allows you to specify which provider configuration to use.

Eg. provider "aws" {

region = "us-east-1"

alias = "us-east-1" }

provider "aws" {

region = "us-west-2"

alias = "us-west-2"

}

resource "aws_s3_bucket" "East bucket" {

provider = aws.useast1

}



Date: / /

## Taint & Untaint

It is used to manage the lifecycle of resources. They help to control whether a resource should be recreated during the next Tf apply. Resource marked as taint displayed in Tf Plan.

(i) Taint :- If a resource marked as "Tainted", it means Tf will destroy & recreate the resource during the next Tf apply. If you want to replace a resource without making changes to the configuration.

Eg :- terraform taint <resource-address>
terraform taint aws_instance.example

(ii) Untaint :- It removes the taint status from the resource. It will not destroy & recreate during next apply. when by ~~mistakenly~~ mistakely resource added as tainted

Eg :- terraform untaint <resource-addr>
terraform untaint aws_instance.eg.