Terraform

- Terraform is an open-source tool used to write Infrastructure as A Code (IaC) to automate provisioning the public cloud infrastructure.

- Terraform is written in Golang and available on GitHub

-This includes both low-level components like compute instances, storage, and networking, as well as high-level components like DNS entries and SaaS features

- Terraform code/configuration is written in type .tf

- Terraform helps in prediction of changes through plans and we can know what changes will be applied before applying it.

- Terraform always keeps the infrastructure state saved and terraform at every apply of the changes to infrastructure is always matched with the state file and it predicts the changes need to be applied.
- Safe to run terraform code many times because terraform only apply the changes for the first if the changes are not there in state.

- Terraform scripts can be version controlled means we keep the terraform scripts in github

Terraform Provider

- A provider is responsible for understanding API interactions and exposing resources. It interacts with the various APIs required to create, update and delete various resources. Terraform configurations must declare which providers they require so that Terraform can install and use them.
- In a complete terraform project we can have only one provider block.

```
provider "aws" {
        access_key = ""
        secret_key = ""
        region = ""
}
```

- If we won't configure access_key and secret_key key in provider block then terraforms will try to fetch the keys from aws_cli that is using the below environment variables.

1. AWS_ACCESS_KEY_ID
2. AWS_SECRET_ACCESS_KEY

Provide Aliases

- we can define multiple configurations for the same provider and select which provider to use in resource and other blocks.

```
provider "aws" {
        region = "ap-south-1"
}

provider "aws" {
        alias = "region1"
        region = "us-east-1"
}

provider "aws" {
        alias = "region2"
        region = "us-east-2"
}
```

How to use provider alias ?

```
        resource "<resource_type>" "<resource_name>" {
                provider = aws.region1
        }

        resource "aws_instance" "my_instance" {
                provider = aws.region2
                ami = ""
                instance_type = "t2.micro"
        }
```

Terraform Registry

- Terraform registry is a repository of modules and resources which are write by terraform w.r.t public clouds and we user these resources for provisioning public clouds.
- Terraform registry is also a community-based registry which are maintained by providers only.

URL: regisrty.terraform.io

Example: aws - ec2 instance -> aws_instance
- s3 bucket -> aws_s3_bucket
- vpc -> aws_vpc

Terraform INIT

- This command we use to initialize a working directory containing terraform script as terraform project.
- This is the first command that should be executed after writing a new terraform script.
  - It is safe to run this command multiple times.

The initializations steps are:

1. Initializing the Backend ....
2. Initializing provider plugins ...
3. Initializing the child modules ...

Terraform plan

- The terraform plan is used to create a execution plan which internally performs a refresh unless user has disabled this refresh.
- Plan will scan all the .tf scripts in the project directory and it manages to determine the changes to be applied to cloud (means it determines the desired state).
- Plan will never do any changes to the actual infrastructure it just gives what desired State will be applied.

To save a plan in a file

terraform plan -out <file_name>

terraform plan -out plan.out

Terraform apply

- The terraform apply command is used to apply the planned changes to the actual cloud infrastructure.
- This will internally run the plan again with the user confirmation the predicted changes will be applied.

To apply changes will out user confirmation

terraform apply -auto-approve

To run a particular, terraform script

terraform apply <script>.tf <script1>.tf

To apply a plan file

terraform apply <plan_file>

Terraform validate

- This is to validate the syntax of the terraform scripts in a project.

Terraform destroy

- This is used to destroy the resources which are managed by this current terraform scripts.
- Crate a destroy plan

terraform destroy -target <resource_type>.<resource_name>
terraform destroy -target aws_instance.ec2_instance2

Terraform state file

- Whenever we build an infrastructure with terraform configuration files (terraform apply) a file gets generates by name terraform.tf state which contains all the configuration data in JSON format.

- When we try to apply / rerun the apply command, terraform will compare the actual state of cloud with the state file and generates the plan.

- local

we can keep the state file locally in the same machine where we execute the terraform code.

- Remote

Also know as backend, instead of keeping state file in local we can keep it in a remote location such as s3, etcd, kubernetes, artifactory etc.

Benefits

- Safer storage: As state file contains sensitive data, Saving in the remote place we can safe guard the data

Example: In **S3** backend we can use encrypted s3 to secure the **state file**. - Versioning/Auditing: Invalid access can be identified or can maintain the version on state file.

- Multiple user of same team can apply the changes by sharing the same state file.

Terraform settings

- used for specifying provider requirements
- we can restrict the provider version
- we configure backend in these settings
- we use terraform block
- default file terraform.tf / versions.tf

```
terraform {

}
```

Terraform variables and outputs

- Local variables

A local value assigned to a name, so we can use it multiple times with in the same module without repeating it.

```
locals {
        service_name = "gmail"
        owner = "Kubernetes"
}
```

- Input variables

Input variable serve as parameters for terraform configuration files/modules
main advantage is input variables are used to configure the modules instead of altering the source code we can pass the input values.
To supress the values in the console output use sensitive = true

```
variable "bucket_name" {
        type = string
        sensitive = true
        default = "s3-backend-bucket-aaaaa-22222"
}
```

- Output variables

- Output variable is like return values of terraform code.
- A child module can use the output of another module as input.
- To supress the values in the console output use sensitive = true

```
output "ec2_public_ip" {
        value = aws_instance.new_ec2.public_ip
}
```

Modules

A module is a container for multiple resources that are used together

All the set of terraform configurations (all .tf files) in a folder is called as module

Root Module

- The .tf files iin the working directories is root module
- Where we run terraform init which contains terraform configurations is root module
- Every terraform configuration has to have atleast one module by default it will be root module.

Child module

A terraform set of configurations defined in sub folder can be called by other modules (root module) to extend the functionality.
These are usually user defined modules.

Terraform Taint

- The terraform taint command is used to mark a terraform managed resource as tainted, which will mark the resource to be destroyed and recreated in the new apply operation.
- Taint will not apply the tainted mark on the actual resource in the infrastructure but it will mark as tainted in the state file.
- If we do terraform plan/apply it will show resources as tainted.

To taint a resource
terraform taint <reasource_type>.<resource_name>

To taint a module resource
terraform taint -module=<module_name> <reasource_type>.<resource_name>terraform taint <terra.arn>
To list the resources
terraform state list

To remove taint
terraform untaint <reasource_type>.<resource_name>

Null resource

- The null resource will have the complete lifecycle support but there will be no change in the cloud configurations.
- We can use triggers to collect the data of some resources.
- We can use connections to connect to resource using provisioners.
- The triggers argument allows specifying an arbitory set of values only when changed, trigger will update the value.

```
resoruce "null_resource" "<resource_name>" {

}
```

Null_data_resource

- This block we are not using in our company we use data block instead of this.

```
data "null_data_source" "values" {
        inputs = {
            all_server_ids = concat(aws_instance.green.*.id, aws_instance.blue.*.id)
            all_server_ips = concat(aws_instance.green.*.private_ip,
aws_instance.blue.*.private_ip)
        }
}
```

Provisioners

- Provisioners are used to copy file/directories from terraform source machine to cloud resource and to run command/script on local or remote resource as a part of resource creation.
- They are similar to "EC2 user-data" which will run once on the creation and if it fails terraform mark this resource are tainted.
- Most provisioners require access to resource via SSH or WinRM and terraform supports nested connections.

```
Connection
    connection {
        type = "ssh"
        user = "root"
        password = ""
        host = ""
    }
}
```

Types of provisioners

File

- The file provisioner is used to copy file or directory from the machine which we execute terraform scripts to the cloud resources.
- This provisioner needs a connection to the cloud resource.

local-exec

- local-exec provisioner will execute commands or script on the machine which we execute terraform scripts.

remote-exec

- remote-exec provisioner will execute commands or script or scripts on the cloud resources.

Inline - this is a list of commands which we want to execute on cloud resources (ec2)

script - This will run a script and takes local script path as a parameter to run on cloud resource.
scripts - This will run a scripts present in a directory and takes directory local path as a parameter to run on cloud resource.

Note: for script and scripts we need use the combination of file and inline remote-exec provider where we use file provisioner to copy script or scripts and then we execute that script on cloud resource.

```
resource "aws_instance" "web" {
  # ...

  provisioner "file" {
        source      = "script.sh"
        destination = "/tmp/script.sh"
  }

  provisioner "remote-exec" {
        inline = [
          "chmod +x /tmp/script.sh",
          "/tmp/script.sh args",
        ]
  }
}
```

data sources / data resource

- A data source is used to get the information dynamically of a resource from cloud provider.
- This resource is read-only and this will be executed every time we run the terraform code.
- We can query the information and get the required field of information using filters.

```
data "aws_ami" "ubuntu" {
        most_recent = true

        filter {
                name   = "name"
                values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
        }

        filter {
                name   = "virtualization-type"
                values = ["hvm"]
        }
}

resource "aws_instance" "example_ecc2" {
        ami = data.aws_ami.ubuntu.id
        type = "t2.micro"
}
```

Terraform workspace

- Workspace in terraform are a way to manage different set state files independently within different workspaces.
- In terraform every configuration that we do should be in a workspace by default it is 'default' workspace.
- Different workspace means completely separate working directory or root module.

To list workspace

```
terraform workspace list
```

To create workspace

```
terraform workspace new <workspace_name>
```

To change workspace

```
terraform workspace select <workspace_name>
```

To delete workspace

```
terraform workspace delete <workspace_name>
```

Advantages:

- with workspace we can deploy terraform scripts to different environments in same working director
- To maintain multiple different set of infrastructure resource with same cloud provider.
- We can isolate state files using workspace.

tf.vars vs variables.tf

variables.tf we can have variable block but variable blocks will not work in tf.vars

tf.vars is to create variables whose scope will be at root module level.

we use variables.tf to declare variables and we use tf.vars to assign value to variables.

Terraform refresh

- Terraform refresh will find the updates done in the cloud resources by comparing the changes with state file.
- This won't modify your real remote objects, but it will modify the Terraform state.lock

Terraform import

The terraform import command is used to import existing resources into Terraform.

- we are using terraforming tool in our company to import the existing configurations.

Install terraforming in RHEL/Centos

```
sudo yum install rubygems
sudo gem install terraforming
```

Terraform loops

(count and for_each)

```
resource "null_resource" "loop_simple" {
        count = 2
}

output "loop_out" {
        value = null_resource.loop_simple
}
```

List count example

```
locals {
        names = ["bucket1","bucket2","bucket3","bucket4"]
}

resoruce "null_resource" "names" {
        count = length(local.names)
        triggers = {
                name = local.names[count.index]
        }
}

output "list_out" {
        value = null_resource.names
}
```

List for_each example

```
locals {
        names = ["bucket1","bucket2","bucket3","bucket4"]
}

resoruce "null_resource" "names" {
        for_each = toset(local.names)
        triggers = {
                name = each.value
        }
}

output "list_out" {
        value = null_resource.names
}
```

Map for_each example

```
locals {
        names = {
                bucket1 = "region1"
                bucket2 = "region2"
                bucket3 = "region3"
                bucket4 = "region4"
        }
}
```

```
resoruce "null_resource" "names" {
        for_each = local.names
        triggers = {
                bucket = each.key
                region = each.value
        }
}

output "list_out" {
        value = null_resource.names
}
```

Terraform plan lock state

Terraform will lock your state for all operations that could write state.

This prevents others from acquiring the lock and potentially corrupting your state

Terraform state file locking

- State file locking is a mechanism in terraform where operation on a specific state file is blocked to avoid conflicts between multiple users performing the same operation
- Once the lock from one user is released, then only any other user can operate on that state file after taking a lock on it. This helps in preventing any corruption of the state file.
- It is a backend operation, so the acquiring of lock on a state file in the backend. If it takes more time than expected to acquire a lock on the state file, you will get a status message as an output.

Concept of Terraform Workflow

The workflows of Terraform are built on top of five key steps: Write, Init, Plan, Apply, and Destroy. Nevertheless, their details and actions vary between workflows.

Terraform Workflow

Write – this is where you create changes to the code.

Init – this is where you initialize your code to download the requirements mentioned in your code.

Plan – this is where you review changes and choose whether to simply accept them.

Apply – this is where you accept changes and apply them against real infrastructure.

Destroy – this is where to destroy all your created infrastructure.