***What is Terraform***

* ***Terraform is a tool*** *that* ***helps you create and manage your computer infrastructure****, like* ***servers, databases, and networks, in a way that's easy to understand and repeatable.***
* *It uses code to define your infrastructure, which means you can treat your infrastructure like software, allowing you to automate its creation and changes*.
* *Terraform is an* ***open-source infrastructure****, as code software tool that provides a consistent* ***CLI workflow*** *to manage hundreds of cloud services*. *Terraform codifies cloud APIs into declarative configuration files.*

**Basic example:**

* *Let's say you want to set up a web server on a cloud platform like Amazon Web Services (AWS).*
* *Instead of* ***manually clicking through the AWS console******to create the server****, you can use* ***Terraform to write a simple configuration file*** *that* ***describes the server's specifications****, like* ***its size, operating system, and security settings****.*
* *By using* ***Terraform, you can automate the provisioning and management of your infrastructure, making it easier to scale, update, and maintain your applications and services.***

**First EC2 Instance using Terraform:**

**AWS Account, Terraform Installation**

* *We required 3 important things How to Authenticate to AWS.*
* *Which region?*
* *Resource Type to launch*

***Discussion On Terraform Statefile, backup of state file and GitHub repo and Destroy target flag:***

* *Terraform init*
* *terraform init -upgrade #if you want upgrade or degrade version*
* *Terraform validate*
* *Terraform plan*
* *Terraform apply*
* *Terraform destroy*
* *Terraform apply -refresh-only*
* *Terraform apply –auto-approve #It will not ask you confirmation before applying*
* *aws configure and aws configure list*

*#****If you want delete target resource you can use below command:***

*terraform destroy -target [resource].[resource Name]*

***terraform destroy -target github\_repository.sample-tf***

***Terraform Provider:***

* *Without provider terraform cannot manage any infrastructure.*
* *Provider are distributed separately from terraform.*
* *Each provider has its own release cycles and version number.*

***#USerdata***

***#!/bin/sh***

***yum update -y***

***yum install docker -y***

***service docker start***

***docker pull nginx:alpine***

***docker run -it -d –name my-first-container -p 80:80 nginx:alpine***

***Note:*** *If you don’t give default values in variable file; it will ask you while performing the output.*

***Note:*** *if you don’t have permission to modify the file; you can overwrite by using below command*

*Terraform plan -var instance-type=”t2.large”*

*Terraform plan -var instance-type=”t2.large” -out myplan.plan*

*Terraform apply myplan.plan*

***Commenting:***

*# this is single line comment and // also*

*# /\*  end \*/ Multiline comment*

***Set environment variable:***

*Terraform searches the environment of its own process for environment variables named TF\_VAR\_ followed by the name of a declared variable.*

$ export TF\_VAR\_image\_id=ami-abc123

$ terraform plan

...

***Environment variables***

* *The terraform.tfvars file, if present.*
* *The terraform.tfvars.json file, if present.*
* *Any \*.auto.tfvars or \*.auto.tfvars.json files, processed in lexical order of their filenames.*
* *Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)*

***Implement variable type as String, number, list, and Map:***

/\* Discussing on variable types of list, map, number

variable "instance\_type" {

  description = "type of instance"

  type = list(string)

  default     = ["t2.micro", "t2.nano", "t2.micro"]

}

# variable list;   instance\_type = var.instance\_type[1] ----> you can mention in main.tf file

\*/

/\*

Discussing on variable Map:

variable "instance\_type" {

  description = "type of instance"

  type = map(string)

  default     = {

    "dev" = "t2.micro",

    "QA" = "t2.nano",

    "test"  = "t2.micro"

    }

}

instance\_type = var.instance\_type["dev"] ----> you can mention in main.tf file

\*/

***Terraform meta-arguments implementing for\_each with maps and set of strings:***

*Resource Meta-Arguments can be used with any resource type to just change the behavior of resources.*

1. *Count*
2. *For\_each*
3. *Depends\_on*
4. *Lifecycle*
5. *Provider*
6. *Provisioners and connections*

***Example 1****: Creation of s3 bucket for environment (****dev, qa and prod****) with for\_each* ***(maps****)*

resource "aws\_s3\_bucket" "mys3perenv" {

for\_each = {

"dev" = "dev-s3-bucket-00741"

"qa" = "qa-s3-bucket-00741"

"prod" = "prod-s3-bucket-00741"

}

bucket = "${each.key}-${each.value}"

acl = "private"

tags = {

value = each.value = each.key

env

s3-bucket-name = "${each.key}-${each.value}"

}

}

***Example2:*** *Creation of multiple IAM user with for\_each (set of strings => toset)*

#IAM creation using for\_each (set of strings)

resource "aws\_iam\_user" "iamuser" {

for\_each = toset (["AKumar", "VGupta", "RKumar", "UmeshSingh"])

name = each.key

#name=each.value

}

***Resource Meta\_argumnets-Lifecycles:***

1. ***Create\_before\_destroy:*** *By default, first resource get destroy and after that desired resource get created.*

resource "aws\_instance" "first-tfserver" {

  ami           = var.aws\_ami

  instance\_type = var.instance\_type

availabilitiy\_zone= “ap-south-1a”

#availabilitiy\_zone= “ap-south-1b”

  count = var.instance\_count

  tags = {

    Name = "First\_TF\_server-${count.index}"

  }

  user\_data = file(var.userdata\_file\_path)

  lifecycle {

    create\_before\_destory = true

  }

}

1. ***Prevent\_destroy:*** *Safety against the accidental deletion of resources.*

resource "aws\_instance" "first-tfserver" {

  ami           = var.aws\_ami

  instance\_type = var.instance\_type

availabilitiy\_zone= “ap-south-1a”

#availabilitiy\_zone= “ap-south-1b”

  count = var.instance\_count

  tags = {

    Name = "First\_TF\_server-${count.index}"

  }

  user\_data = file(var.userdata\_file\_path)

  lifecycle {

    prevent\_destroy = true

  }

}

1. ***Ignore\_change:*** *Not very useful for production env.*

resource "aws\_instance" "first-tfserver" {

  ami           = var.aws\_ami

  instance\_type = var.instance\_type

availabilitiy\_zone= “ap-south-1a”

#availabilitiy\_zone= “ap-south-1b”

  count = var.instance\_count

  tags = {

    Name = "First\_TF\_server-${count.index}"

  }

  user\_data = file(var.userdata\_file\_path)

  lifecycle {

    ignore\_changes = [

tags,

]

  }

}

***Resource Meta\_Arguments Depends\_on:***

*To handle hidden resource or module dependencies that terraform cannot automatically Infer.*

resource "aws\_eip" "lb" {

  instance = aws\_instance.first-tfserver.id

  vpc= true

  depends\_on =[

    aws\_internet\_gateway.vpc-dev-igw

  ]

}

***Terraform Data\_resource:***

*Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.*

*A data source is accessed via a special kind of resource known as a data resource, declared using a data block*

variable "ami-id" (

description = "This variable is used to define ami id"

default = "ami-076754bea03bde973"

}

resource "aws\_instance" "ec2" {

ami = var.ami-id.id

instance\_type = var.instance-type

tags = {

Name = "My-New-Instance"

}

}

***Management using S3 and DynamoDB:***

***Terraform state management:***

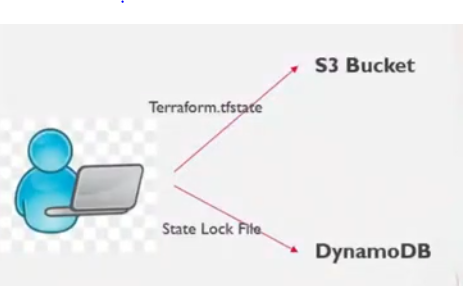
*By Default, terraform implicitly uses a backend called local to store state as a local file on disk.*

*Storing state file in local disk will not allow collaboration.*

*We can store terraform file in git, this is fine.*

*But what about state file??? We cannot store it in central GIT repo, we need to use Central backed like S3 (AWS) or others like consul, Kubernetes, HTTP, ETCD etc.*

*There is multiple backend supported by terraform.*



* *State Management using S3 and DynamoDB*
* *By Default, S3 does not support state locking functionality*
* *For this we need to make use of DynamoDB to accomplish state locking feature.*
* *Accessing state in a remote service generally require some kind of access credentials.*

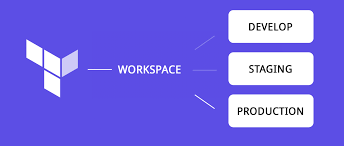
***Terraform has detected you're unconfiguring your previously set "s3" backend.***

*terraform init -migrate-state*

***Terraform Workspace:***

*Terraform Workspace help you keeping* ***your infrastructure consistent and use a single manifest to create many of the same things again and again****. Working with Terraform involves* ***managing collections of infrastructure resources****.*

t2.micro



t2.large

t2.nano

|  |  |
| --- | --- |
| *Terraform workspace -h* | *Terraform help* |
| *Terraform workspace new [name]* | *Creation of workspace* |
| *Terraform workspace delete* | *delete workspace* |
| *Terraform workspace list* | *list workspace* |
| *Terraform workspace select* | *select workspace (switching to another workspace)* |
| *Terraform workspace show* | *Show name of the current workspace* |

resource "aws\_instance" "first-tfserver" {

  ami           = var.aws\_ami

  instance\_type = lookup(var.instance\_type, terraform.workspace)    #you can use different workspaces

  count = var.instance\_count

  user\_data = file(var.userdata\_file\_path)

  tags = {

   Name = "web\_server-${terraform.workspace}"

  }

}

variable "aws\_region" {

  description = "selecting aws region"

  default     = "ap-south-1"

}

variable "instance\_type" {

  description = "type of instance"

  type = map(any)

  default     = "t2.nano"

  dev    = "t2.micro"

  test   = "t2.large"

}

variable "instance\_count" {

  description = "no.of instance"

  default     = 1

}

variable "aws\_ami" {

  description = "Update with valid AMI ID for ap-south-1 region"

  default     = "ami-026255a2746f88074"

}

variable "userdata\_file\_path" {

  default = "userdata-nginx.bash"

}

***Terraform Taint:***

* *The terraform taint command informs Terraform that a particular object has become degraded or damaged.*
* *The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.*

***Use Cases***

* *It could use for re-creating an EC2 instance if someone logged in and made some manual changes.*
* *Marking a resource as Taint, force a rebuild of certain resources without doing a full destroy - though usually only during development phase.*

***Behavior***

* *This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted.*
* *Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.*

*terraform taint aws\_instance.first-tfserver*

*terraform untaint aws\_instance.first-tfserver*

***Terraform Function (String, number, and collection):***

***Terraform Function:***

*The Terraform language includes* ***several built-in functions*** *that you* ***can call from within expressions to transform and combine values.***

*The Terraform* ***language does not support user-defined functions****, and so only the functions built in to the language are available for use.*

***Terraform Functions***

* *Numeric Functions*
* *String Functions*
* *Collection Functions*
* *Encoding Functions*
* *Filesystems Functions*
* *Date and Time Functions*
* *Hash and Crypto Functions*
* *IP Network Functions*
* *Type Conversion Functions*

***Ref:***[*https://developer.hashicorp.com/terraform/language/functions*](https://developer.hashicorp.com/terraform/language/functions)

$ terraform console

> 12+13

25

> 2\*4

8

> min (12,34,4,5)

4

> max(12,34,4,5)

34

> pow(3,2)

9

> trim("?!hello?!", "!?")

"hello"

> trimprefix("helloworld", "hello")

"world"

> trimsuffix("helloworld", "world")

"hello"

> trimspace(" hello\n\n")

"hello"

> join(", ", ["foo", "bar", "baz"])

"foo, bar, baz"

> split(",", "foo, bar, baz")

tolist([

"foo",

"bar",

"baz",

1)

> upper("hello")

"HELLO"

> lower ("HELLO")

"hello"

> concat(["a", ""], ["b", "c"])

[

"a",

"",

"b",

"c",

]

> contains(["a", "b"], "b") #checking which one is belongs to

true

> contains(["a", "b"], "c")

false

> distinct(["a", "b", "c", "d", "a", "b"]) #removing duplicate values

tolist([

"a",

"b",

"c",

"d",

])

> length("helloworld")

10

> length(["a", "b"])

2

> lookup({a="ay", b="bee"}, "a", "what?")

ay

> lookup({a="ay", b="bee"}, "c", "what?")

what?

> merge({a="b", c="d"}, {e="f", c="z"})

{

"a" = "b"

"c" = "z"

"e" = "f"

}

> merge({a="b"}, {a=[1,2], c="z"}, {d=3})

{

"a" = [

1,

2,

]

"c" = "z"

"d" = 3

}

***Terraform provisioners:***

*In Terraform, provisioners are a mechanism used to perform additional actions on the provisioned infrastructure beyond just resource creation.* ***Provisioners are typically used to******initialize, configure, or manage resources after they have been created by Terraform****. Execute scripts and terraform resource*

*There are several types of provisioners available in Terraform:*

***Local-exec Provisioner:*** *This provisioner allows you to* ***execute commands locally on the machine where terraform is running.*** *It is often used for tasks such* ***as running shell scripts or invoking other local commands.***

***Remote-exec Provisioner:******The remote-exec provisioner allows you to execute commands on the provisioned resource over SSH or WinRM****. This provisioner is commonly used to perform tasks such as software installation, configuration, or initialization on remote instances.*

***File Provisioner:*** *The file provisioner* ***allows you to copy files or directories from the local file system to the provisioned resource****. It is useful for tasks such as transferring configuration files or other resources to remote instances.*

***resource "aws\_instance" "example" {***

***ami = "ami-12345678"***

***instance\_type = "t2.micro"***

***provisioner "remote-exec" {***

***inline = [***

***"echo 'Hello, World!' > /tmp/example.txt",***

***"sudo apt-get update",***

***"sudo apt-get install -y nginx"***

***]***

***connection {***

***type = "ssh"***

***user = "ubuntu"***

***private\_key = file("~/.ssh/id\_rsa")***

***host = self.public\_ip***

***}***

***}***

***}***

***Ref:***[*https://developer.hashicorp.com/terraform/language/resources/provisioners/file*](https://developer.hashicorp.com/terraform/language/resources/provisioners/file)

***User\_Data Vs Remote-exec Provisioner?***

* ***User data is typically used for one-time setup tasks during instance bootstrapping****, while* ***remote-exec provisioners are used for ongoing configuration management and administrative tasks on provisioned resources****.*
* *Both user data and remote-exec provisioners have their respective use cases and should be chosen based on the specific requirements of your infrastructure and application deployment.*

***User Data:***

* *User data is a feature provided by cloud providers, such as AWS, Azure, and Google Cloud Platform, that* ***allows you to run scripts or commands during the bootstrapping process of an instance.***
* *In Terraform, user data can be specified within the user\_data attribute of an instance resource block.*
* *User data scripts are typically executed only once, during the instance's initial bootstrapping phase.*
* *User data scripts can be used to perform tasks such as installing software, configuring services, or running initialization scripts on newly launched instances.*
* *User data scripts are ideal for performing one-time setup tasks or for configuring instances in a cloud-native way.*

***Remote-exec Provisioner:***

* *Remote-exec provisioners in Terraform allow you to execute commands or scripts on a provisioned resource (e.g., an EC2 instance) over SSH or WinRM.*
* *Remote-exec provisioners are defined* ***within a resource block using the provisioner block.***
* *Remote-exec provisioners enable you to perform actions on an instance after it has been provisioned by Terraform.*
* *Remote-exec provisioners can be used to perform tasks such as software installation, configuration management, or any other administrative tasks on the provisioned resource.*
* *Unlike user data scripts, remote-exec provisioners can be used to execute commands multiple times, making them suitable for ongoing configuration management or maintenance tasks.*

***Creation & Destroy – Time Failure behavior for provisioners:***

***Provisioners Types:*** *There are 2 primary types of provisioners.*

1. *Creation-Time Provisioner*
2. *Destroy-Time Provisioner*

***Creation-Time Provisioner:***

* *It runs only during creation not during updating or other Lifecyle.*
* *If this fails due to any reason then the resource will be marked as tainted.*

***Destroy-Time Provisioner:***

* *It run before the resource is destroyed*

***Provisioner Failure Behavior:***

*Due to any reason, if provisioners get failed, this will cause the terraform apply as failed and this is the default property.*

***This can be changed by setting the on\_failure settings.***

***Continue =>*** *It will ignore the error and continue with creation or destruction.*

***Fail (default)=>*** *It will raise an error and stop applying. In case of creation-time provisioner, it will mark the resources as taint. So that resource will get replaced in next apply.*

***Fundamental concept of terraform:***

***DRY Principle:***

* *The DRY principle in software development stands for "Don't Repeat Yourself."*
* *It is a fundamental concept that encourages developers to avoid duplicating code or logic in a system.*
* *The goal of following the DRY principle is to reduce redundancy, improve maintainability, and make code easier to understand and modify.*
* *This principle applies not only to traditional software development but also to infrastructure as code (laC) tools like Terraform.*

***Modularization***

* *Instead of writing similar* ***infrastructure code repeatedly, you can create reusable Terraform modules.***
* *Modules* ***allow you to define a set of resources, variables, and outputs in one place*** *and then reuse them in* ***multiple parts of your Terraform configuration****.*
* *This promotes code* ***reusability and reduces duplication.***

resource "aws\_instance" "my-ec2" {

ami = "ami-076754bea03bde973"

instance\_type = “t2.micro”

tags = {

"Name" = "My-EC2"

}

}

module "ec2module" {

source = "../../Modules/ec2"

}

***Variables and Input Parameters***

*Use Terraform variables to parameterize your configurations. Instead of hardcoding values throughout your code, define variables that can be easily changed when needed.*

*This allows you to reuse the same module or configuration with different input values.*

resource "aws\_instance" "my-ec2" {

ami = "ami-076754bea03bde973"

instance\_type tags = {

"Name" = "My-EC2"

}

}

module "ec2module" {

source = "../../Modules/ec2"

instance-type = "t2.medium"

= "var.instance-type"

}

***Locals and Modules in terraform:***

* *if you don't want overwrite instance type in modules, you can use locals; it's prevented the overwritten file.*
* *you can write separate local file or after resource block you can use it.*