Basic Terraform:
Basically, it is used as an Infrastructure management / configuration tool. IAAC: Infrastructure as A Code.
In the client machine, where the terraform will be installed, in that it's better to give one Role of administrator access. The official website of terraform is www.terraform.io
Installation of terraform:
If we install the terraform by using boot-strap method, i.e., by using wget command, it will be downloaded in zipped form, then we have to unzip it by using the command below:
`[unzip terraform……]`
Now it is a binary file that has been installed, so by default all the binary files will be in the path of
`[/usr/local/bin]`
Now, we have to move this file or copy this file in the above path by using the command below:
`[mv terraform /usr/local/bin]`
Now we will make a separate directory or not, and under this we have to write the different .tf files, such as provider.tf, main.tf, variables.tf, and so on.
For provider.tf
We have to search the provider files as per our cloud premises.
For aws, we have to search for [Provider file for aws in terraform]
So now, if we want to create any resource in different aws account, we can simply give the access key and secret access key to undergo with the process.

One more point, whenever we will update any terraform (.tf) file in a particular folder, we have to run **`[terraform init]`** command for initializing the terraform.

After installing:

1. We can write all the **(.tf)** file/s
2. Then we have to give **`[terraform init]`**
3. Even again or whenever we will update or edit anything in any of the **(.tf)** file/s we have to run the **`[terraform init]`** command.
4. Then after executing the infrastructure configuration if we want to see what will get to form, what will the action done by the code/s given in the **(.tf)** files, we can see it by giving the command **`[terraform plan]`**.
5. Now, again we can give **`[terraform validate]`** to see the validation of our code to check the quality of our code/s written in the **(.tf)** files or to check whether if we have any error in our code or not.
6. Then if everything is okay, we can run **`[terraform apply]`** → **`[yes]`** or **`[terraform apply --auto-approve]`**
7. After applying, terraform will create a file automatically known as terraform state file, which will contain the history of the code execution.

```
[root@ip-172-31-89-211 ~]# terraform apply
aws_instance.web: Refreshing state... [id=i-066dc151702113fba]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so
no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
[root@ip-172-31-89-211 ~]# ls -lrth
total 18M
-rw-r--r-- 1 root root  18M Mar  2 19:32 terraform_1.1.7_linux_amd64.zip
-rw-r--r-- 1 root root  223 Mar 12 05:24 provider.tf
-rw-r--r-- 1 root root 3.8K Mar 12 05:30 terraform.tfstate
[root@ip-172-31-89-211 ~]#
```

8. After that if we need to delete the entire thing that we created in our infrastructure, we can give the command as **`[terraform destroy]`** → **`[yes]`** or **`[terraform destroy --auto-approve]`**
9. So, once we destroy the configuration, the state file will be there, but there won't be any content.

10. So, if we want to run the terraform apply command again, it will execute only if the state file is destroyed. If the state file exists, it will say that, already the resource/s have been there. So now, if we want to create the same resource again without destroying the existing one, we have to copy the **(.tf)** files or we can recreate/rewrite the **(.tf)** files in some other path / directory, and then only we can create the same resource again followed by **`[terraform init]`** → `[terraform plan]` → `[terraform validate]` → **`[terraform apply --auto-approve]`**

Some of the basic terraform codes for creating different basic aws resources. To create EC2, VPC. Subnet with different things:

Very Important Note: Once we launch any instance as client machine to run our Terraform codes, we have to attach a role for this EC2 to access the other AWS resources for creating the infrastructure. Or else we can switch to some other's IAM user or by creating an IAM user, then by giving the command [aws configure] followed by access key and secret access key.

TERRAFORM ADV 1:

`[terraform init]` → It will download the plugins needed, modules, and the provider

`[terraform fmt]` → It will check the alignment format of the code
`[terraform validate]` → It will check the syntax of the code.

`[terraform plan]`→ It will show the changes that are going to happen once we apply the code. The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. By default, when terraform creates a plan, it reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
`[terraform apply]` → It will build the infrastructure.

1.1 First topic is Terraform Required Version:

In real time in any organization, they will have one fixed stable version of terraform which they will be using for infrastructure creation.
Anywhere that Terraform lets you specify a range of acceptable versions for something, it expects a specially formatted string known as a version constraint. Version constraints are used when configuring:

- Modules
- Provider requirements
- The `required_version` setting in the terraform block.

A version constraint is a string literal containing one or more conditions, which are separated by commas. Each condition consists of an operator and a version number.

Version numbers should be a series of numbers separated by periods (like 1.2.0), optionally with a suffix to indicate a beta release.
The following operators are valid:

- **`[=]`** (or no operator): Allows only one exact version number. Cannot be combined with other conditions.
- **`[!=]`** Excludes an exact version number.
- **`[>, >=, <, <=]`** Comparisons against a specified version, allowing versions for which the comparison is true. "Greater-than" requests newer versions, and "less-than" requests older versions.
- **`[~>]`** Allows only the *rightmost* version component to increment. For example, to allow new patch releases within a specific minor release, use the full version number: **`~> 1.0.4`** will allow installation of `1.0.5` and `1.0.10` but not `1.1.0`. This is usually called the pessimistic constraint operator.

Example of terraform AWS provider:

```
terraform {
  required_version= "~> 1.2.0"
```

```
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.20.1"
    }
  }
}
terraform {
  required_version = "~> 1.2.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.50.0"
    }
  }
}
```

What are resources in terraform?

Ans: Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records. Resource Blocks documents the syntax for declaring resources.

1.2 Multiple Provider:

So, multiple providers are like providing the information of two or more regions for creating resources in different regions. So, if the scenario is like we have to create different resources in different regions but we have to do it from a single **(.tf)** files, that time we can use the concept of multiple providers for mentioning the different regions with an "Alias" name. Where as in the top, the required_provider is AWS.

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.50.0"
    }
  }
}

# Provider-1 for us-east-1 (Default Provider)
provider "aws" {
  region = "us-east-1"
  profile = "default"
}

# Provider-2 for us-west-1
provider "aws" {
  region = "us-west-1"
  alias = "my-west"
  profile = "default"
}

# Resource Block to Create VPC in us-east-1 which uses default provider
resource "aws_vpc" "vpc-us-east-1" {
  cidr_block = "10.1.0.0/16"
  tags = {
    "Name" = "vpc-us-east-1"
  }
}
```

If no provider with alias name is given, it will catch the default one

```
resource "aws_vpc" "vpc-us-west-1" {
  cidr_block = "10.1.0.0/16"
  provider = aws.my-west
  tags = {
    "Name" = "vpc-us-west-1"
  }
}
```

This resource will be created in "us-west-1"

1.3 Terraform Immutable: (https://youtu.be/II4PFe9BbmE)

The meaning is if we run any (.tf) file/s for the first time, and then again if we do some major change/s in the terraform file, such as changing of the availability zone/s and all, terraform will first delete the existing infrastructure and then it will create the new one as per the codes given in the **(.tf)** file/s

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami               = "ami-079b5e5b3971bd10d"
  instance_type     = "t2.micro"
  #availability_zone = "ap-south-1b"
  availability_zone = "ap-south-1a"
  tags = {
    "Name" = "web"
    #"tag1" = "Update-test-1"
  }
}
```

So, now if we un-tag the second tag, it's a minor change and simply it will add the new tag in the existing infrastructure. But if we un-tag the availability zone, it's a major change, thus, terraform will first delete the existing one and will re-create a new infrastructure indeed.

Cons: It doesn't have any control over the traffic.

Pros: It avoids different risks such as complexity, unstable conditions etc.

1.4 Count in terraform

This is nothing the number of same resource that you want to create in terraform.

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create 5 EC2 Instance
resource "aws_instance" "web" {
  ami           = "ami-079b5e5b3971bd10d" # Amazon Linux
  instance_type = "t2.micro"
  count         = 5
  tags = {
    "Name" = "web"
    #"Name" = "web-${count.index}"
  }
}
```

1.5 For each Map in Terraform

Suppose we want to create multiple resources in a region with different names, for that we can use for each map concept in terraform.

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create 4 S3 Buckets
resource "aws_s3_bucket" "mys3bucket" {

  # for_each Meta-Argument
  for_each = {
    dev  = "my-dapp-bucket-455"           # key -> dev ; value -> "my-dapp-bucket-455"
    qa   = "my-qapp-bucket-455"
    stag = "my-sapp-bucket-455"
    prod = "my-papp-bucket-455"
  }

  bucket = "${each.key}-${each.value}"    # "dev-my-dapp-bucket-455"
  acl    = "private"

  tags = {
    Environment = each.key
    bucketname  = "${each.key}-${each.value}"
    eachvalue   = each.value
  }
}
```

By using for each Map in terraform can we create 3 EC2 instances in a same region but in all the 3 availability zones:

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create 3 EC2 Instances in a same region but all in three availability zones
resource "aws_instance" "my-ec2-vm" {

  # for_each Meta-Argument
  for_each = {
    nm1 = "ap-south-1a"
    nm2 = "ap-south-1b"
    nm3 = "ap-south-1c"
  }

  ami               = "ami-079b5e5b3971bd10d"
```

```
  instance_type     = "t2.micro"
  availability_zone = "${each.value}"

  tags = {
    Name = "${each.key}"
  }
}
```

1.6 For each toset in Terraform

If we want to create 4 resources of similar kind by different, we can use this concept. For example, if we want to create 4 or 5 IAM Users, we can use this concept as given below:

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create 5 IAM Users
resource "aws_iam_user" "myuser" {
  for_each = toset(["TJack", "TJames", "TMadhu", "TDave", "Tmithran"])
  name     = each.key
}
```

1.7 Create_Before_Destroy in Terraform: (Comes under lifecycle)

As terraform is immutable, (.tf) files containing in the same path will destroy the old one and it will build the new resource. But if we add this concept, it will first create the resource and then it will destroy.

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create EC2 Instance
resource "aws_instance" "web" {
  ami               = "ami-079b5e5b3971bd10d" # Amazon Linux
  instance_type     = "t2.micro"
  #availability_zone = "ap-south-1a"
  availability_zone = "ap-south-1b"
  tags = {
    "Name" = "web-1"
  }
  lifecycle {
    create_before_destroy = true
  }
}
```

So, at the end we can add the lifecycle block to availing this feature in terraform. We can add this for any kind of resources we want to build.

1.8 Prevent Destroy in terraform (Comes under lifecycle)

This is nothing but if we add this at the end of the code it won't destroy resource once we create it. It will not even let us to destroy form the console as well. So, after applying this if we run destroy command, it will show an error, and it will say that it can't be destroyed as it has a lifecycle attached with it, which says to prevent destroy. So, for this feature we have to add a lifecycle block at the end of the code, with (prevent_destroy = true) syntax. For example,

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-079b5e5b3971bd10d" # Amazon Linux
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-2"
  }
  lifecycle {
    prevent_destroy = true  # Default is false
  }
}
```

TERRAFORM ADV 2:

1.9 Ignore Changes: (Comes under lifecycle)

This is a process by which we can ignore the manual drift in any of our terraform configuration. Terraform is having a habit, of deleting the new manual drift, when we run the code for the same resource with respect to the (.tf) file/s and state file.

So, now if we want to keep my manual drift/s unchanged we have to add a two lines syntax at the end of the code by mentioning the parameters for which parameter we need this feature and by this, terraform won't change or delete the manual drifts while running the codes for the same resource/s.

Here, any of the Argument Reference that are available here in this link of terraform official document (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance) can be included in the ignore change lifecycle block, so if we do any manual drift in the particular Argument/s mentioned in the ignore change block, will not be changed or destroyed by terraform while running with a new configuration for the same resource/s.

For example:

```
 # Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
```

```
  profile = "default"
}

# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-079b5e5b3971bd10d" # Amazon Linux
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-3"
  }

  lifecycle {
    ignore_changes = [
      # Ignore changes to tags, e.g., because a management agent
      # updates these based on some ruleset managed elsewhere.
      tags,    # here we can include any arguments, in this case we have used only 'Tags'
    ]
  }
}



lifecycle {
    ignore_changes = [tags, any arguments]
  }
```

So, if we run this code and if also in the existing ec2 instance in the console we had added or modified our tags by manual drifting, terraform will not change or delete the new manually created tags in this ec2 instance.

1.10 Depends on in Terraform:

So, its like if we have any dependency for creating a resource at that time, we can use this 'depends on' in the last line of the required resource block in the terraform code to execute it.

For example, I am going to create one EC2 instance, one VPC, and one IAM role by using a same (.tf) file. And I also want my EC2 instance should have the same IAM role attached in it and also the same for VPC. So as per the console we have to create the VPC and IAM role first then after this we have to launch the EC2 instance followed by attaching the IAM and VPC. So, here, the resource EC2 has the dependency on VPC and IAM role, Thus, if we add Depends_On syntax at the end of the EC2 resource block, terraform will first create the VPC and IAM role then it will create the EC2 resource.

Generally, it is used to maintain some linearity in the resource creation in terraform cause, by default terraform will read the **(.tf)** file/s parallelly and will start creating the resource/s in a parallel manner too. So, one example of such **depends on** syntax are as follows:

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = "ap-south-1"
  profile = "default"
}

# Resource-1: Create VPC
resource "aws_vpc" "vpc-dev" {
  cidr_block = "10.0.0.0/16"
  tags = {
    "Name" = "vpc-dev"
  }
}

# Resource-2: Create IGW
```

```
resource "aws_internet_gateway" "vpc-dev-igw" {
  vpc_id = aws_vpc.vpc-dev.id
  depends_on = [aws_vpc.vpc-dev]
}

# Resource-3: Create Elastic IP
resource "aws_eip" "my-eip" {
  vpc      = true
  # Meta-Argument
  depends_on = [aws_internet_gateway.vpc-dev-igw]
}
```

2.1 Basic variables concept in terraform:

This is nothing but, in the terraform code, in resource block or in provider block, we can pass the parameters by using some variables. i.e., we will have a separate variable block in our terraform code which will be automatically triggered by the terraform while creating the resource/s as per the provided variable syntax.

So, in real time we will keep the (.tf) files in an organized manner. Instead of a single terraform code file we will have multiple. One file for the main code i.e., the code for creating the resource/s; one for provider block; one for variables and so on. By default, terraform will read all the (.tf) files and will consider these as a one file followed by creation of the resource in an organized manner.

So, the format of the variable file is given below:

```
# Input Variables
variable "aws_regions_mumbai" {
  description = "Region in which AWS resources to be created"
  type        = string
  default     = "ap-south-1"
}

variable "ec2_ami_id" {
  description = "AMI ID"
  type        = string
  default     = "ami-079b5e5b3971bd10d" # Amazon2 Linux AMI ID
}

variable "ec2_instance_count" {
  description = "EC2 Instance Count"
  type        = number
  default     = 1
}
```

The format of the provider file is given below:

```
# Terraform Block
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  region  = var.aws_regions_mumbai
  profile = "default"
}
```

And finally, we have the main file which is having the code for creating the resources:

```
# Create Security Group - SSH Traffic
resource "aws_security_group" "vpc-ssh" {
  name        = "vpc-ssh"
  description = "Dev VPC SSH"
  ingress {
    description = "Allow Port 22"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
```

```
    description = "Allow all IP and Ports outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# Create Security Group - Web Traffic
resource "aws_security_group" "vpc-web" {
  name        = "vpc-web"
  description = "Dev VPC Web"

  ingress {
    description = "Allow Port 80"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "Allow Port 443"
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    description = "Allow all IP and Ports outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami                    = var.ec2_ami_id
  instance_type          = "t2.micro"
  count                  = var.ec2_instance_count
  user_data              = <<-EOF
    #!/bin/bash
    sudo yum update -y
    sudo yum install httpd -y
    sudo systemctl enable httpd
    sudo systemctl start httpd
    echo "<h1>Welcome to GreensTechnology ! AWS Infra created using Terraform in ap-south-1 Region</h1>" >
/var/www/html/index.html
    EOF
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "myec2vm"
  }
}
```

So, in the variable file the syntax that are important are:

```
variable "aws_regions_mumbai" {   # here we have to give the same that we are going to use in the code [like
our alias name, but should be in a format with ( _ )]
  description = "Region in which AWS resources to be created"    # optional
  type        = string    # should be given, this will help us to know the type of variable, is it of string
type or any number or any list etc.
  default     = "ap-south-1"    # the value of the variable, that we need to use
}
```

And in the code and elsewhere, where we are going to use the variable, we have to write it exactly what we wrote just beside the variable "***" prefixed by "var.". For example,

```
# Provider Block
provider "aws" {
  region  = var.aws_regions_mumbai
  profile = "default"
}
```

1. CLI (Highest Priority)
2. System Env. Variable or Environment variable (Medium Priority)
3. (.tfvars) file
4. Default (Lowest Priority)

2.2 Basic of variables when prompted in terraform:

Now, suppose in the variable file we didn't mention the default value while planning or creating, terraform will ask for those variables to put the value that we want. Otherwise terraform will say that, as the values are not there I can't even plan or create the required resource/s.

```
# Input Variables
variable "aws_region" {
  description = "Region in which AWS resources to be created"
  type        = string
  #default     = "ap-south-1"
}

variable "ec2_ami_id" {
  description = "AMI ID"
  type        = string
  #default     = "ami-079b5e5b3971bd10d" # Amazon2 Linux AMI ID
}

variable "ec2_instance_count" {
  description = "EC2 Instance Count"
  type        = number
  #default     = 1
}

# Assign When Prompted using CLI
variable "ec2_instance_type" {
  description = "EC2 Instance Type"
  type = string
}
```

Here, the default values are commented, so now, if we give plan or apply, it will ask for the values of region, AMI ID, Count of EC2 instance/s, and type of the ec2 instance/s.
So, this concept is called as Variables when prompted.

2.3 Variables overwrite with CLI in Terraform:

The concept of this is, when we put some variable/s and the default value/s of the same as well, even though we can change that default value by using some addons with the CLI command while running the command for terraform plan or terraform apply. Meaning, the priority of the CLI command is more than that of the default valued attached with the variable code in (.tf) file/s or variable.tf file.
Let us see about the priority of the variables with the sequence. (Variable Precedence)
Meaning, which way of providing variable/s will be considered as more important that the other method of providing.

```
# Input Variables
variable "aws_region" {
  description = "Region in which AWS resources to be created"
  type        = string
  default     = "ap-south-1"
}

variable "ec2_ami_id" {
  description = "AMI ID"
  type        = string
  default     = "ami-0915bcb5fa77e4892" # Amazon2 Linux AMI ID
}

variable "ec2_instance_count" {
  description = "EC2 Instance Count"
  type        = number
  default     = 2
```

```
}

variable "ec2_instance_type" {
  description = "EC2 Instance Type"
  type = string
  default = "t3.micro"
}
```

Suppose now we have this code in variable.tf file and the provider.tf and main.tf files are having my other codes fitted as such. But while running the plan command I run the command as:

[terraform plan --var="ec2_instance_type=t3.large" --var="ec2_instance_count=1”]

Where, we have given some CLI commands for the instance type as t3.large and count as 1. While in the variable.tf file we have given type as t2.micro and count as 2. But CLI having the more priority terraform will execute what we given in the CLI format.

So, the format giving the command is:

[terraform plan / apply --auto-approve --var=“name_of_the_variable=Value” --var=“name_of_the_variable=Value”]

1. CLI (Highest Priority)
2. System Env. Variable or Environment variable (Medium Priority)
3. (.tfvars) file
4. Default (Lowest Priority)

If default is also not given, they will ask for the variable/s while planning or applying.

2.4 Variables overwrite with Env. in Terraform:

Now let us see about Linux environment variable:

In Linux if we give **[printenv]** → it will show me all the environment variables present in our Linux machine.

So, we can also create any environment variables in our Linux Environment by giving the command as:

[export variable=value] / [export age=60]

So, now if we give **[printenv]** again we can see this newly created environment variable “age=60” anywhere in the list.

So now, if I run **[echo “Hello my age is $age”]** → the output will be **“Hello my age is 60”**

Now, if we want to delete the Env. Variable that we created or any other Env. Variable, the command is

[unset <variable name>] / [unset age]

Now again if we run **[printenv]** we won’t see the age variable in the list, meaning, it has been deleted.

Now, if we want to set terraform variables in our default client machine in Linux format, we can also do that. And from that also terraform will pick it and will execute the environment variables with more priority than the default variables.

But if we want our Linux environment variables to be executed by terraform the variable name should be given with a special formatted syntax, and the same has been shown below:

[export TF_VAR_<variable name with ‘_’ at each space>=value of the variable]
[export TF_VAR_variable_name=value]

For example:

```
[export TF_VAR_ec2_instance_count=1]
[export TF_VAR_ec2_instance_type=t3.large]
```

Now even though we have a variable.tf file as given bellow, terraform will pick the env_variables while planning or applying command will be given.

```
# Input Variables
variable "aws_region" {
  description = "Region in which AWS resources to be created"
  type        = string
  default     = "ap-south-1"
}
```

```
variable "ec2_ami_id" {
  description = "AMI ID"
  type        = string
  default     = "ami-0915bcb5fa77e4892" # Amazon2 Linux AMI ID
}

variable "ec2_instance_count" {
  description = "EC2 Instance Count"
  type        = number
  default     = 2
}

variable "ec2_instance_type" {
  description = "EC2 Instance Type"
  type = string
  default = "t3.micro"
}
```

Now, if we run plan or apply command terraform will take the instance count as 1 and instance type as t3.large. Because this has been mentioned in the env.variables of Linux machine with terraform format (TF_VAR_<variable name>)
Again, if we want terraform to take the default variable/s value/s we can unset/delete the environment variables from Linux machine by:
[unset TF_VAR_ec2_instance_count]
[unset TF_VAR_ec2_instance_type]

```
# Sample
export TF_VAR_variable_name=value

# SET Environment Variables
export TF_VAR_ec2_instance_count=1
export TF_VAR_ec2_instance_type=t3.large
printenv
echo $TF_VAR_ec2_instance_count, $TF_VAR_ec2_instance_type

# Initialize Terraform
terraform init

# Validate Terraform configuration files
terraform validate

# Format Terraform configuration files
terraform fmt

# Review the terraform plan
terraform plan

# UNSET Environment Variables after demo
unset TF_VAR_ec2_instance_count
unset TF_VAR_ec2_instance_type
printenv
echo $TF_VAR_ec2_instance_count, $TF_VAR_ec2_instance_type
```

1. CLI (Highest Priority)
2. System Env. Variable or Environment variable (Medium Priority)
3. (.tfvars) file
4. Default (Lowest Priority)

2.5 Variables with (.tfvars) file in terraform:

So, this is another type of file we use to use along with (.tf) files but unlike (.tf) files it will be in the format of (.tfvars)

So, here in this along with all the (.tf) files, such as main.tf, provider.tf, and variables.tf files, they will also have a separate file called as **<something>.tfvars** / **terraform.tfvars** file which will have all the variables listed over their along with the required value/s For example:

```
ec2_instance_count      = “1”
ec2_instance_type       = "t3.small"
aws_region              = "us-west-1a"
```

So now, if we execute these files followed by plan & apply, **terraform will consider the variables given in the (.tfvars) file instead of default values mentioned in the variables.tf file**

So, in real time instead of using **CLI [--var=“<name_of_the_variable>=<value>”]** or **environment variables [TF_VAR_<name_of_the_variable>=<Value>]**; they will have a separate (.tfvars) file, which will have all the variables listed over there along with the values in the format viz. [**name_of_the_variable=“Value”**]

1. CLI (Highest Priority)
2. System Env. Variable or Environment variable (Medium Priority)
3. (.tfvars) file
4. Default (Lowest Priority)

2.6 Variables with multiple (.tfvars) file in terraform:

So now, in real time, in one directory path, we may have multiple (.tfvars) file along with (.tf) files as we have lot of environments, or for different purpose such as infra for static content or dynamic content etc.

So, while running the terraform commands we have to mention the terraform by using CLI which (.tfvars) file it has to pick and inject the value/s in to the code of (main.tf), and for the other variables which are not mentioned into those (.tfvars) file, terraform will choose the default value which is given in (variable.tf) file, if not given, it will ask for an input while planning of applying.

So, the format of giving the CLI is:

```
[terraform plan --var-file=<name.tfvars>]
[terraform apply --var-file=<name.tfvars> --auto-approve]
```

Now there also we have priority, when we have multiple (.tfvars) files, and [#if any of those name is (terraform.tfvars), by default terraform will pick the values present in (terraform.tfvars) file] when nothing is mentioned in the CLI for selecting any (.tfvars) file during planning or applying, terraform will take the default value from the **variable.tf** file

So, the priority became like that:

1. CLI (Highest Priority)
2. System Env. Variable or Environment variable (Medium Priority)
3. [--var-file=<name.tfvars>] (CLI) (if nothing has given it will pick up the default value from variable.tf file)
4. (.tfvars) file [when there is one (.tfvars) file]
4. Default (Lowest Priority)

2.7 Variables with multiple (.auto.tfvars) file in terraform:

So it’s also a same (.tfvars) file format only, but the name is like (name.auto.tfvars)

In this case if we have all the variables.tf, (.tfvars), and (.auto.tfvars) files terraform will read both the (.tfvars) and (.auto.tfvars) files and will execute the value/s of (.auto.tfvars) file in the plan or apply commands. So, only the naming format of the files are different.

2.8 Variables with list (type) in terraform:

In variables.tf file we have seen something as type: where we give either string, number, and here we can also give the type as list, map, and files too.

Now, we will see about the variables with list type in terraform and how it works as well.

So, here in the list type, we have to give the variable in the form of list as shown below:

```
# Input Variables
variable "aws_region" {
  description = "Region in which AWS resources to be created"
  type        = string
  default     = "us-east-1"
}

variable "ec2_ami_id" {
  description = "AMI ID"
  type        = string
  default     = "ami-0915bcb5fa77e4892" # Amazon2 Linux AMI ID
}

variable "ec2_instance_count" {
  description = "EC2 Instance Count"
  type        = number
  default     = 2
}

variable "ec2_instance_type" {
  description = "EC2 Instance Type"
  type = list(string)
  #             0           1           2
  default = ["t3.micro", "t3.small", "t3.large"]
}
```

So now, in our file of resource block (main.tf) we have to mention the variable as: **`(var.ec2_instance_type[1])`**. For example,

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami                    = var.ec2_ami_id
  instance_type          = var.ec2_instance_type[0]
  key_name               = "terraform-key"
  count                  = var.ec2_instance_count
```

So, as per the list what we give, terraform will pick up that variable value and it will execute the same. Such as `[0] → t3.micro, [1] → t3.small, and [2] → t3.large`

Thus, in the main.tf file we have to mention the list number as such in the format of **`(var.variable_name[no.])`**

2.9 Variables with map (type) in terraform:

This concept is quite similar as for each map concept, there in that we gave the syntax like such in the same main.tf file:

For_each = {
  Key1 = “value1”
  Key2 = “value2”
  Key3 = “value3”
}

But here in this case, we will give the for each map in the variable.tf file and the same will be mentioned in the resource block/s in the format of **`(var.variable_name[key_name])`**

And in the variable.tf file, we will give the variables in map format of “key” = “value”

So, here in this case, the description will be our choice; and the type will be map(string)

```
variable "ec2_instance_tags" {
  description = "EC2 Instance Tags"
```

```
  type = map(string)
  default = {
    "Name" = "ec2-web"
    "Tier" = "Web"
  }
}

variable "ec2_instance_type_map" {
  description = "EC2 Instance Type"
  type = map(string)
  default = {
    "small-apps" = "t3.micro"
    "medium-apps" = "t3.medium"
    "big-apps" = "t3.large"
  }
}
```

And in the resource block we have to map the variables like such:

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami                    = var.ec2_ami_id
  #instance_type         = var.ec2_instance_type[0] (this is for variables with list type)
  instance_type = var.ec2_instance_type_map["big-apps"]
  key_name               = "terraform-key"
  count                  = var.ec2_instance_count
```

2.10 Variables with file in terraform:

This is a concept where we can add some installation package in the form of (.sh) file by sharing some shell scripts.

For example, in the resource block of EC2 creation, we can add the user data with `[<<-EOF]` like such:

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami                    = var.ec2_ami_id
  instance_type          = var.ec2_instance_type[0]
  key_name               = "terraform-key"
  count                  = var.ec2_instance_count
  user_data              = <<-EOF
    #!/bin/bash
    sudo yum update -y
    sudo yum install httpd -y
    sudo systemctl enable httpd
    sudo systemctl start httpd
    echo "<h1>Welcome to GreencTechnology ! AWS Infra created using Terraform in ap-south-1 Region</h1>"
> /var/www/html/index.html
    EOF
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "myec2vm"
  }
}
```

Instead of the above way, we can also add the user data context in a separate (.sh) file in the form of shell script given below:

```
#! /bin/bash
sudo yum update -y
sudo yum install -y httpd
sudo systemctl enable httpd
sudo service httpd start
echo "<h1>Welcome to GreensTechnology ! AWS Infra created using Terraform in ap-south-1
Region</h1>" | sudo tee /var/www/html/index.html
```

And the same in the resource block file [in (main.tf) file], we have to mention / pass it as a variable like as `file("file path location / name of the file if in same path")`, then Terraform

will automatically inject it in the cade followed by planning or applying it for resource creation. Example has given below:

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami           = var.ec2_ami_id
  instance_type = var.ec2_instance_type
  key_name      = "terraform-key"
  count = var.ec2_instance_count
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "web"
  }
}
```

2.11 Variables with secrets in terraform:

Suppose there are some resources that need secret information to set such as username and password or access key or secret access key. In that case we will use this concept. So, the concept is quite similar to that of (.tfvars) file. Here as we told earlier, along with all the (.tf) files we will also have lot of (.tfvars) file as well. Among them, **`[secrets.tfvars]`** is also one of the (.tfvars) file. But here the one difference is in the variable.tf file we won't give any default value, and instead of this we will use one term / syntax called as **`[sensitive = true]`**

Let's see some examples of this condition and how it differs in the different terraform files.

Main.tf file:

```
# Warning: Never check sensitive values like usernames and passwords into source control.
# Create RDS MySQL Database
resource "aws_db_instance" "db1" {
  allocated_storage   = 5
  engine              = "mysql"
  instance_class      = "db.t2.micro"
  name                = "mydb1"
  username            = var.db_username       # The same syntax has to maintain in both variable.tf and
secrets.tfvars files
  password            = var.db_password       # The same here too
  skip_final_snapshot = true
}
```

Variable.tf file:

```
# Input Variables
variable "aws_region" {
  description = "Region in which AWS Resources to be created"
  type        = string
  default     = "us-east-1"
}

variable "db_username" {
  description = "AWS RDS Database Administrator Username"
  type        = string
  sensitive   = true
}

variable "db_password" {
  description = "AWS RDS Database Administrator Password"
  type        = string
  sensitive   = true
}
```

Secrets.tfvars file:

```
db_username = "admin"
db_password = "admin12345"
```

now, when we run the terraform command for applying or planning, it will ask for the sensitive value (like as when we don't give the default value/s in the variable.tf file)
So, now if we want our terraform to take the values from the [secrets.tfvars], then we have to apply the concept of multiple (.tfvars). So, while running the terraform command we have to pass **`[--var-file=<secrets.tfvars>]`**

So, whenever we give **`sensitive = true`** in the **variable.tf file**, after applying, terraform won't show it in the state file too as well as not in the plan also. So, in the output, in the place of username and password it will show as **`'sensitive'.`**

So, here the important thing is, in real time, these secrets.tfvars file won't be tracked by GIT as well. It will be included in (.gitignore). So, all our files will go the remote repo except our secret files.

And in the terraform state file as well, that variables will be shown as **`'sensitive'.`**

TERRAFORM ADV 3:

2.12 Outputs in terraform:

So, in this concept we can get the output of the parameters of the resource/s created by the terraform, other than this everything is same as previous, only a separate code is required for getting the output. Meaning, in the example of EC2 instance launching, there are lot of things in EC2 which we can see while terraform is planning or applying the command to create the resource viz. public_IP, private_IP, secuirity_group, instance_type, instance_id, like that a lot. So, after creation of the resource/s if we need information of any of the parameters of that / those resource/s terraform will give the same after creating the resource. [This can't be achieved in terraform plan command].

And later we can also direct the output in a separate file path for any other future usage. Let us discuss a small example of that:

Suppose, I am going to create/launch 5 EC2 instances by using terraform, so after creation suppose I need the public IP and private IP of the instances as output, so for this, we have to include the required parameters that we need as an output in output.tf file as an organized manner. So, the format of the output.tf file is given below:

```
# Define Output Values
# Attribute Reference: EC2 Instance Public IP
output "ec2_instance_publicip" {
  description = "EC2 Instance Public IP"
  value = aws_instance.my-ec2-vm.public_ip
}

# Argument Reference: EC2 Instance Private IP
output "ec2_instance_privateip" {
  description = "EC2 Instance Private IP"
  value = aws_instance.my-ec2-vm.private_ip
}

# Argument Reference: Security Groups associated to EC2 Instance
output "ec2_security_groups" {
  description = "List Security Groups associated with EC2 Instance"
  value = aws_instance.my-ec2-vm.security_groups
}
```

So, here in the above code, we wrote the code for getting the output of private IP, public IP, and security group list/s. The value section of the code should be followed by the required format. The format is in the resource block what we mentioned the same should be written here separated by a **`'.'`** followed by the parameter name with **`'_'`** in required place ***[the same we see in the terraform plan list]*** prefixed with unspaced **`'.'`**.

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami                    = var.ec2_ami_id
  instance_type = var.ec2_instance_type
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "web"
  }
}
```

Here, it has written as "**aws_instance**" "**my-ec2-vm**". The same should be written in the output.tf file in the place of **value = aws_instance . my-ec2-vm . parameter_name**

For example, if in the resource block (main.tf file) we had mentioned like such:

```
resource “aws_s3” “my-S3-buck”
```

Then in the **output.tf** file, the same should be written followed by the required parameter names, like such:

```
#argument: bucket name/s
output “s3_bucket_name” {
      description =     “list the name of the bucket/s”
      value       =     aws_s3.my-S3-buck.bucket_names
}
#argument: security group/s
output “s3_security_group” {
      description =     “list the security group associated with the bucket/s”
      value             =     aws_s3.my-S3-buck.security_groups
}
```

One use-case of this output feature: If we make a separate file by listing the output of private or public IP of the instances created by terraform, the same can be used by the ansible as s host/inventory file in jenkins CD pipeline.

2.13 Locals in terraform:

One interview question: What is the difference between a local and a variable?

So, both variables and locals serve the same purpose, both are used to pass some inputs/variables. But variables we are using in an organized manner in separate (.tf) files, where as locals we can use as a lower precedence to pass some variables locally in the terraform files.

One example is:

In variable.tf file we have some variables like such:

```
# App Name S3 Bucket used for
variable "app_name" {
  description = "Application Name"
  type = string
  default = "zomato"
}

# Environment Name
variable "environment_name" {
  description = "Environment Name"
  type = string
  default = "dev"
}
```

And in my resource block (main.tf) file we have the block as such:

```
# Create S3 Bucket - with Input Variables & Local Values
resource "aws_s3_bucket" "mys3bucket" {
  bucket = "${var.app_name}-${var.environment_name}-bucket"
  tags = {
    Name = "${var.app_name}-${var.environment_name}-bucket"
    Environment = var.environment_name

  }
}
```

So, my bucket name will be “zomato-dev-bucket”, now suppose like this I have multiple blocks where the same string is present and for any purpose, I need my bucket name as “zomato-dev-buck”. Now what I have to do is, in each and every place in the resource blocks I have to go and change the bucket as buck.

In this case we can use the locals concept to change the variable in a singled syntax code in locals block, locally in any terraform file/s. So, now in the locals block if I change the bucket as buck, in all the resource blocks, the name will be changed into “zomato-dev-buck” where ever this string is present in the file.

For example,

```
# Define Local Values
locals {
  bucket-name = "${var.app_name}-${var.environment_name}-bucket" # Complex expression
}

# Create S3 Bucket - with Input Variables & Local Values
resource "aws_s3_bucket" "mys3bucket" {
  bucket = local.bucket-name
  acl = "private"
  tags = {
    Name = local.bucket-name
    Environment = var.environment_name
  }
}
```

2.14 Data sources in terraform: (Extremely Important)

This is one of the famous features in terraform by which we can fetch the information of an existing infrastructure in terraform.

For example, suppose we had already an existing VPC in our console. Now we want to fix some subnet groups in the same VPC, OR suppose we have an existing EC2 instance and now we want to fix some security group in the same EC2 instance, for that we can use this data source concept to get the details of the existing resource for doing some addon works.

So, in the data source block we have to give the exact name format of the resource/s as mentioned in the terraform registry. For example: for EC2 → aws_ec2; for VPC→ aws_vpc and so on.

So, the format of the data source block can be given as:

```
variable "vpc_id" {
  description = "Will get VPC ID"
  type = string
}

data "aws_vpc" "selected" {
  id = var.vpc_id
}

resource "aws_subnet" "example" {
  vpc_id            = data.aws_vpc.selected.id
  availability_zone = "ap-south-1a"
  cidr_block        = "10.0.1.0/24"
}
```

3.1 State in terraform (regarding state file)

remote_state→ to store the state file of the terraform in some remote place such as s3 or github.

state locking→ While one user will apply the commands, the state file will be locked so that other user can't able to apply the command.

Interview question: Where do you keep your state file / terraform state file?

Ans: In S3 bucket we will store the state files and the bucket will be mandatorily enabled with bucket versioning. We also have access control list configured with the S3 bucket so that who can modify the bucket who can able to delete the content from the bucket etc.

So now, we can also check the backup of the state file once we run the destroy command as:

`[cat terraform.tfstate.backup]` → this will sow only one step before backup only.

So now, to save the terraform state file in remote place we have to run the code in the (.tf) file, mainly in the provider.tf file, just in the terraform block under terraform and after required_providers in a section of backend. Then we have to give the provider block. For example,

```
# Terraform Block
terraform {
  required_providers {
    aws = {
```

```
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
  # Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket = "my-terraform-states-2106"
    key    = "dev/terraform.tfstate"
    region = "ap-south-1"
  }
}

# Provider Block
provider "aws" {
  region  = var.aws_region
  profile = "default"
}
```

So, the format is:

backend "where to store?" (if S3 bucket, then)

bucket = "bucket name"

key = "path inside the bucket if any folder in which we need to store/<terraform state file name; i.e., terraform.tfstate>".

For ex.: If inside the bucket, dev is one folder and inside the 'dev' we need there to store the state file, then the path should be given in the key as,

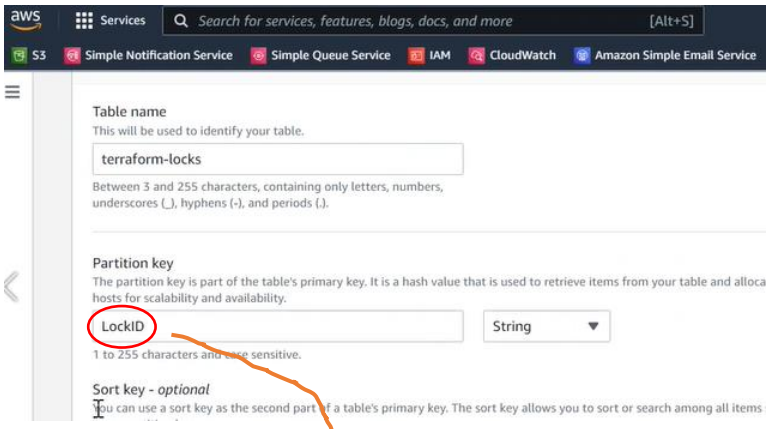
key = "dev/terraform.tfstate"

region = "region of the bucket"

3.2 State locking in terraform:

The concept is, suppose one user is applying the command of [terraform apply] and if at the same time another user run the same command, a conflict may occur. Thus, state locking is the concept by virtue of which when one user is running the apply command automatically the state file will be locked so that any other user can't able to run the apply command. If any other user run the apply command it will show that the state file is locked because, someone else is applying the command.

So, to enable the state locking we need the help of one more tool, that is nothing but DynamoDB. Now, we have to create a table in dynamodb where we have to give only the table name and the partition key and other things as default.



And in the provider.tf file, along with backend we have to give the details for state-locking as well in such a manner:

```
# Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket = "terraform-buck"
    key    = "dev/terraform.tfstate"
    region = "ap-south-1"
    # For State Locking, LockID
    dynamodb_table = "<dynamodb table name>"
  }
```

TERRAFORM ADV 4:

So, once anybody is running the code, a lock-ID will be generated in the DynamoDB table which will be automatically deleted after the completion of the applying work in terraform. This ID is used for doing force-unlock action for the state file.

3.3 Terraform commands regarding state file: (State commands)

3.3.1 [terraform show]

- This will work after applying command.
- This is nothing but it will show the state file as output.
- Generally the same can be seen with Linux command [cat terraform.tfstate], but this is a terraform command providing the same output.
- Importantly, if we configure any remote-state, the state file won't be present in the working directory and we can't able to use the Linux cat command to view the state file. So, at that time we have to use [terraform show] command to see the state file.

3.3.2 [terraform plan -out=<filename>.out]

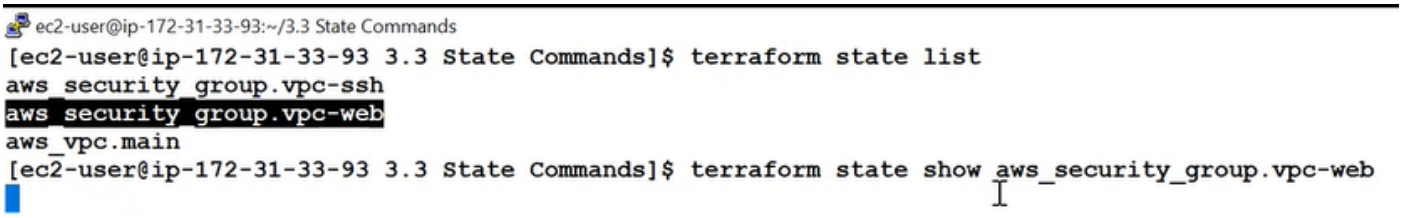
- To get the output of the plan / apply in a separate file (i.e., to redirect the output in a separate file), we can use this command. The file name should be in the format of [<filename>.out] / [test-dev.out].
- Later the same file can be seen by using either Linux [cat <filename>.out] command or [terraform show <filename>.out] command.
- Now if we want to see the output in json format we can give the command as [terraform show -json <filename>.out]
- The same can be done in case of terraform apply command too to store the state file in some other out file if remote-state is not done.

3.3.3 [terraform refresh]

- This will refresh the current status of the resource created by terraform. Meaning, if we do some manual drift or changes in the resource/s created by terraform in the console, that change/s will be reflected in the state file once we run the command **[terraform refresh].** Refresh will update the manual changes done in the infrastructure in the console in the terraform state file.
- But again, if we run plan or apply command, it will again delete the manual changes from the resource.
- The only way to save the manual drift is by adding the same in the resource block of the main.tf file or by adding ignore changes feature in the lifecycle block of the (.tf) file.
- When we run [terraform plan], it initially calls refresh automatically and then it will show the output.
- Refresh command will just show the changes done on manual drift/s. It won't save the same for next apply or plan until we don't change the same in the main.tf file or include ignore changes in lifecycle block.

3.3.4 [terraform state list]

In real time the state file will be huge and there will be lot of resources created at a time with single (.tf) files. And due to the output of the state file will be huge. Now if we want to see the output of few resources or any particular resource, we can use this command followed by manually saying [terraform state show <the resource name from the state list / resource.resourceID>]



So now, we have 'mv' command in the state list as well. The concept is, suppose in my state file, I have changed the resource name by using Linux 'mv' command. For example:

[terraform state mv aws_security_group.vpc-ssh aws_security_group.vpc-ssh-new]

Then for the same resource, the state file name will be changed to a new name, but in the (.tf) file the name has not been changed.

Now, if we run **[terraform plan / apply]** command it will say that, one to add one to delete. Meaning, terraform will again create the old thing and will delete the new thing from the state file. So, whatever is there in the state file, based on this terraform will create and destroy the resource depending on

the code given in the (.tf) file/s. Whatever the code is given terraform will always try to main the same by modifying the state file if manually we do some change/s either in the console or in the state file. Terraform will never change the code from the (.tf) file/s with respect to the manual changes what we did.

Now again if we run the command for removing any resource/s from the state file like:
[terraform state rm aws_security_group.vpc-ssh], What will happen?
→ It will simply remove the resource information from the state file, but it will never destroy the resource in the console once created until we are running the destroy command. So, it will simply delete the resource information from the state file only.
Now, if we run **[terraform plan/apply]** command, what will happen?
→ The default behavior of terraform is when we apply the plan or apply command, terraform will first read the (.tf) file/s present in the directory and then it will compare the state file. If the information of the resource/s given in the code are present in the state file it will show that [0 to add, 0 to destroy]. And if there is any modification or deletion of any information in the state file, terraform will tends to create the same what is present in the (.tf) file/s. So, in this case, the resource information from the state file only got deleted, but the resource is present in the console. Now, while terraform will read the (.tf) file and will compare the state file it will found that one resource is absent and thus, terraform will try to create the missing resource in the state file by following the (.tf) file/s. But it will show an error by saying that a same resource with same name is already existing in the console since, the resource has not been deleted from the console, only it has been removed or deleted from the state file.

3.3.5 [terraform force-unlock <LockID>]

In real time this won't be use, because we will have a conflict if we apply this. It is just a command to know in terraform. If we need in some special case, the same can be used, and the LockID will be collected from the DynamoDB table.

3.3.6 [terraform taint] & [terraform untaint]

This concept we want to use if we want to replace the same resource again due to some issue/s. Suppose I have one EC2 or VPC created by terraform, and there is some issue in the created resource/s, So I want to replace it (meaning, I want to delete the resource and recreate the same), in that case we can use terraform taint command followed by terraform plan and apply.
`[terraform state list]`
[terraform taint <resource.resourceID as per the list >]
`[terraform apply --auto-approve]`
So, again, suppose I tainted a resource wrongly, and I want to untaint it, that can be done simply by running the command:
[terraform untaint <resource.resource ID as per the list>]

3.3.7 Terraform resource targeting for plan, apply, and destroy:

Here by using this concept we can target any resource/s either for panning, or applying, or for destroying. The format is:

- `[terraform plan -target= <resource.resourceID as per the list/written in resource block>, <2nd resource if any>]`
- `[terraform apply -target= <resource.resourceID as per the list/written in resource block>, <2nd resource if any> --auto-approve]`
- `[terraform destroy -target= <resource.resourceID as per the list/written in resource block>, <2nd resource if any> --auto-approve]`

3.4 Workspace in terraform

It is like a branching concept in terraform:
So, we can have multiple workspaces in a single directory path, and we can use single set of (.tf) files for all our workspaces. So, the workspace will be mentioned in our (.tf) files in different manners.

Once we give [terraform init] automatically one workspace will be created named "Default"

Workspace related commands in terraform are as follows:

- [terraform workspace show] → to see in which workspace you are there.
- [terraform workspace list] → to list the workspaces
- [terraform workspace new <workspace name>] → to create a new workspace
- [terraform workspace select <workspace name>] → to switch into another workspace
- [terraform workspace delete <workspace name>] → to delete any workspace

Some example blocks of the main.tf file with the workspace concept:

```
# Create Security Group - SSH Traffic
resource "aws_security_group" "vpc-ssh" {
  name        = "vpc-ssh-${terraform.workspace}"   # Meaning the name will be “vpc-ssh<workspace name>”
  description = "Dev VPC SSH"

  ingress {
    description = "Allow Port 22"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    description = "Allow all ip and ports outboun"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# Create EC2 Instance - Amazon Linux (with IF condition for workspace)
resource "aws_instance" "my-ec2-vm" {
  ami           = "ami-079b5e5b3971bd10d"
  instance_type = var.instance_type
  key_name      = "terraform-key"
  count = terraform.workspace == "default" ? 2 : 1  # Meaning, if the workspace id default, then count will be 2
otherwise the count will be 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "vm-${terraform.workspace}-${count.index}"   #Meaning, the name of the EC2 will be “vm-
<workspace name>-<0/1/2 as per the index>”
  }
}
```

4.1 Local provisioner in terraform:

If we want to execute any command in our client / master terraform machine, in any path that we can do by using terraform with the help of some code in (.tf) file/s. For example, if we want to execute some script in the terraform client machine, in an existing file or after creating a file in any path, that can be done by using local provisioner concept.

```
# Create EC2 Instance - Amazon Linux
resource "aws_instance" "my-ec2-vm" {
  ami           = "ami-079b5e5b3971bd10d"
  instance_type = var.instance_type
  key_name      = "terraform-key"
  #count = terraform.workspace == "default" ? 1 : 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "vm-${terraform.workspace}-0"
  }
```

```
  # local-exec provisioner (Creation-Time Provisioner - Triggered during Create Resource)
  provisioner "local-exec" {
    command = "echo ${aws_instance.my-ec2-vm.private_ip} >> creation-time-private-ip.txt"
    working_dir = "local-exec-output-files/"
    #on_failure = continue
  }

  # local-exec provisioner - (Destroy-Time Provisioner - Triggered during Destroy Resource)
  provisioner "local-exec" {
    when    = destroy
    command = "echo Destroy-time provisioner Instanace Destroyed at `date` >> destroy-time.txt"
    working_dir = "local-exec-output-files/"
  }
}
```

4.2 Remote provisioner in terraform:

Now, if we want to do the same for the newly created resource, it will be called as remote provisioner. Meaning, if we want to execute some command/s in the newly created resource/s we have to provide remote executer and for this we need something extra such as SSH authentication information. For example:
So, the concept is quite simple, if we want to execute something in any ec2, first we have to login to the machine by using public IP and private key. The same we have to provide in the connection block after the resource block and then we have to write the code for remote provisioner.

```
# Create EC2 Instance - Amazon Linux
resource "aws_instance" "my-ec2-vm" {
  ami           = "ami-079b5e5b3971bd10d"
  instance_type = var.instance_type
  key_name      = "terraform-key"
  #count = terraform.workspace == "default" ? 1 : 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "vm-${terraform.workspace}-0"
  }

  # Connection Block for Provisioners to connect to EC2 Instance
  connection {
    type = "ssh"
    host = self.public_ip    # Understand what is "self" →    self.public_ip is nothing but the public IP of the ec2 instance which will be created newly.
    user = "ec2-user"
    password = ""
    private_key = file("private-key/terraform-key.pem")   #Here we have to give the path of the private key will be stored and will be tracked by terraform for later connection
  }

  # Copies the file to Apache Webserver /var/www/html directory
  provisioner "remote-exec" {
    inline = [
      "sleep 120",  # Will sleep for 120 seconds to ensure Apache webserver is provisioned using user_data
      "sudo cp /tmp/file-copy.html /var/www/html"
    ]
  }
}
```

4.3 File provisioner in terraform:

This file provisioner can be used to copy some files from the master/client machine to the newly created machine remotely, and for this also we need a connection block.

```
# Create EC2 Instance - Amazon2 Linux
resource "aws_instance" "my-ec2-vm" {
  ami           = "ami-079b5e5b3971bd10d"
  instance_type = var.instance_type
  key_name      = "terraform-key"
  #count = terraform.workspace == "default" ? 1 : 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
```

```
    "Name" = "vm-${terraform.workspace}-0"
  }

# PLAY WITH /tmp folder in EC2 Instance with File Provisioner
  # Connection Block for Provisioners to connect to EC2 Instance
  connection {
    type = "ssh"
    host = self.public_ip # Understand what is "self"
    user = "ec2-user"
    password = ""
    private_key = file("private-key/terraform-key.pem")
  }
```

```
# Copies the file-copy.html file to /tmp/file-copy.html
 provisioner "file" {
    source      = "apps/file-copy.html"
    destination = "/tmp/file-copy.html"
 }
```

5.1 Modules in Terraform:

This is similar to ansible roles, shared libraries in jenkins, Meaning, we will use modules in terraform to reuse the code. Here, we will have our codes saved in some directory path and from that we will call the directory to run our codes instead of writing them again and again.

So, provider.tf file should be there in the working directory where we are going to run the code for resource creation. In the modules directories we will have only the other files such as main.tf files, variables.tf, variables.tfvars etc.

Now, in our main.tf file of the working directory where we want to create the resource, we have to add the source of the module directory my mentioning the path from the current directory. And the format of mention the code is:

So, in this case, there won't be any resource block, directly we can call the directory.

Now if we need to add extra resource to get create that we can include after the module block in separate resource blocks one by one.

```
# Call our Custom Terraform Module which we built earlier

module "website_s3_bucket" {
  source = "./modules/aws-s3-static-website-bucket"  # This line is Mandatory, and (./) too
  bucket_name = var.my_s3_bucket
  tags = var.my_s3_tags
}
```

So, the format is:

```
Module "any_randam_name" {
       Source = "./naba/devops/git/terra-infra"
       Bucket name = var.variable_value as_mentioned_in_the_variable.tf_file_of_module_directory
       Tags = same as mentioned in the variable.tf in the module directory
}
```

So, here the module is in the path of **(./naba→devops→git→terra-infra)** the '.' Indicates thatfrom the current directory to the source.

5.2 Providers in terraform

Null providers, time providers, aws providers etc.

5.2.1 Null Resource:

If we want to execute any provisioners either file, remote or local without creating any resource at that time we have to provide `null_resource` in the resource block and for the same to achieve, we have to add null provider in the provider block along with aws provider. For example:

```
# Create EC2 Instance - Amazon Linux
resource "aws_instance" "my-ec2-vm" {
  ami           = "ami-079b5e5b3971bd10d"
  instance_type = var.instance_type
  key_name      = "terraform-key"
  #count = terraform.workspace == "default" ? 1 : 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
  tags = {
    "Name" = "vm-${terraform.workspace}-0"
  }
}

# Wait for 90 seconds after creating the above EC2 Instance
resource "time_sleep" "wait_90_seconds" {
  depends_on = [aws_instance.my-ec2-vm]
  create_duration = "90s"
}

# Sync App1 Static Content to Webserver using Provisioners
resource "null_resource" "sync_app1_static" {
  depends_on = [ time_sleep.wait_90_seconds ]
  triggers = {
    always-update =  timestamp()
  }

  # Connection Block for Provisioners to connect to EC2 Instance
  connection {
    type = "ssh"
    host = aws_instance.my-ec2-vm.public_ip
    user = "ec2-user"
    password = ""
    private_key = file("private-key/terraform-key.pem")
  }

 # Copies the app1 folder to /tmp
  provisioner "file" {
    source      = "apps/app1"
    destination = "/tmp"
  }

# Copies the /tmp/app1 folder to Apache Webserver /var/www/html directory
  provisioner "remote-exec" {
    inline = [
      "sudo cp -r /tmp/app1 /var/www/html"
    ]
  }
}
```

Important topics form Terraform for interview.

1. Where are you storing your backend? how are achieving State locking?
2. write a code for ec2, s3, IAM? Include: provider.tf, main.tf, output.tf, var.tf
3. what is terraform Modules?
4. Null Resource in Terraform?
5. Depends on in Terraform?
6. What are the terraform provisioners?
7. Scenario based terraform questions comparing (infra, state, .tf)
8. Data Sources in Terraform?
9. Workspaces in Terraform?

10. Terraform taint untaint?
11. var.tf, .tfvars, auto.tfvars, env variables, cli variables passed? How to pass, the syntax of each
12. lifecycle in Terraform?
    → Create-before-destroy, prevent-destroy, ignore errors
    → Initialize, fmt, validate, plan, apply, and destroy
13. multiple provider? → alias!
14. For Each map / For each
15. For Each Set