What is Terraform?

- Terraform is an infrastructure as code (IAC) tool.
- it helps to provision and manage the cloud platform infrastructure resources automatically with simple code.
- Key features of Terraform include:
- Terraform can manage infrastructure on multiple cloud platforms.
- The human-readable configuration language helps you write infrastructure code quickly.
- Terraform's state allows you to track resource changes throughout your deployments.
- You can commit your configurations to version control to safely collaborate on infrastructure.

Terraform Main commands:

- `init` - - *set up the working dir and download the provider plugin.*

  command is used to initialize a Terraform working directory. Mainly it read the main.tf file and

  1. Download and install the provider plugin.
  2. Module installation.
  3. Backend Initialization: Terraform can store state files remotely, known as "remote backends." When you run terraform init, it initializes the backend according to the configuration specified in your terraform block. This could involve setting up state storage in services like Amazon S3, Azure Blob Storage, Google Cloud Storage, or HashiCorp's Terraform Cloud.
  4. And It create the *".terraform"* directory and *".terraform.lock.hcl"* file in current directory.

- `validate` - - *Syntax and Configuration Check.*

  command is used to check the syntax and validity of the Terraform configuration files in your project directory.

  1. Syntax Check.
  2. Configuration Check.

- `plan` - - *preview the changes.*

  command is used to generate an execution plan based on the current state of your infrastructure and the desired state defined in your Terraform configuration files.
- This command allows you to preview the changes that Terraform will make to your infrastructure when you apply the configuration.

- `apply` - - *to enact the plan changes. (create the resources)*

  command is used to apply the changes described in the Terraform execution plan to your infrastructure. After running *terraform plan* and reviewing the proposed changes, you can use *terraform apply* to enact those changes.

```
terraform apply -auto-approve
```

- `destroy` - - *Resource Deletion.*

  command is used to clean up all resources defined in your Terraform configuration, effectively tearing down the entire infrastructure managed by Terraform.

1. Providers	10. WorkSpaces	19. graph
2. Resources	11. AWS Auth	20. cross account attri
3. Variables	12. Meta-Arguments	21. refresh
4. Statefile	13. Count	22. null_resource
5. Provisioners	14. ForEach	23. Commands
6. Backends	15. Functions	
7. Modules	16. import	
8. Data Sources	17. taint	
9. Locals	18. replace	

```
###  AWS EC2 INSTANCE  ###
### main.tf

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-west-2"
}

resource "aws_vpc" "my_vpc" {
   cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "my_subnet" {
  vpc_id     = aws_vpc.my_vpc.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-west-2a"
}

resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"
  vpc_id      = aws_vpc.my_vpc.id

  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = [aws_vpc.my_vpc.cidr_block]
  }

  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }

}
```

```
resource "aws_network_interface" "my_nic" {
  subnet_id       = aws_subnet.my_subnet.id
  private_ips     = ["10.0.1.50"]
  security_groups = [aws_security_group.allow_tls.id]
}

resource "aws_instance" "my_instance" {
  ami           = "ami-005e54dee72cc1d00" # us-west-2
  instance_type = "t2.micro"
  #key_name     = "user1"
  #vpc_security_group_ids = ["sg-12345678"]
  #subnet_id              = "subnet-eddcdzz4"

  network_interface {
    network_interface_id = aws_network_interface.my_nic.id
    device_index         = 0
  }

  user_data = <<EOF
    #!/bin/bash
    echo "Copying the SSH Key to the server"
    echo -e "ssh-rsa <key-data" >> /home/ubuntu/.ssh/authorized_keys
  EOF

  tags = {
    Name = "web-server-1"
  }

  output "instances" {
    value       = "${aws_instance.my_instance}"
    description = "EC2 details"
  }

}
----------------
resource "aws_ebs_volume" "my_ebs" {
  avaibility_zone = "es-east-1"
  size     = 20
  tags = {
    name = "my_web_ebs"
  }
}
resource "aws_volume_attachment" "ebs_attach" {
   device name ="/dev/sdf"
   volume id =
   instance id =
}
```

Components of Terraform:

1. Providers: Terraform relies on plugins called “providers” to interact with cloud provider, SaaS providers, and other APIs.

```
### main.tf

# Configure the AWS Provider
provider "aws" {
  region = "us-west-2"
}
```

2. Resources: Resources are the most important elements in the terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, subnet, ec2 instances, or higher-level components such as DNS records.

```
### main.tf

resource "aws_vpc" "my_vpc" {
   cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "my_subnet" {
  vpc_id     = aws_vpc.my_vpc.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-west-2a"
}

resource "aws_instance" "my_instance" {
  ami           = "ami-005e54dee72cc1d00" # us-west-2
  instance_type = "t2.micro"
}
```

3. Variables: Using variables in terraform configurations makes our deployments more dynamic.
A Separate file with name “variables.tf” needs to be created in the working directory o store all variables for our used in main.tf configuration file.

```
### variables.tf

variable "region"{
   type = string
   description = "used for selecting region"
   default = " us-west-2"
}

variable "subnet1_cidr " {
  type = string
  description = "This variable defines address space for subnetnet"
}

-  -  -  -  -

### terraform.tfvars

region = " us-west-1"
subnet1_cidr = "10.10.1.0/24"

-  -  -  -  -

### main.tf

# Configure the AWS Provider
provider "aws" {
  region = "${var.region}"
}
```

4. Statefile: After the deployment is finished terraform creates a state file to keep track of current state of the infrastructure.

It will use this file to compare when you deploy/destroy resources, in other words it compares “current state” with “desired state” using this file.

A file with a name of “terraform.tfstate” will be created in your working directory.

Note: when executed the terraform apply command, state file will go to lock mode, to avoid the parallel execution.

“.terraform.tfstate.lock.info” temporary file create on apply

5. Provisioners: Provisioners provide the ability to run additional steps or tasks when a resource is created or destroyed.

This is not a replacement for configuration management tools.

it supports 2 types of connection 1- ssh, 2- winrm

```
### main.tf

resource "null_resource" "copy_file_on_vm" {
 depends_on = [
  aws_instance.web
 ]
 connection {
  type       = "ssh"
  user       = "ubuntu"
  private_key = file("~/.ssh/id_rsa")
  host       = aws_instance.web.public_dns
 }
 provisioner "file" {
  source      = "./file.yaml"
  destination = "/opt/file.yaml"
 }
 provisioner "local-exec" {
  command = "echo ${self.private_ip} >> private_ip.txt"
 }
 provisioner "remote-exec" {
    inline = [
      "touch hello.txt",
      "echo 'Have a great day!' >> hello.txt"
    ]
 }
}
```

6. Backends: terraform state file will be stored on remote location like S3 bucket.

1. Create an S3 bucket with following options.

a. set bucket permissions. b. Enable bucket versioning.
c. Enable encryption.

2. Create a DynamoDB table.

a. Set the column name as “LockID”

3. Include the backend block in the Terraform configuration.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.18.0"
    }
  }

  backend "s3" {
    bucket          = "mycomponents-tfstate"
    key             = "state/terraform.tfstate"
    region          = "eu-central-1"
    encrypt         = true
    dynamodb_table  = "mycomponents_tf_lockid"
  }
}
```

4. Initialize the S3 Backend.

```
"terraform init"
```

5.How to migrate to an AWS S3 remote backend (Optional)

```
"terraform init" "-reconfigure" or "-migrate-state"
```

1. Providers	10. WorkSpaces	19. graph
2. Resources	11. AWS Auth	20. cross account attri
3. Variables	12. Meta-Arguments	21. refresh
4. Statefile	13. Count	22. null_resource
5. Provisioners	14. ForEach	23. Commands
6. Backends	15. Functions	
7. Modules	16. import	
8. Data Sources	17. taint	
9. Locals	18. replace	

7. Modules: Instead of writing similar infrastructure code repeatedly, we can create reusable terraform modules, like a *template-based* deployment.

A module defines a set of parameters which will be passed as key value pairs to actual deployment.

With this approach you can create multiple environments in a very easy way.

```
Azure_Example_Modules
 |
 | -  - modules
            | - - - main.tf
            | - - - variables.tf
 |
 | -  - dev.tf
 | -  - prod.tf


### main.tf

module "module_dev" {
   source = "./modules"
   prefix = "dev"
   vnet_cidr_prefix = "10.20.0.0/16"
   subnet1_cidr_prefix = "10.20.1.0/24"
   rgname = "DevRG"
   subnet = "DevSubnet"
}
```

```
terraform apply --target=module.module.dev
```

8. Data Sources: a data source is a way to fetch information from an external system or service and use it within your Terraform configuration. Data sources allow you to import data into your Terraform configuration that is not directly managed by Terraform itself, such as information about existing resources in a cloud provider, network information, or even data from external APIs.

```
### main.tf

data "aws_instance" "example" {
 instance_id = "i-0123456789abcdef0"
}

resource "aws_ebs_volume" "example" {
 availability_zone = data.aws_instance.example.availability_zone
 size              = 20
 type              = "gp2"
}
```

9. Locals: Locals are similar to variables, but they are scoped to a specific module or resource block rather than being global.
They are particularly useful for storing intermediate values, length names to sort, complex expressions, or repeated configurations that you want to reuse multiple times within your configuration.

And locals are used to overwrite the default variables.

```
locals {
 # Define a reusable VPC ID
 vpc_99 = "vpc-1234567891468543-345644-666788-0000-99999"

 # Define a list of availability zones
 availability_zones = ["us-east-1a", "us-east-1b", "us-east-1c"]
}

resource "aws_subnet" "example_subnet" {
 vpc_id            = local.vpc_99
 availability_zone = local.availability_zones[0]
}
```

```
terraform apply -var instance-type="t2.micro"
```

10. WorkSpaces: to manage multiple statefiles and multiple environments.

```
## List all available workspaces.
terraform workspace list

## Create a new workspace.
terraform workspace new <name>

## Switch to a different workspace.
terraform workspace select <name>

## Delete a workspace.
terraform workspace delete <name>


## Create the dev env resources.
terraform workspace select dev
terraform apply -var-file dev.tfvars
```

```
Azure_Example_Workspaces
 |
 | - - main.tf
 | - - variables.tf
 | - - dev.tfvars
 | -  - prod.tfvars
 |
 | -  - terraform.tfstate.d
          | - - dev
                | - - {} terraform.tfstate



### dev.tfvars

rgname = "NextOpsVideos-Dev"
location = "centralus"
```

11. AWS authentication:

1. AWS IAM User
2. IAM Role
3. Access Keys

Option-1 : directly specify the keys in main.tf file.

```
### main.tf
## Terraform AWS provider configuration using access key

provider "aws" {
 region     = "us-west-2"
 access_key = "your-access-key-id"
 secret_key = "your-secret-access-key"
}

## using IAM role ARN
provider "aws" {
 region = "us-west-2"

 assume_role {
  role_arn = "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
 }
}
```

Option-2 : Export as a variable.

```
## Export as a variable.
export AWS_ACCESS_KEY_ID=your_key;
export AWS_SECRET_ACCESS_KEY=your_secret;
```

Option-3 : specify keys in ~/.aws/config file.

```
### main.tf
[default]
aws_access_key_id=xxxx
aws_secret_access_key=xxxxxx
region=sa-east-1
output=text
```

Option-4 : aws configure command.

```
$ aws configure
AWS Access Key ID [None]: xxxxxxxxxxxxxxxxxxxxxxx
AWS Secret Access Key [None]: xxxxxxxxxxxxxxxxxxxxxxx
```

12. Meta-Arguments:

depends_on: resources are deployed in a specific order.
count: Deploy multiple VMs of same Type.
for_each: Deploy multiple VMs with different size.
provider: specify the cloud provide and region.
liftcycle: used to define resource lifecycle behaviors, such as

1-create before destroy.
2-prevent destroy.
3-ignore_change

```
resource "aws_instance" "example" {
# option-1
 lifecycle {
   create_before_destroy = true
 }
# option-2
 lifecycle {
   prevent_destroy = true
 }
# option-3
 lifecycle {
   ignore_changes = [
     tags["Name"],
     tags["Environment"]
   ]
 }
}
```

13. Count: Deploy multiple VMs of same Type.

```
## main.tf

resource "aws_subnet" "my_subnet" {
 count                = 2
 name                 = "my-subnet-${count.index}"
 vpc_id               = aws_vpc.my_vpc.id
 cidr_block           = ["10.0.${count.index}.0/24"]
 availability_zone    = "us-west-2a"
}

resource "aws_network_interface" "my_nic" {
 count                = 2
 name                 = "my-nic-${count.index}"
 subnet_id            = aws_subnet.mysubnet[count.index].id
 private_ip_address_allocation = "Dynamic"
 security_groups = [aws_security_group.allow_tls.id]
}
```

14. ForEach: Deploy multiple VMs with different size.

```
## variables.tf

variable "resourcedetails" {
 type = map(object({
  rg_name  = string
  name     = string
  location = string
  size     = string
 }))
 default = {
  westus = {
   rg_name  = "westus-rg"
   name     = "west-vm"
   location = "westus2"
   size     = "Standard_B2s"
  }
  eastus = {
   rg_name  = "eastus-rg"
   name     = "east-vm"
   location = "eastus"
   size     = "Standard_B1s"
  }
 }
}
```

```
## main.tf

resource "azurerm_resource_group" "myrg" {
 for_each = var.resourcedetails

 name     = each.value.rg_name
 location = each.value.location
}

resource "azurerm_virtual_network" "myvnet" {
 for_each = var.resourcedetails
 name                = each.value.vnet_name
 address_space       = ["10.0.0.0/16"]
 location            = azurerm_resource_group.myrg[each.key].location
 resource_group_name = azurerm_resource_group.myrg[each.key].name
}
```

15. Functions: Terraform provides several built-in functions that allow you to perform various operations within your configurations. These functions can be used to manipulate strings, lists, maps, and other data types, as well as to interact with resources and perform calculations. Here are some.

```
## main.tf

resource "azurerm_resource_group" "newrg" {
 name     = join("",["${var.prefix}"],["RG01"])
 location = "EastUS"
}

resource "azurerm_storage_account" "newsa" {
 name                     = lower(join("",["${var.prefix}"],["SA01"]))
 resource_group_name      = azurerm_resource_group.newrg.name
 location                 = azurerm_resource_group.newrg.location
 account_tier             = "Standard"
 account_replication_type = "LRS"
}

output "rgname" {
  value = join("",["${var.prefix}"],["RG01"])
}

output "saname" {
  value = lower(join("",["${var.prefix}"],["SA01"]))
}
```

1.String Functions:

format: Formats a string with placeholders.

join: Joins a list of strings into a single string with a delimiter.

split: Splits a string into a list of substrings based on a delimiter.

substr: Extracts a substring from a string.

2. List Functions:

concat: Concatenates lists together.

element: Retrieves an element from a list by index.

length: Returns the length of a list.

slice: Returns a subset of a list.

16. import: allows you to import existing infrastructure into terraform state, enabling you to manage it using terraform going forward.

```
$ terraform import <resource_type.resource-name> <resource-id>
$ terraform import aws_instance.example i-1234567890abcdef0
```

17. taint: This command is **deprecated**. For Terraform v0.15.2 and later, the command it **-replace** option.

```
$ terraform plan -out import <resource_type.resource-name>
<resource-id>
$ terraform import aws_instance.example i-1234567890abcdef0
```

18. replace: For example, if software running inside a virtual machine crash but the virtual machine itself is still running then Terraform will typically have no way to detect and respond to the problem, because Terraform only directly manages the machine as a whole.

In this scenario if you want to re-create the virtual machine, **replace** option is helpful.

```
$ terraform apply -replace="aws_instance.example"
```

19. graph: command is used to generate visible expression of either configuration or execution plan. so whatever activity u want to perform that it will show in flowchart.

```
$ terraform graph | dot -tsvg -o graph.svg
$ terraform graph -type=plan | dot -Tpng >graph.png
```

20. cross account attributes: When managing resources across multiple AWS accounts using Terraform (Ex. VPC Peering)

```
## main.tf

# Provider configuration for AWS account A (source account)
provider "aws" {
 region = "us-east-1"
}

# Provider configuration for AWS account B (target account)
provider "aws" {
 alias  = "account_b"
 region = "us-west-2"
 assume_role {
  role_arn = "arn:aws:iam::ACCOUNT_B_ID:role/ROLE_NAME"
 }
}

# Data source to get VPC ID from account A
data "aws_vpc" "example" {
 provider = aws
 tags = {
  Name = "example-vpc"
 }
}

# Resource in account B that references VPC from account A
resource "aws_instance" "example" {
 provider = aws.account_b
 instance_type = "t2.micro"
 ami = "ami-12345678"
 subnet_id = "subnet-12345678"
 vpc_security_group_ids = ["sg-12345678"]
 # Use data source from account A
 vpc_id = data.aws_vpc.example.id
}
```

21. refresh: This command is **deprecated**, because its default behavior is unsafe.

22. null_resource: In terraform it is one of the resource type. It doesn't create any infrastructure directly but can be used for various purposes such as running provisioners or executing local-exec commands.

```
## main.tf

resource "null_resource" "example" {
 # This null_resource doesn't create any infrastructure
 # It can be used to run local-exec provisioners or perform other actions

 # Trigger this null_resource when a variable changes
 triggers = {
  var_trigger = var.some_variable
 }

 # Run a local-exec provisioner
 provisioner "local-exec" {
  command = "echo Hello, Terraform!"
 }
}
```

Command: **terraform init -upgrade**

so if u want to change the version of the terraform, so without removing lock file u change version using terraform upgarde command

```
$ terraform init -upgrade
```

Command: **terraform fmt**

command is used to align all the syntax and spaces correctly with respect to resources.

```
$ terraform fmt
```

Q1: Terraform variable files order.

Ans: 1. terraform.tfvars file, if present.
2. terraform.tfvars.json file, if present.
3. Any *.auto.tfvars or *.auto.tfvars.json
4. Any -var and -var-file options on the command-line.

1. Providers	10. WorkSpaces	19. graph
2. Resources	11. AWS Auth	20. cross account attri
3. Variables	12. Meta-Arguments	21. refresh
4. Statefile	13. Count	22. null_resource
5. Provisioners	14. ForEach	23. Commands
6. Backends	15. Functions	
7. Modules	16. import	
8. Data Sources	17. taint	
9. Locals	18. replace	