What is Terraform and how does it work? / Explain the lifecycle of a Terraform plan?

Terraform is an open-source Infrastructure as Code (IaC) tool to define and provision infrastructure using HCL" (Hashicorp Configuration Language).

Write/ Define resources in .tf files → Initialize using terraform init → Validate using terraform validate → plan terraform plan → terraform apply → Save a Plan for Later terraform plan -out=tfplan → terraform destroy

What's the difference between Terraform and technologies such as Ansible, Puppet, Chef, etc?

Terraform is considered to be an IaC technology. It's used for provisioning resources, for managing infrastructure on different platforms.

Ansible, Puppet and Chef are Configuration Management technologies. They are used once there is an instance running and you would like to apply some configuration on it like installing an application, applying security policy, etc.

Is there a way to print/see the dependencies between the different resources?

Using terraform graph

What are the ways to lock Terraform module versions?

Answer: **You can use the terraform module registry as a source and specify the attribute version in the module in a terraform configuration file. or**

If you are using the GitHub repository as a source, you must use '? ref' to specify the branch, version, and query string.

Ex → terraform module registry as a source and specify the attribute version

Main.tf

```
module "vpc" {
    source  = "terraform-aws-modules/vpc/aws"   ---->   AWS VPC module from the Terraform Registry
     version = "5.1.0"                # Pin to a specific version

     name = "my-vpc"
     cidr = "10.0.0.0/16"

     azs           = ["us-west-2a", "us-west-2b", "us-west-2c"]
    private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
    public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

    enable_nat_gateway = true
    single_nat_gateway = true

    tags = {
        Terraform   = "true"
        Environment = "dev"
    }
}
    # Ex→ using the GitHub repository as a source and use  ?ref  to specify the branch, version, and query string
module "infra" {
    source = "git::https://github.com/your-org/my-infra-modules.git?ref=main"   -----> pull data from main branch
    # Pass in variables expected by the module (defined in variables.tf)
    vpc_cidr      = "10.0.0.0/16"
    ec2_instance_type = "t2.micro"
     s3_bucket_name = "my-app-bucket"
}
```

```
my-infra-modules/
├── vpc.tf
├── ec2.tf
├── s3.tf
├── variables.tf
└── outputs.tf
```

Vpc.tf

```
resource "aws_vpc" "main" {
    cidr_block = var.vpc_cidr
}
```

ec2.tf

```
resource "aws_instance" "web" {
    ami           = "ami-0c55b159cbfafe1f0"
    instance_type = var.ec2_instance_type
}
```

s3.tf

```
resource "aws_s3_bucket" "bucket" {
    bucket = var.s3_bucket_name
}
```

What is null_resource in Terraform and how we can use null resource in terraform?

The null_resource is a special Terraform resource that doesn't manage real infrastructure, but can be used to run provisioners , such as: shell scripts in local or in remote server , Triggering actions based on change , Deploy non-cloud assets.

```
===== Example: null_resource with local-exec  ======
resource "null_resource" "example" {                  --------> Terraform-managed resource without cloud infrastructure
  provisioner "local-exec" {                          ----------> Executes a command locally on the machine using terraform apply
    command = "echo 'Deploying app version: ${var.app_version}'"
  }

  triggers = {                    ----------->  if any value changes inside the block , the null_resource re-runs
    app_version = var.app_version
  }
}
========== Example:  null_resource  with  remote-exec ===========
resource "null_resource" "remote_example" {
  connection {
    type     = "ssh"
    host     = "192.168.1.10"
    user     = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
  }

  provisioner "remote-exec" {               ----------------> You can also run commands on a remote server:
    inline = [
      "sudo apt update",
      "sudo apt install nginx -y"
      "echo 'Deploying version ${self.triggers.app_version}' > /tmp/app_version.txt",
      "sudo systemctl restart myapp"
    ]
  }
# Re-run  the null_resource  when app_version changes
  triggers = {
    app_version = var.app_version
  }
}
```

You've deployed a virtual machine with Terraform and you would like to pass data to it (or execute some commands). Which concept of Terraform would you use?

Using the provisioner block — specifically, the remote-exec or file provisioners can be used to pass data / to execute the command on a virtual machine.
With the help of provisioner we can run configuration management using Ansible or we can copy the files and execute commands in remote.

What is a "tainted resource"?

If a resource which was successfully created but failed during provisioning. Terraform will fail and mark this resource as "tainted".
Using terraform taint resource.id marks the resource as tainted in the state file manually. So when you run terraform apply the next time, the resource will be destroyed and recreated.

what is .tfvars file in terraform use for?

The .tfvars file in Terraform is used to provide values for input variables defined in your configuration. It helps you separate configuration (values) from infrastructure logic (code), making your code cleaner, more reusable, and easier to manage across environments.

How to check Terraform configuration is syntactically valid?

The terraform validate command is used to check whether a Terraform configuration is syntactically valid. It does not access remote services (like AWS or Azure) to validate the syntax of the code.
Ex → terraform validate -var-file="dev.tfvars" ----------> Validates using variables from dev.tfvars
terraform validate –json ----> Returns output in JSON format

How to setup an infrastructure from scratch in terraform?

What is Drift detection and how to fix it? / How do you interpret a drop in terraform?

Drift detection occur when actual cloud resources differ from the Terraform state file (e.g., someone changes infra manually).

FIX--: terraform refresh to updates the local or remote "terraform.tfstate " file and Contacts your cloud provider (e.g., AWS, Azure) to get the actual live state. → terraform plan → compares the " terraform.tfstate" and your plan output → change accordingly as compared → [redacted] → terraform apply

What are Terraform providers?

Providers are plugins that allow Terraform to interact with APIs (like AWS, Azure, GCP, etc.)

```
terraform {
    Required_provider{
        aws={
            Source = "hashicrop/aws"
            Versio = '~>5.0'
        }
    }
}
provider "aws" {
    region = "us-east-1"          In this configures Terraform to use AWS in us-east-1 region.
}
```

terraform init → Initializes the Terraform project directory and Downloads the necessary providers (e.g., AWS)

What is datasources in Terraform? And how to get data out of a data source?

Data sources used to get data from providers (AWS , Azure). Data sources used for reading, They are not modifying or creating anything. Many providers expose multiple data sources.

```
data "aws_ami" "webEnv" {          # aws_ami is <PROVIDER>_<DATA_SOURCE> and webEnv is name of this data source
    most_recent = true             # 1st argument
    owners      = ["amazon"]       # 2nd argument

    filter {                       # 3rd argument (complex block) → filter is use for combining data source as below
        name   = "name"            → "name" refers to the name of the AMI image when value matches
        values = ["amzn2-ami-hvm-*-x86_64-gp2"]
    }
}
```

How to access data/ attributes from a data source?

Using " data.<PROVIDER>_<DATA_SOURCE>.<NAME>.<ATTRIBUTE> " we can access data/ attributes from a data source.

```
Ex → output "ami_name" {                → we can say ami_name is a output variable
        value = data . aws_ami . webEnv . name   → So ami_name = "amzn2-ami-hvm-2.0.20240702.0-x86_64-gp2" which come as value
     }
```

How to pass values to an input variables?

Using -var option we can pass variable value in CLI → EX -: terraform plan -var="region=us-west-1"

The -var-file option use to pass a file in CLI →EX -: terraform plan -var-file="dev.tfvars"

If you want to set Environment variable use export TF_VAR_<VAR_NAME> → export TF_VAR_<VAR_NAME> = value

We can reference a variable using var.<VAR_NAME> inside terraform code.

To see the output of specific variable terraform output <OUTPUT_VAR>

How to define a variable? / How to define an input variable?

```
variable "app_id" {
    type        = string
    description = " app_id is a string "
    default     = "some_value"
}
```

```
variable "car_model" {
    type = object({
        model=string
        color=string
    })}
    description = "car_model is a dictionary"
    default  = { model=suv , color=red}
}
```

```
variable "list_of_nums" {
    type = list(number)
    default  = [2, 0, 1, 7]
    description = " list_of_nums is a list"
}
```

What is the difference between terraform apply and terraform plan?

---

terraform plan --> Shows the execution plan. Before implementing we can see the changes.
terraform apply --> Executes the changes and create state file

How to create a number of resources based on the length of a list?

```
resource "some_resource" "some_name" {
   count = length(var.some_list)
}
```

How to create list variable "users" with an object containing age and name attribute? How to access the name attribute?

```
variable "users" {
 type = list(object({
   name = string
   age  = number
 }))
default = [{name='saraswati',age=56},{name='hbsahoo',age=60}]
}
```

var.users[0].name will give the name of the variable users o/p ----> saraswati

How to use conditional expressions in Terraform?

---

some_condition ? "value_if_true" : "value_if_false"

Ex -: var.x != "" ? var.x : "yay" ----------> if x is not empty give value of x else give 'yay'

How do you handle sensitive data in Terraform? / How Terraform Generate Secrets?

Use sensitive = true to handle sensitive data in Terraform or generate secrets.

step - 1

========

Encrypt the .tfvars file even if the file dont have sensitive data.

gpg --output secrets.tfvars.gpg --encrypt --recipient <recipient_email_or_key_id> secrets.tfvars

Decrypt during runtime (e.g., in CI/CD or manually)

gpg --quiet --batch --yes --decrypt --output secrets.tfvars secrets.tfvars.gpg

terraform apply -var-file=secrets.tfvars

rm -f secrets.tfvars ------> delete it after use

step - 2

========

Use sensitive = true for variables.

vi secrets.tfvars

db_password = "supersecretpassword123" encrypt this file as above

vi variables.tf

```
variable "db_password" {
        type      = string
        sensitive = true
}
```

vi main.tf -----> where we can use the variable

```
resource "aws_db_instance" "example" {
        identifier       = "example-db"
        engine           = "mysql"
        instance_class   = "db.t3.micro"
        allocated_storage = 20
        username         = "admin"
        password         = var.db_password   ----------> using the variable here
        skip_final_snapshot = true
}
```

decrypt the file as above or use DB_PASSWORD= echo $(aws ssm get-parameter --name "/myapp/config" --with-decryption --query "Parameter.Value" --output text)

Now we can apply terraform apply -var="db_password=$ DB_PASSWORD"

What is Terraform state and why is it important?

Terraform state (terraform.tfstate) keeps track of the infrastructure which Terraform manages.
This file is automatically created and updated by Terraform every time you run terraform apply. So terraform.tfstate file is use to compare your actual infrastructure with your configuration (.tf files) and determine what needs to be created, updated, or destroyed

terraform.tfstate contains:

- **The list of all resources managed by Terraform**
- **Resource metadata (like IDs, dependencies)**
- **Output values**
- **Module paths**
- **Provider-specific information**

Best practice ------> Store terraform.tfstate file remotely (e.g., in S3 with DynamoDB locking) as below.

```
terraform {
    backend "s3" {
        bucket       = "my-terraform-state"
        key          = "dev/terraform.tfstate"
        region       = "us-east-1"
        dynamodb_table = "terraform-lock"
    }
}
```

How can you share data between modules? / What is a Module in Terraform?

Module in terraform allows you to group infrastructure components together so they can be reused across projects or environments. Use outputs as input we can share data between modules. So modules manage these services/resources (VPC, EC2, S3, RDS, EKS etc.) for AWS.

```
my-tf-project/
├── main.tf
├── modules/
│   ├── network/
│   │   ├── main.tf
│   │   └── outputs.tf
│   └── compute/
│       ├── main.tf
│       └── variables.tf
├── outputs.tf
└── variables.tf
```

Lets Share VPC ID from network module to compute module

vi modules/network/main.tf

```
resource "aws_vpc" "main_vpc" {
    cidr_block = "10.0.0.0/16"
    tags = {
        Name = "main-vpc"
    }
}
```

vi modules/network/outputs.tf

```
output "vpc_id"{
    value = aws_vpc.main_vpc.id
}
```

vi modules/compute/variables.tf

```
variable "comp_vpc_id" {
    type = string
}
```

vi modules/compute/main.tf

```
resource "aws_subnet" "main_subnet" {
    vpc_id          = var.comp_vpc_id
    cidr_block      = "10.0.1.0/24"
    availability_zone = "us-east-1a"

    tags = {
        Name = "main-subnet"
    }
}
```

Another example

```
project/
├── main.tf                    # Root module
├── modules/
│   └── ec2_instance/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
```

It is a reusable AWS EC2 instance module

modules/ec2_instance/: main.tf

```
resource "aws_instance" "this" {
  ami           = var.ec2_ami
  instance_type = var.ec2_instance_type
  tags = {
    Name = var.ec2_name
  }
}
```

modules/ec2_instance/variables.tf

```
variable "ec2_ami" {
  type    = string
}
variable "ec2_instance_type" {
  type    = string
  default = "t2.micro"
}
variable "ec2_name" {
  type    = string
}
```

modules/ec2_instance/outputs.tf

```
output "instance_id" {
  value = aws_instance.this.id
}
```

Project/main.tf

```
module "my_ec2" {
  source        = "./modules/ec2_instance"
  ec2_ami       = "ami-12345678"
  ec2_instance_type = "t2.small"
  ec2_name      = "web-server"
}
```

vi main.tf

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

        module "network" {
             source = "./modules/network"
        }

        module "compute" {
             source = "./modules/compute"
             comp_vpc_id = module.network.vpc_id            # <-- Passing output as input
        }
```

How to list resources created with Terraform?

---

terraform state list

How do you rename an existing resource?

---

terraform state mv old_resource   new_resource

How to identify which workspace are you using?

---

terraform workspace show

How do you manage different environments (dev, staging, prod)? / what is workspace ?

Workspaces allow you to use the same code with different state files.

step 1

=======

Use separate workspace for each environments as below (do same for stage and prod)

terraform workspace new dev ----------> create a new dev workspace

terraform workspace select dev

terraform workspace show ----------> it will show the workspace are you using

Step 2

=======

Use different .tfvars files and apply as per environment

```
terraform  apply  -var-file="dev.tfvars"
```

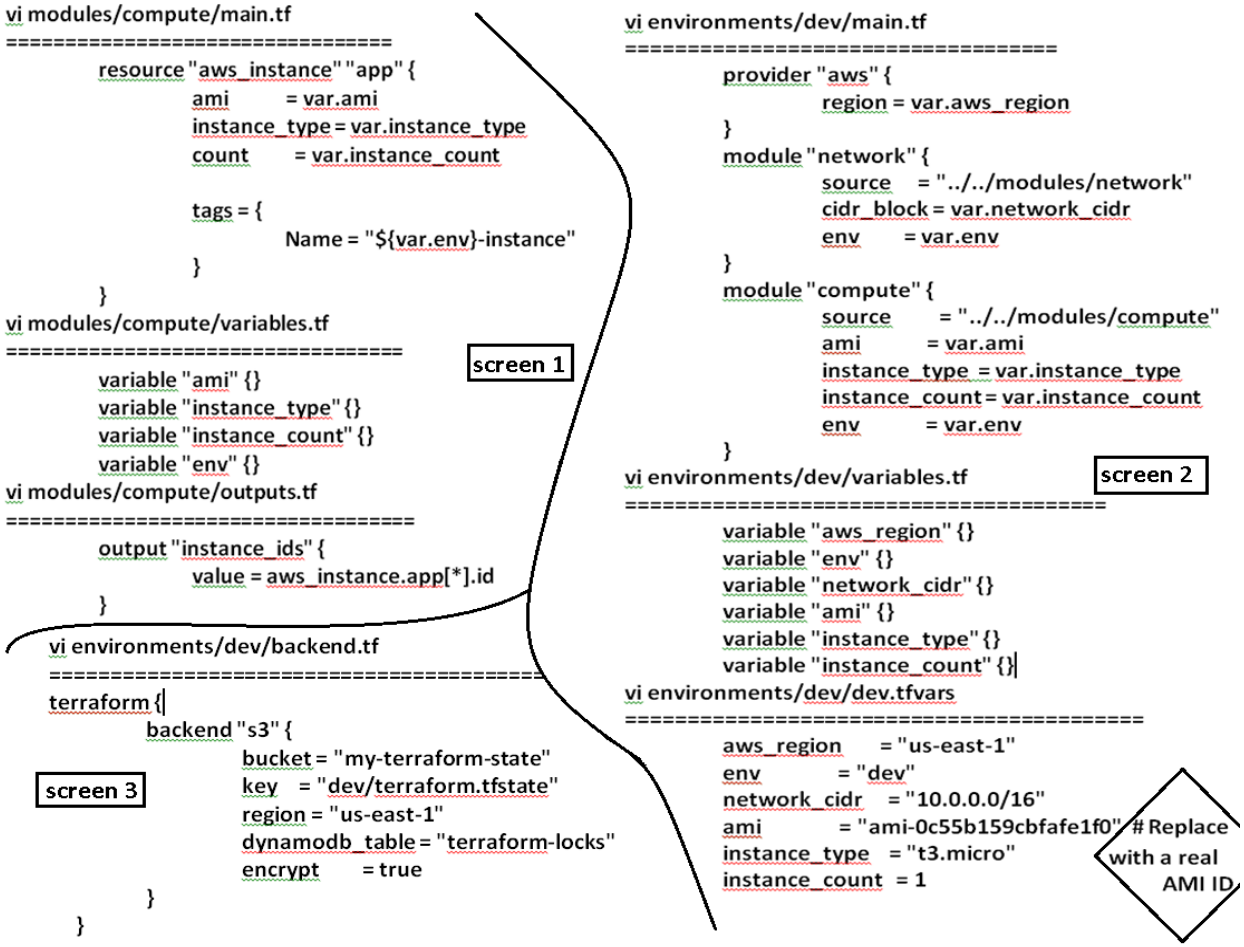
Step 3 Lets Configure dev environment in terraform using modules?

```
=======
        vi modules/network/main.tf
        ==========================
                resource "aws_vpc" "main" {
                        cidr_block = var.cidr_block
                        tags = {
                                Name = "${var.env}-vpc"
                        }
                }

        vi modules/network/variables.tf
        ================================
                variable "cidr_block" {}
                variable "env" {}

        vi modules/network/outputs.tf
        ================================
                output "vpc_id" {
                        value = aws_vpc.main.id
                }
```

```
my_tf_project/
├── modules/           # Reusable modules
│   ├── network/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── compute/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
├── environments/      # Environment-specific
│   ├── dev/                    configurations
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── dev.tfvars
│   ├── staging/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── stage.tfvars
│   └── prod/
│       ├── main.tf
│       ├── variables.tf
│       └── prod.tfvars
└── backend.tf         # Optional: remote  state  config
                                (can also be env-specific)
```



What happens if your state file gets corrupted or deleted?

If it's corrupted: try terraform state pull or recover from backup. So it is advisable to keep remote state file with versioning. if state file get deleted then Terraform loses track of resources

What is "Remote State". When would you use it and how?

Terraform generates a `terraform.tfstate` json file that describes components/service provisioned on the specified provider. Remote State stores this 'terraform.tfstate' file in a remote storage (s3) to enable collaboration amongst team.

What is "State Locking?

Using State locking terraform blocks an operations against a specific state file (.tfstate file) from multiple callers, It allow one caller at a time. That means due to the state locking when you execute terraform apply it create a lock.

What is terraform_remote_state ? /How to share Terraform Remote State ?

The terraform_remote_state data source is a powerful feature in Terraform that allows one project to **access the outputs of another project. Generally Terraform remote state is stored in a backend like Amazon S3, Terraform Cloud.**
Using a data "terraform_remote_state" block we can reuse it.

Step 1 --> Store the State in a Remote Backend

========

```
# backend.tf in project-A
terraform {
    backend "s3" {
        bucket = "my-terraform-state"
        key = "project-a/terraform.tfstate"
        region = "us-west-2"
        dynamodb_table = "terraform-locks"
        encrypt = true
    }
}
```

Run terraform init to configure the backend

Step 2 --> Expose Outputs in the Producing Module

==========

```
# outputs.tf  in  project-A
        output "vpc_id" {
                value = aws_vpc.main.id
        }
```

Step 3 --> Reference the Remote State in Another Project

==========

use the terraform_remote_state data source in project-B

```
# project-b/backend.tf
        data "terraform_remote_state" "project_a" {
                backend = "s3"
                config = {
                        bucket  = "my-terraform-state"
                        key     = "project-a/terraform.tfstate"
                        region  = "us-west-2"
                }
        }
        resource "aws_instance" "example" {
                ami           = "ami-0abcdef1234567890"
                instance_type = "t2.micro"
                subnet_id     = data.terraform_remote_state.project_a.outputs.vpc_id
        }
```

Then run terraform init and terraform apply.

Explain lifecycle rules in Terraform? / is it possible to change the lifecycle rules in Terraform?

In Terraform, lifecycle is a meta-argument within a resource block which tell Terraform how to handel the changes to that resource. Yes we can to change the lifecycle rules in Terraform.

lifecycle Option	Purpose
create_before_destroy	Ensures the new resource is created before the old one is destroyed.
prevent_destroy	Prevents the resource from being accidentally destroyed.
ignore_changes	Ignores changes to specific attributes during terraform apply.
replace_triggered_by	Forces replacement if another resource/input changes (v1.2+).

vi main.tf (step 1)

```
resource "aws_instance" "example" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"

  tags{
        Name = "MyWebServer"
        Environment = "Dev"
  }

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [tags, instance_type]
    replace_triggered_by  = [ var.force_recreate_flag ]
  }
}
```

vi variables.tf (step 2)

```
variable "force_recreate_flag" {
  description = "A dummy value to force resource recreation when changed."
  type        = string
  default     = "initial"
}
```

we can create terraform.tfvars file as below or can pass the force_recreate_flag in command line

vi terraform.tfvars (step 3)

```
force_recreate_flag = "change-1"
```

Now execute terraform plan and terraform apply

from command line

```
terraform apply -var="force_recreate_flag=force-2025-07-29"
```

How to deploy EC2 with Terraform?

```
provider "aws" {
        region = "us-east-1"
}
resource "aws_instance" "web" {
        ami           = "ami-0c55b159cbfafe1f0"
        instance_type = "t2.micro"
        key_name      = "my-key"
        tags = {
                Name = "Terraform-Web"
        }
}
```

execute the command terraform init then terraform plan then terraform apply

What is terraform import and when would you use it?

When you already have an EC2 instance running outside Terraform (e.g. created manually or by another system), and you want to manage it with Terraform, you don't need to recreate it. Instead, you can "map" or import it into Terraform.

How to perform terraform import?

---

step 1 --> Write a basic provider configuration and blank resource block in your Terraform code:

```
provider "aws" {
        region = "us-west-2" # Replace with your region
}

resource "aws_instance" "web" {
                # the block is blank
}
```

step 2 --> Find your instance ID from the AWS Console or CLI:

```
aws ec2 describe-instances --query "Reservations[*].Instances[*].InstanceId" --output text
```

Step 3 --> Then import instance ID into Terraform

```
terraform import    aws_instance.web    i-0123456789abcdef0
```

Step 4 --> Run terraform plan to Inspect What's Missing

```
terraform plan
```

Step 5 --> Manually update your main.tf by copying the attributes from the plan output

```
        resource "aws_instance" "web" {
                ami         = "ami-0abcdef1234567890"
                instance_type = "t3.micro"

                subnet_id           = "subnet-0a12b345c678de9f0"
                vpc_security_group_ids = ["sg-0123abc"]

                key_name     = "my-key"


                tags = {
                        Name = "MyImportedInstance"
                }

                # Add other attributes seen in the terraform plan output

                root_block_device {
                        volume_type         = "gp3"    ##### EBS volume type
                        volume_size         = 100
                        delete_on_termination = true
                        encrypted           = true
                }
                # think this part we added after cheking the plan
        }
```

step 6 --> execute terraform state command to see all the attributes

```
terraform state show    aws_instance.web
```

Step 7 --> once Validate and then Apply

```
terraform plan                  ---> plan shows no changes, that means you've correctly reconstructed the config
terraform apply
```

What is depends_on and how you can use it?

Terraform normally auto-detects dependencies based on how resources reference each other.
But sometime we need to specify explicit dependency using depends_on
When the depends_on meta-argument is used it ensure one resource/module is created or destroyed before another.

Example : create an EC2 Instance That Depends on IAM Role (resource dependency example)

```
provider "aws" {
        region = "us-west-2"
}
resource "aws_iam_role" "example" {
        name = "example-role"

        assume_role_policy = jsonencode({
                Version = "2012-10-17",
                Statement = [{
                        Action = "sts:AssumeRole",
                        Effect = "Allow",
                Principal = {
                        Service = "ec2.amazonaws.com"
                }
                }]
        })
}
resource "aws_instance" "example" {
        ami   = "ami-0c55b159cbfafe1f0" # Example AMI
        instance_type = "t2.micro"

        # Explicit dependency even though there's no direct reference
        depends_on = [aws_iam_role.example]
}
```

Example : Module dependency

```
module "network" {
        source = "./modules/vpc"
}
module "compute" {
        source = "./modules/ec2"
        depends_on = [module.network]
}
```

What is the use of terraform validate?

The terraform validate command is used to check whether your Terraform configuration files are syntactically and structurally valid.

so we should execute terraform validate before exeuting plan and apply

What are the different loops in Terraform? where to use them?

In terraform there is count , for , for_each loop.

What is the Difference: count vs for_each

count --> Count takes numenrical value and create identical resources untill the numenrical value reached / the iteration not complete simple lists.

for_each --> for_each takes key value pair and iterat using those key-value pair to create/delete the resource. For_each will iterate for maps or sets.

Example with count --> S3 bucket using count
==========
vi variables.tf

```
variable "bucket_prefix" {
        type    = string
        default = "my-bucket"
}
```

vi main.tf

```
resource "aws_s3_bucket" "example" {
        count  = 3
        bucket = "${var.bucket_prefix}-${count.index}"
        acl    = "private"
}
```

terraform apply will create the below resource
my-bucket-0
my-bucket-1
my-bucket-2

Example with for_each --> S3 bucket using count
==========
vi variables.tf

```
variable "bucket_map" {
        type = map(string)
        default = {
                dev   = "dev-bucket"
                prod  = "prod-bucket"
                stage = "stage-bucket"
        }
}
```

vi main.tf

```
resource "aws_s3_bucket" "example" {
        for_each = var.bucket_map

        bucket = each.value
        acl    = "private"

        tags = {
                Environment = each.key
                Name        = each.value
        }
}
```

terraform apply will create the below resource
dev-bucket → tagged as Environment = dev
prod-bucket → tagged as Environment = prod
stage-bucket → tagged as Environment = stage

What is dynamic block and how you can use it?
==========
To avoid code repetition we use dynamic block by using "dynamic" keyword. We use dynamic block when the number of nested block (list / map) is not fix. We need to use for_each , BLOCK_TYPE and content.

1st stage

```
variable "ingress_rules" {
 default = [
  {
   from_port = 80
   to_port   = 80
   protocol  = "tcp"
  },
  {
   from_port = 443
   to_port   = 443
   protocol  = "tcp"
  }
 ]
}
```

2nd stage

```
resource "aws_security_group" "example" {
 name = "example"
 dynamic "ingress" {                    -------> dynamic block
  for_each = var.ingress_rules
  content {
   from_port   = ingress.value.from_port
   to_port     = ingress.value.to_port
   protocol    = ingress.value.protocol
   cidr_blocks = ["0.0.0.0/0"]
  }
 }

egress {
   description = "Allow all outbound traffic"
   from_port   = 0
   to_port     = 0
   protocol    = "-1"
   cidr_blocks = ["0.0.0.0/0"]
 }
tags = {
   Name = "example-sg"
 }
}
```

Anatomy/summery of a dynamic Block

```
dynamic "BLOCK_TYPE" {
 for_each = EXPRESSION
 content {
  # inner configuration using BLOCK_TYPE.value.*
 }
}
```

3rd stage

What's a VPC Endpoint? How to configure VPC endpoint?
==========
Using VPC endpoint we can connect different services of AWS (S3 , DynamoDB) without requiring an internet gateway, NAT device, VPN connection.

EX → vpc endpoint for S3

```
resource "aws_vpc_endpoint" "s3_endpoint" {
 vpc_id       = aws_vpc.main.id
 service_name = "com.amazonaws.${var.region}.s3"
 vpc_endpoint_type = "Gateway"

 route_table_ids = [aws_route_table.public.id]
}
```

Ex → vpc endpoint for SSM, EC2

```
resource "aws_vpc_endpoint" "ssm_endpoint" {
 vpc_id          = aws_vpc.main.id
 service_name    = "com.amazonaws.${var.region}.ssm"
 vpc_endpoint_type = "Interface"
 subnet_ids      = [aws_subnet.private.id]

 security_group_ids = [aws_security_group.allow_https.id]
 private_dns_enabled = true
}
```

Ex → example showing how to create a VPC and a Gateway endpoint:

```
provider "aws" {
 region = "us-east-1"
}
resource "aws_vpc" "main" {
 cidr_block = "10.0.0.0/16"
}
resource "aws_subnet" "public" {
 vpc_id     = aws_vpc.main.id
 cidr_block = "10.0.1.0/24"
}
resource "aws_route_table" "public" {
 vpc_id = aws_vpc.main.id
}
resource "aws_vpc_endpoint" "s3_endpoint" {
 vpc_id          = aws_vpc.main.id
 service_name    = "com.amazonaws.us-east-1.s3"
 vpc_endpoint_type = "Gateway"
 route_table_ids  = [aws_route_table.public.id]
}
```

What is load-balancer and how you configure in terraform?

==========

A load balancer is a system that automatically distributes incoming network or application traffic across multiple servers to ensure high availability, reliability, and performance.

Example of configuring an AWS Application Load Balancer (ALB) using Terraform.

```
======== 1. Create the ALB
resource "aws_lb" "app_lb" {
 name               = "app-load-balancer"
 internal           = false
 load_balancer_type = "application"
 security_groups    = [aws_security_group.lb_sg.id]
 subnets            = var.public_subnets

 enable_deletion_protection = false

 tags = {
  Name = "app-load-balancer"
 }
}
========2. Define Target Group
resource "aws_lb_target_group" "app_tg" {
 name     = "app-target-group"
 port     = 80
 protocol = "HTTP"
 vpc_id   = var.vpc_id

 health_check {
  path                = "/"
  interval            = 30
  timeout             = 5
  healthy_threshold   = 2
  unhealthy_threshold = 2
  matcher             = "200"
 }
}
```

```
========3. Attach EC2 Instances (or Autoscaling Group)
resource "aws_lb_target_group_attachment" "app_tg_attachment" {
 target_group_arn = aws_lb_target_group.app_tg.arn
 target_id        = aws_instance.web.id
 port             = 80
}
======== 4. Create Listener ================
resource "aws_lb_listener" "app_listener" {
 load_balancer_arn = aws_lb.app_lb.arn
 port              = 80
 protocol          = "HTTP"

 default_action {
  type             = "forward"
  target_group_arn = aws_lb_target_group.app_tg.arn
 }
}
```

1. Scenario: Managing Environments with Terraform

Q: You're working on a project with multiple environments (dev, staging, prod). How would you use Terraform to manage and isolate these environments?

==========

I would structure the Terraform configuration using workspaces or separate state files for each environment. For example:

Use separate directories (e.g., environments/dev, environments/staging, environments/prod) with environment-specific variables.

Use Terraform workspaces if sharing the same configuration files but want separate state per environment.

Store state files in an S3 backend with DynamoDB for state locking, using different keys for each environment (e.g., terraform-state/dev/terraform.tfstate).

Variables like instance size, DB password, or endpoint URLs would be defined in environment-specific .tfvars files.

2. Scenario: Provisioning a Serverless App
Q: How would you use Terraform to deploy a serverless application with AWS Lambda and API Gateway?
==========
I'd define the following resources in Terraform:
aws_lambda_function: Upload the Lambda package (ZIP or S3), set runtime, handler, and IAM role.
aws_api_gateway_rest_api, aws_api_gateway_resource, and aws_api_gateway_method: Define the API structure.
aws_api_gateway_integration: Integrate the API with the Lambda function.
aws_lambda_permission: Allow API Gateway to invoke the Lambda.
Terraform example:

```
resource "aws_lambda_function" "my_lambda" {
 function_name = "myFunction"
 role        = aws_iam_role.lambda_exec.arn
 handler     = "index.handler"
 runtime     = "nodejs18.x"
 filename    = "function.zip"
}

resource "aws_api_gateway_rest_api" "api" {
 name = "MyAPI"
}
# Add resources, methods, integration, and permissions
```

---

3. Scenario: Managing PostgreSQL in RDS
Q: You are asked to provision an RDS PostgreSQL instance. How do you ensure security and parameter tuning using Terraform?
==========
I would use the aws_db_instance resource for RDS, with options like:
Storage encryption enabled.
Public accessibility set to false.
Place it in private subnets using a subnet_group.
Attach security groups to control inbound access.
Use a parameter group (via aws_db_parameter_group) to manage DB parameters.
Example:

```
resource "aws_db_instance" "postgres" {
 engine          = "postgres"
 instance_class   = "db.t3.micro"
 username        = var.db_user
 password        = var.db_pass
 allocated_storage = 20
 vpc_security_group_ids = [aws_security_group.db_sg.id]
 db_subnet_group_name  = aws_db_subnet_group.main.name
 parameter_group_name  = aws_db_parameter_group.pg.name
}
```

4. Scenario: Deploying ECS with Load Balancer
Q: How would you define an ECS Fargate service behind an Application Load Balancer using Terraform?
==========
I'd use the following Terraform resources:
aws_ecs_cluster, aws_ecs_task_definition, and aws_ecs_service.
Create a load balancer using aws_lb and target group with aws_lb_target_group.
Attach service to ALB via aws_lb_listener.
Example workflow:
Define ECS Cluster and Task Definition.
Define ALB, Listener, and Target Group.
Configure ECS Service with launch_type = "FARGATE" and link to Target Group.
Security groups would allow traffic only from the ALB to ECS tasks.

5. Scenario: Handling State Management
Q: How would you manage and protect Terraform state files in a team setting?
==========
I'd use remote backends like S3 for state storage and DynamoDB for state locking:

```
terraform {
  backend "s3" {
    bucket       = "my-terraform-states"
    key          = "dev/app/terraform.tfstate"
    region       = "us-east-1"
    dynamodb_table = "terraform-locks"
  }
}
```

This ensures:
Shared state access.
State locking to prevent race conditions.
Versioning can be enabled on the S3 bucket for rollback.
Access to the backend bucket and DynamoDB is restricted via IAM policies.

---

6. Scenario: Managing Dependencies Across Resources
Q: You have an S3 bucket and a Lambda function that depends on it. How does Terraform handle this dependency?
==========
Terraform handles implicit dependencies through resource references. If I assign the S3 bucket ARN or name to a Lambda environment variable or source, Terraform knows to create the S3 bucket first.
Example:

```
resource "aws_s3_bucket" "code" {
  bucket = "my-code-bucket"
}

resource "aws_lambda_function" "app" {
  s3_bucket = aws_s3_bucket.code.bucket
  ...
}
```

If needed, explicit dependencies can be added using depends_on.

---

7. Scenario: Secrets Management
Q: How would you manage sensitive data like DB passwords or API keys in Terraform?
==========
I'd avoid hardcoding secrets. Instead:
Use Terraform variables marked as sensitive = true.
Fetch secrets from AWS Secrets Manager or SSM Parameter Store using the data block.
Use tools like terraform-provider-vault if integrating with HashiCorp Vault.
Example:

```
data "aws_secretsmanager_secret_version" "db_pass" {
  secret_id = "prod/db_password"
}
```

8 How to setup an infrastructure from scratch in terraform? give file structure and code for each resource
==========
Setting up infrastructure from scratch using Terraform involves organizing your project with a clear file structure and writing Terraform configuration files for each part of your infrastructure — like VPC, subnets, compute, IAM, etc.
Here's a basic but complete example of how to set up infrastructure in AWS using Terraform, including:
VPC
Subnets
Internet Gateway
Route Tables
EC2 instance
Security Groups
Provider and backend setup

Step 1: File Structure

```
terraform-infra/
│
├── main.tf
├── variables.tf
├── outputs.tf
├── provider.tf
├── terraform.tfvars
│
├── modules/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── ec2/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── security_group/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
```

Step 2: Root Files

---

provider.tf

```
provider "aws" {
  region = var.aws_region
}

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }

  backend "local" {
    path = "terraform.tfstate"
  }
}
```

---

variables.tf

```
variable "aws_region" {
  default = "us-east-1"
}

variable "vpc_cidr" {
  default = "10.0.0.0/16"
}

variable "public_subnet_cidr" {
  default = "10.0.1.0/24"
}

variable "ami_id" {
  description = "AMI ID for EC2"
}

variable "instance_type" {
  default = "t2.micro"
}
```

---

terraform.tfvars

```
aws_region = "us-east-1"
ami_id     = "ami-0c02fb55956c7d316" # Replace with a valid AMI in your region
```

---

main.tf

```
module "vpc" {
  source           = "./modules/vpc"
  vpc_cidr         = var.vpc_cidr
  public_subnet_cidr = var.public_subnet_cidr
}

module "security_group" {
  source      = "./modules/security_group"
  vpc_id      = module.vpc.vpc_id
}

module "ec2" {
  source       = "./modules/ec2"
  ami_id       = var.ami_id
  instance_type = var.instance_type
  subnet_id    = module.vpc.public_subnet_id
  sg_id        = module.security_group.sg_id
}
```

---

outputs.tf

```
output "instance_public_ip" {
  value = module.ec2.public_ip
}
```

□ Modules

□ modules/vpc/main.tf

```
resource "aws_vpc" "main" {
  cidr_block = var.vpc_cidr
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.public_subnet_cidr
  map_public_ip_on_launch = true
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route" "internet_access" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.igw.id
}

resource "aws_route_table_association" "a" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

modules/vpc/variables.tf

```
variable "vpc_cidr" {}
variable "public_subnet_cidr" {}

modules/vpc/outputs.tf

output "vpc_id" {
 value = aws_vpc.main.id
}

output "public_subnet_id" {
 value = aws_subnet.public.id
}
```

---

modules/security_group/main.tf

```
resource "aws_security_group" "allow_ssh" {
 name        = "allow_ssh"
 description = "Allow SSH inbound traffic"
 vpc_id      = var.vpc_id

 ingress {
  from_port   = 22
  to_port     = 22
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }

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

modules/security_group/variables.tf

```
variable "vpc_id" {}

modules/security_group/outputs.tf

output "sg_id" {
 value = aws_security_group.allow_ssh.id
}
```

---

modules/ec2/main.tf

```
resource "aws_instance" "web" {
 ami                    = var.ami_id
 instance_type          = var.instance_type
 subnet_id              = var.subnet_id
 vpc_security_group_ids = [var.sg_id]

 tags = {
  Name = "Terraform-EC2"
 }
}
```

modules/ec2/variables.tf

```
variable "ami_id" {}
variable "instance_type" {}
variable "subnet_id" {}
variable "sg_id" {}
```

modules/ec2/outputs.tf

```
output "public_ip" {
  value = aws_instance.web.public_ip
}
```

Step 3: Initialize and Deploy

```
cd terraform-infra
terraform init
terraform plan
terraform apply
```

Result

This setup provisions:

A VPC and subnet with internet access

A security group allowing SSH

An EC2 instance in that subnet

Would you like me to generate a downloadable zip of the project structure or extend it to include more services (RDS, ECS, S3, etc)?