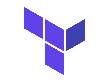
** Terraform Notes**

**Infrastructure as Code (IaC)** is the practice of managing and provisioning infrastructure (such as servers, networks, and databases) using code rather than manual processes. IaC allows infrastructure to be defined, versioned, and deployed in a consistent, automated manner, reducing human errors and increasing scalability.

**Key Benefits of IaC:**

✅ **Automation** – Eliminates manual configuration.  
✅ **Consistency** – Prevents configuration drift.  
✅ **Scalability** – Easily replicate environments.  
✅ **Version Control** – Track changes using tools like Git.  
✅ **Faster Deployments** – Speeds up infrastructure provisioning.

**Popular IaC Tools:**

* **Terraform** (declarative, multi-cloud)
* **AWS CloudFormation** (AWS-native)
* **Ansible** (configuration management + IaC)
* **Pulumi** (uses familiar programming languages)
* **Chef/Puppet** (more focused on configuration management)

**Terraform**

Terraform is an **Infrastructure as Code (IaC)** tool developed by **HashiCorp** that allows you to define, provision, and manage infrastructure using a declarative configuration language. It helps automate cloud resources like servers, databases, and networking components across multiple cloud providers.

**Key Features of Terraform**

✅ **Declarative Approach** – Define what you want, and Terraform figures out how to achieve it.  
✅ **Multi-Cloud Support** – Works with AWS, Azure, GCP, Kubernetes, and on-prem infrastructure.  
✅ **State Management** – Keeps track of infrastructure changes in a .tfstate file.  
✅ **Modular and Reusable** – Supports modules for reusability and scalability.  
✅ **Plan and Apply** – Preview changes before applying them.

***Terraform follows a structured workflow for managing infrastructure as code (IaC). Here’s a step-by-step flow:***

1. **Write Configuration:** Define infrastructure using HashiCorp Configuration Language (HCL) in .tf files.

provider "aws" {  
 region = "us-east-1"  
}  
resource "aws\_instance" "web" {  
 ami = "ami-12345678"  
 instance\_type = "t2.micro"  
}

1. **Initialize the Working Directory:** terraform init  
   This: a. Downloads required provider plugins.  
    b. Sets up backend storage (if configured).
2. **Validate Configuration:** terraform validate

This checks the syntax and validity of .tf files.

1. **Plan the Execution:** terraform plan

This previews changes before applying them.

1. **Apply the Configuration:** terraform apply

This provisions resources as defined in .tf files. Terraform prompts for confirmation unless -auto-approve is used.

It will take your confirmation whether: Yes or No

1. **Verify Deployment:** terraform show
2. **Destroy Infrastructure:** terraform destroy
3. **Syncs the state file with real-world resources:** terraform refresh
4. **Check State:** terraform state list

**Simple Terraform Script to Create AWS EC2 instance**

provider "aws" {

region = "us-east-1"  
  
}

resource "aws\_instance" "web" {  
 ami = "ami-0c55b159cbfafe1f0" **# Replace with a valid AMI ID**  
instance\_type = "t2.micro"

tags = {

Name = "Terraform-EC2"  
 }  
}

**Terraform Configuration:** Terraform configurations are written in HCL (HashiCorp Configuration Language) and define resources, providers, variables, and outputs.

**Terraform variables:** In Terraform, variables allow you to define values that can be reused across configurations, making your infrastructure more flexible and maintainable. Terraform supports different types of variables

**Defining Variables:** You can define variables in Terraform using the variable block:

variable "instance\_type" {  
 description = "Type of EC2 instance"  
 **type** = string  
 default = "t2.micro"  
}

**Variable Types:** Terraform supports several variable types:

1. String
2. Number
3. Boolean
4. List
5. Map
6. Object

**How to Pass Variables**

Many ways to do this:

**First Method:** create a var file: **var.tf**

**variable "region" {**

type = string

default = "us-east-1"

}

**variable "instance\_type"** {

type = string

default = "t2.micro"

}

Now, to use these variables in main code:

provider "aws" {

region = **var.region**

}

resource "aws\_instance" "example" {

ami = "ami-123456" # Replace with a valid AMI ID

instance\_type = **var.instance\_type**

}

**Second Method: Use 🡪** terraform.tfvars

terraform.tfvars is an external file that provides values for variables.

It is useful for overriding default values without modifying variables.tf.

**Example** terraform.tfvars file:

region = "us-west-2"

instance\_type = "t3.medium"

**You have defined variable in this file.**

**Now, simply:**

**terraform apply -var-file="terraform.tfvars"**

**Terraform output**: In Terraform, the terraform output command is used to retrieve the values of outputs defined in your Terraform configuration. These outputs are specified in the outputs.tf file or within the module configuration.

**Usage**

1. **List all outputs**: terraform output
2. **Get a specific output:** terraform output <output\_name>

**Just Create a new Output file: Output.tf**

output "instance\_ip" {

value = aws\_instance.EC2\_instance\_name.public\_ip

}

This will show the public IP of EC2 instances when they are launched as output in the logs.

**Here’s a simple Terraform script to create an AWS S3 bucket:**

provider "aws" {

region = "us-east-1" # Change this to your preferred region

}

**resource "aws\_s3\_bucket"** "my\_bucket" {

bucket = "my-unique-bucket-name" # Change this to a globally unique name

**acl**  = "private" # its

}

**output** "bucket\_name" {

value = aws\_s3\_bucket.my\_bucket.id

}

**The ACL (Access Control List)** in Terraform for an S3 bucket defines the level of access control for the bucket. Some common ACL values are:

* **private** – Default, only the bucket owner has access.
* **public-read** – Everyone can read objects in the bucket.
* **public-read-write** – Everyone can read and write objects (not recommended).
* **authenticated-read** – Only authenticated AWS users can read objects.
* **bucket-owner-read** – Bucket owner has read access, others may write.
* **bucket-owner-full-control** – Bucket owner has full access.

**✅ Full Terraform Code to Upload a File to S3:**

**# Create a new S3 bucket**

resource "aws\_s3\_bucket" "my\_bucket" {

bucket = "anujlibucket-007-unique" **# Ensure its globally unique**

}

**# Upload a file to the S3 bucket**resource "aws\_s3\_object" "my\_uploaded\_file"  
  
 {bucket = aws\_s3\_bucket.my\_bucket.bucket # 👍 Better to use .bucket herekey = "uploads/myfile.txt"source = "local/path/to/myfile.txt" **etag =** filemd5("local/path/to/myfile.txt")  
}

**# Output the bucket name**  
output "bucket\_name"   
{ value = aws\_s3\_bucket.my\_bucket.id  
}

**What is etag in aws\_s3\_object?**The etag in the aws\_s3\_object resource is used to track changes to the file you're uploading. It acts like a checksum (fingerprint) of the file.

**💡 Why use it?**Terraform uses etag to know if the file has changed, so it can re-upload it during the next terraform apply.

**📌 Optional, but useful**

The etag is **optional**, but using it helps ensure **idempotency** — Terraform will only upload the file if it changes.

**Random** provider in Terraform: it's super useful for generating random strings, numbers, passwords, UUIDs, etc., often used for unique resource names.

It is used to generate:

1. Random String
2. Random Password
3. Random Integer
4. Ensure Unique Resource Names
5. Random EC2 ID, Bucket ID

Here’s a simple example of how to launch an EC2 instance using Terraform and a **random variable** (e.g., for generating a unique name using the random\_pet provider):

provider "aws" {

region = "us-east-1"

}

**resource "random\_pet" "name" { #Random variable created**

**length = 2**

**separator = "-"**

}

resource "aws\_instance" "example" {

ami = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI (for us-east-1)

instance\_type = "t2.micro"

tags = {

**Name = "ec2-${random\_pet.name.id}" #Pass the variable here**

}

}

**Terraform State Management:** Terraform maintains the state of infrastructure in a file called terraform.tfstate. This file helps Terraform map configuration to actual resources.

**Key Concepts**: **🔹 Types of State Storage:**

1. Local State: By default, Terraform stores terraform.tfstate locally.
2. Remote State: For **collaboration**, **locking**, and **backup**. Terraform can store state in:
   * AWS S3
   * Azure Storage
   * Google Cloud Storage
   * HashiCorp Terraform Cloud.

**🔹 What is the .tfstate file?**

In Terraform, the **.tfstate file** (Terraform state file) is a **critical file** that keeps track of the infrastructure resources Terraform manages. Here's a breakdown:

* It's a **JSON** file automatically created and updated by Terraform.
* It **stores the current state** of your infrastructure as known by Terraform.
* Helps Terraform **map** real-world resources to your configuration files (.tf files).

**🔹 Why is it important?**

1. **Tracking Resource Metadata**: It keeps info like resource IDs, dependencies, and attributes.
2. **Diff Calculations**: When you run terraform plan, Terraform compares your .tf config with the .tfstate file to determine changes.
3. **Resource Management**: It ensures that changes you apply go to the right resources (e.g., correct EC2 instance, bucket, etc.).

**🔹 Types of State Storage**

* **Local** (default): Stored on your local disk as terraform.tfstate.
* **Remote**: Recommended for teams (e.g., S3, Azure Blob, GCS).

**🔹 Common Commands**

* terraform show — View the current state.
* terraform state list — List resources in the state.
* terraform state rm <resource> — Remove a resource from the state (without destroying it).
* terraform refresh — Sync state file with real infrastructure.

In Terraform, the backend block is used to **configure where Terraform stores the .tfstate file**. This allows you to move state from your local machine to a **remote backend**, which is crucial for **team collaboration, security, and state locking**.

**Syntax of backend Block:** You define it **inside the terraform block**, like this:

backend "s3"

{

bucket = "my-terraform-state-bucket" **#Bucket ID**

key = "env/dev/terraform.tfstate" **#Khn pe save krni hai file apni**

region = "us-west-2"

}

**🔹 Why Use a Backend?**

* **Remote state storage** (e.g., S3, Azure Blob, GCS)
* **State locking** (prevents multiple people from running apply at the same time)
* **Secure storage**
* **History and versioning** of state

**🔹 Popular Backends**

| **Backend** | **Features** |
| --- | --- |
| **local** | Default, stores .tfstate on disk |
| **s3** | Works with AWS, supports locking with DynamoDB |
| **azurerm** | Azure Blob Storage |
| **gcs** | Google Cloud Storage |
| **remote** | Terraform Cloud |
| **consul** | Key-value store backend |

**Data Source in terraform: Data Source** is used to **fetch and use information from**:

1. **An external source** – This could be something like an API or a file.
2. **An existing resource within your cloud infrastructure** – Like an already existing AWS VPC, AMI, security group, etc., that you didn’t create in your Terraform code but still want to reference.

**🔧 Problem Statement :** Man lo ki aap AWS par ek VPC aur kuch Security Groups already create kar chuke ho manually ya kisi aur Terraform project se. Ab aap naye Terraform code me inhi existing resources ka use karna chahte ho, bina unhe firse create kiye. Agar aap unka reference lena chahte ho — jaise ki VPC ka ID ya Security Group ka name — to kaise karoge?

Isi tarah, jab aap EC2 instance create karte ho, to aapko **AMI ID** chahiye hoti hai. Har region me AMI ID alag hoti hai, aur ye frequently change bhi hoti hai (jaise new Ubuntu version aaye). Aap manually ID de sakte ho, lekin wo reliable nahi hoga — kal ko AMI delete ho gaya to problem.

Agar aap directly resource block likhoge, to Terraform bolega ki ye resource uske control me nahi hai, aur wo usko create karne ki koshish karega.

**💡 Data Source Solution Summary:** Ye aapko allow karta hai existing resources ko read karne ke liye. Aap naye infrastructure me unka reference use kar sakte ho. Bina dobara unhe create kiye, aur bina Terraform ke state ko confuse kiye.

**Example:** Here’s how you might use a data source to fetch an existing AWS AMI:

data "aws\_ami" "ubuntu" {

most\_recent = true

filter {

name = "name"

values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-\*"]

}

filter {

name = "virtualization-type"

values = ["hvm"]

}

owners = ["099720109477"] # Canonical

Then, you can reference it in a resource like this:

resource "aws\_instance" "example" {

ami = data.aws\_ami.ubuntu.id

instance\_type = "t2.micro"

}

**Terraform Variable – Detailed :** Agar har baar har jagah hardcode karega, toh code messy ho jaayega. Variables se tu clean, dynamic, aur easily manageable code likh sakta hai.

🧩 **Types of Variables in Terraform:**

**1. Input Variables:** Use to pass values into your Terraform configuration.

**# Define**

variable "instance\_type" {

description = "EC2 instance type"

type = string

default = "t2.micro"

}

**# Use**

resource "aws\_instance" "example" {

instance\_type = var.instance\_type

ami = "ami-123456"

}

**2. Output Variables:** Used to display information after terraform apply.

output "instance\_id" {

value = aws\_instance.example.id

}

**3. Environment Variables (via CLI)**

You can override vars using:

terraform apply -var="instance\_type=t3.micro"

**OR**

**Set Environment Variable (Before apply)**

**In your terminal (VS CODE):** export TF\_VAR\_key=value

**OR**

**use .tfvars file:**

# terraform.tfvars

instance\_type = "t3.micro"

**And call with**: terraform apply -var-file="terraform.tfvars"

**terraform.tfvars :** Ye ek special file hai jisme tum apne variables ke values define karta hai. Terraform is file ko automatically read kar leta hai during plan or apply.

**🧾 terraform.tfvars:**

# terraform.tfvars

instance\_type = "t2.micro"

region = "us-east-1"

env = "dev"

**Ab jab tum bologe**: terraform apply

Terraform khud ye file padh lega. No need for -var or -var-file.

Ek module ke andar multiple .tfvars files ho sakti hain — lekin Terraform automatically sirf kuch hi files ko read karta hai, aur baaki tujhe manually batani padti hai.

Using:

**terraform apply -var-file="dev.tfvars"   
terraform apply -var-file="prod.tfvars"**

Here, dev.tfvars and prod.tfvars are files present in same folder.

📦 **\*.auto.tfvars — Kya Hai Bhai?** **Mostly used for PROD/DEV environment:**

Ye **special file** hoti hai Terraform me, jiska naam agar .auto.tfvars se end karta hai, toh **Terraform usko automatically load kar leta hai** — bina tujhe -var-file bolna pade. 😎

Tu terraform apply likhega — aur bas!

✅ Ye file **auto-load** ho jaayegi.

**⚖️ Priority Order (High → Low):**

| **Priority** | **Source** | **Auto-Loaded?** | **Notes** |
| --- | --- | --- | --- |
| 🔝 1 | -var or -var-file in CLI | ❌ No | Manual override |
| 2 | terraform.tfvars | ✅ Yes | Default file |
| 3 | \*.auto.tfvars | ✅ Yes | Multiple files allowed |
| 4 | TF\_VAR\_<name> environment vars | ✅ Yes | Good for CI/CD |
| 5 | default = in variable block | ✅ Yes | Backup value |

**🧾 Yes, Terraform Can Take User Input — Here's How:**

**✅ 1. Using Input Variables**

variable "instance\_type" {

description = "EC2 instance type"

type = string

}

When you run: **terraform apply**

**Terraform will show:**

var.instance\_type

EC2 instance type

**Enter a value:**

You can also use Validation with these inputs: **Use Case: Variable validation for aws\_type**

Suppose tu chah raha hai ki user sirf specific instance types hi de — like t2.micro, t3.micro, t3.small.

**Example:**  **Complete Variable Block with Validation:**

variable "aws\_type" {

description = "What is the type of instance"

type = string

**validation   
{**

**condition = contains(["t2.micro", "t3.micro", "t3.small"], var.aws\_type)**

**error\_message = "Only 't2.micro', 't3.micro', or 't3.small' are allowed as valid instance types."**

**}**

**}**

**🎯 Why Use Validation in Terraform Variables?**

**1. ✅ Prevent Invalid Input:** Validation se tu restrict kar sakta hai ki user sirf **valid, allowed** values hi de. Jaise: instance type me sirf t2.micro, t3.micro chale — aur koi nahi.  
 **2. 💡 Improve Error Messages:** Custom error message se user ko **clear bata sakta hai kya galti hai**. Default error se confusion hota hai. Validation ke saath: "Only 'dev', 'stage', or 'prod' values are allowed." — neat & clear!

**3. 🔐 Protect Sensitive Workflows:** Galat input se teri infra down bhi ho sakti hai ya sensitive data expose ho sakta hai.  
Validation = **guardrail** 🚧

**Terraform Variable Data Types (Typesystem):**

**1. Primitive Types (Basic Types)**

| **Type** | **Example** | **Use Case** |
| --- | --- | --- |
| **string** | "hello" | Names, tags, IDs |
| **number** | 5, 3.14 | Sizes, counts |
| **bool** | true, false | Conditional logic |

**2. Complex Types (**Collection & Structural**)**

**🧺 Collection Types:** Use to store multiple values.

| **Type** | **Example** | **Notes** |
| --- | --- | --- |
| **list(string)** | ["a", "b", "c"] | Ordered values |
| **set(number)** | set(1, 2, 3) | Unordered, unique values |
| **map(string)** | { key = "value" } | Key-value pairs |

**🧱 Structural Types:** Used for grouping related values together.

| **Type** | **Example** | **Notes** |
| --- | --- | --- |
| **object({...})** | { name = "dev", size = 10 } | Defined keys and types |
| **tuple([...])** | [ "a", 123, true ] | Fixed types at fixed positions |

**What are Local Variables in Terraform?**

Terraform me ‘**local’** keyword ka use hota hai jab:

* Tumhe **temporary calculations** karni ho
* Ek **reusable expression** chahiye ho
* Naam se code readable banana ho

**Syntax:**

locals {

instance\_name = "web-${var.env}"

is\_prod = var.env == "prod"

cost = 10 \* var.instance\_count

}

You can access it like this:

resource "aws\_instance" "example" {

tags = {

Name = local.instance\_name

}  
}

**Mini Project: Server Naming Convention**

**Goal:** Har environment ke liye server ka naam alag ho, format ho: <env>-<project>-<region>

# 👉 1. Variables

variable "env" {

description = "Environment name"

type = string

default = "dev"

}

variable "project" {

description = "Project name"

type = string

default = "myapp"

}

variable "region" {

description = "AWS region"

type = string

default = "us-west-2"

}

# 👉 2. Local Variables

locals {

server\_name = "${var.env}-${var.project}-${var.region}"

is\_prod = var.env == "prod"

}

# 👉 3. Output

output "final\_server\_name" {

value = local.server\_name

}

output "is\_production?" {

value = local.is\_prod

}

🧾 Output You'll Get:

final\_server\_name = "dev-myapp-us-west-2"

is\_production? = false

**What You Practiced**

* 📥 Input lena via variable
* 🧠 Logic banana via locals
* 🔎 Result dikhana via output

**Terraform functions**

**🧮 1. Numeric Functions**

* abs(-5) → 5
* ceil(4.2) → 5
* floor(4.9) → 4
* max(3, 7, 1) → 7
* min(3, 7, 1) → 1

**📚 2. String Functions**

* upper("hello") → "HELLO"
* lower("HELLO") → "hello"
* trim(" hi ") → "hi"
* replace("foo-bar", "-", "\_") → "foo\_bar"
* split("-", "a-b-c") → ["a", "b", "c"]
* join(",", ["a", "b", "c"]) → "a,b,c"

**📋 3. Collection Functions (list/map)**

* length([1, 2, 3]) → 3
* contains(["a", "b"], "a") → true
* merge({a = 1}, {b = 2}) → {a = 1, b = 2}
* keys({a = 1, b = 2}) → ["a", "b"]
* values({a = 1, b = 2}) → [1, 2]
* lookup(map, key, default) → safe value fetching

**🧠 4. Logical Functions**

* anytrue([false, true]) → true
* alltrue([true, true]) → true
* can(expression) → returns true if no error
* coalesce("", "backup") → returns first non-empty → "backup"

**⏳ 5. Date & Time**

* timestamp() → "2025-04-23T08:30:00Z"
* timeadd("2025-04-23T08:30:00Z", "2h") → adds 2 hours

**🔀 6. Encoding/Decoding**

* base64encode("hello") → "aGVsbG8="
* base64decode("aGVsbG8=") → "hello"
* jsonencode({a = 1}) → '{"a":1}'

**Example of each function:**

locals {

# Input data (list and string for demo)

values = [1, 2, 3, 4, 5]

greeting = "hello"

my\_map = { "name" = "raj", "age" = 25 }

}

**# 1. Numeric Function - `max` & `min`**

locals {

max\_value = max(local.values...)

min\_value = min(local.values...)

}

output "max\_value" {

value = local.max\_value # 5

}

output "min\_value" {

value = local.min\_value # 1

}

**# 2. String Function - `upper`**

locals {

upper\_greeting = upper(local.greeting)

}

output "upper\_greeting" {

value = local.upper\_greeting # "HELLO"

}

**# 3. Collection Function - `length`**

locals {

list\_length = length(local.values)

}

output "list\_length" {

value = local.list\_length # 5

}

**# 4. Logical Function - `anytrue`**

locals {

any\_true\_check = anytrue([false, false, true])

}

output "any\_true\_check" {

value = local.any\_true\_check # true

}

**# 5. Date Function - `timestamp`**

locals {

current\_time = timestamp()

}

output "current\_time" {

value = local.current\_time # current timestamp like "2025-04-23T08:30:00Z"

}

**# 6. Map Function - `lookup`**

locals {

user\_name = lookup(local.my\_map, "name", "Unknown")

}

output "user\_name" {

value = local.user\_name # "raj"

}

* **Split Function Example:**

split(separator, string)

**separator**: Wo character ya string jisse split karna hai (jaise comma, space, hyphen, etc.)

**string**: Wo string jo split karni hai.

locals {

sentence = "Terraform is awesome"

words = split(" ", local.sentence)

}

output "split\_words" {

value = local.words # ["Terraform", "is", "awesome"]

}

| **File** | **Purpose** | **Use count/for\_each?** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| main.tf | Resources define karo | ✅ YES |

|  |  |  |
| --- | --- | --- |
| variables.tf | Variable declare karo | ❌ NO (sirf define) |

|  |  |  |
| --- | --- | --- |
| terraform.tfvars | Variable values do | ❌ NO (sirf assign) |

**Handling Dynamic/Multiple data in Terraform**

**Dynamic Data Handle Karne Ke Tareeke – Creating multiple resources*: 2 Ways***

***1.*** *Count in Terraform*

***2.*** *For\_each in Terraform*

**1. Count in Terraform:** Count ek top-class tool hai dynamic data handle karne ke liye in Terraform. Ye especially tab kaam aata hai jab tujhe repeat karna ho resource based on list length, flags, ya conditions

***Create 4 EC2 Instances with count:***

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "web" {

count = 4

ami = "ami-0c55b159cbfafe1f0" # ✅ Replace with valid AMI for your region

instance\_type = "t2.micro"

tags = {

Name = "web-server-${count.index + 1}" # 🏷️ Output: web-server-1, web-server-2,...

}

}

| **Part** | **Meaning** |
| --- | --- |
| count = 4 | 4 instances banenge automatically |
| count.index | 0 to 3 index milta har instance ko |
| count.index + 1 | Clean numbering (1 se shuru karne ke liye) |
| tags["Name"] | Har instance ka unique name |

**MINI PROJECT: Terraform Setup: VPC + Subnets + EC2 (2 per subnet)**

provider "aws" {

region = "us-east-1"

}

**# 1️ Create VPC**

resource "aws\_vpc" "main" {

cidr\_block = "10.0.0.0/16"

tags = {

Name = "main-vpc"

}

}

**# 2️ Create 2 Subnets**

resource "aws\_subnet" "subnets" {

count = 2

vpc\_id = aws\_vpc.main.id

cidr\_block = cidrsubnet(aws\_vpc.main.cidr\_block, 4, count.index)

availability\_zone = data.aws\_availability\_zones.available.names[count.index]

tags = {

Name = "subnet-${count.index + 1}"

}

**# 🔍 Get available AZs**

data "aws\_availability\_zones" "available" {}

**# 3️ Create 2 EC2 in each subnet (total 4)**

resource "aws\_instance" "ec2" {

count = 4

ami = "ami-0c55b159cbfafe1f0" **# ✅ Replace with valid AMI in your region**

instance\_type = "t2.micro"

subnet\_id = aws\_subnet.subnets[floor(count.index / 2)].id # ⚡ Maps 2 instances per subnet

tags = {

Name = "ec2-${count.index + 1}"

}

}

**How the Logic Works:**

| **EC2 Index** | **Subnet Index (floor(index / 2))** |
| --- | --- |
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |

So EC2s 0 and 1 go to Subnet 0, EC2s 2 and 3 go to Subnet 1 ✅

**🔥 Final Output:**

* **1 VPC** 10.0.0.0/16
* **Subnet-1** → 10.0.0.0/20 → EC2-1, EC2-2
* **Subnet-2** → 10.0.16.0/20 → EC2-3, EC2-4

**For\_each:** Terraform mein for\_each ek looping construct hai jo resources, modules, ya blocks ko dynamically multiple instances mein create karne ke liye use hota hai — especially jab aapke paas ek list ya map of items ho jinpe aapko repeat karna ho.

resource "resource\_type" "name" {

for\_each = var.some\_map\_or\_set

**# Access current item with each.key (for map) or each.value**

name = each.key

value = each.value

}

**Important Notes:**

* for\_each **maps or sets** pe kaam karta hai, **lists** pe nahi — agar list hai toh toset() ka use karo.
* each.key and each.value dono available hote hain agar aap map use kar rahe ho.
* Agar set use kar rahe ho toh sirf each.value hota hai.

**Example with Map:**

variable "instances" {

default = {

web = "t2.micro"

app = "t2.small"

db = "t2.medium"

}

}

resource "aws\_instance" "servers" {

for\_each = var.instances

ami = "ami-0c55b159cbfafe1f0"

instance\_type = each.value

tags = {

Name = each.key

}

}

**Yahan kya scene hai?**

* Map hai: har key ek instance ka naam hai, aur value uska type.
* each.key = "web", "app", "db"
* each.value = "t2.micro", "t2.small", "t2.medium"

**Example with Set (List of Strings):**

variable "bucket\_names" {

default = ["logs", "media", "backup"]

}

resource "aws\_s3\_bucket" "buckets" {

for\_each = toset(var.bucket\_names)

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

acl = "private"

}

**Kya ho raha hai yahan?**

* toset convert karta hai list ko set mein (because for\_each list directly accept nahi karta).
* Har value pe resource banega.
* each.value = "logs", "media", "backup" (ek-ek baar loop chalega).

**Points yaad rakhne waale:**

| **Point** | **Matlab** |
| --- | --- |
| for\_each | loop lagata hai har item ke liye |
| List use karni ho | toset() lagao |
| Map use kar rahe ho | toh key/value dono milte hain |
| each.key | map ke andar ka naam ya ID |
| each.value | actual data jo use karna hai |

**🔧 Terraform Modules kya hota hai?**

Soch le tu apna ek ghar bana raha hai — ghar mein bedroom, kitchen, bathroom sab hota hai na? Har cheez ka apna ek layout hota hai.

Waise hi **Terraform module** ek *reusable code block* hai. Tu baar-baar wahi resources likhne ki jagah, ek module bana ke har jagah use kar sakta hai. Time bhi bachega, aur code bhi clean lagega.

**🧱 Module ka Structure:**

Ek module basically ek folder hota hai jisme:

* main.tf (main logic)
* variables.tf (inputs)
* outputs.tf (outputs)

rakhte hain. Tu ise apne main code mein bula sakta hai.

**Pro Tip:**

* Tu modules ko GitHub pe bhi daal sakta hai aur source mein URL deke import kar sakta hai.
* Modules ke andar bhi modules ho sakte hain (Nested modules bro, inception level stuff 😄).

**Terraform Registry :** Terraform Registry ek **official online marketplace** hai jahan:

* Terraform ke pre-built modules milte hain (AWS, Azure, GCP, etc. ke liye)
* Providers milte hain (matlab Terraform ko AWS, GitHub, etc. se baat karwana ho toh)

**Terraform Resource Lifecycle:** Terraform ka lifecycle block basically rules batata hai ki ek resource ko *kab recreate karna hai, kab ignore karna hai*, aur kaise usko treat karna hai during changes.

resource "aws\_instance" "my\_ec2" {

ami = "ami-12345678"

instance\_type = "t2.micro"

lifecycle {

**create\_before\_destroy** = true

**prevent\_destroy** = true

**ignore\_changes** = [instance\_type]

}

}

**🔐 Prevent\_destroy**

Iska matlab: **“Bhai, yeh resource delete karne ki himmat mat karna!”**

Terraform tujhe error de dega agar tu ise destroy karne ki koshish kare.

**♻️ create\_before\_destroy**

Iska matlab: **"Naya resource pehle bana le, purana baad mein todna!"**

Yeh useful hai jab tu downtime avoid karna chahta hai. Jaise load balancer ya prod DB ko replace karte waqt.

**🙈 ignore\_changes**

Iska matlab: **"Bhai, agar yeh cheez change ho bhi gayi manually, chhod de. Plan mein mat dikha."**

Tu use karta hai jab koi cheez manually update hoti hai (jaise tags, security group rules) aur tu nahi chahta ki Terraform har baar complain kare.

**🤘 replace\_triggered\_by (Terraform v0.13+)**

Isse tu force kar sakta hai kisi resource ko replace karne jab koi **dusra** resource change ho

resource "aws\_instance" "app" {

lifecycle {

replace\_triggered\_by = [aws\_launch\_template.app\_template]

Matlab: Agar app\_template update hua → yeh aws\_instance bhi recreate hoga.

**Real-Life Scenario:**

Soch le tu prod mein ho. Agar EC2 instance type change karega, par downtime allowed nahi hai. Toh tu bolega:

* **Create naya, fir purana tod (create\_before\_destroy)**
* **Kisi ne manually tag badla, chhod de (ignore\_changes)**

**Terraform state modification:**

**📘 Pehle yeh samajh:**

Terraform ek terraform.tfstate file rakhta hai – isme likha hota hai:

* Kaunsa resource bana hai,
* Kya uska current status hai,
* Aur real world se kaise match ho raha hai.

Ab agar tu manually AWS Console se kuch kar de, ya resource ka naam badal de code mein, toh Terraform bolega:

“Bhai yeh resource toh alag hai, main dobara banaunga.”

Aur tu kahega:

“Nahi bhai, already bana hua hai!”

**🛠️ Kaam ki Commands**

**1. terraform state list**

Tere state file mein abhi kaunse resources hain – yeh dikhata hai.

terraform state list

**2. terraform state show <resource>**

Ek particular resource ka full detail dekhne ke liye.

terraform state show aws\_instance.my\_ec2

**3. terraform state mv**

Kisi resource ko ek naye naam ya naye module mein shift karne ke liye

terraform state mv aws\_instance.old\_name aws\_instance.new\_name

Ya agar tu module ke andar shift kar raha hai:

terraform state mv aws\_instance.my\_ec2 module.ec2.aws\_instance.my\_ec2

**4. terraform state rm**

Terraform ki yaadon se resource hata dega (destroy nahi karega, sirf bhool jaayega)

terraform state rm aws\_instance.my\_ec2

Iska matlab: Resource AWS pe toh hai, par Terraform bhool gaya ki usne banaya.

**5. terraform import**

Real-world mein existing resource ko Terraform state mein add karna

terraform import aws\_instance.my\_ec2 i-0123456789abcdef0

Jaise kisi aur ne AWS console se EC2 bana diya ho, tu bolta:

“Terraform bhai, isko apna resource maan le!”

**🔐 Warning**

* **Direct .tfstate file mat edit karna**, agar kar bhi raha hai toh backup le pehle.
* state mv, state rm, aur import power tools hain – production mein carefully chalana.

**🧠 Real-life Scene:**

Maan le tu apne code mein ye likha tha:

resource "aws\_instance" "web" { ... }

Aur baad mein naam change kar diya:

resource "aws\_instance" "web\_server" { ... }

Toh Terraform bolega: “Pehla delete, naya bana!”

Par tu bolega: “Nahi bhai, sirf naam change kiya hai.”

Toh command chalaye: **terraform state mv aws\_instance.web aws\_instance.web\_server**

**🌟 Terraform import**

Jab koi resource already real-world mein bana hua ho (AWS, Azure, etc.), aur tu chahta hai ki **Terraform usko track kare**, tab use karte hain terraform import.

Soch le kisi aur ne AWS Console se EC2 bana diya.  
Tu bole:  
*"Terraform bhai, isko apna maan le, manage kar ab se!"*

**🔧 Basic Syntax:**

terraform import <resource\_type>.<resource\_name> <real\_world\_resource\_id>

**🧪 Examples ke saath samajh:**

**1. ✅ EC2 Instance Import:**

# main.tf

resource "aws\_instance" "my\_ec2" {

ami = "ami-12345678"

instance\_type = "t2.micro"

}

Terraform ko batana hai ki i-0abc123def456ghij tu manage kare:

terraform import aws\_instance.my\_ec2 i-0abc123def456ghij

**2. ✅ S3 Bucket Import:**

resource "aws\_s3\_bucket" "my\_bucket" {

bucket = "mera-awesome-bucket"

}

Command:

terraform import aws\_s3\_bucket.my\_bucket mera-awesome-bucket

**3. ✅ Module ke andar Import:**

Agar resource kisi module ke andar hai:

terraform import 'module.vpc.aws\_vpc.main' vpc-0a1b2c3d4e5f6g7h

📝 Quotes zaroori hain agar path mein . ya [ jaisa kuch ho.

**4. ✅ Security Group Import:**

terraform import aws\_security\_group.my\_sg sg-0aaabbbcccdddeee

**🔥 Import se pehle:**

**Step-by-step checklist:**

1. Pehle Terraform config (main.tf) mein resource likh le.
2. Fir terraform init kar le.
3. Tab terraform import command chalaye.
4. Fir terraform plan se check kar ki koi unwanted diff toh nahi aa raha.
5. Agar zarurat ho toh terraform state show <resource> se actual imported config dekh le.

**😱 Warning :**

* terraform import sirf **state file mein add** karta hai resource ko.  
  **Code (main.tf)** tu khud likhta hai — match hona chahiye warna terraform plan mein diff dikhega.
* Jo import kiya, woh destroy bhi hoga agar config galat diya.

**⚠️ Common Mistake:**

Tu resource import karta hai, par .tf file mein woh resource match nahi karta — Terraform bolega:

“Bhai tu bol raha hai manage karo, par tera config alag hai! Destroy karke naya bana doon?”

So always **sync code and reality**!