TERRAFORM OUTSIDE

TERRAFORM FMT

- **easily format your Terraform code to a canonical format and style.**
- **it will be applied to all files in current folder.**
- **For subfolders use [terraform fmt --recursive]**

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

resource "aws_instance" "one" {
count = 5
ami = "ami-03eb6185d756497f8"
instance_type = "t2.micro"
}
```

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

resource "aws_instance" "one" {
  count         = 5
  ami           = "ami-03eb6185d756497f8"
  instance_type = "t2.micro"
}
```

TERRAFORM TAINT

- It allows for you to manually mark a resource for recreation
- in real time some times resources fails to create so to recreate them we use taint
- in new version we use -replace option
- TO TAINT: terraform taint aws_instance.one[0]
- TO UNTAINT: terraform untaint aws_instance.one[0]

- terraform state list
- terraform taint aws_instance.one[0]
- terraform apply --auto-approve

- TERRAFORM REPLACE:
- terraform apply --auto-approve -replace="aws_instance.one[0]"

TERRAFORM IMPORT

- when we create resource manually terraform wont track that resource.
- Import command can used to import the resource which is created manually.
- can only import one resource at a time (FROM CLI)
- it can import both code to config file and state file.
- FOR STATE FILE: terraform import aws_instance.one (not prefarable)
- FOR STATEFILE & CODE:

  terraform plan-generate-config-out=ec2.tf

  terraform apply

```
import {
  to = aws_instance.one
  id = "i-12345678"
}
```

TERRAFORM WORKSPACES

- Terraform workspaces allow you to maintain multiple environments (e.g., development, testing, production)
- using the same Terraform configuration but with different states.
- Each workspace has its own state file (terraform.tfstate.d folder)
- resources created/managed in one workspace do not affect other workspaces.
- the default workspace in Terraform is default
- Each workspace is isloated (seperated with each other)

- terraform workspace list : to list the workspaces
- terraform workspace new dev : to create workspace
- terraform workspace show : to show current workspace
- terraform workspace select dev : to switch to dev workspace
- terraform workspace delete dev : to delete dev workspace

NOTE:

1. we need to empty the workspace before delete
2. we cant delete current workspace, we can switch and delete
3. we cant delete default workspace

TERRAFORM STATE COMMANDS

- The terraform state command is used for advanced state management.
- There are some cases where you may need to modify the Terraform state.
- Rather than modify the state directly use these commands.

- terraform state list : to list the resources
- terraform state show aws_subnet.two : to show specific resource info
- terraform state mv aws_subnet.two aws_subnet.three : to rename block
- terraform state rm aws_subnet.three : to remove state information of a resource
- terraform state pull : to pull state file info from backend

TERRAFORM DEBUGGING

- Terraform automates the infrastructure.
- issues like misconfigurations or dependency errors can occur.
- Debugging helps understand issues and fix them efficiently.
- You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs, with TRACE being the most verbose.

- export TF_LOG=TRACE
- export TF_LOG_PATH="logs.txt"
- terraform apply

TERRAFORM STATE MANAGEMENT

TERRAFORM STATE FILE

- Terraform Stores the infrastructure infromation on state file.
- it will automatically refreshed when we run plan, apply & destroy.
- In Terraform, a backend is a configuration that determines how and where Terraform stores its state file and how it manages operations like apply, plan, and destroy.
- By default Terraform uses local backend.
- it stores state file in terraform.tfstate in local folder.
- it stores information in json format.
- if we delete any resource it stores infromation in terraform.tfstate.backup.

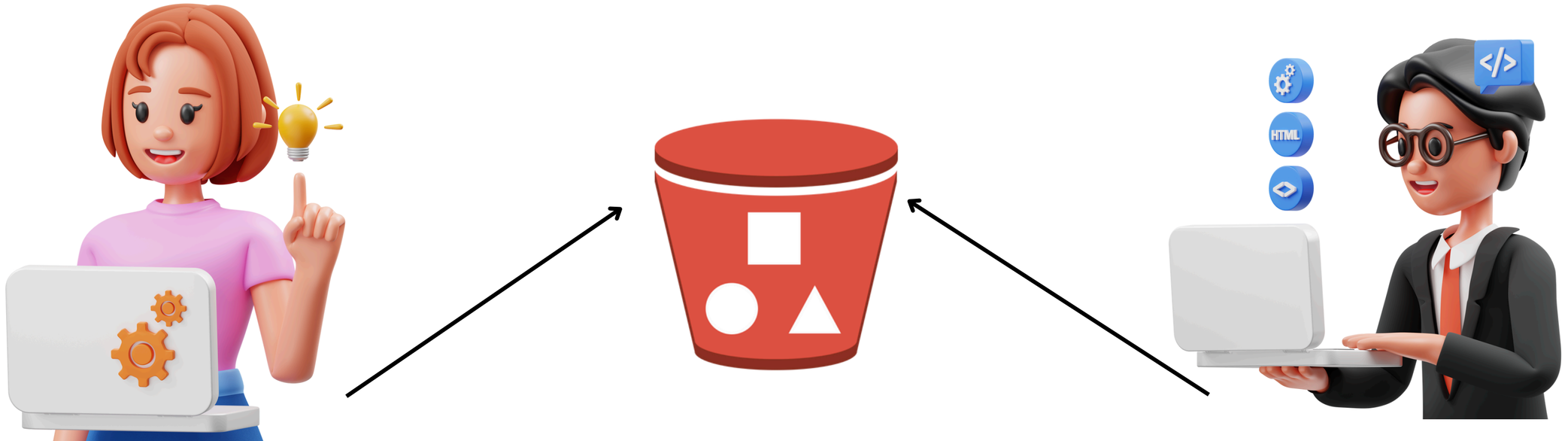
TERRAFORM STATE FILE LOCKING

- in Real time once we complete our work we need to lock state file.
- it ensures that only one operation can be executed at a time.
- once you lock state file you cant modify the infrastructure anymore.
- When two people working on state file at a time it will be locked automatically.
- unfortunately if two people runs apply at same time unpredictable results, like creating duplicate resources or destroying the wrong infrastructure.
- Not all Terraform backends support locking.

Backend	Supports Locking	Notes
Amazon S3 + DynamoDB	Yes	Requires DynamoDB table for locking.
Google Cloud Storage	Yes	Native support for state locking.
Azure Blob Storage	Yes	Native support for state locking.
HashiCorp Consul	Yes	Uses Consul’s locking mechanism.
Terraform Cloud	Yes	Automatic locking with Terraform Cloud.
Alibaba Cloud OSS	Yes	Supports remote state locking.
Etcd	Yes	Distributed locking via etcd.
Local Backend	No	No support for locking (single-user only).
Git (as backend)	No	Not recommended, no locking support.

TERRAFORM S3 BACKEND

- Terraform used local backend to manage state file.
- But in that local backend only one person can able to access it.
- in Real time it’s often necessary to have a centralized, consistent, and secure storage mechanism for the state file.
- Amazon S3 is a popular choice for this, and when combined with DynamoDB for locking, it ensures safe, consistent operations.



WHY LOCKING HAPPEND

- when 2 developers work on the same project with same state file then the locking will be happend.
- if state file is locked only first operation execute and second operation waits.
- to remove state lock use: terraform force-unlock <LOCK_ID>
- after adding dynamodb run: terraform init -reconfigure

ADVANTAGES

- Global Access
- Team Collaboration
- Secure and Scalable
- Centralized State Management
- State File Versioning
- Disaster Recovery

```
terraform {
  backend "s3" {
    bucket = "rahamterrabucketforstatefile"
    key    = "prod/terraform.tfstate"
    region = "us-east-1"
    dynamodb_table = "mytableforlock"
  }
}
```

- **Add that block to existing code and run terraform init -upgrade**
- **dynamodb -- > create table -- > Partition key: LockID -- > create**
- **now after apply state file will go to s3 bucket**
- **dev-1 type destroy and dev-2 type apply now state file locked.**
- **you can check lock-id in new items of table.**
- **once destroy done for dev-1 state file will be unlocked and dev-2 can work.**

REMOTE BACKEND WILL COVERED ON HCP

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

terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "swiggy0099"

    workspaces {
      name = "terraform"
    }
  }
}

resource "aws_instance" "one" {
  ami           = "ami-0208b77a23d891325"
  instance_type = "t2.micro"
  tags = {
    Name = "raham-server"
  }
}
```

terraform login -- > yes
copy the link and paste on browser.
generate token and give to terminal
add backend code and give commands
see the output from HCP CLOUD.

MIGRATING FROM S3 TO LOCAL BACKEND

- if we want state file to back on local use below method.
- remove backend code from main.tf
- run terraform init -migrate-state

TERRAFORM REFRESH

- This command will be use to refresh the state file.
- terraform compares the current state to desired state, it it found any changes on the current state it will update values to state file.
- when we run plan, apply or destroy refresh will perform automatically.
- if a server is manually created running terraform apply -refresh-only would detect those changes and update the state file to reflect the current state of the resource, but it won't attempt to change the infrastructure to match the Terraform configuration.
- if you dont want to refresh while apply & destroy use
  terraform apply/destroy -refresh=false

TERRAFORM BACKEND BLOCK

- By default there is no backend configuration block within Terraform configuration Because Terraform will use it's default backend - local
- This is why we see the terraform.tfstate file in our working directory.
- FOR PARTIAL BACKEND:

  path = "state_data/terraform.dev.tfstate"
- FROM CLI:

  terraform init -backend-config="path=state_data/terraform.prod.tfstate" -migrate-state
- If want we can specify multiple partial backends too.

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

TERRAFORM SENSITIVE DATA

- By default the local state is stored in plain text as JSON.
- There is no additional encryption beyond your hard disk.
- Terraform can store sensitive information in plain text.
- Amazon S3 & Terraform cloud for you can enable encryption

```
variable "first_name" {
  type = string
   sensitive = true
   default = "Terraform"
}
```

check state file it will show data.

BEST PRACTICE

- Treat State as Sensitive Data
- Encrypt State Backend
- Control Access to State File

TERRAFORM BLOCKS

TERRAFORM VALIDATE

- **used validates the configuration files in your working directory.**
- **it will show error when we havent given the values for variables.**

  command: terraform validate

TERRAFORM PLAN

- **used to save plan in a file for future reference.**

  command: terraform plan -out myplan
- **to apply : terraform apply myplan**
- **to destroy: terraform plan -destroy**

PROVIDER BLOCK

- By default provider plugins in terraform change version for every few weeks.
- when we run init command, it download latest plugins always.
- some code will not work with old plugins, so we need to update them.
- To get latest provider plugins : https://registry.terraform.io/browse/providers.
- when you add a new provider terraform init is must.
- terraform providers: to list the providers which required to run code.
- to create infra on any cloud all we need to have is provider.

TYPES:

- 1. OFFICIAL : MANAGED BY TERRAFORM
- 2. PARTNER : MANAGE BY 3RD PATRY COMPANY
- 3. COMMUNITY: MANAGED BY INDIVIDUALS

registry.terraform.io/providers/hashicorp/aws/latest

Providers / hashicorp / aws / Version 5.69.0 Latest Version

aws

Overview Documentation USE PROVIDER

aws

Official by: HashiCorp

Public Cloud

Lifecycle management of AWS resources, including EC2, Lambda, EKS, ECS, VPC, S3, RDS, DynamoDB, and more. This provider is maintained internally by the HashiCorp AWS Provider team.

VERSION	PUBLISHED	SOURCE CODE
5.69.0	**3 days ago**	hashicorp/terraform-provider-aws

How to use this provider

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

Terraform 0.13+

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.69.0"
    }
  }
}
```

- Terrafrom supports installing multiple providers plugins at a time.
- terraform init -upgrade

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">5.67.0"
    }
    azurerm = {
      source  = "hashicorp/azurerm"
      version = ">5.67.0"
    }
    google = {
      source  = "hashicorp/google"
      version = ">5.67.0"
    }
  }
}
```

TERRAFORM BLOCK

- This block is used to set global configurations and settings for the Terraform.
- It usually includes details such as required providers, backend configuration, and version constraints.
- terraform -v
- terraform -version
- terraform --version

```
terraform {
    required_version = ">= 1.8.0"
}
```

LOCAL BLOCK

- A local block is used to define values.
- if a value is repeating multiple times we can define it here.
- This makes our code cleaner and easier to understand.
- simply define value once and use for mutiple times.

```
provider "aws" {
}

locals {
  env = "prod"
}

resource "aws_vpc" "one" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "${local.env}-vpc"
  }
}
```

```
resource "aws_subnet" "two" {
  vpc_id     = aws_vpc.one.id
  cidr_block = "10.0.0.0/24"
  tags = {
    Name = "${local.env}-subnet"
  }
}

resource "aws_instance" "three" {
  subnet_id     = aws_subnet.two.id
  ami           = "ami-00b8917ae86a424c9"
  instance_type = "t2.micro"
  key_name      = "jrb"
  tags = {
    Name = "${local.env}-server"
  }
}
```

TERRAFORM CODE COMMENTING

- We use comments to make others code to understand easily.
- Terraform supports three different syntaxes for comments.
- # -- > single line comment
- // -- > single line comment
- /* */ -- > multi line comment
- Note: if we put comments for code, terraform thinks code is not existed and it will destroy the resource.

```
# this is single line comment

// this is also single line comment

/* this is
multi-line comment
*/
```

TLS PROVIDER

- it provides utilities for working with Transport Layer Security keys & certificates.
- It provides resources that allow private keys, certificates & CSR.
- Add tls on your own and try this below code.

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "devopsbyraham.pem"
}
```

TERRAFROM PROVISIONERS:

- **executes commands/scripts in both local and remote servers.**

LOCAL-EXEC:

- **executes command/script on local machine (where terraform is installed)**
- **it will execute the command when resource is created**

```
resource "aws_instance" "one" {
  ami           = "ami-04823729c75214919"
  instance_type = "t2.micro"
  tags = {
    instance_name = "rahaminstance"
  }
  provisioner "local-exec" {
    command = "echo my name is raham"
  }
}
```

REMOTE-EXEC:

- executes command/script on remote machine.
- once the server got created it will execute the commands and scripts for installing the softwares and configuring them and deploying app also.

```
resource "aws_instance" "one" {
  ami           = "ami-04823729c75214919"
  instance_type = "t2.micro"
  key_name      = "yterraform"
  tags = {
    Name = " rahaminstance"
  }

  provisioner "remote-exec" {
    inline = [
      "sudo yum update -y",
      "sudo yum install git maven tree httpd -y",
      "touch file1"
    ]

    connection {
      type        = "ssh"
      user        = "ec2-user"
      private_key = file("~/.ssh/id_rsa")
      host        = self.public_ip
    }
  }
}
```

ALIAS & PROVIDERS:

- we can map resource blocks to particular provider blocks.
- used to create resources on different regions.

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

resource "aws_instance" "one" {
  ami                    = "ami-04823729c75214919"
  instance_type          = "t2.micro"
  tags = {
    Name = "N-VIRGINIA"
  }
}

provider "aws" {
region = "ap-south-1"
alias = "mumbai"
}

resource "aws_instance" "two" {
  provider               = aws.mumbai
  ami                    = "ami-08718895af4dfa033"
  instance_type          = "t2.micro"
  tags = {
    Name = "MUMBAI-SERVER"
  }
}
```

DATA BLOCK

- A data block is used to read resource information that is already existed.
- Data Block will not create new resources, it fetch info from existing resources.
- it can fetch info from other workspaces also.

EX:

- Fetching information about existing resources (e.g., AMIs, VPCs, subnets).
- Using data in other blocks (e.g., resources, outputs, or modules).
- Referencing external services or configurations without creating them.

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

# Data block to fetch the latest Amazon Linux 2 AMI
data "aws_ami" "example" {
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["137112412989"] # AWS account ID for Amazon
}

# Using the data fetched to create an EC2 instance
resource "aws_instance" "example" {
  ami           = data.aws_ami.example.id
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleInstance"
  }
}
```

TERRAFORM MODULES

MODULES

- it divides the code into folder structure.
- Modules are group of multiple resources that are used together.
- This makes your code easier to read and reusable across your organization.
- we can publish modules for others to use.
- each module will be having sperate plugins.
- modules plugins will be store on .terraform/modules/

TYPES:

- Root Module: This is the main directory where Terraform commands are run. All Terraform configurations belong to the root module.
- Child Modules: These modules are called by other modules.

main.tf

```
provider "aws" {
}

module "my_instance" {
  source = "./modules/instances"
}

module "s3_module" {
  source = "./modules/buckets"
}
```

vim modules/instances/main.tf

```
resource "aws_instance" "three" {
  count           = 2
  ami             = "ami-00b8917ae86a424c9"
  instance_type = "t2.medium"
  key_name        = "yterraform"
  tags = {
    Name = "n.virginia-server"
  }
}
```

vim modules/buckets/main.tf

```
resource "aws_s3_bucket" "abcd" {
  bucket = "devopsherahamshaik0099889977"
}
```

MODULE SOURCES

- **Modules can be sourced from a number of different locations, including both local and remote sources.**
- **we can get modules form Terraform Registry, HTTP urls, S3 buckets & Local.**

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

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.13.0"
}
```

MODULE INPUT & OUTPUTS

- **We can add Input and Output Blocks for Terraform Modules**
- **Root module can refer both variables & values of child modules.**
- **Child modules cant refer variables, but it can refer its values.**

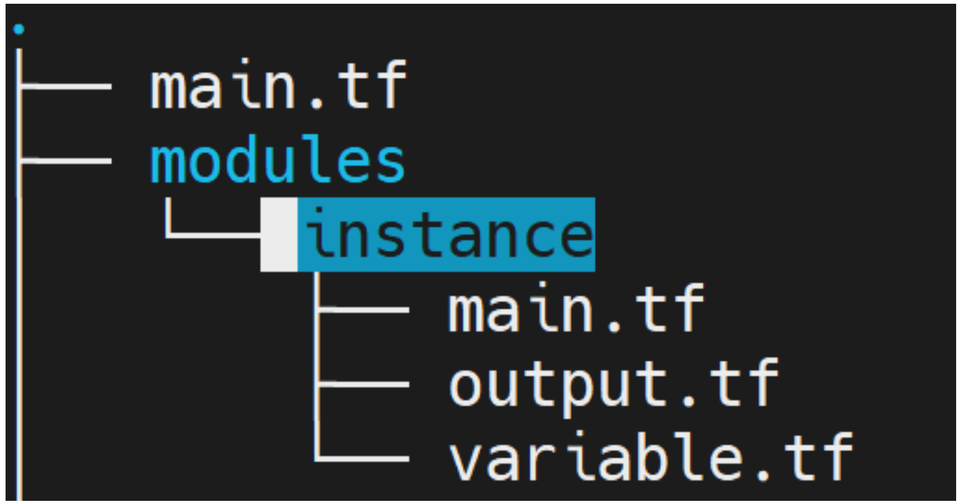
main.tf

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

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

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

MODULE STRUCTURE



cat modules/instance/main.tf

```
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = var.instance_type

  tags = {
    Name = var.instance_name
  }
}
```

cat modules/instance/variable.tf

```
variable "ami_id" {
  default     = "ami-0e54eba7c51c234f6"
}

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

variable "instance_name" {
  default     = "dev-server"
}
```

cat modules/instance/output.tf

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

PUBLIC MODULES

- The module must be on GitHub and must be a public repo.
- NAMING FORMAT: terraform-<provider>-<name> (terraform-aws-ec2-instance)
- must have a description.
- module structure will be main.tf, variables.tf, outputs.tf.
- x.y.z tags for releases.

TERRAFORM GRAPH

- it shows the relationships between objects in a Terraform configuration.
- using the DOT language.
- once create infra run the command: terraform graph
- Go to Google -- > Graphwiz online & copy paste the code

HCP CLOUD

INTRO

- HCP means HashiCorp Cloud Platform
- it is a managed platform to automate cloud infrastructure.
- it provide privacy, security and isloation.
- it supports multiple providers like AWS, Azure, and Google Cloud.
- it offers a suite of open-source tools for managing infrastructure, including Terraform, Vault, Consul, and Nomad.
- We can use Different Code Repos for a Project.
- We can use Variable sets to apply same variables to all the workspaces.

MAIN PRODUCTS

- **Terraform: To create infrastructure**
- **Vault: to managing secrets and protecting sensitive data.**
- **Nomad: to schedule, deploy, and manage applications.**
- **Consul: to secure service-to-service communication and networking.**
- **Packer: to create images for launching servers.**

ACCOUNT CREATION

- Go to google & type : **HCP CLOUD ACCOUNT SIGNIN**
- **EMAIL & PASSWORD**
- **VERIFY THE EMAIL -- > CREATE ORG**
- **CLICK ON TERRAFORM**
- **CONTINUE WITH HCP ACCOUNT**
- **CREATE A GITHUB ACCOUNT**
- **create repo -- > name -- > add new file -- > write terraform code -- > commit**

WORKING:

- CREATE AN ORGINIZATION
- CREATE YOUR WORKSPACE
- INTEGRATE YOUR VCS -- > GITHUB -- > SELECET REPO -- > NEXT -- > CONTINUE
- ADD VARAIBLES -- >
- AWS_ACCESS_KEY_ID : MARK AS ENV VARS -- > SENSITIVE -- > SAVE
- AWS_SECRET_ACCESS_KEY: MARK AS ENV VARS -- > SENSITIVE -- > SAVE

NOTE: MARK THEM AS ENV VARIBALE AND MAKE SURE NO SPACES ARE GIVEN

- RUNS -- > NEW RUN -- > START -- > CONFIRM AND APPLY
- IT WILL AUTOMATICALLY PLAN & WE NEED TO APPLY BY MANUAL
- SECOND TIME WHEN WE CHANGE CODE IT WILL AUTOMATICALLY PLAN
- PLAN & APPLY
- DESTROY

Cost Estimation policy:

- Cost Estimation policy: we can estimate the cost of infra before we create it.
- by default this feature is disable, we need to enable
- click on terraform logo -- > orginization -- > settings -- > Cost Estimation -- > enable

SENTINEL POLICY:

- To create any resource you can define rules through sentinel policy.
- It verify with before terraform apply.
- Sentinel is a policy as code framework.
- to enforce governance and compliance in your infrastructure as code.
- These rules can check for a variety of conditions, such as security compliance, resource tagging, and more.

TERRAFORM CLOUD OFFERS:

Free

UP TO

500 resources

per month

Cloud

Get started with all capabilities needed for infrastructure as code provisioning.

No credit card required

Start for free

Standard

STARTING AT

$0.00014

per hour per resource

Cloud

For professional individuals or teams adopting infrastructure as code provisioning.

Enterprise support included

Start for free

First 500 resources per month are free

Plus

Custom

Cloud

For enterprises standardizing and managing infrastructure automation and lifecycle, with scalable runs.

Enterprise support included

Contact sales

Enterprise

Custom

Self-managed

For enterprises with special security, compliance, and additional operational requirements.

Enterprise support included

Contact sales

TERRAFORM CLOUD FEATURES:

1. Workspaces
2. Projects
3. Runs
4. Variables and Variable Sets
5. Policies and Policy Sets
6. Run Tasks
7. Single Sign-On (SSO)
8. Remote State
9. Private Registry
10. Agents
11. Role-Based Access Control
12. Version Control Integration
13. Observability

TERRAFORM CLOUD ENTERPISE FEATURES:

- **Private Module Registry: Includes a private registry for sharing modules securely across teams.**
- **Policy as Code: Integrates Sentinel for enforcing policies during provisioning.**
- **Enhanced Automation: Offers advanced run triggers, notifications, and support for custom workflows.**

- **Multi-Organization Support: Allows multiple teams or departments to manage their own infrastructure within a single account.**
- **Advanced Collaboration: Provides role-based access controls and team management features, allowing for fine-grained permissions.**
- **Enhanced Security: Features like SSO (Single Sign-On), audit logs, and compliance tools.**

SECURE SECRETS IN TERRAFORM CODE:

- **TIP 1: Do Not Store Secrets in Plain Text**
- **TIP 2: Mark Variables as Sensitive**
- **TIP 3: Environment Variables**
- **TIP 4: Secret Stores (e.g., Vault, AWS Secrets manager)**

- **NOTE: Even after marked a value as sensitive in tf file, they are stored within the Terraform state file.**
- **It is recommended to store security creds outside of terraform.**

VARIABLE PRECEDENCE:

- DEFINES THE LEVEL PRIORITY FOR VARIABLES IN TERRAFORM
- TERRAFORM WILL PICK THE VARIABLES BY BELOW ORDER

1. CLI
2. AUTO.TFVARS
3. TERRAFORM.TFVARS.JSON
4. TERRAFORM.TFVARS
5. ENV VARIABLE
6. VARIABLE.TF

ENV VARIABLE:

export TF_VAR_instance_type="t2.medium"

BUILT IN FUNCTIONS:

- **Terraform has built-in functions that you can call from within expressions to transform and combine values.**
- **FUNCTIONS WILL HAVE () AND VALUES ON IT**
- **EACH VALUE IS SEPERATE BY ,**

- **terraform console**
- **> max(20, 40, 99)**
- **99**
- **> min(20, 40, 99)**
- **20**
- **> lower("HELLO")**
- **hello**

LIST VS SET:

LIST:

Duplicates allowed

Indexing(starting from 0)

SET:

Duplicates not allowed:

No indexing

```
provider "aws" {
}

resource "aws_instance" "example" {
  count         = length(var.instance_types)
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = tolist(var.instance_types)[count.index]
}
variable "instance_types" {
  type    = LIST(string)
  default = [ "t2.nano", "t2.micro", "t2.large", "t2.medium",]
}
```

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

resource "aws_instance" "example" {
  for_each      = var.instance_type
  ami           = "ami-063d43db0594b521b"
  instance_type = each.value
}

variable "instance_type" {
  type          = set(string)
  default       = ["t2.micro", "t2.medium", "t2.micro"]
}
```

MAP:

- used to pass both key and value to variables.
- use mostly used to assign tags for resources.
- KEY-VALUE can also called as object or Dictionary.

```
provider "aws" {
}

resource "aws_instance" "example" {
  ami           = "ami-0fff1b9a61dec8a5f"
  instance_type = "t2.micro"
  tags          = var.instance_tags
}

variable "instance_tags" {
  type = map(string)
  default = {
    Name   = "raham"
    client = "Swiggy"
  }
}
```

META ARGUMENTS:

- **PARALLELISM: by default terraform follows parallelism.**
- **it will execute all the resources at a time.**
- **by default parallelism limit is 10.**
- **terraform apply -auto-approve -parallelism=1**

DEPENDS_ON:

- one resource creation will be depending on other.
- this is called explicit dependency.

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

resource "aws_instance" "one" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "raham-server"
  }
}

resource "aws_s3_bucket" "two" {
  bucket     = "terraformbucketabcd123"
  depends_on = [aws_instance.one]
}
```

COUNT:

- used to create similar objects.
- NOTE: it wont create resources with different configs.

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

resource "aws_instance" "one" {
  count         = 3
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "dev-server-${count.index+1}"
  }
}
```

FOR_EACH:

- it is a loop used to create resources.
- we can pass different configuration to same code.
- it will create resource with less code.

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

resource "aws_instance" "one" {
  for_each      = toset(["dev-server", "test-server", "prod-server"])
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "${each.key}"
  }
}
```

LIFECYCLE:
PREVENT DESTROY:

- **Used to prevent the resource to not be deleted.**
- **it is bool.**
- **NOTE: CHECK WITH TERRAFORM APPLY**

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

resource "aws_instance" "one" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "raham-server"
  }
  lifecycle {
    prevent_destroy = true
  }
}
```

IGNORE CHANGES:

- **when user modify anything on current state manually changes will be ignored.**
- **NOTE: MODIFY NAME MANUALLY AND CHECK WITH TERRAFORM APPLY**

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

resource "aws_instance" "one" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "raham-server"
  }
  lifecycle {
    ignore_changes = all
  }
}
```

CREATE BEFORE DESTROY:

- **by default when we modify some properties terraform will first destroy old server and create new server later.**
- **if we follow create before destroy lifecycle the new resource will create first and later old resource is going to be destroyed.**
- **NOTE: Change the code of AMI for Running Server and Run Apply command**

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

resource "aws_instance" "one" {
  ami           = "ami-0208b77a23d891325"
  instance_type = "t2.micro"
  tags = {
    Name = "raham-server"
  }
  lifecycle {
    create_before_destroy = true
  }
}
```

DYNAMIC BLOCK:

- it is used to reduce the length of code and used for reusabilty of code in loop.

```
provider "aws" {
}


locals {
 ingress_rules = [{
  port       = 443
  description = "Ingress rules for port 443"
  },
  {
   port       = 80
   description = "Ingree rules for port 80"
 },
 {
   port       = 8080
   description = "Ingree rules for port 8080"
 }]
}
```

```
resource "aws_instance" "ec2_example" {
  ami                    = "ami-0c02fb55956c7d316"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.main.id]
  tags = {
    Name = "Terraform EC2"
  }
}

resource "aws_security_group" "main" {

  egress = [
    {
      cidr_blocks      = ["0.0.0.0/0"]
      description      = "*"
      from_port        = 0
      ipv6_cidr_blocks = []
      prefix_list_ids  = []
      protocol         = "-1"
      security_groups  = []
      self             = false
      to_port          = 0
  }]
```

```
  dynamic "ingress" {
    for_each = local.ingress_rules

    content {
      description = "*"
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }

  tags = {
    Name = "terra sg"
  }
}
```