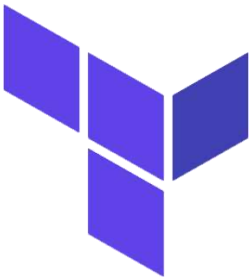


Terraform Handwritten Notes

Parmeshwar Gudadhe

Linkedin.com/parmeshwargudadhe

linkedin : parmeshwargudadhe

Terraform Workflow :

1) init : terraform init

- Used to initialize a working directory containing terraform config files.
- This is the first command that should be run after writing a new terraform configuration.
- Downloads providers.

2) validate : terraform validate

- Validates the terraform configurations files in that respective directory to ensure they are syntactically valid and internally consistent.

3) plan : terraform plan

- Creates an execution plan
- Terraform performs a refresh and determines what actions are necessary to achieve the desired state specified in configuration files.

4) apply : terraform apply

- Used to apply the changes required to reach the desired state of the configuration.
- By default, apply scans the current directory for the configuration and applies the changes appropriately.

5) destroy : terraform destroy

- Used to destroy the Terraform-managed infrastructure.
- This will ask for configuration before destroying.

Terraform Example 01:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      #version = "~> 3.21" # recommended in production.
    }
  }
}

provider "aws" {
  profile = "default"
  region  = "us-east-1"
}

resource "aws_instance" "ec2demo" {
  ami           = "ami-085eea7689237 1e01"
  instance_type = "t2.micro"
}
```

Check default aws credentials -

```
cat $HOME/.aws/credentials.
```

```
$ terraform init
$ terraform validate
$ terraform plan
$ terraform apply
```

LinkedIn: parmeshwarghadadhe

__/__/____

Terraform Configuration language Syntax :-

- Understand Terraform Language Basics:
  1) Understand Blocks.
  2) Understand Arguments, Attributes & Meta-Arguments.
  3) Understand Identifiers.
  4) Understand Comments.
- Terraform language Basics - Files
  * Code in the Terraform language is stored in plain text files with the .tf file extension.
  * There is also a JSON-based variant of the language that is named with .tf.json file extension.
  * We can call the files containing terraform code as Terraform Configuration files or Terraform Manifests.

- Terraform language Basics - Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
    # Block Body
    <IDENTIFIER> = <EXPRESSION> # Argument
}
```

```
# AWS Example
resource "aws_instance" "ec2demo" {
    ami = "ami-04d29b6f966df1537"
    instance_type = "t2.micro"
}
```

Block Type → resource

Block labels → "aws_instance" "ec2demo"

Arguments → ami = ... , instance_type = ...

Single line comments with # or //

Multiline Comments

```
/*
  line-1
  line-2
*/
```

__/__/__

Arguments, Attributes and Meta-Arguments :-

Arguments configure a particular resource; because of this, many arguments are resource specific. Arguments can be required or optional, as specified by the provider. If you do not supply a required argument, Terraform will give an error and not apply the configuration.

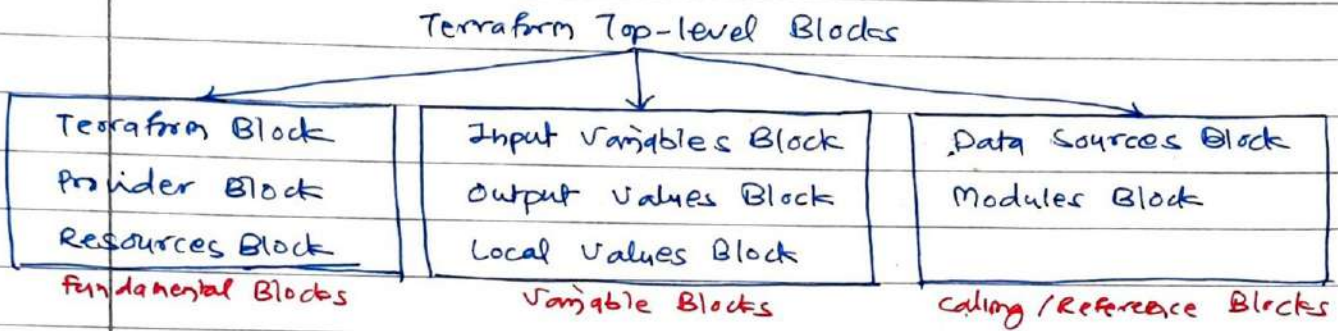
Attributes are values exposed by an existing resource. References to resource attributes take the format resource_type.resource_name.attribute_name. Unlike arguments which specify an infrastructure object's configuration, a resource's attributes are often assigned to it by the underlying cloud provider or API.

Meta-arguments change a resource's behaviour, such as using a count meta-argument to create multiple resources. Meta-arguments are a function of Terraform itself and are not resource or provider-specific.

- depends_on
- count
- for_each
- provider
- lifecycle

Terraform Top-level Blocks :-

- Terraform language uses a limited number of top-level block types, which are blocks that can appear outside of any other block in a TF configuration file.
- Most of Terraform's features are implemented as top-level blocks.



Terraform fundamental Blocks :-

1) Terraform Block :

* Special block used to configure some behaviours of terraform.
* Specifying a required Terraform version.
* Specifying provider requirements
* Configuring a Terraform backend (Terraform state configuration), where our terraform configuration state need to be stored, what is the backend, important block.

2) Provider Block :

* HEART of Terraform
* Terraform relies on providers to interact with Remote Systems.
* Delare providers for terraform to install providers and use them.
* Providers configurations belong to Root Module.

__/__/____

3) Resource Block :-

* Each Resource Block describes one or more infrastructure objects.

* Resource Syntax: How to declare resources?

* Resource Behaviour: How terraform handles resource declarations?

* Provisioners: We can configure resource post-creation actions.

1) Terraform Blocks-

This block can be called in 3 ways. All means the same:

Terraform Block

Terraform Settings Block

Terraform Configuration Block

Each terraform block can contain a number of settings related to terraform's behaviour.

Within a terraform block, only constant values can be used; arguments may not refer to named objects such as resources, input variables, etc and may not use any of the terraform language built-in functions.

Terraform 0.13 and later :-

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

linkedin: parmeshwarg4dadhe

__/__/____

```
terraform {
  # Required Terraform Version
  required_version = "~> 0.14.3"
  # Required Providers and their versions
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.21" # optimal but recommended
    }
  }
  # Remote Backend for storing Terraform state in S3 bucket
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
  # Experimental Features (Not required)
  experiments = [example]
  # Passing metadata to providers (Super Advanced
  provider_meta "my-provider" {
    hello = "world"
  }
}
```

Required Terraform Version

Provider Requirements

Terraform Backend

__/__/____

Creating a simple terraform block and play with required_version -

- required_version focuses on underlying Terraform CLI installed on your desktop.
- If the running version of Terraform on your local desktop doesn't match the constraints specified in your terraform block, Terraform will produce an error and exit without taking any further actions.
- By changing the versions try terraform init and observe whats happening.

A version constraints is a string literal containing one or more conditions, which are seperated by commas.

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

Play with Terraform Versions-

```
required_version = "~> 0.14.3"
required_version = "= 0.14.4"
required_version = ">= 0.13"
required_version = "= 0.13"
required-version = "~> 0.13"
```

linkedin : parmeshwargyadadhe

__/__/____

Play with Provider version:

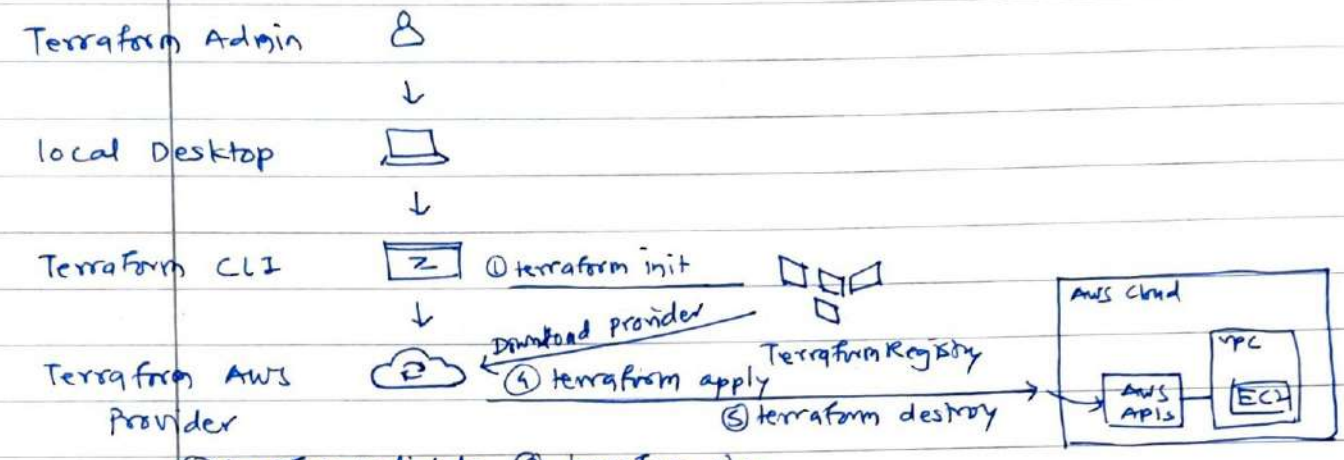
```
version = "~> 3.0"
version = ">= 3.0.0, < 3.1.0"
version = ">= 3.0.0, <= 3.1.0"
version = "~> 2.0"
version = "~> 3.0"
```

Terraform init with upgrade option to change provider version : terraform init -upgrade

Example :

```
terraform {
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 4.0"
        }
    }
}
```

Terraform Providers



- Providers are HEART of Terraform
- Every Resource Type (example: EC2 instance), is implemented by a Provider.
- Without Providers Terraform cannot manage any infrastructure.
- Providers are distributed seperately from Terraform and each provider has its own release cycles and version numbers.
- Terraform Registry is publicly available which contains many Terraform Providers for most major Infra platforms.

Provider Requirements

```
terraform {
  required_version = "~> 0.14.11"
  required_provider {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```

Provider Configuration

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

Dependency Lock file

.terraform.lock.hcl

__/__/__

Required Providers:-

```
# Terraform Block
terraform {
    required_version = "~> 0.14.3"
    required_providers {
        myaws = {
            source = "hashicorp/aws"
            version = "~> 2.0"
        }
    }
}
# Provider Block
provider "myaws" {
    profile = "default"
    region = "us-east-1"
}
```

local names

Local Names:

Local Names are Module Specific and should be unique per-module.

Terraform configurations always refer to local name of provider outside required-provider block.

Users of a provider can choose any local name for it (myaws, aws1, aws2).

Recommended way of choosing local name is to use preferred local name of that provider. (for AWS provider: hashicorp/aws, preferred local name is aws)

Source:

It is the primary location where we can download the Terraform provider.

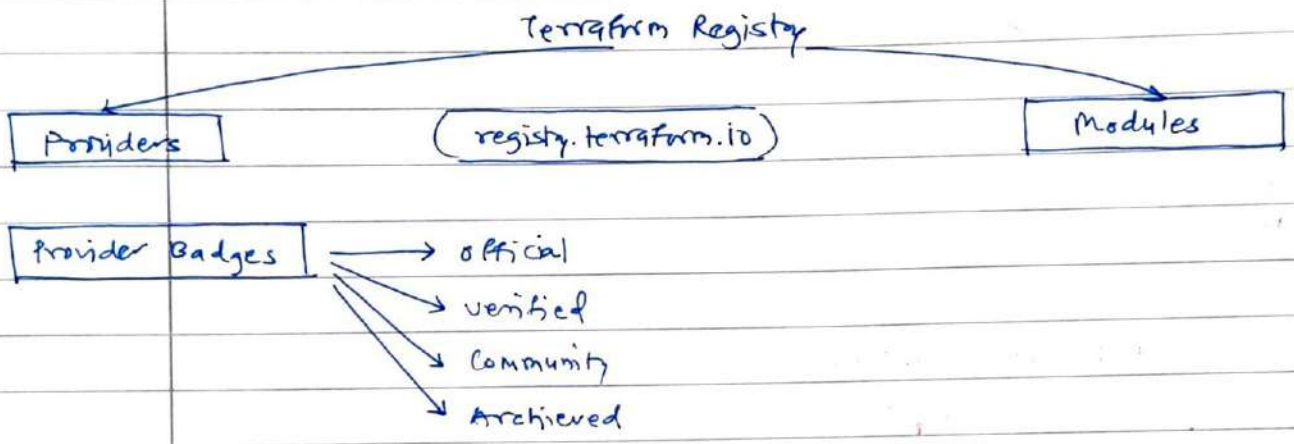
Source address consists of three parts delimited by slashes (/)

__/__/____

[<HOSTNAME>/]<NAMESPACE>/<TYPE>

registy.terraform.io / hashicorp/aws

Registry Name is optional as default is going to be Terraform Public Registy.



linkedin: parmeshwarguadhe

__/__/____

Provider Block

c1.versions.tf

```
# Terraform Block
  terraform {
    required_version = "~> 1.13.0"
    required_providers {
      aws = {
        source = "hashicorp/aws"
        version = "~> 4.1.0"
      }
    }
  }
# Provider Block
  provider "aws" {
      region = "us-east-1"
      profile = "default"
    }
/*
Note-1: AWS Credentials Profile (profile= default) configured on
your local desktop $HOME/.aws/credentials
*/
```

c2-vpc.tf

```
# Resource Block
resource "aws_vpc" "myvpc" {
    cidr_block = "10.0.0.0/16"
    tags = {
      "Name" = "myvpc"
    }
}
```

__/__/____

Terraform Multiple Providers :-

We can define multiple configurations for the same provider and select which one to use on a per-resource or per-module basis.

Ex. We can create one vpc in one region and one vpc in another region.

```
# Provider-1 for us-east-1 (Default Provider)
provider "aws" {
    region    = "us-east-1"
    profile   = "default"
}
# Provider-2 for us-west-1
provider "aws" {
    region    = "us-east-1"
    profile   = "default"
    alias     = "aws-west-1"
}
```

The primary reason for this is to support multiple regions for cloud platforms.

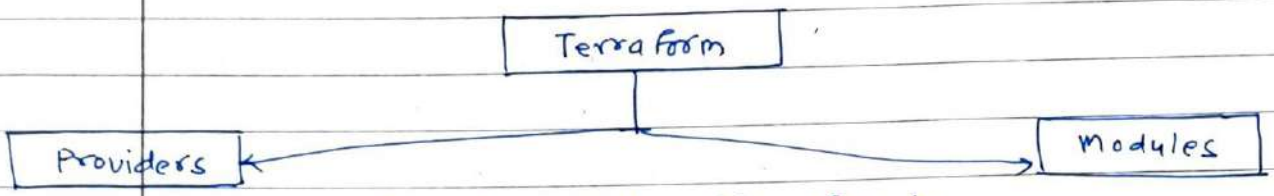
We can use the alternate provider in a resource, data or module by referencing it as <providername>.<alias>.

```
# Resource Block to create VPC in us-west-1
resource "aws_vpc" "vpc-us-west-1" {
    cidr_block = "10.2.0.0/16"
    #<Provider Name>.<alias>
    provider   = aws.aws-west-1
    tags       = {
        "Name" = "vpc-us-west-1"
    }
}
```

LinkedIn : parmeshwargudadhe

__/__/____

Terraform Dependency Lock File :-



Terraform configuration refers to two different kinds of external dependency that come from outside of its own codebase.

- Version Constraints within the configuration itself determine which versions of dependencies are potentially compatible.
- Dependency Lock file: After selecting a specific version of each dependency using version constraints Terraform remembers the decisions it made in a dependency lock file so that it can (by default) make the same decision again in future.
- Location of lock file: Current Working Directory.
- Very Important: Lock file currently tracks only Provider dependencies. For modules continue to exact version constraints to ensure that Terraform will always select the same module version.
- Checksum Verification: Terraform will also verify that each package it installs matches at least one of the checksums it previously recorded in the lock file, if any, returning an error if none of the checksums match.

Provider	Version Constraint	terraform init (no lock file)	terraform init (lock file)
AWS	>= 2.0	latest version (3.18.0)	lock file versin (2.50.0)
random	3.0.0	3.0.0	Lock file versin (3.0.0)

Importance of Dependency Lock File

__/__/____

If terraform did not find a lock file, it would download the latest version of the providers that fulfill the version constraints you defined in the required-providers block inside Terraform Settings Block.
If we have lock file, the lock file causes Terraform to always install the same provider version, ensuring that runs across your team or remote sessions will be consistent.

Creating S3 bucket with existing Lock File AWS provider version!

c1_version.tf

```
# Terraform Setting Block
terraform {
  # Terraform version
  required_version = "~> 0.14.6"
  required_providers {
    # AWS Provider
    aws = {
      source  = "hashicorp/aws"
      version = ">= 2.0.0"
    }
    # Random Provider
    random = {
      source  = "hashicorp/random"
      versin  = "3.0.0"
    }
  }
}
# Provider Block
provider "aws" {
  region  = "us-east-1"
  profile = "default" # Defining it for default profile is optional
}
```

c2. s3bucket.tf

```
# Resource Block: Create Random Pet Name
resource "random_pet" "petname" {
    length    = 5
    seperator = "_"
}
# Resource Block: Create AWS S3 bucket
resource "aws_s3_bucket" "sample" {
    bucket = random_pet.petname.id
    acl    = "public-read"
    region = "us-east-1" # Comment this if we are going
                         to use Aws provider v3.x version.
}
```

__/__/____

Terraform Resources Introduction:

- Terraform Language Basics
- Terraform Resource Syntax
- Terraform Resource Behaviour
- Terraform State
- **Terraform Resources**
- Resource Meta-Argument count
- Resource Meta-Argument depends-on
- Resource Meta-Argument for_each
- Resource Meta-Argument lifecycle

Terraform Resource Syntax:

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
    #Block Body
    <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Block Type → `resource` | Block Label → `"aws_instance" "ec2demo"` {

```
# AWS Example
resource "aws_instance" "ec2demo" {
    ami           = "ami-04d29b6f966df1537"
    instance_type = "t2.micro"
}
```

Argument → `ami = "ami-04d29b6f966df1537"` / `instance_type = "t2.micro"`

- Top level & Block inside Blocks
- Top level Blocks: resource, provider
- Block Inside Block: provisioners, - resource specific blocks like tags

- Based on Block Type block labels will be 1 or 2

Example:

Resource - 2 labels

Variables - 1 label

Parmeshwar Gudadhe

__/__/____

Template

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
    # Block Body
    <IDENTIFIER> = <EXPRESSION> # Argument
```

AWS Example

```
resource "aws_instance" "ec2-demo" {
    ami           = "ami-04d29b6f966df1537"
    instance_type = "t2.micro"
}
```

Argument Name OR Identifier

Argument Value OR Expression

Resource Syntax in Detail :

```
# Resource Block to Create VPC
resource "aws_vpc" "vcp_us-west-1" {
    provider   = aws.aws-west-1
    cidr_block = "10.2.0.0/16"
    tags = {
        "Name" = "vpc-1"
    }
}
```

① ② ③ ④

① Resource Type: It determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports.

② Resource Local Name: It is used to refer to this resource from elsewhere in the same Terraform module, but has no significance outside that module's scope.

The resource type and name together serve as an identifier for a given resource and so must be unique within module.

③ Meta-Argument: Can be used with any resource to change the behaviour of resources.

④ Resource Arguments: Will be specific to resource type. Argument values can make use of Expressions or other Terraform Dynamic Language features.

Terraform Resource Behaviour:

Create Resource: Create resources that exist in the configuration but not associated with a real infrastructure object in the state.

Destroy Resource: Destroy resources that exist in the state but no longer exist in the configuration.

Update in-place Resources: Update in-place resources whose arguments have changed.

Destroy and re-create: Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

Format Terraform configuration files

terraform fmt

Observations: *.tf files will change to format them if any format changes exists.

__/__/____

Parmeshwar Gadadhe

c1-versions.tf

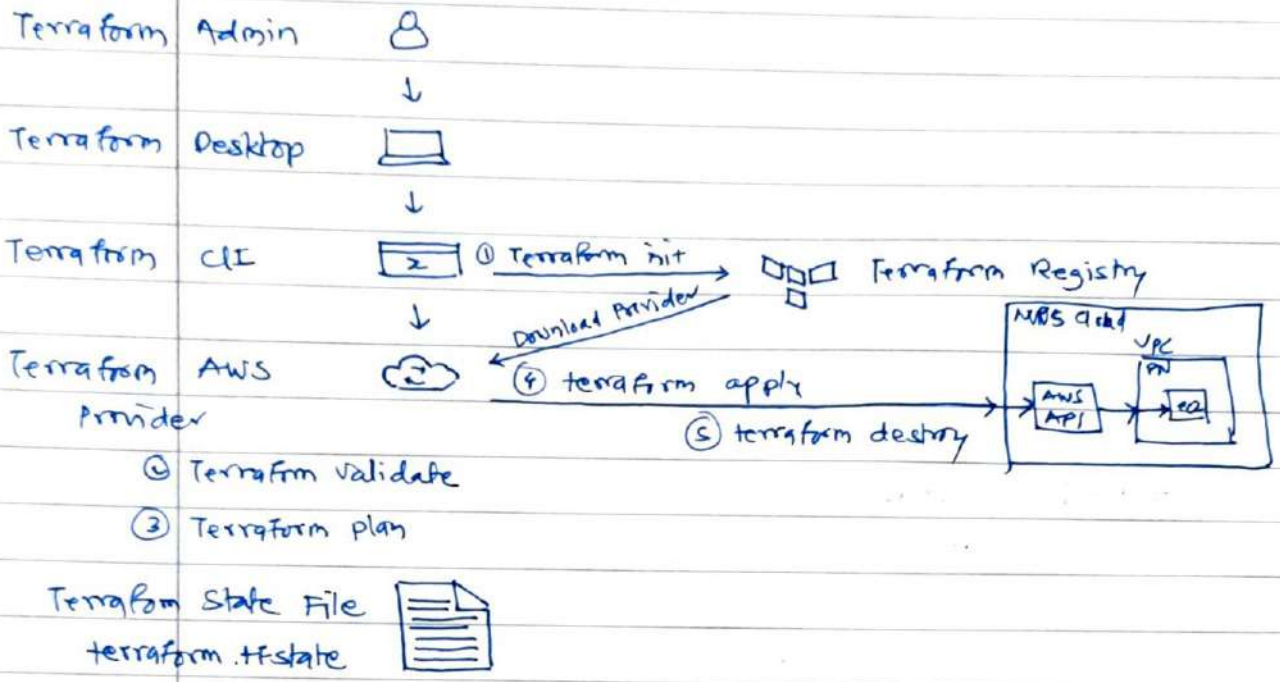
```
# Terraform Block
terraform {
    required_version = "~> 0.14.6"
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 3.0"
        }
    }
}
# Provider Block
provider "aws" {
    region = "us-east-1"
    profile = "default"
}
```

c2-ec2-instance.tf

```
# Create EC2 instance
resource "aws_instance" "my-ec2-vm" {
    ami = "ami-08e0ca9822195bepa"
    instance_type = "t2.micro"
    availability_zone = "us-east-1a"
    tags = {
        "Name" = "Web"
    }
}
```

__/__/__

Understanding Terraform State :



- ✓ Terraform must store state about your managed infrastructure and configuration.
- ✓ This state is used by Terraform to map real world resources to your configuration (.tf files), keep track of metadata and to improve performance for large infrastructures.
- ✓ This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.
- ✓ The primary purpose of terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.
- ✓ When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance. and then potentially update or delete that object in response to future configuration changes.

linkedin: parmeshwargudadhe

__/__/____

Terraform Resource Behaviour Update-In-Place:

```
# Create EC2 Instance
resource "aws_instance" "my_ec2_vm" {
    ami               =  "ami-047a5f1a2770816ef"
    instance_type     =  "t2.micro"
    availability_zone =  "us-east-1a"
    tags = {
        "Name"        =  "Web"
        "tag1"        =  "This is updated tag / test1"
    }
}
```

```
# Review the terraform plan
terraform plan
```

Observation: You should see "~ update in-place"

"Plan: 0 to add, 1 to change, 0 to destroy."

```
# Create / Update Resources
terraform apply -auto-approve
```

Observation: "Apply complete! Resources: 0 added, 1 changed, 0 destroyed"

Important Note: Here we have seen example for update in-place.

✓ Current state: Resources without any change is current state of the resources.

✓ Desired state: What changes are going to be done after terraform apply

__/__/____

Terraform Resource Behaviour Destroy and Re-create Resources:

```
# Before
availability_zone   =  "us-east-1a"

# After
availability_zone   =  "us-east-1b"

# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
   ami             = "ami-047a51fa27710816e"
   instance_type = "t2.micro"
  #availability_zone= "us-east-1a"
   availability_zone= "us-east-1b"
   tags = {
      "Name"    =  "web"
    }
}

# Review the terraform plan
terraform plan
Observation:
  1) -/+ destroy and then create replacement
  2) # aws_instance.my-ec2-vm must be "replaced"
  3) # aws_instance.my-ec2-vm must be "replaced" - This
parameter forces replacement.
  4) "Plan: 1 to add, 0 to change, 1 to destroy."

# Create/Update Resources
terraform apply -auto-approve
Observation: "Apply complete! Resources: 1 added, 0 changed
                                        1 destroyed."
```

linkedin: parmeshwargudadhe

__/__/____

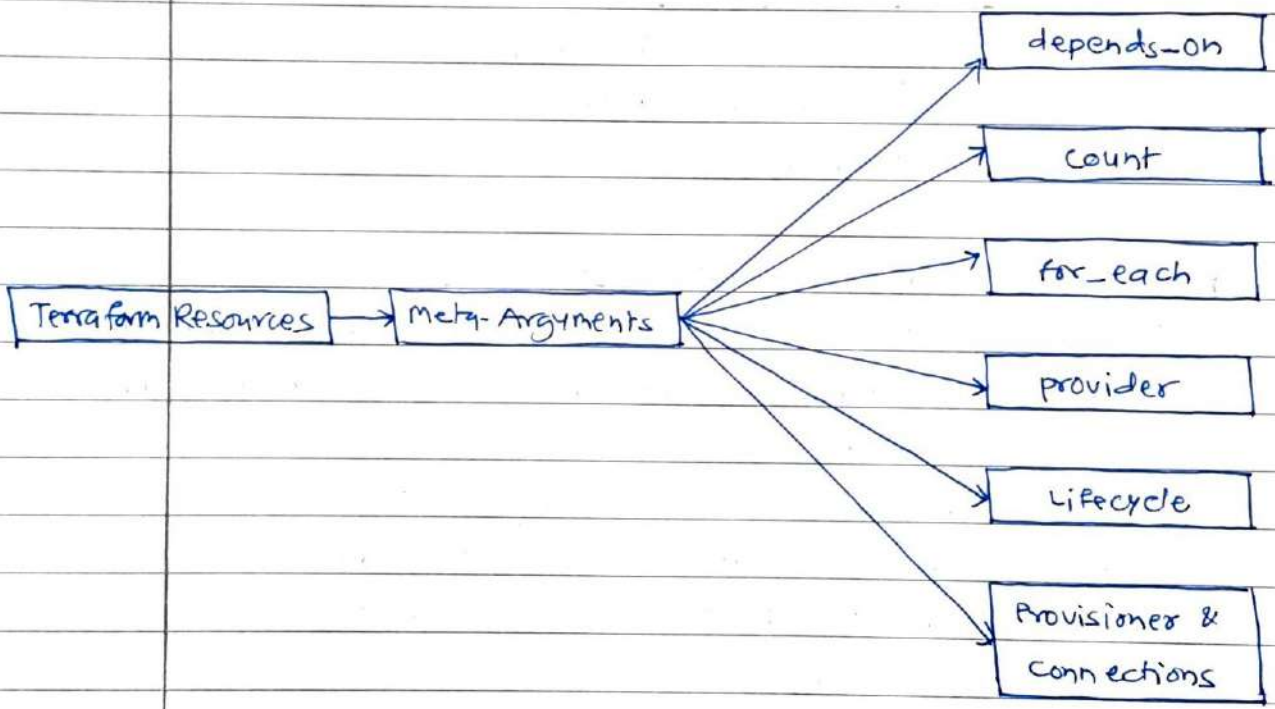
Resource: Destroy Resource:

Destroy Resource
terraform destroy
Observation:
1) - destroy
2) # aws_instance.my-ec2-vm will be destroyed.
3) Plan: 0 to add, 0 to change, 1 to destroy
4) Destroy complete! Resource: 1 destroyed.

Desired States and Current States (High-level Only):

- Desired state: local Terraform Manifest (All *.tf files)
- Current state: Real Resources present in your cloud.

Resource Meta Arguments:



✓ Meta-Arguments can be used with any resource type to change the behaviour of resources.

__/__/____

depends-on : To handle hidden resource or module dependencies that terraform can't automatically infer

count : for creating multiple resource instances according to a count.

for_each : To create multiple instances according to map or set of strings.

provider : For selecting a non-default provider configuration.

lifecycle : Standard Resource behaviour can be altered using special nested lifecycle block within a resource block body.

Provisioners and Connections : For taking extra actions after resource creation (Example: install some app on server or do something on local desktop after resource is created at remote destination.

* Resource Meta-Argument - depends-on :
- ✓ Use the depends_on meta-argument to handle hidden resource or module dependencies that Terraform can't automatically infer.
- ✓ Explicitly specifying a dependency is only necessary when a resource or module relies on some other resource's behaviour but doesn't access any of that resource's data in its arguments.
- ✓ This argument is available in module blocks and in all resource blocks, regardless of resource type.
- ✓ The depends_on meta-argument, if present, must be a

list of references to other resources or child modules in the same calling module.

✓ Arbitrary expressions are not allowed in the depends_on argument value, because its value must be known before Terraform knows resource relationships and thus before it can safely evaluate expressions.

✓ The depends_on arguments should be used only as a last resort. Add comments for future reference about why we added this.

* Use Case: What are we going to implement?
  - # Resource-1 : Create VPC
  - # Resource-2 : Create Subnets
  - # Resource-3: Internet Gateway
  - # Resource-4: Create Route Table
  - # Resource-5: Create Route in Route Table for Internet Access.
  - # Resource-6: Associate the Route Table with the Subnet.
  - # Resource-7: Create SecurityGroup
  - # Resource-8: Create EC2 Instance with Sample web App.
  - # Resource-9: Create Elastic IP depends_on

EIP may require IGW to exist prior to association use depend_on to set an explitit dependency on the IGW.

Hands-on: depends-on

✓ Step-01: Pre-requisite - Create a EC2 Key Pair:
Create EC2 Key pair terraform-key and download .pem file and put ready for SSH login.

✓ Step-02: c1-versions.tf - Create Terraform & Provider Blocks:
Create Terraform Block.
Create ProviderBlock.

c1-versions.tf

```
# Terraform Block
terraform {
  required_version = "~> 0.14.6"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
# Provider Block
provider "aws" {
  region = "us-east-1"
  profile = "default"
}
```

✓ Step-03 : c2-vpc.tf - Create VPC Resources :

Create VPC manually and understand all the resources we are going to create. Delete that VPC and start writing the VPC template using terraform.

Create VPC Resources listed below !

- Create VPC
- Create Subnet
- Create Internet Gateway
- Create Route Table
- Create Route in Route Table for Internet Access.
- Associate Route Table with Subnet.
- Create Security Group in the VPC with port 80, 22 as inbound open.

c2-vpc.tf

```
# Resource Block
# Resource-1: Create VPC
resource "aws_vpc" "vpc_dev" {
    cidr_block = "10.0.0.0/16"
    tags = {
        "Name" = "vpc-dev"
    }
}
# Resource-2: Create Subnets
resource "aws_subnet" "vpc-dev-public-subnet-1" {
    vpc_id = aws_vpc.vpc_id
    cidr_block = "10.0.1.0/24"
    availability_zone = "us-east-1a"
    map_public_ip_on_launch = true
}
# Resource-3: Internet Gateway
resource "aws_internet_gateway" "vpc-dev-igw" {
    vpc_id = aws_vpc.vpc_dev.id
}
# Resource-4: Create Route Table
resource "aws_route_table" "vpc-dev-public-route-table" {
    vpc_id = aws_vpc.vpc-dev.id
}
# Resource-5: Create Route in Route Table for Internet Access
resource "aws_route" "vpc-dev-public-route" {
    route_table_id = aws_route_table.vpc-dev_public-rute-table.id
    destination_cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.vpc-dev_igw.id
}
```

__/__/____

```
# Resource-6: Associate the Route Table with the Subnet
resource "aws_route_table_association" "vpc-dev-publicroute-table-ass" {
  route_table_id = aws_route_table.vpc-dev-public-route-table.id
  subnet_id      = aws_subnet.vpc-dev-public-subnet-1.id
}
# Resource-7: Create Security Group
resource "aws_security_group" "dev-vpc-sg" {
  name        = "dev-vpc-default-sg"
  vpc_id      = aws_vpc.vpc-dev.id
  description = "Dev VPC default Security Group"
  ingress {
    description = "Allow port 22"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    description = "Allow port 80"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] }
  egress {
    description = "Allow all ip and ports outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

✓ Step-04: c3-instance.tf - Create EC2 Instance Resource

- Review apache-install.sh

```
#! /bin/bash
sudo yum update -y
sudo yum install -y httpd
sudo service httpd start
sudo systemctl enable httpd
echo "<h1> Hellow World! </h1>" > /var/www/html.html
```

- Create EC2 Instance Resource

c3-instance.tf

```
# Resource-8: Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
  ami = "ami-0be2609da8855g1ec"
  instance_type = "t2.micro"
  subnet_id = aws_subnet.vpc-dev-public-subnet-1.id
  key_name = "terraform-key"
    #user_data = file("apache-install.sh")
  user_data = <<- EOF
    #! /bin/bash
    sudo yum update -y
    sudo yum install httpd -y
    sudo systemctl enable httpd
    sudo systemctl start httpd
    echo "<h1> Helloworld! </h1> > /var/www/html/index.html
    EOF
  vpc_security_group_ids = [aws_security_group.dev-vpc-sg.id]
}
```

✓ Step-05: c4-elastic-ip.tf - Create Elastic IP Resource

- Create Elastic IP Resource
- Add a Resource Meta-Argument depends-on ensuring Elastic IP gets created only after AWS Internet Gateway in a VPC is present or created.

Resource-9: Create Elastic IP

```
resource "aws_eip" "my-eip" {
    instance    =  aws_instance.my-ec2-vm.id
    vpc         =  true
    depends_on  =  [ aws_internet_gateway.vpc-dev-igw ]
}
```

Step-06: Execute Terraform Commands to Create Resource using Terraform.

```
# Initialize Terraform
terraform init
# Terraform Validate
terraform validate
# Terraform Plan to verify what it is going to create/update/destroy
terraform plan
# Terraform Apply to Create EC2 Instance
terraform apply
```

Step-07: Verify the Resources

- Verify VPC
- Verify EC2 Instance
- Verify Elastic IP
- Review the terraform.tfstate file
- Access Apache Webserver Static page using Elastic IP

```
# Access Application
http://<AWS-ELASTIC-IP>
```

Step-08: Destroy Terraform Resources:

```
# Destroy Terraform Resources
terraform destroy
# Remove Terraform Files
rm -rf .terraform*
rm -rf terraform.tfstate*
```

linkedin: parmeshwarguddadhe

__/__/__

* Introduction to Resources Meta-Argument count:

- ✓ If a resource or module block includes a count argument whose value is a whole number, Terraform will create that many instances.
- ✓ Each instance has a distinct infrastructure object associated with it and each is separately created, updated, or destroyed when the configuration is applied.
- ✓ The count meta-argument accepts numeric expressions. The count value must be known before Terraform performs any remote resource actions.
- ✓ count.index: The distinct index number (starting with 0) corresponding to this instance.
- ✓ When count is set, Terraform distinguishes between the block itself and the multiple resources or module instances associated with it. Instances are identified by an index number, starting with 0. Ex. aws_instance.myvm[0]
- ✓ Module support for count was added in Terraform 0.13, and previous versions can only use it with resources.
- ✓ A given resource or module block cannot use both count and for_each.

→ Use Case: What are we going to implement?

Use Meta-Argument count to create multiple EC2 instances:

```
# Create EC2 instance
resource "aws_instance" "web" {
    ami = "ami-047a51fa27710816e"
    instance_type = "t2.micro"
    count = 5
    tags = {
      #"Name" = "web"
      "Name" = "web-${count.index}"
    }
}
```

count

count.index

aws_instance.web[0]
aws_instance.web[1]
aws_instance.web[2]
aws_instance.web[3]
aws_instance.web[4]

__/__/____

Hands-on: count:

- In general, 1 EC2 Instance Resource in Terraform equals to 1 EC2 instance in Real AWS cloud.
- 5 EC2 Instance Resources = 5 EC2 Instances in AWS cloud.
- With meta-Argument count this is going to become simple.

```
# Create EC2 Instance
resource "aws_instance" "web"
    ami = "ami-097a51a27710816e"
    instance_type = "t2.micro"
    count = 5
    tags = {
      #"Name" = "web"
    } "Name" = "web-${count.index}"
}
```

Execute Terraform Commands:

```
# Initialize Terraform
terraform init
# Terraform Validate
terraform validate
# Terraform Plan whats going / create / update / destroy.
terraform plan
# Terraform Apply to Create EC2 Instance.
terraform apply.
```

Also check using count index web-0, web-1, web-2, web-3, web-4.

```
# Destroy Terraform Resources
terraform destroy
# Remove Terraform Files
rm -rf .terraform*
rm -rf terraform.tfstate*
```

linkedin: parmeshwargudadhe

__/__/____

Resource Meta-Argument ~~for~~ each :

- If a resource or module block includes a for_each argument whose value is a map or set of strings, Terraform will create one instance for each member of that map or set.
- Each instance has a distinct infrastructure object associated with it, and each is seperately created, updated, or destroyed when the configuration is applied.
- In blocks where for_each is set, an additional each object is available in expressions, so you can modify the configuration of each instance.

each.key - The map key (or set member) corresponding to this instance.

each.value - The map value corresponding to this instance. (If a set was provided, this is the same as each.day)

- For set of strings, each.key = each.value

```
for_each = toset(["Jack", "James"])
each.key = Jack
each.value = James
```

- For maps, we use each.key & each.value

```
~~Each~~.value
for_each = {
dev = "my-dapp-bucket"
}
each.key   = dev
each.value = my_dapp-bucket
```

- Module support for for_each was added in Terraform 0.13 and previous versions can only use it with resources.
- A given resource or module block cannot use both count and for_each.

__/__/____

Usecase-1 :: for_each Maps

Resource-1: Use Meta-Argument for_each with Maps to create Multiple S3 buckets using single Resource.

```
# Create S3 Bucket per environment with for_each and Maps
resource "aws_s3_bucket" "myc3bucket" {
  for_each = {
    dev  = "my-dapp-bucket"
    qq   = "my-qapp-bucket"
    stag = "my-sapp-bucket"
    prod = "my-papp-bucket"
  }
  bucket = "${each.key}-${each.value}"
  acl    = "private"
  tags   = {
    eachvalue   = each.value
    Environment = each.key
    bucketname  = "${each.key}-${each.value}"
  }
}
```

Define for_each with Map with Key value pairs

Use each.key and each.value for the S3 Bucket name

Use each.key & each.value for S3 Bucket tags

AWS

Buckets (4)

Name	Region
dev.my-dapp-bucket	us-east-1
prd.my-papp-bucket	us-east-1
qq.my-qapp-bucket	us-east-1
stag.my-sapp-bucket	us-east-1

linkedin: parmeshwargudadhe

__/__/____

Usecase-2: for_each Set of Strings (toset)

Resource-1: Use Meta-Argument for_each with Set of String to create multiple IAM users using single Resource.

```
# Create 4 IAM Users
resource "aws_iam_user" "myuser" {
    for_each = toset (["Param", "Harshy", "shubham", "Sayali"])
    name     = each.key
}
```

__/__/____

★ Resource - Meta-Argument Lifecycle:-

✓ Lifecycle is a nested block that can appear within a resource block.

✓ The lifecycle block and its contents are meta-arguments, available for all resource block regardless of type.

create_before_destroy

```
resource "aws_instance" "web" {
    ami = "ami-912bcfsp90f7Asq"
    instance_type = "t2.micro"
    availability_zone = "us-east-1a"
   #availability_zone = "us-east-1b"
    tags = {
    "Name" = "web-1"
    }
    lifecycle {
     create_before_destroy = true
    }
}
```

prevent_destroy

```
resource "aws_instance" "web"
    ami = "ami-912bcfsp90f7Asq"
    instance_type = "t2.micro"
    tags = {
    "Name" = "web-2"
    }
    lifecycle {
       prevent_destroy = true
    }
}
```

linkedin: parmeshwargudadhe

__/__/__

ignore changes

```
resource "aws_instance" "web" {
    ami = "ami-091bcb5fa77e4892"
    instance_type = "t2-micro"
    tags = {
      "Name" = "web-3"
    }
    lifecycle {
      ignore_changes = [
        # Ignore changes to tags, eg. because a management agent.
        # update these based on some ruleset managed elsewhere.
        tags,
      ]
    }
}
```

★ Terraform Variables :-

There are three types of terraform variables -

1) Terraform Input Variables
2) Terraform Output Values
3) Terraform Local Values.

1) Terraform Input Variables :-

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code and allowing to be shared between different configurations.

① Input variables - Basics
② Provide Input Variables when prompted during terraform plan or apply.
③ Override default variable values using CLI argument -var
④ Override default variable values using Environment Variables (TF_var_aa)
⑤ Provide Input variables using terraform.tfvars files.
⑥ Provide Input variables using <any-name>.tfvars file with CLI argument -var-file
⑦ Provide Input variables using auto.tfvars files.
⑧ Implement complex type constructors like list & maps in Input variables.
⑨ Implement Custom Validation Rules in Variables
⑩ Protect Sensitive Input variables.

05 Input Variables Basics - Hands-on :

① apache-install.tf

```
#!/bin/bash
sudo yum update -y
sudo yum install httpd -y
sudo systemctl enable httpd
sudo systemctl start httpd
echo "<h1> Hello World! </h1>" > /var/www/html/index.html
```

② versions.tf

```
# Terraform Block
terraform {
    required_version = "~> 1.3.0"
    required_providers {
        aws = {
            source  = "hashicorp/aws"
            version = "~> 4.50.0"
        }
    }
}
# Provider Block
provider "aws" {
    region  = var.aws_region
    profile = default
}
```

③ variables.tf

```
# Input Variables
variables "aws_region" {
    description = "In this region resources will be created"
    type        = string
    default     = "us-east-1"
}
```

linkedin: parmeshwargudadhe
__/__/____

```
variables "ec2_ami_id" {
    description = "AMI ID"
    type        = string
    default     = "ami-0915bcb5fa77e4892"
}

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

④ security-group.tf

```
# Create Security Group for SSH Traffic
resource "aws_security_group" "vpc-ssh" {
    name        = "vpc-ssh"
    description = "Dev VPC SSH"
    ingress {
        description = "Allow Port 22"
        from_port   = 22
        to_port     = 22
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    egress {
        description = "Allow all IP and Ports outbound"
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
```

__/__/____

```
# Create Security Group for Web Traffic
resource "aws_security_group" "vpc_web" {
    name        = "vpc-web"
    description = "Dev VPC Web"
    ingress {
        description = "Allow port 80"
        from_port   = 80
        to_port     = 80
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    ingress {
        description = "Allow port 443"
        from_port   = 443
        to_port     = 443
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    egress {
        description = "Allow all IP and ports outbound"
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
        cidr_blocks = ["0.0.0.0./0"]
    }
}
```

linkedin: parmeshwargudadhe
__/__/____

⑤ ec2-instance.tf

```
# Create EC2 Instance
resource "aws_instance" "my-ec2-vm" {
    ami           = var.ec2_ami_id
    instance_type = "t3.micro"
    key_name      = "PARAM"
    count         = var.ec2_instance_count
    user_data     = <<-EOF
        #!/bin/bash
        sudo yum update -y
        sudo yum install httpd -y
        sudo systemctl enable httpd
        sudo systemctl start httpd
        echo "<h1> Hello World! </h1>" >> /var/www/html/index.html
        EOF
    vpc_security_group_ids = [aws_security_group.vpc-ssh.id, aws_security_group.vpc-web.id]
    tags = {
      "Name" = "myec2vm"
    }
}
```

__/__/____

02 Input variables Assign when Prompted :

- ✓ Add a new variable in variables.tf named ec2_instance_type without any default value.
- ✓ As the variable doesn't have any default value when you execute terraform plan or terraform apply it will prompt for the variable.

① variables.tf

```
:
:
variable "ec2_instance_type" {
    description = "This is instance type"
    type        = string.
}
```

② ec2_instance.tf

```
resource "aws_instance" "webapp"
    ami           = var.ec2_ami_id
    instance_type = var.ec2_instance_type
    keyname       = "PARAM"
    count         = var.ec2_instance_count
                :
                :
```

terraform init

terraform validate

terraform fmt

terraform plan

linkedin: parmeshwargudadhe.

__/__/____

03 Input Variables override default value with cli argument -var :-

- ✓ Keep default variable value as is inside .tf file
- ✓ We are going to override the default values defined variables.tf by providing new values using the -var argument using CLI.

```
# Initialize Terraform
terraform init

# Validate Terraform configuration files
terraform validate.

# Format Terraform configuration files.
terraform fmt

# Option-1 (Always provide -var for both plan and apply)
# Review the terraform plan
terraform plan -var="ec2_instance_type=t2.micro"
               -var="ec2_instance_count=1"

# Create Resources (Optional)
terraform apply -var="ec2_instance_type=t2.micro"
                -var="ec2_instance_count=1"

# Option-2 (Generate plan file with -var and use that with apply)
# Generate Terraform plan file
terraform plan -var="ec2_instance_type=t2.micro"
               -var="ec2_instance_count=1" -out v3out.plan

# Create / Deploy Terraform Resources using Plan file
terraform apply v2out.plan
```

__/__/____

04 Input Variables Override with Environment Variables:

✓ Set environment variables and execute terraform plan to see if it overrides default values.

```
# Sample: SET Environment Variables
export TF_VAR_ec2_instance_count=1
export TF_VAR_ec2_instance_type = t2.Micro
echo $TF_VAR_ec2_instance_count, $TF_VAR_ec2_instance_type

# Initialize Terraform
    terraform init

# Validate Terraform Configuration files
    terraform validate

# Format Terraform Configuration files
    terraform fmt

# Review the terraform plan
    terraform plan

# UNSET Environment variables after hands-on
    unset TF_VAR_ec2_instance_count
    unset TF_VAR_ec2_instance_type
    echo $TF_VAR_ec2_instance_count, $TF_VAR_ec2_instance_type
```

linkedin : parmeshwargudadhe

__/__/____

05 Assign Input variables from terraform.tfvars :-

- ✓ Create a file named terraform.tfvars and define variables.
- ✓ If the file name is terraform.tfvars, terraform will auto-Load the variables present in this file by overriding the default in variables.tf

terraform.tfvars

```
ec2_instance_count = 2
ec2_instance_type = t3.small
```

```
# Initialize Terraform
terraform init

# Validate Terraform Configuration files
terraform validate

# Format Terraform Configuration files
terraform fmt

# Review the terraform plan
terraform plan

# Create Resources
terraform apply
```

06. Assign Input Variables with -var-file argument:

✓ If we plan to use different names for .tfvars files, then we need to explicitly provide the argument -var-file during the terraform plan or terraform apply.

✓ We will use following thing in this example:-

- variables.tf: aws_region variable will be picked with default value.
- terraform.tfvars: ec2_instance_count variable will be from this file.
- web.tfvars: ec2_instance_type variable will be picked from this file.
- app.tfvars: ec2_instance_type variable will be picked from this file.

```
# Initialize terraform
terraform init

# Validate Terraform configuration files
terraform validate

# Format Terraform Configuration files.
terraform fmt

# Review the terraform plan
terraform plan -var-file="web.tfvars"
terraform plan -var-file="app.tfvars"
```

file.auto.tfvars

linkedin: parmeshwargudadhe

__/__/____

07 Auto load input variables with *.auto.tfvars files:

- ✓ We will create a file with extension as *.auto.tfvars.
- ✓ With this extension, whatever may be the file name, the variables inside these files will be auto loaded during terraform plan or terraform apply.

web.auto.tfvars

```
ec2_instance_type = "t2.micro"
```

```
# Initialize Terraform
terraform init

# Validate Terraform Configuration files
terraform validate

# Format Terraform Configuration files
terraform fmt

# Review the terraform plan
terraform plan
```

08 Implement Complex Type Constructors like list and maps :-

08.01 : Implement Variable Type as list :-

- list (or tuple) : a sequence of values, like ["us-east-1a", "us-east-1c"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
- Implement list function for variable ec2_instance_type.

```
# Implement list function in variables.tf
variable "ec2_instance_type" {
    description = "Ec2 Instance Type"
    type        = list(string)
    default     = ["t3.micro", "t3.small", "t3.medium"]
}

# Reference values from list in ec2_instance.tf
instance_type = var.ec2_instance_type[0]    - t3.micro
instance_type = var.ec2_instance_type[1]    - t3.small
instance_type = var.ec2_instance_type[2]    - t3.medium

    # Initialize Terraform
    terraform init

    # Validate Terraform Configuration files
    terraform validate

    # Format Terraform Configuration files
    terraform fmt

    # Review the terraform plan
    terraform plan
```

08.02: Implement Variable Type as Map:

- Map (or object): a group of values identified by named labels, like { name = "Mabel", age = 52 }.
- Implement Map function for variable ec2_instance_tags

```
# Implement Map function for tags: variables.tf
variables "ec2_instance_tags" {
    description = "EC2 Instance tags"
    type        = map(string)
    default     = {
        "Name" = "ec2-web"
        "Tier" = "Web"
    }

# Reference Values from Map in ec2-instance.tf
tags = var.ec2_instance_tags

# Implement Map function for Instance Type variables.tf
# Important Note: Comment "ec2-instance_type" variable with list function
variable "ec2-instance_type_map" {
    description = "EC2 Instance Type using maps"
    type        = map(string)
    default     = {
        "small-apps"  = "t3.micro"
        "medium-apps" = "t3.medium"
        "big-apps"    = "t3.large"
    }

# Reference Instance type from Maps Variables: resources.tf
instance_type = var.ec2_instance_type_map["small-apps"]
instance_type = var.ec2_instance_type_map["medium-apps"]
instance_type = var.ec2_instance_type_map["big-apps"]
```

09 Implement Custom Validation Rules in Variables :-

✓ The terraform console command provides an interactive console for evaluating expressions.

09.01 Learn Terraform Length Function:

→ length determines the length of a given list, map, or string.

→ If given a list or map, the result is the number of elements in that collection. If given a string, the result is the number of characters in the string.

```
# Go to Terraform Console
  terraform console
# Test Length Function
  Template: length()
  length ("hi")
  length ("hello")
  length ["a", "b", "c"] # list
  length ({"key" = "value"}) # map
  length ({"key1" = "value1", "key2" = "value2"}) # map
```

09.02 Learn Terraform Substring Function:

→ substr extracts a substring from a given string by offset and (maximum) length.

```
# Go to Terraform Console.
  terraform console
# Test Length Substr Function
  Template : substr(string, offset, length)
  substr ("Hello", 1, 4)
  substr ("Hello World!", 0, 6)
  substr ("Hello World!", 0, 1)
  substr ("Hello World!", 0, 0)
  substr ("Hello World!" 0, 10)
```

linkedin: parmeshwargudadhe

__/__/____

09.03 Implement Validation Rule for ec2_ami_id variable:

```
variable "ec2_ami_id" {
    description = "AMI ID"
    type        = string
    default     = "ami-0be2609ba883822ec"
    validation {
      condition = length(var.ec2_ami_id) > 4 &&
                  substr(var.ec2_ami_id, 0, 4) == "ami-"
      error_message = "The ec2_ami_id value must be a valid
                       AMI ID, string with \"ami-\"."
    }
}
```

```
terraform init

terraform validate

terraform fmt

terraform plan
```

10 Protect Sensitive Input variables :-

- ✓ When using environment variables to set sensitive values, keep in mind that those values will be in your environment and command-line history
  Example: export TF_VAR_db_username=admin
  TF_VAR_db_password=insecurepassword
- ✓ When you use sensitive variables in your Terraform configuration, you can use them as you would any other variable.
- ✓ Terraform will redact these values in command output and log files, and raise an error when it detects that they will be exposed in other ways.
- ✓ Important Note-1 : Never check-in secrets.tfvars to git repositories.
- ✓ Important Note-2 : Terraform state file contains values for these sensitive variables terraform.tfstate. You must keep your state file secure to avoid exposing this data.

versions.tf

```
terraform {
  required_version = "~> 1.3.0"
  required_provider {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.50.0"
    }
  }
}
provider "aws" {
  region = var.aws_region
  profile = "default"
}
```

linkedin: parmeshwargudadhe

__/__/____

rds.tf

```
resource "aws_db_instance" "db1" {
    allowcated_storage  = 5
    db_name             = "mydb1"
    engine              = "mysql"
    engine_version      = "5.7"
    instance_class      = "db.t2.micro"
    username            = var.db_username
    password            = var.db_password
    skip_final_shapshot = true
}
```

variables.tf

```
variables "aws_region" {
    description = "aws region"
    type        = string
    default     = "us-east-1"
}
variables "db_username" {
    description = "DB username"
    type        = string
    sensitive   = true
}
variable "db_password" {
    description = "DB Password"
    type        = string
    sensitive   = true
}
```

secrets.tfvars

```
db_username = "admin"
db_password = "insecurepasswd"
```

linkedin: parmeshwargudadhe

__/__/____

11 Understand about File function :-

- file reads the contents of a file at the given path and returns them as a string.

Syntax: file(path)

Example: > file("${path.module}/hello.txt")
Hello World!

resource.tf

```
resource "aws_instance" "webapp" {
  ami           =
  instance_type =

  user_data     = file("apache_install.sh")
```

② Terraform Output Values :-

Output values are like the return values of a Terraform module and have several uses.

1. A root module can use outputs to print certain values in the CLI output after running terraform apply.
2. A child module can use outputs to expose a subset of its resource attributes to a parent module.
3. When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state data source.

output.tf

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

# Argument Reference: EC2 Instance Private IP
  output "ec2_instance_publicip" {
      description = "EC2 Instance Private IP"
      value       = aws_instance.webapp.private_ip
  }

# Argument Reference: Security Groups associated to EC2 instance
  output "ec2_instance_security_group" {
      description = "List Security Group associated with EC2 instance"
      value       = aws_instance.webapp.security_groups
  }

# Attributes Reference - Create Public DNS URL with http:// appended
  output "ec2_publicdns" {
      description = "Public DNS URL of an EC2 instance"
      value       = "http://${aws_instance.webapp.public_dns}"
      # sensitive = true  # Uncomment it during step-04 execution
  }
```

linkedin: parmeshwargudadhe

__/__/____

Query Terraform Outputs :-

- Terraform will load the project state file, so that using terraform output command we can query the state file.

```
terraform output
terraform output ec2_instance_public_ip
terraform output ec2_instance_public_dns
```

Output Values - Suppressing Sensitive Values in outputs :-

- We can redact the sensitive outputs using sensitive = true in output block.
- This will only redact the cli output for terraform plan & apply.
- When you query using terraform output, you will be able to fetch exact values from terraform.tfstates files.
- Add sensitive=true for output ec2_publicdns

outputs.tf

```
# Attributes Reference - Create Public DNS URL with http:// appended
output "ec2_publicdns" {
    description = "Public DNS URL of an EC2 Instance"
    value       = "http://${aws_instance.my-ec2-vm.public_dns}"
    sensitive   = true
}
```

Test the flow -

```
# Terraform Apply
terraform apply -auto-approve
```

Observation: You should see the value as sensitive.

```
# Query using terraform output
terraform output ec2_publicdns
```

Observation: You should get non-redacted original value from terraform.tfstate file.

locals

__/__/__

Terraform Variables : Local Values

Understand - DRY Principle

DRY means Don't Repeat Yourself

Using local value we will be able to reduce the repeatative code complex code expressions to simple names and we use those simple names to where it is required.

Local Values!

- ✓ A local value assigns a name to an expressions so you can use that name multiple times within a module without repeating it.
- ✓ Local values are like a function's temporary local variables

We can define locals like this -

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

- ✓ Once a local value is declared, you can reference it in expressions as local.<NAME>.

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    service = local.service_name
    owner   = local.owner
  }
}
```

__/__/____

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

- ✓ Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration.
- ✓ If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.
- ✓ The ability to easily change the value in a central place is the key advantage of local values.
- ✓ In short, use local values only in moderation.

versions.tf

```
# Terraform Block
  terraform {
    required_version = "~> 1.3.0"
    required_provider {
      aws = {
        source = "hashicorp/aws"
        version = "~> 4.50.0"
      }
    }
  }

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

LinkedIn: parmeshwargudadhe

variables.tf

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

# App Name S3 bucket used for
variable "app-name" {
    description = "Application Name"
    type        = string
}

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

bucket.tf

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

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

__/__/____

Terraform Datasources :-

- ✓ Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.
- ✓ Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another seperate Terraform configuration.
- ✓ If there is another terraform project and from that if you want to gather some information and use in this current terraform project also you can use that.
- ✓ A data source is accessed via a special kind of resource known as a data resource, declared using a data block.

Example:

```
# Get latest AMI ID for Amazon Linux 2 OS
data "aws_ami" "amznlinux" {
    most_recent     = true
    owners          = ["Amazon"]
    filters {
        name        = "name"
        values      = ["amzn2-ami-hvm-*"]
    }
    filters {
        name        = "root-device-type"
        values      = ["ebs"]
    }
    filter {
        name        = "virtualization-type"
        values      = ["hvm"]
    }
    filter {
        name        = "architecture"
        values      = ["x86_64"]
    }
}
```

✓ Each data resource is associated with a single data source, which determines the kind of object (or objects) it reads and what query constraint arguments are available.

✓ Data resources have the same dependency resolution behaviour as defined for managed resources. Setting the depends_on meta argument within data block defers reading of the data source until after all changes to the tependencies have been applied.

✓ We can refer the data resource in a resource as depicted. ↓

```
# Create EC2 instance - Amazon linux 2 latest
 resource "aws_instance" "my-ec2-vm" {
    ami           = data.aws_ami.amzlinux.id
    instance_type = var.ec2_instance_type
    key_name      = "terraform-key"
    user_data     = file("apache_install.sh")
    vpc_security_group = [aws_security_group
    tags = {
        "Name" = "am2-linux-vm"
    }
 }
```

• Meta-Arguments for Datasources :-

✓ Data resources support the provider meta-argument as defined for managed resources, with the same syntax and behaviour

✓ Data resources do not currently have any customization settings available for their lifecycle, but the lifecycle nested block is reserved in case any are added in future versions.

✓ Data resources support count and for_each meta-arguments as defined for managed resources, with the same syntax and behaviour.

✓ Each instance will seperately read from its data source with its own variant of the constraint of the constraint arguments, producing an indexed result

__/__/____

- Create a Datasource to fetch latest AMI ID:
  - ✓ Create or review manifest ami-datasource.tf
  - ✓ Go to AWS Mgmt Console > Services > EC2 > Images > AMI
  - ✓ search for "Public Images" > Provide AMI ID
    - We can get AMI Name format
    - We can get Owner Name
    - Visibility
    - Platform
    - Root Device Type
    - and many more info here.
  - ✓ Accordingly using this information build your filters in datasource.

- Reference the datasource in ec2-resource.tf

```
ami = data.aws_ami.amzlinux.id
```

- Test using terraform commands:

```
terraform init
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
terraform destroy -auto-approve
```

- Reference - Terraform Documentation: Data Source: aws_ami

linkedin: parmeshwarguddadhe

__/__/____

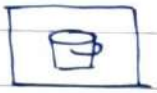
Terraform State :

Topics : I) Terraform Remote State Storage.
II) Terraform Commands from State Perspective.

What is terraform backend ?

Backend are responsible for storing state and providing an API for state locking.

Terraform
State storage

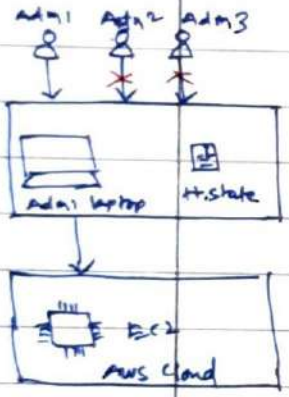


AWS S3 Bucket

Terraform
State locking

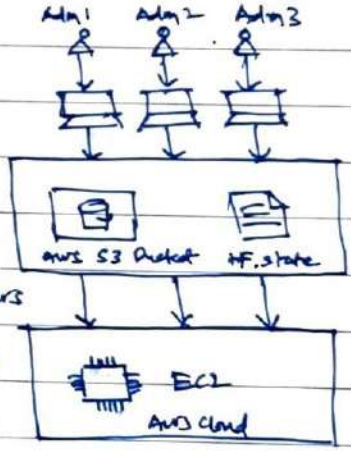
AWS Dynamo DB

Local State File



* multiple team members cannot update the infrastructure as they don't have access to state file
* This means we need store the state file in a shared location.

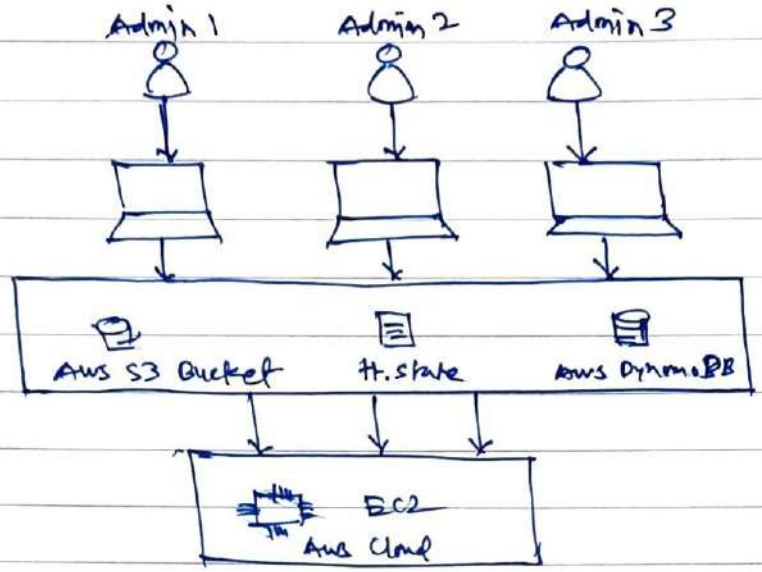
Remote State File



* Using Terraform Backend concept we can use Aws S3 as the shared storage for state files
* If two team members are running Terraform at the same time, you may run into race condition as multiple terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.
* Implement State locking.

__/__/__

Terraform Remote State File with State locking :-



- ✓ Not all backends support state locking. AWS S3 supports state locking.
- ✓ State locking happens automatically on all operations that could write state.
- ✓ If state locking fails, Terraform will not continue.
- ✓ You can disable state locking for most commands with the lock flag but is not recommended.
- ✓ If acquiring the lock is taking longer than expected, Terraform will output a status message.
- ✓ If terraform doesn't output a message, state locking is still occuring if your backend support it.
- ✓ Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Terraform Commands – State Perspective :–

1) terraform show
2) terraform refresh
3) terraform plan
4) terraform force-unlock
5) terraform taint
6) terraform untaint
7) terraform state
8) terraform apply target.

linkedin: parmeshwargudadhe

__/__/____

Terraform Backend: Remote State Storage

Terraform Remote State File with State locking:

```
# Terraform Block
terraform {
  required_version = "~> 1.3.0"
  required_providers = {
    aws = {
      source  = "hashicorp/aws"
      version = "4.50.0"
    }
  }
  # Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket = "terraform-stacksimply"
    key    = "dev/terraform.tfstate"
    region = "us-east-1"
  }

    # Enable during step-09
    # For State locking
    dynamodb_table = "terraform-dev-state-table"
  }
}
```

Terraform State Storage to Remote Backend

Terraform State locking

__/__/____

Terraform Remote State Storage & locking : Hands-on

- Create S3 Bucket :
  - Go to services -> S3 -> Create Bucket
  - Bucket name: terraform-stack-backup
  - Region: US-East (N. Virginia)
  - Block setting for Block Public Access: leave to defaults
  - Bucket Versioning: Enable
  - Rest all leave to defaults
  - Click on create Bucket
  - Create Folder
    - Folder Name: dev
    - Create Folder

- Terraform Backend Configuration :
  - Terraform Backend as S3
  - Add the below listed Terraform backend block in Terraform Setting block main.tf

provider.tf

```
terraform {
  required_version = "~> 1.3.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.50.0"
    }
  }
  backend "s3" {
    bucket = "terraform_state-backup"
    key    = "dev/terraform.tfstate"
    region = "us-east-1"
  }
}
```

__/__/____

Bucket Versioning - Test :-

- Update in variables.tf

```
variable "instance_type" {
    description = "EC2 Instance type-sizing"
    type        = string
  #default     = "t2.micro"
    default     = "t3.micro"
}
```

Execute Terraform Commands :-

```
terraform plan
terraform apply -auto-approve
# Verify S3 Bucket for terraform.tfstate file
bucket_name/dev/terraform.$Fstate
```

observation: New version of terraform.$fstate file will be created.

Destroy Resources :-

```
terraform destroy -auto-approve
```

- ## Terraform State Locking Introduction :-

Terraform provides locking to prevent concurrent runs against the same state. Locking helps make sure that only one team member runs terraform configuration. locking helps us. prevent conflicts, data loss and state file corruption due to multiple runs on same state file.

Create Dynomo DB Table :-

- Table Name : terraform-dev-state-table
- Partition key (primary key) : LockID (Type as string)
- Table setting : Use default settings (checked)
- Click on Create

Update DynamoDB Table entry in backend in main.tf :

- Enable state locking in versions.tf

```
# Adding Backend as S3 for remote state Storage with state locking
backend "s3" {
    bucket  = "terraform-state-backup"
    key     = "dev2/terraform.tfstate"
    region  = "us-east-1"

    # For state locking
    dynamodb_table = "terraform-dev-state-table"
}
```

- Execute Terraform Commands :

```
# Initialize Terraform
  terraform init

# Review the terraform plan
  terraform plan
```

observation :

1) Below messages displayed at start and end of command.
   Acquiring state lock. This may take a few moments...
   Releasing state lock. This may take a few moments...
2) Verify DynamoDB Table -> Items tab

```
# Create Resources
  terraform apply -auto-approve
# verify S3 bucket for terraform.tfstate file
  bucket-name/dev2/terraform.tfstate
```

Observation :

Now versin of terraform.tfstate file will be created in dev2 folder.

```
# Destroy Resources
```

linkedin: parmeshwargudadhe

__/__/____

Understand more about Terraform Backends:-

Terraform Backends:

✓ Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored, etc.

✓ Where Backends are Used:

- Backend configuration is only used by Terraform CLI.
- Terraform Cloud and Terraform Enterprise always use their own state storage when performing Terraform runs, so they ignore any backend block in the configuration.
- For Terraform Cloud users also it is always recommended to use backend block in Terraform configuration for commands like terraform taint which can be executed only using Terraform CLI.

✓ Terraform Backends:

There are two things backends will be used for

1) where state is stored.
2) where operations are performed.

Store State

✓ Terraform uses persistent state data to keep track of the resources it manages.

✓ Everyone working with a given collection of infrastructure resources must be able to access the same state data (shared state storage).

State locking

✓ State locking is to prevent conflicts and inconsistencies when the operations are being performed.

Operations

✓ "operations" refers to performing API requests against infrastructure services in order to create, read, update or destroy resources. apply destroy etc.

✓ Not every terraform subcommand performs API operations; many of them only operate on state data.

✓ Only two backends actually perform operations: local and remote.

✓ The remote backend can perform API operations remotely, using terraform Cloud or Terraform Enterprise.

Terraform State Commands.

__/__/____

- Backend Types :-

1) Enhanced Backends :

✓ Enhanced backends can both store state and perform operations. There are only two enhanced backends: local and remote

✓ Example for Remote Backend Performing Operations: Terraform Cloud, Terraform Enterprise.

2) Standard Backends :

✓ standard backends only store state, and rely on the local backend for performing operations.

✓ Example: AWS S3, Azure RM, Consul, etcd, gcp http and many more.

- Terraform State Commands :-

- Pre-requisite : cl-versions.tf

✓ Update your backend block with your bucket name, key and region.

✓ Also update dynamodb table name.

```
# Terraform Block
  terraform {
      :
  }
  backend "s3" {
      bucket  = "terraform-state-backup"
      key     = "statecommands/terraform.tfstate"
      region  = "us-east-1"
      dynamodb_table = "terraform-dev-state-table"
  }
}
      :
```

- **Terraform show Command to review Terraform plan files :-**
  - ✓ The terraform show command is used to provide human-readable output from a state or plan file.
  - ✓ This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as.
  - ✓ Terraform plan output files are binary files, we can read them using terraform show command.

```
terraform init

terraform plan -out=v1plan.out

terraform show v1plan.out

terraform show -json v1plan.out
```

- **Terraform show command to Read State files :-**
  - ✓ By default, in the working directory if we have terraform.tfstate file, when we provide the command terraform show it will read the state file automatically and display output.

```
terraform apply -auto-approve

terraform show
```

Observation : It should display the state file

__/__/____

Understand terraform refresh in detail :-

- This commands comes under Terraform Inspecting state.
- Understanding terraform refresh clears a lot of doubts in our mind and terraform state file and state feature.
- The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.
- This can be used to detect any drift from the last-known state, and to update the state file.
- This does not modify infrastructure, but does modify the state file. If the state is changed, this may cause changes to occur during the next plan or apply.
- terraform refresh: Update local state file against resources in cloud.
- Desired State: Local Terraform Manifest (All *.tf files)
- Current state: Real Resources present in your cloud.
- Command order of Execution: refresh, plan, make a decision, apply.
- Why? lets understand that in detail about this order of execution.

Add a new tag to EC2 instance using AWS Management Console :-

"demotag" = "refreshtest"

Execute terraform plan :

. You should see terraform state file updated with demo tag

Execute refresh

terraform refresh

Review terraform state file

1) terraform show

2) Also verify s3 bucket, new version of terraform.tfstate file will be created.

linkedin: parmeshwargudadhe

- Why you need to the execution in this order (refresh, plan, make a decision, apply) ?
  - ✓ There are changes happened in your infrastructure manually and not via terraform.
  - ✓ Now decision to be made if you want those changes or not.
  - ✓ choice-1 : If you want those changes proceed with terraform apply so manually changes we have done on our cloud EC2 instance will be removed.
  - ✓ choice-2 : If you want those changes, refer terraform.tfstate file about changes and embed them in your terraform manifests (example: versions.tf) and proceed with flow (refresh, plan, review execution plan and apply)

- I picked choice-2, So i will update the tags in c4-ec2-instance.tf :-
  - ✓ Update in version.tf

```
tags = {
  "Name" = "am2-linux-vm"
  "demotag" = "refreshtest"
}
```

- Execute the commands to make our manual change official in terraform manifests and tfstate files perspective.

```
terraform refresh

terraform plan

terraform apply -auto-approve
```

__/__/____

Terraform State Command :-

Terraform state list and show commands :

- ✓ These two commands comes under Terraform Inspecting state.
- ✓ terraform state list : This command is used to list resources within a Terraform state.
- ✓ terraform state show : This command is used to show the attributes of a single resources in the Terraform state.

List Resources from Terraform state
terraform state list

Show the attributes of a single resource from Terraform state.
terraform state show data.aws_ami.amzlinux
terraform state show aws_instance.my-ec2-vm

Terraform State mv command :-

- ✓ This command comes under Terraform moving Resources.
- ✓ This command will move an item matched by the address given to the destination address.
- ✓ This command can also move to a destination address in a completely different state file.
- ✓ very dangerous command.
- ✓ Very advanced usage command.
- ✓ Result will be unpredictable if concept is not clear terraform state file mainly desired state and current state and current state.
- ✓ Try this in production environments, only when everything worked well in lower environments.

Terraform list Resources
terraform state list

Terraform state move Resources to different name __/__/____

terraform state mv -dry-run aws_instance.my-ec2-vm aws_instance.my-ec2-vm-vm-new
terraform state mv aws_instance.my-ec2-vm aws_instance.my-ec2-vm-new
ls -lrta

Observations: 1) It should create a backup file of terraform.tfstate as something like this "terraform.tfstate.1661192958?.backup"
1) Renamed the named of "my-ec2-vm" in state file to "my-ec2-vm-new"
2) Run terraform plan and observe what happens in next run of terraform plan and apply.

WRONG APPRIACH

WRONG APPROACH OF MOVING TO TERRAPORM PLAN AND APPLY AFTER CHANGE
terraform state mv CHANGE.

WE NEED TO UPDATE EQUIVALENT RESOURCE in terraform manifests to match the same new name.

Terraform Plan
terraform plan

Observations: It will show "Plan: 1 to add, 0 to change, 1 to destroy."
1. to add: New EC2 instance will be added
1 to destroy: Old EC2 instance will be deleted / destroyed.

DON'T DO TERRAFORM APPLY because it shows make changes. Nothing changed other than state file local naming of a resource.

RIGHT APPROACH

Update "c3-ec2-instance.tf"
Before: resource "aws_instance" "my-ec2-vm" {
After: resource "aws_instance" "my-ec2-vm-new" {
Update all references of this resources in your terraform manifests.
Update c5-outputs.tf
Before: value = aws_instance.my-ec2-vm.public_ip
After: value = aws_instance.my-ec2-vm-new.public_ip
Before: value = aws_instance.my-ec2-vm.public_dns
After: value = aws_instance.my-ec2-vm_new.public_dns
terraform plan observation: Message-1: ① No changes. Infrastructure is up-to-date
② This means that Terraform did not detect any differences between your configuration and real physical resources that exists. As a result, no actions need to be performed.

__/__/__

Terraform state rm command :-

- ✓ This commands comes under Terraform moving Resources.
- ✓ The terraform state rm command is used to remove items from the Terraform state.
- ✓ This command can remove single resources, single instances of a resource, entire modules and more.

Terraform list Resources
terraform state list

Remove Resources from Terraform state
terraform state rm -dry-run aws_instance.my-ec2-vm-new
terraform state rm aws_instance.my-ec2-vm

Observations: 1) Firstly takes backup of current state file before removing (example: terraform.tfstate.1611930284.backup)
2) Removes it from terraform.tfstate file.

Terraform Plan
terraform plan

Observation: It will tell you that resources is not state file but same is present in your terraform manifest (ec2-instance.tf - DESIRED STATE). Do you want to re-create it?

This will re-create new EC2 instance excluding one created earlier and running.

Make a choice: ① You want this resource to be running on cloud but not be managed by terraform. Then remove its references in terraform manifests (DESIRED STATE). So that the one running in AWS cloud (current state infra) this instance will be independent of terraform.

② You want a new resource to be created without deleting other one (Non-terraform managed resource now is current state). Run terraform plan and apply. LIKE THIS WE NEED TO MAKE DECISIONS ON WHAT WOULD BE OUR CHANGE OUTCOME OF REMOVING A RESOURCE FROM STATE.

PRIMARY REASON for this is command is that respective resource need to be removed from as terraform managed.

terraform plan
terraform apply.

linkedin: parmeshwargudadhe
__/__/____

Terraform state replace-provider command:

- ✓ This commands comes under Terraform Moving Resources.

Terraform state pull / push command:

- ✓ This command comes under Terraform Disaster Recovery Concept.
- ✓ terraform state pull:
  - ✓ The terraform state pull command is used by manually download and output the state from remote state.
  - ✓ This command also works with local state.
  - ✓ This command will download the state from its current location and output the raw format to stdout.
- ✓ terraform state push: The terraform state push command is used to manually upload a local file to remote state.

Other State Commands (Pull / Push)

```
terraform state pull
terraform state push
```

Terraform force-unlock command:

- ✓ This command comes under Terraform Disaster Recovery Concept
- ✓ Manually unlock the state for the defined configuration.
- ✓ this will not modify your infrastructure.
- ✓ This command removes the lock on the state for the current configuration.
- ✓ The behaviour of this lock is dependent on the backend (aws s3 with dynamodb for state locking etc.) being used.
- ✓ Important Note: Local state files cannot be unlocked by another process.

Manually unlock the state.

```
terraform force-unlock LOCK-ID
```

Terraform Taint / untaint Commands __/__/____

Terraform taint & untaint commands:

- These commands comes under Terraform Forcing Re-creation of Resources.
- When a resource declaration is modified, Terraform usually attempts to update the existing resource in place (although some changes can require destruction and re-creation, usually due to upstream API limitations.)
- Example: A virtual machine that configures itself with cloud-init on startup might no longer meet your needs if the cloud-init configuration changes.
- terraform taint: The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and re-created on the next apply.
- terraform untaint:
  - ✓ The terraform untaint command manually unmarks a Terraform-managed resource as tainted, restoring it as the primary instance in the state.
  - ✓ This ~~reserve~~ reverses either a manual terraform taint or the result of provisioners failing on a resource.
  - ✓ This command will not modify infrastructure, but does modify the state file in order to unmark a resource as tainted.

```
# List Resources from state.
terraform state list
# Taint a Resource
terraform taint <RESOURCE_NAME_IN_TERRAFORM_LOCALLY>
terraform taint aws_instance.my-ec2-vm-new
# Terraform Plan
terraform plan
```

Observation:

Message: "-/+ destroy and then create replacement"

linkedin: parmeshwargudadhe

__/__/____

Untaint a Resource

terraform untaint <RESOURCE_NAME_IN_TERRAFORM_LOCALLY>

terraform untaint aws_instance.my-ec2-vm-new

Terraform Plan

terraform plan

Observation:

Message: "No changes, Infrastructure is up-to-date."

Terraform Resource Targeting - Plan, Apply (-target) Option

- ✓ The -target option can be used to focus Terraform's attension only a subset of resources.
- ✓ This targetting capability is provided for exceptional circumstances, such as recovering from mistakes or working around Terraform limitations.
- ✓ It is not recommended to use -target for routine operations, since this can lead to undetected configuration drift and confusion about how the true state of resources relates to configuration.
- ✓ Instead of using -target as a means to operate on isolated portions of very large configurations, prefer instead to break large configurations into several smaller configurations that can each be independently applied.

lets make two changes

change-1: Add new tag in c4-ec2-instance.tf

"target" = "Target-Test-1"

change-2: Add additional inbound rule in "vpc-web" security group for port 8080.

```
ingress {
    description = "Allow Port 8080"
    from_port = 8080
    to_port = 8080
    protocol = "tcp"
    cidr_block = ["0.0.0.0/0"]
}
```

__/__/____

Change-3 : Add new EC2 Resource

```
# New VM
resource "aws_instance" "my-demo-vm" {
    ami = data.aws_ami.am2linux.id
    instance_type = var.instance_type
    tag = {
      "Name" = "demo-vm1"
    }
}
```

List Resources from state.

```
terraform state list
```

Terraform plan

```
terraform plan -target=aws_instance.my-ec2-vm-new
```

observation:

0) message: "Plan: 0 to add, 2 to change, 0 to destroy."
1) It is updating Change-1 because we are targetting that resource "aws_instance.my-ec2-vm-new"
2) It is updating change-2 "vpc-web" because its a dependent resource for "aws_instance.my-ec2-vm-new"
3) It is not touching the new resource which we are creating now. It will be in terraform configuration but not getting provisioned when we are using -target

Terraform Apply

```
terraform apply -target = aws_instance.my-ec2-vm-new
```

linkedin: parmeshwarguddadhe

__/__/____

Terraform Destroy & Clean_Up :-

```
# Destroy Resources
terraform destroy -auto-approve
# Clean-Up Files
rm -rf .terraform*
rm -rf terraform v1plan.out
rm -rf v2plan.out
```

Ensure below two lines are in commented state in version.tf

① This is required for students demo for this entire section to implement in a step by step manner from beginning

```
# "demo_tag" = "freshestlist" # Enable during
# "target" = "Target-test"
```

② Rename the local name for ec2 instance from "my-ec2-vm-new" to "my-ec2-vm"

changed du

__/__/__

Terraform Workspaces - CLI Based

- ✓ Terraform starts with a single workspace named "default"
- ✓ This workspace is special both because it is the default and also because it cannot ever be deleted.
- ✓ By default, we are working in default workspace.
- ✓ Named workspaces allow conveniently switching between multiple instances of single configuration within its single backend.
- ✓ They are convenient in a number of situations, but cannot solve all problems.
- ✓ A common use for multiple workspaces is to create a parallel, distinct copy of a set of infrastructure in order to test a set of changes before modifying the main production infrastructure.
- ✓ For example, a developer working on complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.
- ✓ Terraform will not recommend using workspaces for larger infrastructures inline with environments pattern like dev, qa, staging. Recommended to use separate configuration directories.
- ✓ Terraform CLI workspaces are completed different from Terraform Cloud Workspaces.

Terraform Workspace Commands :-

terraform workspace show : shows current workspace.
terraform workspace list : list all present workspaces in tf working directory.
terraform workspace new : create new workspace.
terraform workspace select : switch between terraform workspaces
terraform workspace delete : to delete workspace.
usecase-1 : local Backend
use case_2 : Remote Backend

linkedin: parmeshwargudadhe

__/__/____

Create manifests to support multiple workspaces:-

- Ideally, Aws don't allow to create a security group names with same name twice
- With that said, we need to change our security group names in our ec2-security-group.tf
- At the same time, just for reading convenience we can also have our EC2 instance Name tag also updated inline with workspace name.
- What is ${terraform.workspace} ? - It will get the workspace name.
- Popular Usage-1: Using the workspace name as part of naming or tagging behaviour.
- Popular Usage-2: Referencing the current workspace is useful for changing behaviour based on the workspace. For example, for non-default workspaces, it may be useful to spin up smaller cluster sizes.

```
# Change-1: Security Group Names
    name    = "vpc-ssh-${terraform.workspace}"
    name    = "vpc-web-${terraform.workspace}"
# Change-2: For non-default workspaces, it may be useful to spin up smaller
            cluster sizes.
    count   = terraform.workspace == "default" ? 2 : 1
```

This will create 2 instances if we are in default workspace and in any other workspaces it will create 1 instance.

```
# Change-3: EC2 Instance Name tag
    "Name"  = "vm-${terraform.workspace}-${count.index}"
# Change-4: Outputs
    value = aws_instance.my-ec2-vm.*.public_ip
    value = aws_instance.my-ec2-vm.*.public_dns
```

You can create list of all of the values of a given attribute for the items in the collection with a star. For instance, aws_instance.my-ec2-vm.*.id will be a list of all of the Public IP of the instances.

__/__/____

Create resources in default workspaces :-

- ✓ Default Workspace: Every initialized working directory has at least one workspace.
- ✓ If you haven't created other workspaces, it is a workspace named default.
- ✓ For a given working directory, only one workspace can be selected at a time.
- ✓ Most Terraform commands (including provisioning and state manipulation commands) only interact with the currently selected workspace.

Terraform init
terraform init

List Workspaces
terraform workspace list

Output current Workspace using now
terraform workspace show

Terraform Plan
terraform plan

Observation: This should show us two instances based on the statement in EC2 instance resource "count = terraform.workspace == "default" ? 2 : 1" because we are creating this in default workspace

Terraform Apply.
terraform apply -auto-approve

Verify
Verify the same in AWS Management console

Observation:

1. Two instances should be created with name as "vm-default-0, vm-default-1"
2. Security groups should be created with name "vpc-ssh-default, vpc-web-default")
3. Observe the outputs on CLI, you should see list of Public IP and PublicDNS.

linkedin: parmeshwarguladhe

Create New Workspace and Provision Infra:-

```
# Create New Workspace
  terraform workspace new dev
# Verify the folder
  cd terraform.tfstate.d
  cd dev
  ls
  cd ../../
# Terraform Plan
  terraform plan
```

Observation: This should show us creating only 1 instance based on statement " count = terraform.workspace == "default" ? 2 : 1" as we are creating this in non-default workspace named dev

```
# Terraform Apply
  terraform apply -auto-approve
# Verify Dev Workspace statefile
  cd terraform.tfstate.d/dev
  ls
  cd ../../
```

Observation: You should find "terraform.tfstate" in "current-working-directory/terraform.tfstate.d/dev" folder

Verify EC2 instance in AWS mangement console.

Observation:

1. Name should be with "vm-dev-0"
2. Security Group names should be as "vpc-ssh-dev, vpc-web-dev"

__/__/__

Switch workspace and destroy resources :-

✓ Switch workspace from dev to default and destroy resources.

Show current workspace
terraform workspace ~~list~~ show

List Workspaces
terraform workspace list

Workspace select
terraform workspace select default

Delete Resources from default workspace
terraform destroy

Verify
D Verify in AWS management console (both instances and security groups should be deleted)

Delete dev workspace :-

✓ We cannot delete 'default' workspace.

✓ We can delete workspaces which we created (dev, qa, etc)

Delete Dev Workspace
terraform workspace delete dev

Switch to Dev Workspace
terraform destroy -auto-approve

Delete dev Workspace.
terraform workspace delete dev

observation :

Workspace "dev" is your workspace.

You cannot delete the currently active workspace. Please switch to another workspace and try again.

Switch to default Workspace.
terraform workspace select default
terraform workspace delete dev

Verify.
In AWS Management console.

Terraform Workspaces in combination with Terraform Backend: (Remote state storage):

- Review terraform manifest (primarily versions.tf):

v2-remote-backend

Only change in the template is in version.tf, we will have the remote backend block which we learned during section Remote-State-Storage-and-Locking

```
# Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket  = "terraform-state-backup"
    key     = "workspaces/terraform.tfstate"
    region  = "us-east-1"
# For State Locking
    dynamodb = "terraform-dev-state-table"
  }
```

- Provision infra using default workspace :-

```
terraform init
terraform workspace list
terraform workspace show
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
```

Review State file in S3 Bucket for default workspace.

Go to AWS Management Console > Services > S3 > terraform-state-backup > workspaces > terraform.tfstate

- Destroy resources in both workspaces (default, dev) :-

```
terraform workspace show
terraform destroy -auto-approve
terraform workspace show
terraform workspace select default
terraform workspace show
terraform destroy -auto-approve
terraform workspace delete dev
```

__/__/____

Try deleting default workspace and understand what happens :-

Try deleting default workspace

```
terraform workspace delete default
```

Observation :

1) Workspace "default" is your active workspace.

2) You cannot delete the currently active workspace. Please switch to another workspace and try again.

Create new workspace

```
terraform workspace new test1
```

Show current Terraform workspace

```
terraform workspace show
```

Try deleting default workspace now

```
terraform workspace delete default
```

Observation :

1) Can't delete default state

Clean-Up :-

Switch workspace to default

```
terraform workspace select default
```

Delete test1 workspace

```
terraform workspace delete test1
```

Clean-Up Terraform local folders

```
rm -rf .terraform*
```

Clean-Up state files in S3 bucket

Go to S3 Bucket and delete files.

Terraform Provisioners

__/__/____

Terraform Provisioners :-

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers.

- ✓ Passing data into virtual machines and other compute resources.
- ✓ Running configuration management software (packer, chef, ansible) after creating resources.
- ✓ Provisioners are a last Resort.
- ✓ First-class Terraform provider functionality may be available
- ✓ Creation-Time Provisioners.
- ✓ Destroy-Time Provisioners.
- ✓ Failure Behaviour: Continue: Ignore the error and continue with creation or destination.
- ✓ Failure Behaviour: Fail: Raise an error and stop applying (the default behaviour). If creation provisioner, taint resource.

Types of Provisioners :-

1) File Provisioners
2) remote-exec Provisioners
3) local-exec Provisioners

Connection Block :-

- ✓ Most provisioners require access to the remote resource via SSH or WinRM and expect a nested connection block with details about how to connect.
- ✓ Expressions in connection blocks cannot refer to their parent resource by name. Instead, they can use the special self object.

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

1) File Provisioners :-

- ✓ File provisioners is used to copy file or directories from the machine executing Terraform to the newly created resource.
- ✓ The file provisioners support both ssh and winRM type of connections.

2) local-exec Provisioners :-

- ✓ The local-exec provisioner invokes a local executable after a resource is created.
- ✓ This invokes a process on the machine running Terraform, not on the resource.

```
# local-exec provisioner (Creation-Time Provisioners - Triggered during Resource creation)
provisioner "local-exec" {
    command = "echo ${aws_instance.my-vm.private_ip} >> file.txt"
    working_dir = "local-exec-output-files/"
    # on failure = continue
}

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

3) remote-exec Provisioner :-

- ✓ The remote-exec provisioner invokes a script on a remote resource after it is created.
- ✓ This can be used to run a configuration management tool, bootstrap into a cluster, etc.

```
# Copies the file-copy.html file to /tmp/file-copy.html
  provisioner "file" {
    source = "apps/file-copy.html"
    destination = "/tmp/file-copy.html"
  }
# Copies the file to Apache Webserver /var/www/html directory.
  provisioner "remote-exec" {
   inline = [
     "sleep 120"
     "sudo cp /tmp/file-copy.html  /var/www/html/"
   ]
  }
```

Null Resources and Provisioners :-

- ✓ If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null_resource.
- ✓ Instances of null_resource are treated like normal resources, but they don't do anything.
- ✓ Same as other resource, you can configure provisioners and connection details on a null_resource.

```
# Wait for 90 seconds after creating the above
  resource "time_sleep" "wait_90_seconds" {
      depends_on = [aws_instance.my-ec2-vm]
      create_duration = "90s"
  }
# Sync appl static content to Webserver using
  resource "null_resource" "sync_appl_static" {
      depends_on = [time_sleep.wait_90_seconds]
      triggers  = {
        always-update = timestamp()
      }
```

```
# Connection Block for provisioners to EC2
  connection {
      type     = "ssh"
      host     = aws_instance.my-ec2-vm.public_ip
      user     = "ec2-user"
      password = ""
      private_key = file("private-key/terraform-key")
    }
# Copies the appl folder to /tmp
  provisioner "file" {
      source      = "apps/appl"
      destination = "/tmp"
    }
# Copies the /tmp/appl folder to Apache Webserver /var/www/html
  provisioner "remote-exec" {
      inline = [
        "sudo cp -r /tmp/appl /var/www/html"
      ]
  }
```

linkedin: parmeshwargudadhe

- File Provisioner & Connection Block :-

✓ Understand about file provisioner and connection block.

✓ Connection Block:

We can have connection block inside resource block for all provisioners - [or] We can have connection block inside a provisioner block for that respective provisioner.

✓ self Object:

✓ Important Technical Note: Resource references are restricted here because references create dependencies referring to a resource by name within its own block would create a dependency cycle.

✓ Expressions in provisioner blocks cannot refer to their parent resource by name. Instead, they can use the special self object.

✓ The self object represents the provisioner's parent resource and has all of that resource's attributes.

Connection Block for Provisioners to connect to EC2 Instance.

```
connection {
    type     = "ssh"
    host     = self.public_ip
    user     = "ec2-user"
    password = ""
    private_key = file("private-key/terraform-key.pem")
}
```

Create multiple provisioners of various types :-

- ✓ Creation-Time Provisioners:
- ✓ By default provisioners run when the resource they are defined within is created.
- ✓ Creation-time provisioners are only run during creation, not during updating or any other lifecycle.
- ✓ They are meant as a means to perform bootstrapping of a system.
- ✓ If a creation-time provisioner fails, the resource is marked as tainted.
- ✓ A tainted resource will be planned for destruction and recreation upon the next terraform apply.
- ✓ Terraform does this because a failed provisioner can leave a resource in a semi-configured state.
- ✓ Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting
- ✓ You can change this behaviour by setting the on_failure attribute, which is covered in detail below.

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

# Copies the ~~appl folder to~~ string in content in /tmp/file.log.
  provisioner "file" {
      content = "ami-used: ${self.ami}"
      destination = "/tmp/file.log"
    }

# Copies the appl folder to /tmp - FOLDER COPY
  provisioner "file" {
      source = "app/appl"
    destination = "/tmp"
  }
```

linkedin: parmeshwargudadhe

__/__/____

```
# Copies all files and folders in apps/app2 to /tmp - CONTENTS OF FOLDER
  WILL BE COPIED
  provisioner "file" {
    source = "apps/app2/" # When "/" at the end is added - CONTENTS OF
                            FOLDER WILL BE COPIED.
    destination = "/tmp"
  }
# Copies the string in content into /tmp/file.log
  provisioner "file" {
    source = "ami used: ${self.ami}" # Understand what is "self"
    destination = "/tmp/file.log"
  }
# Copies the file-copy.html file to /tmp/file-copy.html
  provisioner "file" {
    source = "apps/file-copy.html"
    destination = "/tmp/file-copy.html"
  }
# Copies the appl folder to /tmp - FOLDER COPY
  provisioner "file" {
    source = "apps/appl"
    destination = "/tmp"
  }
# Copies all files and folders in apps/app2 to /tmp -CONTENTS OF FOLDER
  WILL BE COPIED.
  provisioner "file" {
    source = "apps/app2/" # When "/" at the end is added - CONTENTS
    destination = "/tmp"          OF FOLDER WILL BE COPIED.
  }
```

__/__/____

Create Resources using Terraform commands.

Terraform Initialize
terraform init

Terraform Validate
terraform validate

Terraform Format
terraform fmt

Terraform Plan
terraform plan

Terraform Apply
terraform apply -auto-approve

Verify - login to EC2 Instance
chmod 400 private_key/terraform_key.pem
ssh -i private-key/terraform_key.pem ec2-user@ip
ssh -i private-key/terraform-key.pem ec2-user@12.12.12.12
Verify /tmp for all file copied
cd /tmp
ls -lrta /tmp

Clean-up
terraform destroy -auto-approve
rm -rf terraform.tfstate*

Failure Behaviour: Understand Decision making when provisioner fails (continue/fail) :-

✓ By default, provisioners that fail will also cause the Terraform apply itself to fail. The on_failure setting can be used to change this. The allowed values are:

1) continue: Ignore the error and continue with creation or destruction.
2) fail: (Default Behaviour) Raise an error and stop applying (the default behaviour). If this is a creation provisioner, taint the resource.

linkedin: parmeshwarguadadhe

__/__/____

- ✓ Try copying a file to Apache static content folder "/var/www/html" using file-provisioner.
- ✓ This will fail because, the user you are using to copy these files is "ec2-user" for amzon linux vm. This user doesn't have access to folder "/var/www/html" top copy files.
- ✓ We need to use sudo to do that.
- ✓ All we know is we cannot copy it directly, but we know we have already copied this file in "/tmp" using file provisioner.
- ✓ Try two scenarios
  - ✓ No on_failure attribute (same as on_failure = fail) - default what happens. It will raise an error and stop applying. IF this is a creation provisioner, it will taint the resource.
  - ✓ When on_failure = continue, will continue creating resources
  - ✓ Verify: verify terraform.tfstate for "status": "tainted"

```
# Test-1 Without on_failure attribute which will fail terraform apply
# Copies the file-copy.html file to /var/www/html/file-copy.html
    provisioner "file" {
        source      = "apps/file-copy.html"
        destination = "/var/www/html/file-copy.html"
    }
### Verify: Verify terraform.tfstate for "status": "tainted"
# Test-2 With on_failure = continue
# Copies the file-copy.html file to /var/www/html/file-copy.html
    provisioner "file" {
        source      = "apps/file-copy.html"
        destination = "var/www/html/file-copy.html"
        on_failure  = continue
    }
# Verify: Verify terraform.tfstate for "status": "tainted"
```

__/__/____

terraform plan

terraform apply -auto-approve

login to EC2 instance
Verify /tmp, /var/www/html for all files copied.

terraform destroy -auto-approve
rm -rf .terraform*
rm -rf terraform.tfstate*

Destroy time provisioners :-

- ✓ Discuss about this concept.
- ✓ Inside a provisioner when you add this statement when = destroy it will provision this during the resource destroy time.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
      when = destroy
      command = "echo 'Destroy-time provisioner'"
      }
}
```

linkedin: parmeshwargudadhe

__/__/__

Remote-exec Provisioner:

✓ The remote-exec provisioner invokes a script on a remote resource after it is created.

✓ This can be used to run a configuration management tool, bootstrap into a cluster, etc.

Pre-requisites :-

✓ Create a EC2 key pair with name terraform-key and copy the terraform-key.pem file in the folder private-key in terraform-manifest folder.

✓ Connection Block for provisioners uses this to connect to newly created EC2 instance to copy files using file provisioner, execute scripts using remote-exec provisioner.

* Create / Review Provider configuration

- Use case:

1. We will copy a file named file-copy.html using File provisioner to "/tmp" directory.
2. Using remote-exec provisioner, using linux commands we will copy the file to Apache webserver static content directory /var/www/html and access it via browser once it is provisioned.

```
provisioner "file" {
    source      = "app/file-copy.html"
    destination = "/tmp/file-copy.html"
}
provisioner "remote-exec" {
    inline = [
      "sleep 120",
      "sudo cp /tmp/file-copy.html /var/www/html"
    ]
}
```

__/__/____

- **local-exec provisioner:**
  - ✓ The local-exec provisioner invokes a local executable after resource is created.
  - ✓ This invokes a process on the machine running Terraform, not on the resource.

- **Prerequisites:**
  - ✓ Create a EC2 key pair with name terraform-key and copy the terraform-key.pem file in the folder private-key in terraform-manifest folder.
  - ✓ Connection Block for provisioners uses this to connect to newly created EC2 instance to copy files using file provisioner, execute scripts using remote-exec provisioner

- **Review local-exec provisioner code:**
  - ✓ We will create one provisioner during creation-time. It will output private ip of the instance in to a file named creation-time-private-ip.txt.
  - ✓ We will create one more provisioner during destroy time. It will output destroy time with date in to file named destroy-time.txt

ec2-instance.tf

```
provisioner "local-exec" {  # Creation-time provisioner
    command = "echo ${aws_instance.my-ec2-vm.private_ip} >> creation-time-private-ip.txt
    working_dir = "local-exec-output-files/"
  #on_failure = continue
provisioner "local-exec" {  # destroy-time provisioner
    when    = destroy
    command = "echo Destroy-time provisioner Instance Destroyed at 'date' >> destroy-time.txt
    working_dir = "local-exec-output-files/"
}
```

linkedin: parmeshwargudadhe

__/__/____

- **Null Resource**:

✓ Null Resource will be used in combination with provisioners.

1) **Null provider**: The null provider is a rather unusual provider that has constructs that intentionally do nothing. This may sound strange, and indeed these constructs do not need to be used in most cases, but they can be useful in various situations to help orchestrate tricky behaviour or work around limitations.
2) **Null Resource**: The null_resource resource implements the standard resource lifcycle but ~~no~~ takes no further action.
3) ~~Null~~ **Time provider**: The time provider is used to interact with time-based resources. The provider itself has no configuration options.
4) **Provisioners without a Resource**: If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null_resource.
Instances of null_resources are treated like normal resources, but they don't do anything.

✓ **Usecase**: Force a resource to update based on a changed null_resource
✓ Create time_sleep resource to wait for 90 seconds after EC2 instance creation.
✓ Create Null Resource with required provisioners
  - File Provisioners: Copy apps/appl folder to /tmp
  - Remote Exec provisioner: Copy appl folder /tmp to /var/www/html
✓ Over the process we will learn about
  - null_resource
  - time_sleep resource
  - We will also learn how to force a resource to update based on a changed null_resource using timestamp function and triggers in null_resource.

__/__/__

- Define null provider in Terraform Settings Block:

✓ Update null provider info listed below in c1-versions.tf

```
null = {
    source = "hashicorp/null"
    version = "~> 3.0.0"
}
```

- Define Time Provider in Terraform Settings Block:

✓ Update time provider info listed below in c1-versions.tf

```
time = {
    source = "hashicorp/time"
    version = "~> 0.6.0"
}
```

- Create / Review the c4-ec2-instance.tf terraform config:-

① Create Time Sleep Resource

✓ This resource will wait for 90 seconds after EC2 instance creation.

✓ This wait time will give EC2 instance to provision the Apache Webserver and create all its relevant folders.

✓ Primarily if we want to copy static content we need Apache webserver static folder /var/www/html

```
resource "time_sleep" "wait_90_seconds" {
    depends_on = [aws_instance.my-ec2-vm]
create  creation_duration = "90s"
}
```

② Create Null Resource:

- ✓ Create Null Resource with triggers with timestamp() function which will trigger for every terraform apply
- ✓ This Null Resource will help us to sync the static content from our local folder to EC2 instance as and when required.
- ✓ Also only changes will applied using only null_resource when terraform apply is run. In other words, when static content changes, how will we sync those changes to EC2 instance using terraform - this is one simple solution.
- ✓ Primarily the focus here is to learn following -
  - null_resource
  - null_resource trigger
  - How trigger works based on timestamp() function?
  - Provisioners in Null Resource

```
# Sync Appl Static Content to Webserver using Provisioners
resource "null_resource" "sync_appl_static" {
    depends_on = [ time_sleep.wait_90_seconds ]
    triggers   = {
        always_update = timestamp()
    }

    # Connection Block for provisioners to connect to EC2 instance.
    connection {
        type = "ssh"
        host = aws_instance.my_ec2_vm.public_ip
        user = "ec2_user"
        password = ""
        private_key = file("private_key/terraform_key.pem")
    }
```

```
# Copies the appl folder to /tmp
  provisioner "file" {
      source      = "apps/appl"
      destination = "/tmp"
    }
# Copies the /tmp/appl folder to Apache server /var/www/html dir.
  provisioner "remote-exec" {
      inline = [
        "sudo cp -r /tmp/appl /var/www/html"
        ]
    }
}
```

- Execute Terraform Commands :-

```
terraform init
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
# verify
  ssh -i terraform.pem ec2-user@ip
  ls -lrt /tmp
  ls -lrt /tmp/appl
  ls -lrt /var/www/html
  ls -lrt /var/www/html/appl
  http:// public-ip/appl/file1.html
  http:// public-ip/appl/file2.html
```

linkedin: parmeshwargudadhe

__/__/____

- Create new file locally in appl folder:

- Create a new file named file3.html
- Also updated file1.html with some additional info.
- file3.html

```
<h1> Appl file3 </h1>
```

- file1.html

```
<h1> Appl File1 - Updated </h1>
```

terraform, plan, apply, verify.

- **Terraform Modules:**

✓ Modules are containers for multiple resources that are used together. A module consists of a collection of .tf files kept together in a directory.

✓ Module are the main way to package and reuse resource configurations with Terraform.

✓ Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

✓ A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration.

✓ A module that has been called by another module is often reffered to as a child module.

✓ Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

✓ In addition to modules from the local filesystem, Terraform can load modules from a public or private registry.

✓ This makes it possible to publish modules for others to use and to use modules that others have published.

Terraform Registry - Publicly Available Modules

✓ The Terraform Registry hosts a broad collection of publicly available Terraform modules for configuring many kinds of common infrastructure.

✓ These modules are free to use, and Terraform can download them automatically if you specify the appropriate source and version in a module call block.

Private Module Registry in Terraform Cloud and Enterprise

✓ Members of your organisation might produce modules specifically crafted for your own infrastructure needs.

__/__/____

- Define outputs from a EC2 instance module:
  - ✓ cs-outputs.tf: We will output the EC2 instance module attributes (Public DNS and Public IP)

```
# Output variable definitions
output "ec2_instance_public_ip" {
  description = "Public IP addresses of EC2 instances"
  value       = module.ec2_cluster.*.public_ip
}

output "ec2-instance_public_dns" {
  description = "Public IP addresses of EC2 instances"
  value       = module.ec2_cluster.*.public_dns
}
```

- Execute Terraform Commands

```
terraform init
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
```

Verify

1) Verify in AWS management console, if EC2 got created.
2) Update default security group of VPC to allow port 80 from internet.
3) Access Apache server
   http://public-ip-vm1
   http://public-ip-vm2

__/__/____

- **Tainting Resources in a Module:**
  - ✓ The taint command can be used to taint specific resources within a module.
  - ✓ Very Very Important Note: It is not possible to taint an entire module. Instead, each resource within the module must be tainted separately.

```
terraform state list
terraform taint <Resource-Name>
terraform taint module.ec2_cluster.aws_instance.this[0]
terraform plan , a
```

Observation: first destroyed and re-created.

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

- **Clean-Up Resources and local working directory:**

```
terraform destroy -auto-approve
rm -rf .terraform*
rm -rf terraform.tfstate*
```

- **Meta-Arguments for Modules:**
  - ✓ Meta-Argument concepts are going to be same as how we learned during Resources section. refer resources section
    - count
    - for_each
    - providers
    - depends_on

__/__/__

- **Building Terraform Resuable Modules:**
  - Introduction:

✓ Build a Terraform Module

- Create a Terraform module.
- Use local Terraform modules in your configuration.
- Configure modules with variables.
- Use module outputs.
- We are going to write a local re-usable module for the following usecase.

✓ Usecase: Hosting a static website with AWS S3 buckets.

1. Create an S3 Bucket
2. Create Public Read policy for the bucket.
3. Once above two are ready, we can deploy static content.
4. For steps 1 and 2 we are going to create a re-usable module in Terraform.

✓ How are we going to do this!

- We are going to do this in 3 sections.
- Section 1: Full manual: Create static website on S3 using AWS Management Console and host static content and test.
- Section 2: Terraform Resources: Automate section-1 using Terraform Resources.
- Section 3: Terraform Modules: Create a re-usable module for hosting static website by referencing section-2 terraform configuration files.

__/__/____

Step 01 — Hosting a Static website with AWS S3 using AWS mgmt console:

- ✓ Reference Sub-folder: v1-create-static-website-on-S3-using-aws-mgmt-console.
- ✓ We are going to host a static website with AWS S3 using AWS management console.

Step-02-01 — Create AWS S3 Bucket

- ✓ Go to AWS Services > S3 > Create Bucket.
- ✓ Bucket Name: mybucket-1045 (Note: Bucket name should be unique across AWS)
- ✓ Region: US. East (N. Virginia)
- ✓ Rest all leave to defaults
- ✓ Click on Create Bucket.

Step 02-02 — Enable static website hosting

- ✓ Go to AWS Services > S3 > Buckets > mybucke-1045 > Properties Tab > At the end
- ✓ Edit to enable Static website hosting
- ✓ Static website hosting: enable
- ✓ Index document: index.html
- ✓ Click on Save changes.

Step-02-03 — Remove Block public access (bucket settings)

- ✓ Go to AWS Services > S3 > Bucket > mybucke-1045 > Permission Tab
- ✓ Edit Block public access (bucket settings)
- ✓ Uncheck Block all public access
- ✓ Click on Save changes.
- ✓ Provide text confirm and click on Confirm.

__/__/____

- Step-02-04 **Add Bucket policy for public read by bucket owners:**

✓ Update your bucket name in the below listed policy.

✓ Location: v1-create-static-website-on-s3-using-aws-mgmt-console/policy-public-access-for-website.json

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::mybucket-1045/*"
            ]
        }
    ]
}
```

✓ Go to AWS Services > S3 > Buckets > mybucket-1045 > Permission Tab

✓ Edit > Bucket policy > Copy paste the policy above with your bucket name.

✓ Click on Save changes.

__/__/____

Step-02-05 Upload index.html

- ✓ location: v1-create-static-website-on-s3-using-aws-mgmt-console/index.html
- ✓ Go to AWS Services > S3 > Buckets > mybucket-1045 > objects Tab
- ✓ Upload index.html

Step-02-06 Access Static Website using S3 Website Endpoint

- ✓ Access the newly uploaded index.html to S3 bucket using browser.

Endpoint Format

http://example-bucket.s3-website.Region.amazonaws.com/

Replace values (Bucket Name, Region)

http://mybucket-1045.s3-website-us-east-1.amazonaws.com/

Step-02-07 Conclusion

- ✓ We have used multiple manual steps to host a static website on AWS.
- ✓ Now all the above manual steps automate using Terraform in next step.

Step-03 Create Terraform Configuration to Host a Static Website on AWS S3:

- ✓ Reference-Sub-folder: v2-host-static-website-on-s3-using...
- ✓ We are going to host a static website on AWS S3 using generate terraform configuration files.

Step-03-01 Create Terraform Configuration Files step by step

1. versions.tf
2. main.tf
3. variables.tf
4. outputs.tf
5. terraform.tfvars

__/__/____

Step-03.02 **Execute Terraform Commands & Verify the bucket:**

```
terraform init
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
```

verify

1. Bucket has static website hosting enabled.
2. Bucket has public read access enabled using policy.
3. Bucket has "Block all public access" unchecked.

Step-03-03 **Upload index.html and test:**

Endpoint Format

http://example-bucket.s3-website.Region.amazonaws.com/

Replace Values (Bucket Name, Region)

http://mybucket-1406.s3-website.us-east-1.amazonaws.com/

Step.03.04 **Destroy and clean-Up:**

```
terraform destroy -auto-approve
rm -rf .terraform*
rm -rf terraform.tfstate*
```

Step.03-05 **Conclusion:**

- ✓ Using above terraform configurations we have hosted a static website in AWS S3 in seconds.
- ✓ In next step, we will convert these terraform configuration files to a module which will be re-usable just by calling it.

__/__/____

Step-04 **Build a Terraform Module to Host a Static Website on AWS S3:**

- ✓ Reference Sub-folder : V3-build-a-module-to-host-static-....
- ✓ We will build a Terraform module to host a static website on AWS S3.

step-04-01 **Create Module Folder structure:**

- ✓ We are going to create modules folder and in that we are going to create a module named aws-s3-static-bucket
- ✓ We will copy required files from previous section for this respective module
- ✓ Terraform Working Directory : V3-build-a-module-to-host-static-website-on-aws-s3
  - modules
    - ✓ Module-1 : aws-s3-static-website-bucket
      - main.tf
      - variables.tf
      - outputs.tf
      - README.md
      - LICENSE
- ✓ Inside module/aws-s3-static-website-bucket copy below listed three files from V2-host-static-website-on-s3-using-terraform-manifests.
  - main.tf
  - variables.tf
  - outputs.tf

step-04-02 **Call Module from Terraform Work Directory (Root Module):**

- ✓ Create Terraform configuration in Root Module by calling the newly module.
- ✓ c1-versions.tf
- ✓ c2-versions.tf
- ✓ c3-s3bucket.tf
- ✓ c4-outputs.tf

```
module "website_s3_bucket" {
    source = "./modules/aws-s3-static-website-bucket"
    bucket_name = var.my_s3_bucket
    tags = var.my_s3_tags
}
```

Step-04-03 **Execute Terraform Commands:**

```
terraform init
```

Observation:

1. Verify ".terraform", you will find "modules" folder in addition to "providers" folder.
2. Verify inside ".terraform/modules" folder too.

```
terraform validate
terraform fmt
terraform plan
terraform apply -auto-approve
# Verify
  # Endpoint Format
http://example-bucket.s3-website.Region.amazonaws.com/
# Replace values (Bucket Name, Region)
http://mybucket-1047.s3-website.us-east-1.amazonaws.com/
```

Step-04-05 **Destroy and Clean-Up**

```
terraform destroy -auto-approve
rm -rf .terraform*
rm -rf terraform.tfstate*
```

Step-04-06 **Understand terraform get command:**

✓ We have used terraform init to download providers from terraform registry and at the same time to download modules present in local modules folder in terraform working directory.

__/__/____

- ✓ Assuming we already have initialized using terraform init and later we have created module configs, we can terraform get to download the same.
- ✓ Whenever you add a new module to a configuration, Terraform must install the module before it can be used.
- ✓ Both the terraform get and terraform init commands will install and update modules.
- ✓ The terraform init command will also initialize backends and install plugins.

```
# Delete modules in .terraform folder
ls -lrt .terraform/modules
rm -rf .terraform/modules
ls -lrt .terraform/modules
# Terraform Get
terraform get
ls -lrt .terraform/modules
```

Step-04-07 Major difference between Local and Remote Module:

- ✓ When installing a remote module, terraform will download it into the .terraform directory in your configuration's root directory.
- ✓ When installing a local module, terraform will instead refer directly to the source directory.
- ✓ Because of this, terraform will automatically notice changes to local modules without having having to re-run terraform init or terraform get.

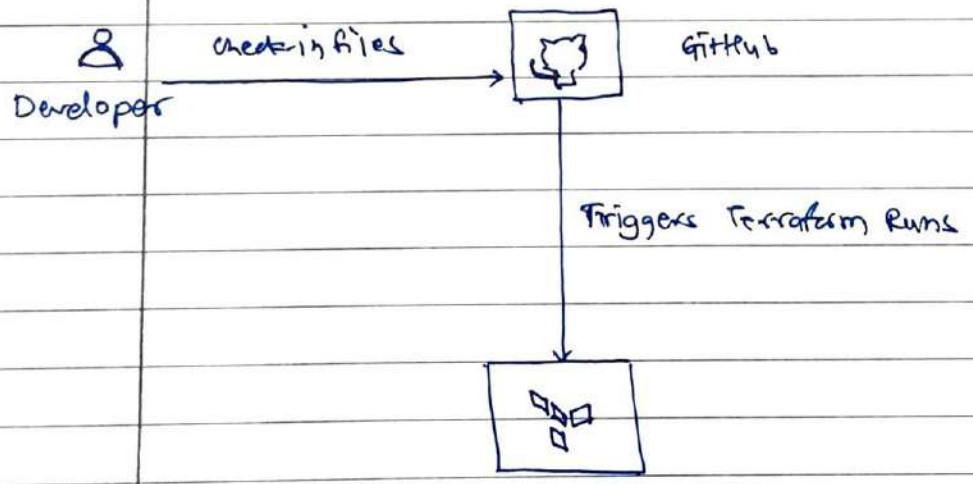
Step-05 Conclusion:

- ✓ Created a Terraform Module.
- ✓ Used local Terraform modules in your configuration.
- ✓ Configured modules with variables.
- ✓ Used module outputs.

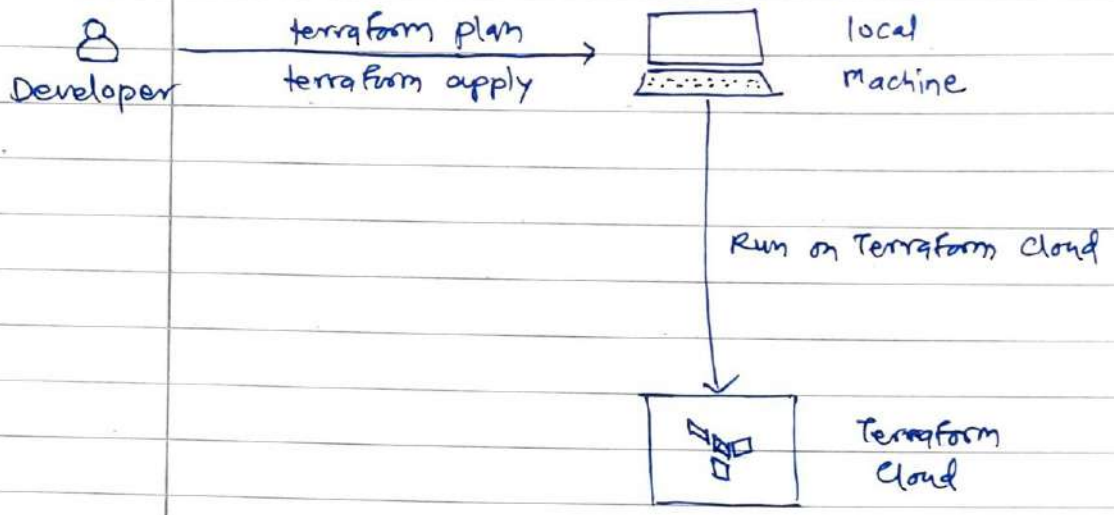
__/__/____

- **Terraform Cloud or Terraform Enterprise:-**
  - ✓ Terraform Cloud and Terraform Enterprise are different distributions of the same application.
  - ✓ It manages Terraform runs in a consistent and reliable environment, and includes easy access to shared state and secret data, access controls for applying changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations.
  - ✓ Terraform Cloud is available as a hosted service at https://app.terraform.io
  - ✓ They offer free accounts for small teams, and paid plans with additional feature sets for medium-sized businesses.
  - ✓ Large enterprises can purchase Terraform Enterprise, their self-hosted distribution of Terraform Cloud.

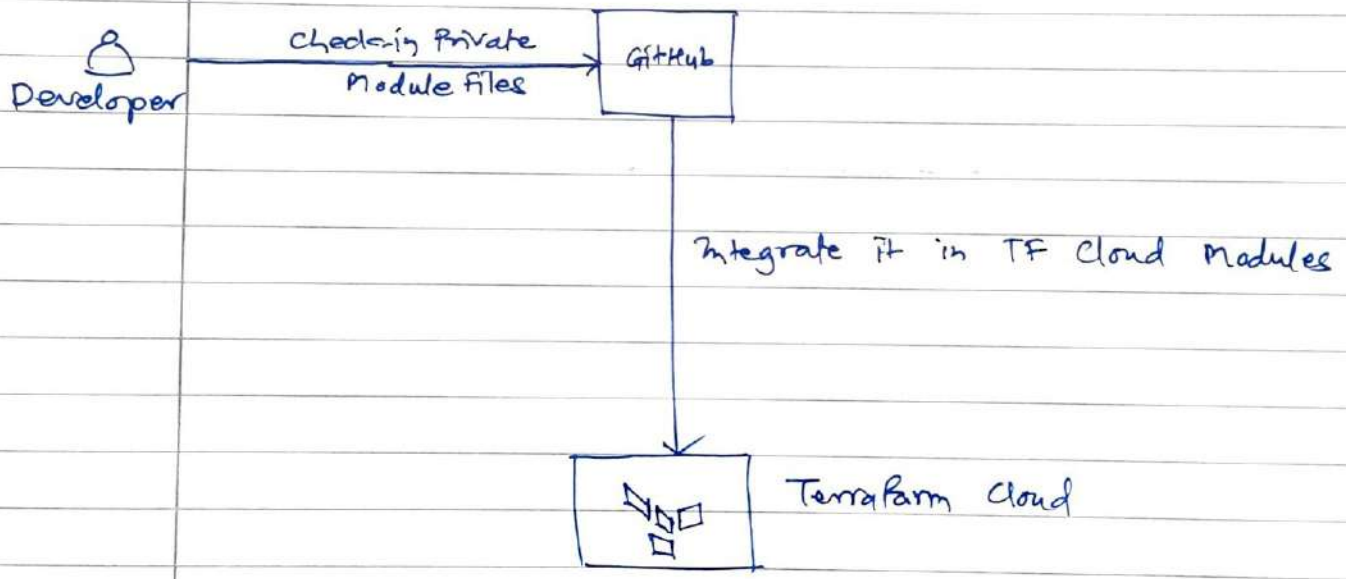
Terraform VCS-Driven Workflow:-



Terraform CLI-Driven Workflow:



Publish Private Module Registry in Terraform Cloud :-



__/__/____

Terraform Cloud Version Control Workflow :-

Terraform VCS Integration :

Terraform Cloud is more powerful when you integrate it with your version control system (VCS) provider (GitHub, Bitbucket, GitLab).

- ✓ Terraform Cloud can automatically initiate Terraform runs.
- ✓ Terraform Cloud makes code review easier by automatically predicting how pull requests will affect infrastructure.
- ✓ Publishing new versions of a private Terraform module is as easy as pushing a tag to the module's repository.

Setup GitHub Repository Local & Remote :-

- Terraform Cloud and GitHub Integration :

Step-01: Introduction :

- ✓ Create GitHub Repository on github.com.
- ✓ Clone GitHub Repository to local desktop.
- ✓ Copy & Check-in Terraform configurations in to GitHub Repository.
- ✓ Create Terraform Cloud Account.
- ✓ Create Organization.
- ✓ Create Workspace by integrating with GitHub.com Git Repo we created.
- ✓ Learn about Workspace related Queue Plan, Runs, States, Variables and Settings.

__/__/__

Step-02: Create new github Repository:

- ✓ URL: github.com
- ✓ Click on Create a new repository
- ✓ Repository Name: terraform-cloud-demo1
- ✓ Description:
- ✓ Repo Type: Public / Private
- ✓ Initialize this repository with:
- ✓ CHECK - Add a README file
- ✓ CHECK - Add .gitignore
- ✓ Select .gitignore Template: Terraform
- ✓ CHECK - choose a license.
- ✓ Select license: Apache 2.0 License
- ✓ click on Create Repository.

Step-03: Review .gitignore created for Terraform

- ✓ Review .gitignore created for Terraform projects.

Step-04: Clone GitHub Repository to local Desktop.

- ✓ git clone https://github.com/.../....git

Step-05: Copy files from terraform-manifests to local repo & check-in Code.

- ✓ List of files to be copied.
  - ✓ apache-install.sh
  - ✓ c1-versions.tf
  - ✓ c2-variables.tf
  - ✓ c3-securitygroups.tf
  - ✓ c4-ec2instance.tf
  - ✓ c5-outputs.tf
  - ✓ c6-ami-datasource.tf
- ✓ Source location: Section-01-01 - Inside terraform manifest folder
- ✓ Destination location: Newly created github repository folder in your local desktop terraform-cloud-demo1

__/__/____

✓ Verify locally before commiting to GIT Repository.

terraform init
terraform validate
terraform plan

✓ Check-in to Remote Repository / GitHub

git status
git add .
git commit -am "TF files first commit"
git push
Verify the same on Remote Repository.
https://github.com/.../....git

Step-06: Sign-Up for Terraform Cloud - Free Account and Login

✓ Sign-Up URL : https://app.terraform.io/signup/account
✓ Username :
✓ Email:
✓ Password:
✓ Login URL : https://app.terraform.io

Step-07: Create Organisation:

✓ Organisation Name: clouddost
✓ Email address:
✓ click on Create Organisation.

Step-08 Create New Workspace

✓ Get into newly created Organisation.
✓ Click on New Workspace.
✓ Choose your workflow: V
  ✓ Version Control Workflow
✓ Connect to VCS.
  ✓ Connect to a version control provider: github.com

- ✓ New Window: Authorize Terraform Cloud: Click on Authorize Terraform cloud Button.
- ✓ New Window: Install Terraform Cloud.
- ✓ Select radio button: Only select ~~permissions~~ repositories.
- ✓ Selected 1 Repository: ..../terraform-cloud-demo1
- ✓ Click Install.

✓ Choose a Repository
- ✓ ..../terraform-cloud-demo1

✓ Configure Settings:
- ✓ Workspace Name: terraform-cloud-demo1 (Whatever populated automatically leave to defaults)
- ✓ Advanced Settings: leave to defaults.

✓ click on Create Workspace.

✓ You should see this message "Configuration uploaded success.."

Step-09: Configure Variables:

✓ Variable: aws_region
- ✓ key: aws_region
- ✓ value: us-east-1

✓ Variable: instance_type
- ✓ key: instance_type
- ✓ value: t2.micro

step-10: Configure Environment variables:

registry.terraform.io/providers/hashicorp/aws/latest/docs#environment-variables

✓ Environment Variables AWS_ACCESS_KEY_ID
- ✓ key: AWS_ACCESS_KEY_ID
- ✓ Value: xxxxxxxxxxxxxxxxx

✓ Enviroment Variable: AWS_ACCESS_KEY_SECRET.
- ✓ key: AWS_SECRET_ACCESS_KEY
- ✓ Value: yyyyyyyyyyyyyyyyy

__/__/____

Step-11: **Click on Queue Plan:**

- ✓ Go to Workspace → Run → Queue Plan
- ✓ Review the plan generated in full screen.
- ✓ Add comment: First run.
- ✓ Click on Confirm & Apply
- ✓ Add Comment: First run approved.
- ✓ Click on Confirm plan.
- ✓ Review Apply log output in full screen.
- ✓ Add Comment: Successfully Provisioned, verified in AWS.

Step-12 **Change Number of Instance:**

- ✓ Make some changes on local repo and push the code to github. terraform cloud will automatically plan the updated code just we need to approved the plan.

Step-13 **Review Workspace Settings:**

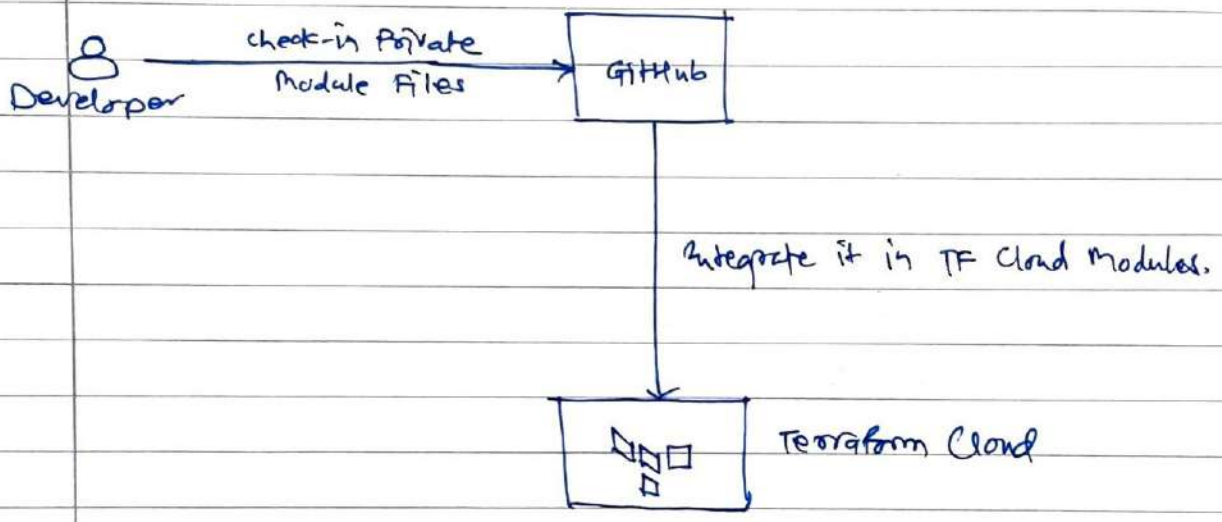
- ✓ General Setting.
- ✓ locking
- ✓ Notifications
- ✓ Run Triggers.
- ✓ SSH Key
- ✓ Version Control

14. **Destruction and Deletion:**

- ✓ Goto → Workspace → Setting → Destruction and Deletion
- ✓ click on Queue Destroy Plan to delete the resources on cloud.
- ✓ Goto → Workspace → Runs → Click on Confirm and Apply.
- ✓ Add comment: Approved for deletion.

__/__/__

- **Terraform Private Module Registry :-**
  - ✓ Terraform cloud's private module registry helps you share Terraform modules across your organisation.
  - ✓ It includes supports for module versioning, a searchable and filterable list of available modules, and a configuration designer to help you build new workspaces faster.
  - ✓ By design, the private module registry works much like the public Terraform Registry. If you're already used the public registry, Terraform cloud's registry will feel familiar.



__/__/____

Terraform Cloud and Enterprise Capabilities:-

step-01: Introduction:

- ✓ Create and version a GitHub repository for use in the private module registry.
- ✓ Import a module into your organisation's private module registry.
- ✓ Construct a root module to consume modules from the private registry.
- ✓ Over the process also learn about terraform login command.

step-02: Create new github Repository for s3-website terraform module:

step-02-01 Create new GitHub Repository.

- ✓ Naming Convension for Modules
  - ✓ terraform-PROVIDER-MODULE_NAME
  - ✓ sample- terraform-aws-s3-website
- ✓ Add .gitignore with template - Terraform
- ✓ Select licences and Create Repository.

step-02-02 Create New Release Tag 1.0.0 in Repo

- ✓ github Repo → Releases → Create a new release
- ✓ Tag version: 1.0.0
- ✓ Release Title: Release-1 terraform-aws-s3-website
- ✓ Write: Terraform Module for Private Registry - terraform-aws-s3-website
- ✓ Publish Release.

step-03 Clone Github Repository to local machine:

- ✓ git clone https://github.com/.../....git

step-04: Copy file from terraform-manifests to local repo & check-in

- ✓ Source location: 11-02.../terraform-s3-website-bucket
- ✓ Destination-location: local repo - terraform-aws-s3-website.

__/__/____

Step-05 Add VCS Provider as GitHub using OAuth App in TF Cloud:

step-05-01 Add VCS Provider as GitHub using OAuth App in TF Cloud:

- ✓ Login to Terraform Cloud.
- ✓ Click on Module Tab → Click on Add Module → Select GitHub (Custom)
- ✓ Should redirect to URL: https://github.com/settings/applications/new in new browser tab.
- ✓ Application Name: Terraform Cloud (hctaprep)
- ✓ Homepage URL: https://app.terraform.io
- ✓ Application description: Terraform Cloud Integration with Github using OAuth.
- ✓ Authorization callback URL: https://app.terraform.io/auth/.../.../callback
- ✓ Click Register Application.
- ✓ Make a Note of Client ID: _____ (Sample for Ref)
- ✓ Generate new Client Secret: _____

Step-05-02: Add the below in Terraform Cloud:

- ✓ Name: github-terraform-modules
- ✓ Client ID: _____
- ✓ Client Secret: _____
- ✓ Click on Connect and Continue.
- ✓ Authorize Terraform Cloud (hctaprep) - Click on Authorize
- ✓ SSH Keypair (Optional): click on Skip and Finish.

step.-06: Import the Terraform Module from GitHub

- ✓ In above step; we have completed the VCS setup with github.
- ✓ Now import TF module from github
- ✓ login to TF Cloud.
- ✓ Module Tab → Add Module → Select GitHub → select
- ✓ Choose a Repository terraform-s3-website
- ✓ Click on Publish Module.

__/__/____

Step-07: Review newly imported Module:

- ✓ login to Terraform Cloud → Click on Modules Tab
- ✓ Review The Module Tabs on Terraform Cloud.
  - ✓ Readme
  - ✓ Inputs
  - ✓ Outputs
  - ✓ Dependencies
  - ✓ Resources
- ✓ Also review the following
  - ✓ versions
  - ✓ Provision Instructions.

Step-08: Create a configuration that uses the Private Registry Module using Terraform CLI

Step-08-01 Call Module from Terraform Work Directory (Root Module)

- ✓ Create Terraform Configuration in Root Module by calling the newly published module in Terraform Private Registry.
- ✓ c1-versions.tf
- ✓ c2-variables.tf : Review and discuss
- ✓ c3-s3bucket.tf
- ✓ c4-outputs.tf

```
module "website_s3_bucket" {
  source = "app.terraform.io/hctaprep/s3-website-internal/aws"
  version = "1.0.0"
  # insert required vaibles here
  bucket_name = var.my_s3_bucket
  tags = var.my_s3_tags
}
```

__/__/____

- Share Module in Private Module Registry:-

Step-01: Intraduction:

- ✓ Create and version a GitHub repository for use in the private registry.
- ✓ Import a module into your organization's private module Registry.
- ✓ Construct a root module to consume modules from the private registry.
- ✓ Over the process also learn about terraform login command.

step-02-01 Create new GitHub Repository

- ✓ URL: github.com
- ✓ terraform ~~Name~~ PROVIDER-MODULE-NAME
- ✓ Sample: terraform-aws-s3-website
- ✓ Repository Name: terraform-aws-s3-website
- ✓ Descriphion: Terraform Modules to be shared in Private Registry.
- ✓ Repo Type: Public / Private
- ✓ Initialize this repository with:
  - ✓ UN-CHECK - Add a README file.
  - ✓ CHECK - Add .gitignore
  - ✓ Select .gitignore Template: Terraform
  - ✓ CHECK - Choose a license
  - ✓ Select license - Aaphe 2.0 license (optional)
  - ✓ click on Create repository.

step-02-01 ~~clone GitHub Repository to local Desktop~~:

Create New Release Tag 1.0.0 in Repo:

- ✓ Tag version: 1.0.0
- ✓ Release Title: Release-1 terraform-aws-s3-website
- ✓ Write: Terraform Module for Private Registry - terraform-aws-s3-website.

__/__/____

Step-03: Clone GitHub Repository to local Machine:

git clone https://github.com/.../.....git

Step-04: Copy files from terraform-manifests to local repo & check-in code:

- ✓ Copy code to new git local repo from course.
- ✓ Check-in the code to remote repository.

Step-05-01 Add VCS Provider as GitHub using OAuth App in TF Cloud:

- ✓ login Terraform Cloud.
- ✓ goto /create organisation.
- ✓ Registry > Publish Module
- ✓ ~~Connect GitHub > Choose Repository.~~
- ✓ Select GitHub (Custom)
- ✓ github.com/settings/applications/new
- ✓ Application Name: Terraform Cloud (hctaprep)
- ✓ Homepage URL: https://app.terraform.io
- ✓ Application description: Terraform cloud integration with GitHub using OAuth.
- ✓ Authorization callback URL: http://app.terraform.io/auth).....
- ✓ Click on Register Application.
- ✓ Make a note of Client ID: _____
- ✓ Generate new Client secret: _____

Step-05-02: Add the below in Terraform Cloud.

- ✓ Name: github-terraform-modules
- ✓ Client ID:
- ✓ Client Secret:
- ✓ Click on Connect and Continue.
- ✓ Authorize Terraform Cloud (hctaprep): Click on Authorize
- ✓ SSH Keypair: Click on Skip and Finish.

__/__/____

Step-06: Import the Terraform Module from GitHub:

- ✓ In above step, we have completed the VCS setup with github.
- ✓ Now lets go ahead and import the Terraform module from GitHub.
- ✓ Login to Terraform Cloud.
- ✓ Click on Module Tab > Click on Add Module > Select GitHub (github-terraform-modules) (PRE-POPULATED) → Select it.
- ✓ Choose a Repository: terraform-module-s3-website.
- ✓ Click on Publish Module.

Step-07: Review newly imported Module:

- ✓ Login TF Cloud → Click on Module Tab
- ✓ Review the Module Tabs on Terraform Cloud.
  - ✓ Readme
  - ✓ Inputs
  - ✓ Outputs
  - ✓ Dependencies.
  - ✓ Resources.
- ✓ Also review the following
  - ✓ Versions
  - ✓ Provisions Instructions.

Step-08-01 Call Module from Terraform Work Directory (Root Module)

- ✓ Create Terraform Configuration in Root Module by calling the newly published module in Terraform Private Registry.
- ✓ c1-version.tf
- ✓ c2-version.tf: Review Bucket
- ✓ c3-version.tf
- ✓ c4-version.tf

→

__/__/____

```
module "website_s3_bucket" {
  source  = "app.terraform.io/hctaprep/s3-website-internal/aws"
  version = "1.0.0"
  # insert required variable here.
  bucket_name = var.my_s3_bucket
  tags = var.my_s3_tags.
}
```

Step-08.02 Execute Terraform Commands:

terraform init

It should fail, cli not have access to private module registry in terraform cloud.

CLI - terraform login

and follow the steps to authorize.

terraform init

terraform validate

terraform fmt

terraform plan

terraform apply -auto-approve

Verify buckets and resources created, read access, public access etc.

Step-08.03: Upload index.html to s3 bucket and test

Step-08.04: Destroy and clean-up

Step-09 Do this assignment