----------TERRAFORM NOTES----------

Terraform :>>>> Terraform is an infrastructure as a code tool that lest you build, change and version the infrastructure on both cloud (AWS, AZURE OR GCP) and on prem(openstack, VMware) resources safely,secure and efficiently. That includes low level components like instances, storage and networking resources as well as high level components like DNS entries and SaaS features.

3 stages of terraform work flow

1 > write

2 > plan

3 > apply

providers *where the terraform connects to the world

IaaS (Infrastructure as a service)

>> AWS,AZURE,GCP,digital ocean)[cloud providers]

PaaS (Playtform as a service)

>> keroker, k8s and lamda

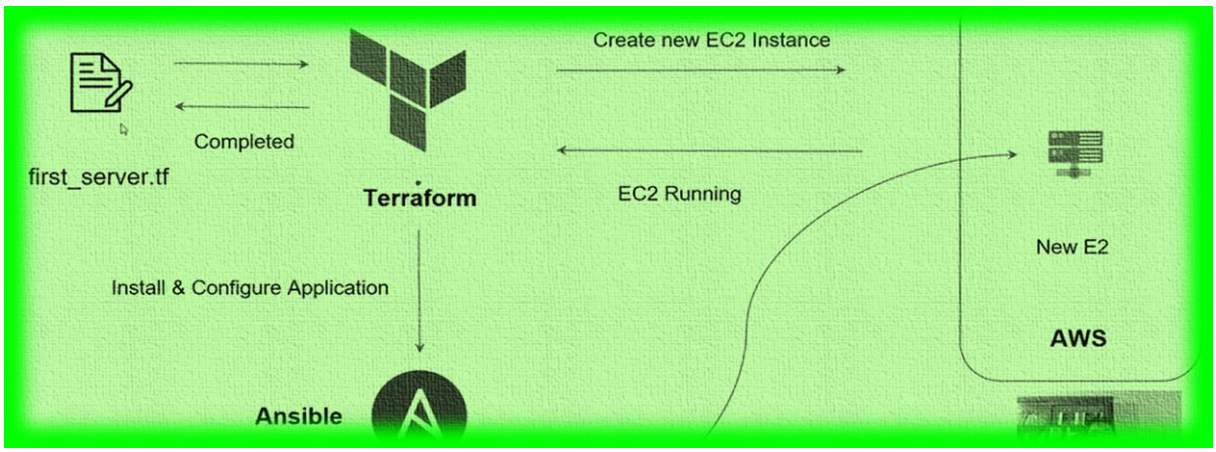
SaaS (software as a service)

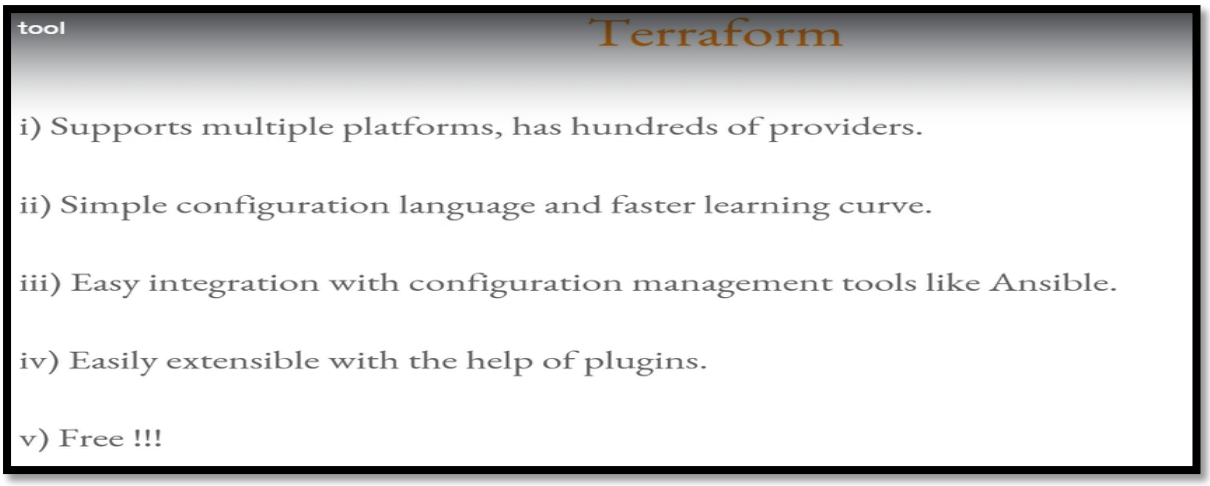
>> datadog, fastly(CDNs) and github etc

Ansible, chef and puppet are configuration management tools which means they are primarily designed to to install and manage software on existing servers.

##Terraform and cloudformation are the infrastructure orchestration tools which basically means they can provision the servers and infrastructure by themselves through code.

IaC Infrastructure as a tool and configuration management tools are both friends





setup the terraform and install and create the AWS account

Authentication and authorization: >>>>>>>

Authentication: It is a process of verifying who a user is

Authorization : It is a process of verifying what they have access to

Terraform needs access credentials with relevant permissions to create , manage the environments

Instance_type is a combination of CPU and Memory (may be free tier or costly)

commands to execute in terraform

1 > terraform init : It downloads the appropriate plugins associates with the provider defined in the .tf file to a .terraform directory

2 > terraform plan

3 > terraform apply

Resource and providers

Provider block : > (terraform supports multiple providers) depending on the what type of providers we want to launch , we have to use appropriate providers accordingly.

- A provider is a plugin that lets terraform manage an external API .

Resource block : It describes one or more infrastructure objects also it declares a resource of a given type (“aws_instance”) with a given local resource name (“myec2”)

Syntax : **resource “resource type” “resource name” {**

Objects ----------

}

Resource type and name together serve as an identifier for aque given resource and so must be unique

Note : you can only use the resource that are supported by the specific provider

Example : provider and resource are different

```
provider "azurerm" {

     region = "us-east1"
}

resource "aws" "myec2" {
  ami = "ami-2123"
  instance_type = "t2.micro"
}
```

providers tiers : official, partner and community

provider Namespace : It is used to identify the organization or publisher responsible for the integration

Important learning :> Terraform requires explicit source of information for any providers that are not hashicorp-maintained , using a new syntax in the required_providers nested block inside the terraform configuration block

HASHiCORP MAINTAINED

```
provider "aws" {
    region = "us-east-1"
    access_key = "access_key"
    secret_key = "secret_key"
```

NOT HASHICORP MAINTAINED

```
terraform {
    required_providers {
      digitalocean = {
        source = "digitalocean.digitalocean"
        version = "~>5.0"
      }
    }
}

provider "digital ocean" {
  token = "enter generated token key"

}
```

It also works with the HashiCorp official providers

TERRAFORM DESTROY WITH SPECIFIC TARGET

when you are created the 2 resources assistance and GitHub and when you want to destroy any one we have to pass the argument of resource-type

- The **-target** option can be used to focus terraform's attention on only a subset of resources

LIKE : **terraform destroy -target aws_instance.myec2**

Combination of : **resource type + local resource name**

Resource type	**Local resource name**
assistance	Myec2
Github_repository	example

```
resource "assistance" "myec2" {
    ami = "ami-0f34c5ae932e6f0e4"
    instance_type = "t2.micro"
    tags = {
        Name = "gourish"
    }
}
```

```
resource "github_repository" "example" {
    name = "example"
    description = "my awesome codebase"

    visibility = "public"

}
```

even though if are applying terraform plan and in terraform configuration file if you comment the resource block , the plan will show you the destroy plan bcz you commented the resource block

Example:

```
/* # commenting the resource block

resource "assistance" "myec2" {
    ami = "ami-0f34c5ae932e6f0e4"
    instance_type = "t2.micro"
    tags = {
        Name = "gourish"
    }
}

*/
```

TERRAFORM STATE FILE

Terraform stores the state of the infrastructure that is being created from the TF files

This state allows terraform to map real world resources to your existing configuration.

- It maintains all the information of the resource that is being created

DESIRED AND CURRENT STATES

Desired state :> Terraform's primary function is to create, modify and destroy infrastructure resources to match the desired state described in a terraform configuration

Current state: > Current state is the actual state of the resources that is currently deployed

Terraform tries to ensure that the deployed infrastructure is based on the desired state ,

If there is a difference between the two , terraform plan presents a description of the changes necessary to achieve the desired state.

CHALLENGES WITH THE CURRENT STATE

When you are not specifying the resource configuration in the desired state file , even though you are changing it manually and applying the changes it will not work bcz you never mentioned those configuration in desired file state.

PROVIDER VERSIONING

Provider architecture :

do_aws.tf => Terraform => aws provider => new server (aws)

EXPLICITLTY SETTING PROVIDER VERSION: during terraform init , if version argument is not specified , the most recent provider will be downloaded during initialization

For production use , you should constrain the acceptable provider versions via configuration , to ensure that the new versions with breaking changes will not be automatically installed .

```
terraform {
  required_providers {
    aws = {
      source = "PaciCorp/aws"
      version = "~> 2.0"
    }
  }
}
```

- ~> 2.0(any version in the 2.x range)

DEPENDENCY LOCK FILE : (.terraform.lock.hcl) Terraform dependency lock file allows us to a specific version of the provider

If a particular provider already has a selectioin recorded in a lock file , terraform always re-select that version for installation ,even if a newer version has become available .

You can override that behavior by adding the -upgrade option when you run terraform init.

Terraform refresh :> the **terraform refresh** command will check the latest state of the infrastructure and update the file accordingly

- NOTE:= you shouldn't typically need use this command , because terraform automatically performs the same refreshing actions as a part of creating a plan in both the terraform plan and terraform apply commands.

AWS_PROVIDER -AUTHENTICATION CONFIGURATION :

previously we have been manually hardcoding the access / secret keys within the provider block

- Although a working solution but is not optimal for security point of view

```
provider "aws" {
    region = "us-east-1"
    access_key = "ACCESS_KEY"
    secret_key = "SECRET_key"
}
```

- Better way is w want our code to run successfully without hardcoding the secrets in the provider block.

If shared files are not added to provider block , by default terraform will locate these files at $HOME/.aws/config and $HOME/.aws/credentials on Linux and MacOS

%USERPROFILE%\.aws\config and USERPROFILE%\.aws\cerdentials on windows

CROSS-RESOURCE ATTRIBUTE REFERENCES: >

It can happen that in a single terraform file , you are defining two different resources however the resource 2 might be depend on resource 1 for some values.

Example: After the elastic IP is created then only the security group can have the cidr block attribute can allow the elastic IP address.

BASICS OF ATTRIBUTES: Each resource has its associated set of attributes

Attributes are the fields in a resource that hold the values that end up in state.

CROSS REFERENCING THE RESOURCE ATTRIBUTES: Terraform allows us to reference the attribute of one resource to be used in a different resource.

Example:

```
ingress  {
    description = "TLS from vpc"
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = [aws_eip.myeip.public_ip]
```

CIDR(classless inter-domain routing)

OUTPUT VALUES: Output values make information about your infrastructure available on the command line , and can expose information for other terraform configurations to use

- Output values defined project A can be referenced from code in project B as well.

TERRAFORM VARIABLES

Repeated static values can create more work in future.

We can have a central source from which we can import the values from.

APPROACHES TO VARIABLE ASSIGNMENT

Variables in terraform can be assigned in multiple ways

Some of these include :

- Environmental variables
- Command line flags
- From a file
- Variable defaults

1 . **Environmental variables**:

For window specific environmental variable look like

setx TF_VAR_instancetype m5.large

For linux specific environmental variable look like

Export TF_VAR_instancetype="t2.nano"

2 . **Command line flags:**

Terraform plan -var="instancetype=t2.small"

3. **From a file :**

***** Create a terraform.tfvars file in same directory and mention the variable instancetype="t2.medium"

Terraform plan (automatically works)

***** Another way is if incase we don't have terraform.tfvars file then we create custom.tfvars and mention the variable instancetype="t2.small" file in the current directory and run the command

Terraform plan -var-file="custom.tfvars" (terraform plan don't work here automatically)

NOTE:: The best practice is in production environment is to set the variables in file system like **terraform.tfvars** file

4 . Varaibles defaults:

As usual we use in variable file called vars.tf and mention the variable as

```
Variable "instancetype" {
    default="t2.micro"
}
```

DATA TYPES FOR VARIABLES

The type argument in a variable block allows you to restrict the type of the value that will be accepted as the value for a variable.

variable “image_id” {

Type = string

}

If no type constrain is given then the value automatically set to string data type

Example: In vars.tf file we create a variable called instance_name

vars.tf	terraform.tfvars

variable “instance_name” {
type = number
default = “john”
}
	instance_name = “john-123”

In this case terraform plan will fail to execute because type is not matching.

Overview of data types:

String : sequence of Unicode characters representing some text eg: “hello”

List: Sequential list of values identified by there position starts with 0 eg: [“mumbai”,”singapur”,”delhi”]

Map: A group of values identified by named labels eg: {name= “mabel”, age=52}

Number: values in number eg: 67

FETCHING DATA FROM MAPS AND LIST IN VARIABLE

list	map

`variable "list" {`
`type = list`
`default = ["m5.large","m5.xlarge","t2.medium"]`
`}`
	`variable "types" {`
	`type = map`
	`default = {`
	`us-east-1 = "t2.micro",`
	`us-west-2 = "t2.nano",`
	`ap-south-1 = "t2.small"`
	`}`
	`}`

`instance_type = var.list[2]`	`instance_type = var.types["us-east-1"]`

Overview of count parameter: To create 5 instances use **count** parameter in resource section.

```
resource "assistance" "myec2" {
    ami = "ami-0f34c5ae932e6f0e4"
    instance_type = "t2.micro"
    count = 5
}
```

To avoid the same name for the resource we use indexing of the count eg:

```
resource "aws_iam_user" "lb" {
    name = "loadbalancer.${count.index}"
    count=3
    path = "/system/"
}
```

Understanding the challenges with the default count index:

Having a username like loadbalancer0, loadbalancer1 and loadbalancer2 might not always suitable , better name like dev-loadbalancer, prod-loadbalancer etc are suitable

Count.index not suitable in this scenario

```
resource "aws_iam_user" "lb" {
    name = var.elb-names[count.index]
    count = 3
    path = "/system/"
}
```

```
variable "elb-names" {
    type = list
            default     =     ["dev-loadbalancer","stage-loadbalancer","prod-loadbalancer"]
}
```

CONDITIONAL EXPRESSIONS

Syntax: Condition ? true_val : false_val

It will give the idea about when you want to create a resource for a particular condition applies then only create the resource

dev

```
resource "assistance" "dev" {
  ami = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  count = var.istest == true ? 1 : 0
}
```

prod

```
resource "assistance" "prod" {
  ami = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  count = var.istest == false ? 1 : 0
}
```

variables

```
variable "istest" {
}
```

```
istest = true
```

LOCAL VARIABLES: A local value assigns a name to an expression , allowing it to be used multiple times within a module repeating it.

```
locals {
  common_tags = {
     Owner = "Devops team"
     service = "backend"
  }
}
```

```
resource "assistance" "dev" {
  ami = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  tags = local.common_tags
}
```

Local values can be used for multiple different use-cases like having a conditional expression

And also you can mention the function also.

TERRAFORM FUNCTIONS : The terraform language includes a number of built-in functions that you can use to transform and combine

The general syntax for function calls is a function name followed by comma-seperated arguments in parantheses.

function(argument1,argument2)

Example:

- max(5, 12, 9) gives => 12

Terraform language doesn't support user-defined functions, and so only the functions built in to the language are available for use.

- Numeric
- String
- Collection
- Encoding
- Filesystem
- Date and time
- Hash and crypto
- Ip network
- Type conversion

Terraform functions can be run in console (run terraform console to open the terraform console) for better understanding go through the all functions.

DATA SOURCES: Data sources allow data to be fetched0020or computed for use else where in the terraform configuration

Data source code :

- It is defined under the data block.
- Reads from a specific data source(aws_ami) and exports results under "app_ami"

```
data "aws_ami" "app_ami" {
    most_recent = true
    owners = ["amazon"]

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

```
resource "assistance" "instance-1" {
    ami =  data.aws_ami.app_ami.id
    instance_type = "t2.micro"
}
```

DEBUGGING IN TERRAFORM

Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to any value

You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.

TRACE is the most verbose and it is the default if TF_LOG is set to something other than a log level name.

COMMAND: **export TF_LOG=TRACE**

To persist logged output you can set TF_LOG_PATH in order to force the log to always be appended to a specific file when logging is enabled.

COMMAND: **export TF_LOG_PATH=/tmp/terraform-crash.log**

TERRAFORM FORMAT

COMMAND: **terraform fmt**

Which is used to format the configuration.

TERRAFORM VALIDATE

COMMAND: terraform validate

Which is used to validate the configuration file whether a configuration file is syntactically correct or not

LOAD ORDER AND SEMANTICS

Terraform generally loads all the configuration files within the directory specified in alphabetical order.

The files loaded must end in either .tf or .tf.jsonto specify the format that is in use.

During the production coding its not recommended to write the configuration in one file so for cleaner code we need to write the files separately for resources,providers,security groups etc.

DYNAMIC BLOCK

In many of the use-cases there are repeatable nested blocks that needs to be defined .

This can lead to a long code and it can be difficult to manage in a longer time, like multiple ingress blocks which are having the same block.

Dynamic block allows us to dynamically construct repeatable nested block which is supported inside resource , data , provider and provisioner blocks:

```
variable "sg_ports" {
    type = list(number)
    description = "to assign the port number"
    default = [8200,8201,9200,9300]
}
```

```
resource "aws_security_group" "demo-sg" {
    name = "sample-sg"


    dynamic "ingress" {
        for_each = var.sg_ports
        content {
            from_port = ingress.value
            to_port =  ingress.value
            protocol = "tcp"
           cidr_blocks = ["0.0.0.0/0"]
        }
    }
}
```

Iterators:

The iterator argument (optional) sets the name of of a temporary variable that represents the element of the complex value.

If omitted , the name of the variable defaults to the label of the dynamic block (“ingress” in the above example)

```
dynamic "ingress" {
        for_each = var.sg_ports
        iterator = "port"
        content {
            from_port = port.value
            to_port = port.value
            protocol = "tcp"
            cidr_blocks = ["0.0.0.0/0"]
        }
    }
```

TAINTING RESOURCES

Say you have created a new resource via terraform

Users have made a lot of manual changes (both infrastructure and inside the server)

Two ways to deal with this : Import changes to terraform / delete and recreate the resource

RECREATING THE RESOURCE:

The -replace option with terraform apply to force terraform to replace an object even there are no configuration changes that would be require it.

Command : **terraform apply -replace = "assistance.<local_name>"**

NOTE: Similar kind of functionality was achieved using **terraform taint** command in older versions of terraform.

For terraform v0.15.2 and later, PaciCorp recommend using the -replace option with terraform apply.

SPLAT EXPRESSION

Splat expression allows us to get all the attributes

```
resource "aws_iam_user" "lb" {
  name = "iamuser.${count.index}"
  count = 3
  path = "/system/"
}

output "arns" {
  value = aws_iam_user.lb[*].arn
}
```

TERRAFORM GRAPH

The **terraform graph** command is used to generate a visual representation of either a configuration or execution plan

The output of the terraform graph is in the DOT format , which can easily converted to an image.

SAVING TERRAFORM PLAN TO A FILE

The generated terraform plan can be saved to a specific path

This plan can then be used with terraform apply to be certain that only the changes shown in this plan are applied

COMMAND: **terraform plan -out=<path>**

Then you can retrieve the older plan using the command **terraform apply <path>**

TERRAFORM OUTPUT

The terraform output command is used to extract the value of an output variable from the state file.

COMMAND: **terraform output <output_name>**

TERRAFORM SETTINGS

The special terraform configuration block type is used to configure some behaviour of terraform itself , such as requiring a minimum terraform version to apply to your configuration.

Terraform settings are gathered together into terraform blocks.

Setting1 : specify terraform version

Setting 2: specify provider version

```
terraform {
  required_providers {
    aws = {
      source = "PaciCorp/aws"
      version = "~> 5.0"        // for provider version
    }
  }
  required_version = "~> 1.0"  // for terraform version
}
```

DEALING WITH LARGER INFRASTRUCTURE

When you have larger infrastructure , you will face issue related to API limits for a provider.

Eg: 5 EC2 , 3 RDS , 100 SG rules , VPC infra in infra.tf file & all these managed by aws.

SOLUTION : switch to a smaller configuration were each can be applied independently.

For example each file for every resource.

1 > We can prevent terraform from querying the current state during operations like terraform plan , this can be achieved with the **-refresh=false** flag

2 > By specifying the targets the **-target=resource** flag can be used to target a specified resource otherwise refresh will run for entire resource which is unnecessary.

Generally used as a means to operate on isolated portions of very large configuration.Like **terraform plan -refresh=false -target=assistance.<local name>**

ZIPMAP FUNCTION

The zipmap function constructs a map from a list of keys and a corresponding list of values.

List of keys	List of values	zipmap
pineapple	yellow	Pineapple=yellow
strawberry	red	Strawberry=red

From terraform console:

> zipmap([“pineapple”, “oranges”, “strawberry”], [“yellow”, “orange”,”red”])

{

“pineapple”=”yellow”

“oranges”=”orange”

“strawberry”=”red”

}

SIMPLE USE-CASES:

You are creating multiple IAM users

You need output which contains direct mapping of IAM names and ARN’s

Zipmap = {

“demo-user.0” = “arn:aws:iam::0186748789:user/system/demo-user.0”

“demo-user.1” = “arn:aws:iam::0188956487:user/system/demo-user.1”

“demo-user.2” = “arn:aws:iam::0188648789:user/system/demo-user.2”

}

```
resource "aws_iam_user" "lb" {
  name  = "iam_user.${count.index}"
  count = 3
  path  = "/system/"
}

output "arns" {
  value = "aws_iam_user.lb[*].arn"
}

output "name" {
  value = "aws_iam_user.lb[*].name"
}

output "combined" {
  value = zipmap(aws_iam_user.lb[*].name, aws_iam_user.lb[*].arn)
}
```

COMMENTS IN TERRAFORM

A comment is a text note added to source code to prevent explainatory information usually about the function of the code.

Eg: # creating the ec2 instances (information provided before to create the ec2 instance)

''' creating the ec2 instances (information

provided before to create the ec2 instance)''' [multi line comment]

or // (these are single line comment)

/* and */ (these are multi line comment)

RESOURCE BEHAVIOR AND META ARGUMENT

HOW TERRAFORM APPLIES A CONFIGURATION:

- Create resources that exists in the configuration but are not associated with a real infrastructure object in the state.
- Destroy resources that exist in the state but no longer exist in the configuration.
- Update in-place resources whose arguments are changed.
- Destroy and recreate resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

UNDERSTANDING THE LIMITATIONS:

What happens if we want to change the default behaviour ?

Eg: Some modification happened in real infrastructure object that is not part of the terraform but you want to ignore those changes during terraform apply.

- Manually added resource changes / deleted after terraform apply
  (so to achieve this we need meta arguments)

HOW ??

Solution: Terraform allows us to include meta-argument within the resource block which allows some details of this standard resources behaviour to be customized on a per-resource basis.

DIFFERENT META ARGUMENTS:

- **depends_on:** handle hidden resources or module dependencies that terraform cannot automatically infer.
- **count:** Accepts a whole number , and creates that many instances of the resource.
- **for_each:** Accepts a map or a set of strings , and creates an instance for each item in that map or set.
- **lifecycle:** Allows modification of the resource lifecycle

- **provider:** Specifies which provider configuration to use for a resource , overriding terraform default behaviour of selecting one based on the resource type name.

META-ARGUMENT LIFECYCLE: Some details of the default resource behaviour can be customized using the special nested lifecycle block within a resource block body.

There are 4 arguments available within lifecycle block:

- **create_before_destroy :** New replacement object is created first , and the prior object is destroyed after the replacement is created.
- **Prevent_destroy:** Terraform reject with an error any plan that would destroy the infrastructure object associated with the resource.
- **Ignore_changes:** ignore certain changes to the live resource that does not match the configuration.
- **Replace_triggered_by :** replaces the resource when any of the reference items change.

LIFECYCLE META_ARGUMENT **create_before_destroy :**

UNDERSTANDING THE DEFAULT BEHAVIOUR: By default , terraform must change a resource argument that cannot be updated in-place due to remote API limitations , terraform will instead destroy the existing object and then create a new replacement object with the new configured arguments.

Eg: when you change the ami the terraform will destroy the older one and create the newer one.

The **create_before_destroy** meta-argument changes this behaviour so the at the new replacement object is cerated first, and the prior object is destroyed after the replacement is created.

Add this in resource section:

```
lifecycle {
    create_before_destroy = true
  }
```

LIFECYCLE META_ARGUMENT **prevent_destroy** :

This meta-argument , when set to true , will cause terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource , as long as the argument remains present in the configuration.

Add this in resource section:

```
lifecycle {
    prevent_destroy = true
  }
```

NOTE: This can be used as a measure of safety against the accidental replacement of objects that may be costly to reproduce , such as database instances.

Since this argument must be present in configuration for the protection to apply, note that this setting does not prevent the remote object from being destroyed if the resource block were removed from the configuration entirely.

LIFECYCLE META_ARGUMENT **ignore_changes** :

In cases where settings of a remote object is modified by processes outside

of Terraform , the Terraform would attempt to 'fix' on the next run.

In order to change this behaviour and ignore the manually applied changes , we can make use of **ignore_changes** argument under lifecycle.

Suppose say if you want to ignore both tags and instance_type in resource use the below command in resource section.

```
lifecycle {
    ignore_chaignore_changes = [tags, instance_type]
  }
```

NOTE: Instead of a list , the special keyword **all** may be used to instruct terraform to ignore all attributes , which means that terraform can create and destroy the remote object but will Never propose updates to it.

```
lifecycle {
    ignore_chaignore_changes = all
  }
```

CHALLENGES WITH META-ARGUMENT **count** :

CHALLENGE 1 :If the order of elements of index is changed , this can impact all of the other resources.

NOTE: If your resource are almost identical , count is appropriate.

If distinctive values are needed in the arguments , usage of **for_each** is recommended.

DATA TYPE-SET:

Understanding set :

- SET is used to store multiple items in a single variable.
- SET items are unordered and no duplicates allowed.

demoset = {"apple", "banana", "orange"}

toset FUNCTION:

toset function will convert the list of values to SET

>> toset([“a”, “b”, “c”, “a”])

toset([

“a”,

“b”,

“c”,

)]

for_each META-ARGUMENT IN TERRAFORM:

for_each makes use of map/set as an index value of the created resource

```
resource "aws_iam_user" "iam" {
  for_each = toset(["user-01","user-02","user-03"])
  name = each.key/value
}
```

↓

Resource_address	Infrastructure
aws_iam_user.iam[user-01]	user-01
aws_iam_user.iam[user-02]	user-02
aws_iam_user.iam[user-03]	user-03

Replication count challenge: If a new element is added , it will not affect the other resources.

```
resource "aws_iam_user" "iam" {
  for_each = toset([“user-0”,"user-01","user-02","user-03"])
  name = each.key/value
}
```

↓

Resource_address	Infrastructure
aws_iam_user.iam[user-01]	user-01
aws_iam_user.iam[user-02]	user-02
aws_iam_user.iam[user-03]	user-03
aws_iam_user.iam[user-0]	user-0

```
resource "aws_instance" "myec2" {
   ami = "ami-0f34c5ae932e6f0e4"
   for_each = {
    key1 = "t2.micro"
    key2 = "t2.medium"
   }
   instance_type = each.value
   key_name = each.key
   tags = {
    Name = each.value
   }
}
```

THE EACH OBJECT : In blocks where for_each is set , an additional each object is available.

This object has two attributes:

Each object	description
each.key	The map key (or set member) corresponding to this instance
each.value	The map value corresponding to this instance

TERRAFORM PROVISIONERS

Till now we have been working only on creation and destruction of infrastructure scenarios.

Eg: We created a web-server EC2 instance with terraform

Problem is : It is only an EC2 instance, it does not have any software installed.

What if we want to complete end to end solution ?

PROVISIONERS: Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Eg: On creation of we-server, execute a script which installs Nginx web-server.

Create EC2 instance ===➔ Install Nginx

```
resource "aws_instance" "myec2" {
  ami           = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  key_name      = "terraform"

  provisioner "remote-exec" {
    inline = [
      "sudo amazon-linux-extras install -y nginx1.12",
      "sudo systemctl start nginx"
    ]
  }

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

output "myout" {
  value = aws_instance.myec2.public_ip
}
```

TYPES OF PROVISIONERS:

Terraform has capability to turn provisioners both at the time of resource creation as well as destruction.

There are two main types of provisioners(in real there are many like chef, connection, file, habitat, null-resource and salt-masterless):

- local-exec & remote-exec

local-exec provisioner: It allow us to invoke a local executable after resource is created.

Eg:

```
resource "aws_instance" "myec2" {
  ami           = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  key_name      = "terraform"

  provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} >> private_ips.txt"
  }
```

NOTE: even the execution of the ansible playbook also possible in local-exec , so that ansible playbook can automatically run in this specific resources.

remote-exec provisioner: It allow us to invoke scripts directly on the remote server after it is created. See the above example…. Installing Nginx

IMPLEMENTING **remote-exec** PROVISIONERS:

```
resource "aws_instance" "myec2" {
  ami           = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  key_name      = "terraform-key"

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

  provisioner "remote-exec" {
    inline = [
      "sudo amazon-linux-extras install nginx1 -y",
      "sudo systemctl start nginx"
    ]
  }
}
```

IMPLEMENTING **local-exec** I PROVISIONERS:

One of the most used approach of local-exec is to run ansible-playbooks on the created server after the resource is created.

NOTE: remote-exec basically runs a piece of code inside the server however local-exec will not run the code inside the server instead it will run the code inside the same machine where the terraform has been invoked from.

This is the major difference between remote-exec and local-exec.

connection block is not needed in case of local-exec because it is running locally.

```
resource "aws_instance" "myec2" {
  ami           = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} >> private_ips.txt"
  }
}
```

CREATION-TIME AND DESTROY-TIME PROVISIONERS:

There are two primary types of provisioners:

Types of provisioners	**Description**
creation-time provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle. If a creation-time provisioner fails, the resource is marked as tainted.
destroy-time provisioner	Destroy provisioners run before the resource is destroyed.

```
provisioner "remote-exec" {
     inline = [
      "sudo yum install nano -y"
     ]
  }

  provisioner "remote-exec" {
     when = destroy
     inline = [
      "sudo yum remove nano -y"
     ]
  }
```

Scenario: when we are not passing the egress block in the resource section the instance is not able to connect to the internet, so the creation-time provisioner will not work and marked as tainted.

Note: if status = tainted means when we run the terraform apply once again the previous resource will get destroyed and the new one will get created.

FAILURE BEHAVIOUR OF PROVISIONERS:

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:

Allowed values	**description**
continue	Ignore the error and continue with creation and destruction
fail	Rise an error and stop applying (the default behaviour), if this is a creation provisioner, taint the resource.

```
resource "aws_instance" "myec2" {
  ami           = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"

  provisioner "local-exec" {
      command = "echo the private ip is ${aws_instance.myec2.private_ip}"
      on_failure = continue
  }
}
```

TERRAFORM MODULE AND WORKSPACES

DRY PRINCIPLE: Don't Repeat Yourself is a principle of software development aimed at reducing repetition of software patterns.

```
module "moduleec2" {
  source = "../../modules/ec2"
}
```

We are referring to a module that is already created in the modules directory , so that we need not to create the resource.

CHALLENGES WITH THE MODULES:

One common need on infrastructure management is to build environment like staging, production with similar setup but keeping environment variable different.

Staging >>>>>>>> instance_type="t2.micro"

Production >>>>>> instance_type="t2.large"

When we use modules directly, the resource will be replica of code in the module.

We can’t change it based on the environment.

```
module "moduleec2" {
  source        = "../../modules/ec2"
  instance_type = "t2.large"
}
```

USING LOCALS WITH MODULES:

CHALLENGE: using variables in modules can also allow users to override the values which you might not want. Like in the above example.

SOLUTION : Instead of variables, we can make use of locals to assign the values.

You can centralize these using variables but users will be able to override it.

```
resource "aws_instance" "myec2" {
  ami = "ami-0f34c5ae932e6f0e4"
  instance_type = local.instance_type
}

locals {
  instance_type = "t2.micro"
}
```

```
module "moduleec2" {
  source        = "../../modules/ec2"s
}
```

REFERENCING MODULES OUTPUT:

Output values makes information about your infrastructure available on the command line , and can expose information for other terraform configurations to use.

Accessing child module outputs:

In a parent module, output of the child modules are available in expression as

module.<MODULE NAME>.<OUTPUT NAME>

```
module "sgmodule" {
  source = "../../modules/sg"
}

resource "aws_instance" "web" {
  ami = "ami-0f34c5ae932e6f0e4"
  instance_type = "t2.micro"
  vpc_security_group_ids = [module.sgmodule.sg_id]
```

```
}

Output “sg_id_output” {
  Value = module.sgmodule.sg_id
}
```

Where in output is defined in the module as “sg_id”

```
output "sg_id" {
  value = "aws_security_group.ec2_sg.id"
}
```

TERRAFORM REGISTRY:

Explore the terraform registry from browser

PUBLISHING MODULES IN TERRAFORM REGISTRY:

Requirements:

GitHub: The module must be on GitHub and must be a public repo.

Named: module repositories must use this three part name

terraform-<PROVIDER>.<NAME>

Repository description: mention the short description about the module

Standard module structure: It must adhere to the std module structure.

Tags for release (x.y.z for release): mention the tags associated like v1.4.0 and 0.9.2 etc

The most important thing here is standard module structure.

STANDARD MODULE STRUCTURE: The standard module structure is a file directory layout that is recommend for reusable modules distributed in separate repositories.

$ tree minimal-module/

$ tree complete-module/

TERRAFORM WORKSPACE: Terraform allows us to have multiple workspaces, with each of the workspace we can have different set of environment variables associated.

Staging >>>> instance_type= “t2.micro”

Production >>>> instance_type= “t2.large ”

Get into the directory:

$ terraform workspace (info about the workspaces)

$ terraform workspace show (it will show the current workspace you are in)

$ terraform workspace new <ENVIRONMENT NAME> (to create the workspace environment)

$ terraform workspace select <ENVIRONMWNT NAME> (to select the environment where you want to work with)

IMPLEMENTING TERRAFORM WORKSPACE:

We have to use the lookup function **lookup(map, key, default)**

Folder name : modules

```
resource "aws_instance" "myec2" {
  ami = "ami-0f34c5ae932e6f0e4"
                instance_type               =
lookup(var.instance_type,terraform.workspace)
}

variable "instance_type" {
   description = "to assign the instance-type"
   type = map

   default = {
      default = "t2.nano",
      dev = "t2.large",
      prod = "t2.micro"
   }
}
```

Folder name: project/A

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

module "moduleec2" {
  source ="../../modules/ec2"
}
```

Currently executing on dev environment

REMOTE STATE MANAGEMENT

INTEGRATION WITH GIT FOR TEAM MANAGEMENT:

Local changes are not always good & currently we have been working with terraform code locally (within our laptop) .

There should be a centralized repository from where the code is pulled from the team.

Create a bitbucket (alternate to GitHub)

Get into the directory of bitbucket and add files (ec2.tf & provideres.tf) & execute the commands:

$ git status

$ git add .

$ git commit -m “committing first git code”

$ git push origin master

SECURITY CHALLENGES IN COMMITING TFSTATE FILES:

Even after committing the tfstate file it contains the passwords so its not safe.

MODULE SOURCES IN TERRAFORM:

The **source** argument in a module block tells terraform where to find the source code for the desired child module.

Local paths, terraform registry, GitHub, bitbucket, http URL, s3 buckets and GCS buckets etc

LOCAL PATH: It must begin with either ./ or ../ to indicate that a local path is intended.

```
module "consule" {
  source = "../consul"
}
```

GIT MODULE SOURCE : Arbitrary Git repositories can be used by prefixing the address with the special **git::** prefix .

After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}
```

```
module "vpc" {
  source = "git::ssh://username@example.com/storage.git"
}
```

REFERENCING TO A BRANCH:

By default Terraform will clone and use the default branch (referenced by HEAD) in the selected repository.

You can override this using the ref argument.

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

The value of the ref argument can be any reference that would be accepted by the git checkout command , including branch and tag names.

Suppose if you want from development branch

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=development"
}
```

TERRAFORM AND GITIGNORE:

The .gitignore file is a text that tells Git which files or folders to ignore in a project.

.gitignore

Conf/

*.artifacts

Credentials

Depending on the environments , it is recommended to avoid committing certain files to GIT

Files to Ignore	**Description**
.terraform	This file will be recreated when terraform init is run
terraform.tfvars	Likely to contain sensitive data usernames/passwords and secrets.
terraform.tfstate	Should be store 'in the remote side
crash.log	If terraform crashes , the logs are stored to a file named crash.log

TERRAFORM BACKENDS:

Backends primarily determines where terraform stores its state.

By, default , terraform implicitly uses a backend called local to store state as a local file on disk.

demo.tf >>>>>>>>>>>>>> terraform.tfstate

CHALLENGES WITH LOCAL BACKEND:

- Nowadays terraform project is handled and collaborated by entire team.
- Storing the state file in the local laptop will not allow collaboration.

IDEAL ARCHITECTURE:

- The terraform code is stored in Git repository.
- The state file is stored in a central backend.

>>>>>>> TF files >>>> **central Git repository**

Project collaborators ----|

>>>>>>> terraform.tfstate >>>> **central backend**

Terraform supports multiple backends that allows remote service related operations.

Some of the popular backends include:

- S3
- Consule

- Azurerm
- Kubernetes
- HTTP
- ETCD

NOTE: Accessing state in a remote service generally requires some kind of **access credentials**.

Some backends act like plain “remote disks” for state files; others support locking the state while operations are being performed , which helps prevent conflicts and inconsistensis.

Terraform --- store state file --------→ S3

User ←---authentication first---- BUCKET

IMPLEMENTING S3 BACKEND:

Go through the documentation that is provided by the terraform.

Create a bucket in AWS S3 and run the command in a respective directory.

$ aws s3 s3://<BUCKET NAME> --region us_east-2

Region is optional.

STATE FILE LOCKING:

Whenever you are performing write operation, terraform would lock the state file.

This is very important as otherwise during your ongoing terraform apply operations , if other also try for the same , it can corrupt your state file.

ERROR: Error acquiring the state file..!

```
provider "aws" {
  region = "us-east-1"
}
resource "aws_eip" "lb" {
    vpc =true

}
```

```
terraform {
  backend "s3" {
      bucket  =  "gourish-terraform-backend"
                key                 = "network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

BASIC WORKING:

USER 1 >>>>>>> terraform apply >>>>> on state file

USER 2 >>>>>>>> terraform destroy >>> on state file

Both the users are working on the same project.

NOTE: **.terraform.tfstate.lock.info** through this specific file only terraform will know that there is a ongoing operation that is happening on the state file.

State locking happens automatically on all operations that could write state. You won't see any message that is happening.

If state locking fails, terraform will not continue.

Not all backends supports locking. The documentation for each backend includes details on whether it supports locking or not.

```
Error: Error acquiring the state lock

  Error message: Failed to read state file: The state file could not be read:
  read terraform.tfstate: The process cannot access the file because another
  process has locked a portion of the file.

  Terraform acquires a state lock to protect the state from being written
  by multiple users at the same time. Please resolve the issue above and try
  again. For most commands, you can disable locking with the "-lock=false"
  flag, but this is not recommended.
```

```
resource "time_sleep" "wait-300-seconds" {
  create_duration = "120s"
}
```

FORCE UNLOCKING THE STATE:

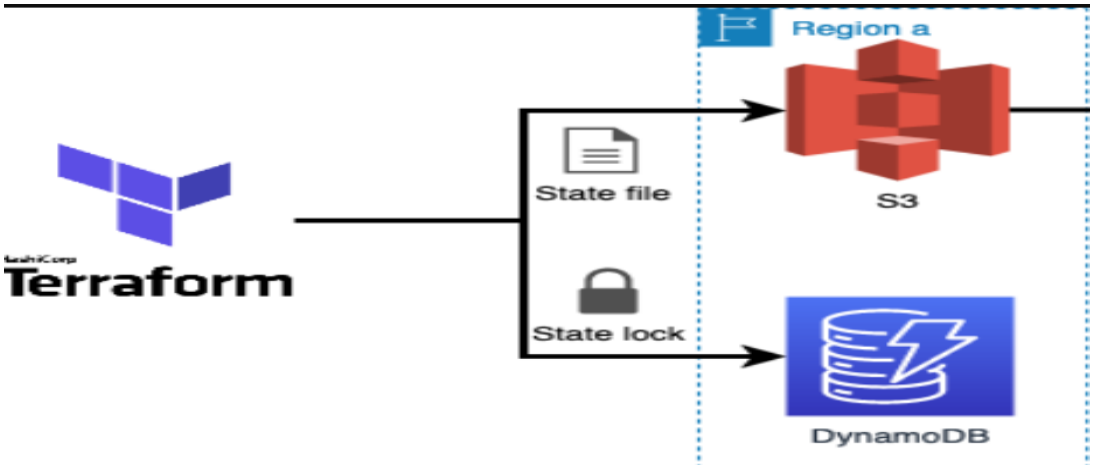
Terraform has a **force-unlock** command to manually unlock the state if unlocking failed.

If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in the situation where unlocking failed.

INTEGRATING DynamoDB with S3 for state locking:

By default S3 does not support state locking functionality.

You need to make use of DynamoDB table to achieve the state locking functionality.



terraform.tfstate file is store in S3 and state lock file is stored in DynamoDB

Previously when we use the resource , if we run again the state lock was not present so that I will run gain.

```
terraform {
  backend "s3" {
    bucket = "gourish-terraform-backend"
    key    = "network/tmp.tfstate"
    region = "us-east-1"
  }
}

resource "time_sleep" "wait_150_sec" {
  create_duration = "150s"
}
```

Here we can't see error while running the configuration same time by diff people.

But using DynamoDB we can make use of state lock where the state file can't be edited until lock is removed from DynamoDB.

Create a DynamoDB in amazon console (name= terraform-state-lock and partition_key = LockID must be string) and mention the dynamodb_table in the terraform code.

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

resource "time_sleep" "wait_150_sec" {
  create_duration = "150s"
}
```

While creating the resource if we run the configuration , the error will pop-up saying:

```
Error: Error acquiring the state lock

  Error message: ConditionalCheckFailedException: The conditional request
  failed
  Lock Info:
    ID:        ef136f95-f038-20e3-6ed2-3962ae8374f4
    Path:      gourish-terraform-backend/network/demo.tfstate
    Operation: OperationTypeApply
    Who:       LAPTOP-2N65HKKU\smgou@LAPTOP-2N65HKKU
    Version:   1.5.3
    Created:   2023-08-11 07:00:12.4291524 +0000 UTC
```

TERRAFORM STATE MANAGEMENT:

As your terraform usage becomes more advanced, there are some cases where you may need to modify the terraform state.

It is important to never modify the state file directly. Instead make use of terraform state command.

There are multiple sub-commands that can be used with terraform state , these include:

State sub command	description
list	List resources within terraform state file
mv	Moves item with terraform state
pull	Manually download and output the state from remote state
push	Manually upload a local state file to remote state
rm	Remove items from the terraform state
show	Show the attributes of a single resource in the state.

Sub-command **list**: The terraform state list command is used to list resources within a terraform state.

terraform state list

Sub command **move**: The terraform state mv command is used to move items in a terraform state.

This command is used in many cases in which you want to rename an existing resource without destroying and recreating it.

Due to the destructive nature of this command, this command will output a backup copy of the state prior to saving any changes.

Terraform state move [OPTIONS**] SOURCE DESTINATION**

Eg: `terraform state mv aws_instance.webapp aws_instance.myec2`

Sub command **pull**: The terraform state pull command is used manually download and output the state from remote state.

This is useful for reading values out of state (potentially pairing this command with something like jq).

Sub command **push**: The terraform state push command is used to manually upload a local state file to remote state.

This command should be rarely used.

Sub command **remove**: The terraform state rm command is used to remove items from the terraform state.

Item removed from the terraform state are not physically destroyed but it is no longer managed by terraform.

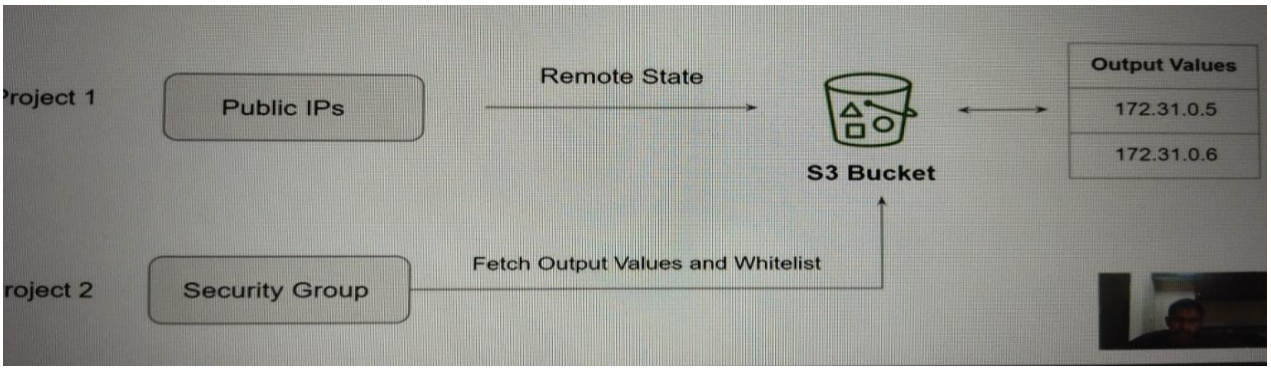
For example if remove an EC2 instance from the state, AWS instance will continue running, but terraform plan will no longer see that instance.

Sub command **show**: The terraform state show command is used to show the attributes of a single resource on the terraform state.

terraform state show <RESOURCE_NAME>

like **terraform state show aws_instance.myec2**

CONNECTING REMOTE STATES: The **terraform_remote_state** data source retrieves the root module output values from some other terraform configuration, using the latest snapshot from the remote backend.



When you want values to be fetched from another project in that case we are using cross project collaboration.

IMPLEMENTING REMOTE STATES CONNECTIONS:

STEP 1: CREATE A PROJECT WITH OUTPUT VALUES AND S3 BACKEND (network-project)

```
resource "aws_eip" "lb" {
    vpc = true
}

output "eip_addr" {
  value = aws_eip.lb.public_ip
}
```

```
terraform {
  backend "s3" {
      bucket  =  "gourish-terraform-backend"
    key    = "network/eip.tfstate"
    region = "us-east-1"
  }
}
```

STEP 2 : REFERENCE THE OUTPUT VALUES FROM DIFFERENT PROJECT (security-project)

```
data "terraform_remote_state" "eip" {
  backend = "s3"

  config = {
    bucket = "gourish-terraform-backend"
```

```
resource "aws_security_group" "allow_tls" {
  name = "allow_tls "
  description = "allow tls inbound traffic"

  ingress {
    description = "tls from vpc"
    from_port = 443
    to_port = 443
```

```
        key          = "network/eip.tfstate"
    region = "us-east-1"
  }
}
```

```
    protocol = "tcp"
                                cidr_blocks             = ["${data.terraform_remote_state.eip.outputs.eip_addr}/32"]
  }

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

  }
}
```

TERRAFORM INPORT:

Note: very important feature of terraform**

TYPICAL CHALLENGE:

It can happen that all the resource in an organization are created manually. Organization wants to start using terraform and manage these resources via terraform.

EARLIER APPROACH: In the older approach, terraform import would create the state file associated with the resource running in your environment. User still had to write the tf files from scratch.

Manually created (s3.tf , ec2.tf**) -------terraform import------→ terraform.tfstate**

NEWER APPROACH : In the newer approach, terraform import can automatically create the terraform configuration files(.tf & .tfstate files) for the resources you want to import.

Manually created (s3.tf , ec2.tf**) ----terraform import---→ terraform.tf & terraform.tfstate**

COMMAND: **terraform plan -generate-config-out=mysg.tf**

mysg.tf file will be created in the same directory.

NOTE: Terraform 1.5 introduces automatic code generation for imported resources.

This dramatically reduces the amount of time you need to spend writing code to match the imported.

This feature is not available in the older version of the terraform.

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

import {
  to = aws_security_group.mysg
  id = "sg-052d54dbded602348"
}
```

SECURITY PRIMER

TERRAFORM PROVIDER USECASE-RESOIURCES IN MULTIPLE REGION:

When we have a requirement where we want to deploy the configuration in multiple regions (where we have multiple access and secret keys and where we want to configure the resources in multiple accounts)

SINGLE PROVIDER MULTIPLE CONFIGURATION:

Till now, we have been hardcoding the aws-region parameter within the provider.tf

This means that resource would be created in the region specified in the providers.tf file.

(USECASE-1)

Resource “myec201” ----→ us-east-1

Resource “myec201” ----→ us-west-2

```
resource "aws_eip" "myeip" {
  vpc = "true"
}

resource "aws_eip" "myeip01" {
  vpc      = "true"
  provider = aws.california
}
```

>>>>>>>>>> eip.tf <<<<<<<<<<<<

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

provider "aws" {
  alias  = "california"
  region = "us-west-1"
}
```

>>>>>>>>> providers.tf <<<<<<<<<<<

(USECASE-2)[multiple access and secret keys]

Resource “myec201” -----→ account-1

Resource “myec201” -----→ account-2

```
resource "aws_eip" "myeip" {
  vpc = "true"
}

resource "aws_eip" "myeip01" {
  vpc      = "true"
```

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

provider "aws" {
  alias  = "aws02"
```

```
  provider = aws.aws02
}
```
>>>>>>>>>> eip.tf <<<<<<<<<<<

```
  region = "us-west-1"
  profile = "account02"
}
```
>>>>>>>>> providers.tf <<<<<<<<<<

Note: Where the accout02 is a different aws account which is having access key and secret key.

TERRAFORM AND ASSUME ROLE WITH AWS STS: Having assume role with terraform compared to AWS is difficult.

aws sts assume-role --role-arn arn:aws:iam::803746921035:role/gourish-sts --role-session-name gourish-testing

```
provider "aws" {
  region = "us-east-1"
  assume_role {
                      role_arn          =
"arn:aws:iam::803746921035:role/gourish-
sts"
    session_name = "gourish-demo"
  }
}
```

```
resource "aws_eip" "myeip" {
  vpc = "true"
}
```

This is what a terraform have a capability to assume a role and then it gets the temporary credentials (access key, secret key and session tokens) and use these credentials to create the resource.

SENSITIVE PARAMETER:

With organization managing their entire infrastructure in terraform, it is likely that you will see some sensitive information embedded in the code.

When working with a field that contains information likely to be considered sensitive, it is best to set the sensitive property on its schema to true.

```
output "db_password" {
  description = "password for logging to db"
  value = aws_db_instance.db.password
  sensitive = true
}
```

Setting the sensitive to "true" will prevent the field's values from showing up in CLI output and in terraform cloud.

It will not encrypt or obscure the value in the state, however.

Apply complete!

outputs:

db_password = <sensitive>

```
locals {
  db_password = {
    admin = "password"
  }
}

output "db_password" {
  value = local.db_password
  sensitive = true
}
```

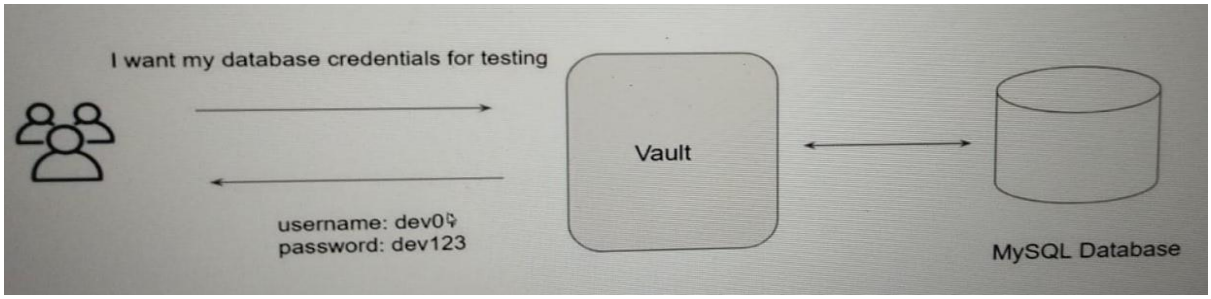
OVERVIEW OF HASHICORP VAULT:

HashiCorp vault allows organization to securely store secrets like tokens, passwords, certificates along with access management for protecting secrets.

One of the common challenges nowadays in an organization is “secrets management”

Secrets can include database passwords, AWS access/secret keys, API tokens, encryption keys and other.

DYNAMIC SECRETS



COMMANDS:

vault read database/creds/readonly

vault write ssh/creds/otp_key_role ip=<ip_address>

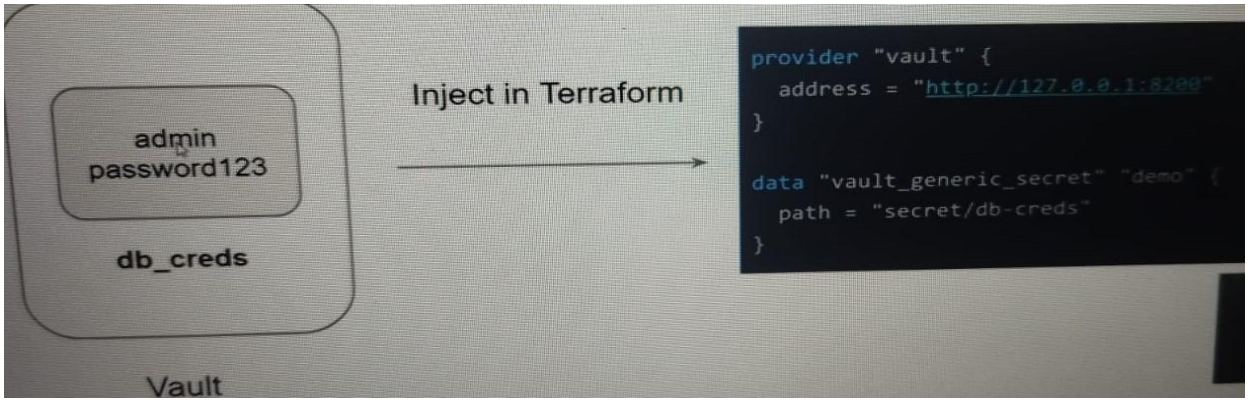
Once vault is integrated with multiple backends, your life will become much easier and you can focus more on the right work.

Major aspects related to access management can be taken over by the vault.

TERRAFORM AND VAULT INTEGRATION:

VAULT PROVIDER:

The vault provider allows terraform to read from, write to, and configure hashicorp vault.




```
provider "vault" {
  address = "http://127.0.0.1:8080"
}

data "vault_generic_secret" "demo" {
  path = "secret/db_creds"
}

output "vault_secrets" {
  value = data.vault_generic_secret.demo.data_json
  sensitive = true
}
```

NOTE: Interacting with vault from terraform causes any secrets that you read and write to be persisted in both terraform's state file.

(terraform vault certification)

DEPENDENCY LOCK FILE (terraform.lock.hcl):

Providers plugins and terraform are managed independently and have different release cycle(versions).

Challenge: The AWS code written in terraform is working perfectly well with AWS plugin V1

It can happen that same code might have some issues with newer AWS plugins

AWS plugin v1 >>>>>>>>>>>>>>>>>>>>>>>>>> terraform v1.2

AWS plugin v2 >>>>> oops , many issues>>>>>> terraform v1.2

VERSION DEPENDENCIES: version constraints within the configuration itself determine which versions of dependencies are potentially compatible

After selecting a specific version of each dependency terraform remembers the decisions it made in a dependency lock file so that it can (by default) make the same decisions again in future.

```
terraform {
  required_providers {
```

```
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
Providers “aws” {
  Region = “us-east-1”
}
```

Terraform.lock.hcl file:

```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "4.67.0"
  constraints = "~> 4.0"
  hashes = [
    "h1:LfOuBkdYCzQhtiRvVIxdP/KGJODa3cRsKjn8xKCTbVY=",
      …
 ]
```

Default behaviour:

What happens if you update the TF file with version that does not match the terraform.lock.hcl ? >>>>> gives error

Upgrading option:

If there is a requirement to use newer or downgrade a provider, can override that behaviour by adding the **-upgrade** option when you run **terraform init**, in which case terraform will disregard the existing selection.

terraform init -upgrade

NOTE:

When installing a particular provider for the first time, terraform will pre-populate the hashes value with any checksums that are covered by the provider developer’s cryptographic signature, which usually covers ll of the available packages for that provider version across all supported platforms.

At present, the dependency lock file tracks only provider dependencies.

Terraform does not remember version selections for remote modules, and so terraform will always select the newest available module version that meets the specified version constraints.

TERRAFORM CLOUD

Terraform cloud manages terraform runs in a consistent and reliable environment with various features like access control, private registry for sharing modules, policy controls and others.

Create terraform cloud account:

OVERVIEW OF SENTINEL:

Sentinel is a policy-as-code framework integrated with the HashiCorp Enterprise products.

It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

NOTE: sentinel policies are paid features.

terraform plan -----sentinel checks --→ terraform apply

High level structure :

Policy ----------→ Policy sets --------→ workspace

(block ec2 without tags)

REMOTE BACKENDS:

Terraform-cloud backend operation types

The remote backend stores terraform state an d may be used to run operations in terraform cloud.

Terraform cloud also be used with local operations, in which case only state is stored in the terraform cloud backend.

Backends operations = local & remote

REMOTE OPERATIONS: When using full remote operations like terraform plan or terraform apply can be executed in terraform cloud's run environment, with log output streaming to the local terminal.

COMMANDS: all these commands are running in terraform cloud.

terraform login (provide the token) in a current directory.

terraform init

terraform plan

terraform apply

```
provider "aws" {
  region     = "us-east-1"
  access_key = "enter_acces_key"
  secret_key = "enter_secret_key"
}
```

```
terraform {
  cloud {
    organization = "gourish-org"

    workspaces {
      name = "remote-operation"
```

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  path = "/system/"
}
```

Iam.tf

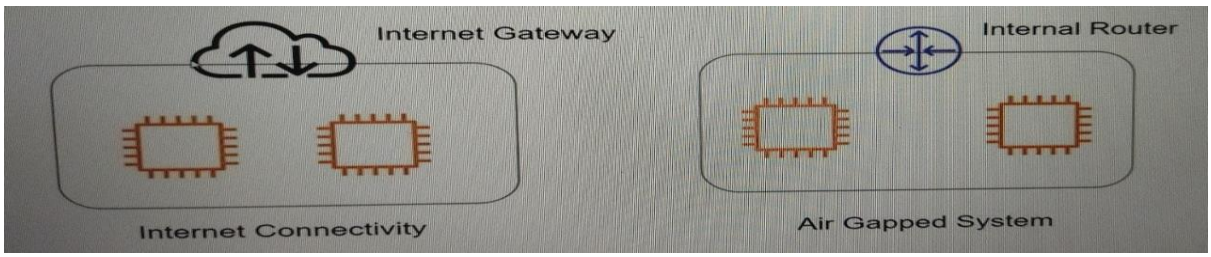
```
    }
  }
}
```

Remote-backend.tf

AIR GAPPED ENVIRONMENT:

An air gap is a network security measure employed to ensure that a secure computer network is physically isolated from unsecured networks, such as public internet.

Air gapped environments are used in many areas like military, stock exchange and industrial control system etc.



TERRAFORM ENTERPRISE INSTALLATION METHODS:

Terraform enterprise installs using either an online or air gapped method and as the name infer, one requires internet connectivity, the other does not.

****