Terraform

What is Terraform?

* Tool that allows you to automate the provisioning and management of infrastructure on any cloud platform or service using declarative configuration files.

* Terraform is Infrastructure as a code

* To automate the resources of the cloud we use the terraform.

* Basically, we can do a process in the cloud, using 3 ways, consider them manual or automation.

1) Start and maintain resources using console

a) In this approach, maintenance overhead is high, as we need to assign all the properties to the resources manually.

2) Use Cloud CLI, and start and maintain resources (Need to execute commands in Cloud CLI)

3) We can use the help of SDKs of other programming languages like python, which can execute azure from its scripts, thus we can automate.

4) We can use ARM templates, to automate the resources. ARM => Azure resource manager (Service which is offered by azure, for managing resources)

If we try to maintain, these resources manually, we need to configure all manually, we need to configure the servers, networks, storage etc, and if any new user want to replicate this virtual machine, it is difficult to understand the configurations, because as said these are done manually, there may or may not be documentation. So, when coming to ansible, we will be having yaml files and for terraform we will have HCL and state files or JSON files, so it is easy to figure out configurations, and it is easy to re-use for other VM's. If any other team, want to create a resource with same configuration, they can simply use already existing configuration files, instead of creating them manually.

We can also use ARM's (Azure resource manager) to structure the resource management process, but ARM is only confined to Azure, so if we want something, which is cross platform supported and useful to create and manage the resources and develop the infrastructure, we can use Terraform. Terraform supports many cloud providers, like Amazon Web Services, IBM Cloud, Google Cloud Platform, Linode , Microsoft Azure , Oracle Cloud Infrastructure , or VMware vSphere as well as open Stack.

Terraform basically supports high-level configuration language called Hashicorp Configuration language, but it also supports JSON as a language through we can accept the requests.

Terraform supports AWS, IBM Cloud (Bluemix), GCP, Linode , Azure , Oracle Cloud Infrastructure , VMWare vSphere , Open Stack.

Ansible is configuration management tool, terraform is infrastructure management as a code.

Ansible is mainly used for configuration management and automation. It helps you to install, update and configure software on your servers or other devices. It also lets you automate tasks like provisioning, deployment and security. Ansible uses simple YAML syntax to write playbooks that define the desired state of your infrastructure. Ansible is agentless, which means you don't need to install any software on the target systems to manage them. Ansible is imperative, which means you specify the steps to achieve the desired state.

Terraform is mainly used for infrastructure as code and orchestration. It helps you to create, modify and destroy cloud resources like servers, networks, storage, etc. It also lets you manage dependencies and logical relationships between resources.

Terraform uses HCL language to write configuration files that define the desired state of your infrastructure.

Terraform is agent-based, we need to install a software called Terraform CLI on system to interact with cloud providers.

Terraform uses a declarative approach, meaning you define the desired state of your infrastructure, and terraform figures out the steps to achieve that state. This is different from Ansible, which is more procedural, requiring you to define the steps to reach the desired state.

Terraform is declarative and figures out the steps to reach the desired state on its own, while Ansible is procedural and requires you to define the steps.

Both Terraform and Ansible have their own strengths and are often used together in the industry to manage different aspects of infrastructure.

Terraform is excellent for provisioning and managing the state of infrastructure, particularly in a cloud environment. It’s often used to set up the infrastructure from scratch.

Ansible, on the other hand, excels at configuration management and application deployment. Once the infrastructure is set up (possibly by Terraform), Ansible can ensure that all systems are configured correctly and maintain this state over time.

Terraform have state files, which have the current status information of their already created resources, and configuration files, which have the entire build details, which is used to build the infrastructure.

Main advantage of terraform comes at place of destroying the resources, if we try to delete a resource like VM, all the underlying resources won't be deleted automatically, but terraform deletes the resource and its underlying dependent resources. And even while creating a resource, it helps us to create network groups, security groups and the rest.

Resource Groups:

In general, to create a resource group, we need to create a subscription. Resource Group is created in the context of the subscription.

Usage of Resource group: In general, in any company, we will have set of resources depending on the type of the server, we are on, either it is dev, prod, test etc ..., so for every type of server, we can create a resource group and add the required resources in respective resource groups. So, whenever we want to deploy an infrastructure on any server, we can directly use this resource group and deploy it directly as we are having some template of resources.

Resource Group is created for an application for every environment.

Resource Group is a container that holds related resources for an Azure solution. It stores metadata about the resources. It also helps us in sharing access to other team members using RBAC (Role-Based Access Control). So only defined users can access the resources of the resource groups. Resource Groups can contain storage, database cache and API's etc as its resources. We have to select subscription while creating the resource group,

and we can also create the region of where the resource group going to create. After creating the resource group, we can add individual resources into the group.

If we want to create a resource group through terraform, basically, we need details of subscription, resource group name and the region of the cloud provider where we want to create the resource group. With these details we can create a resource group through Azure GUI & Terraform & Azure CLI (Cloud Shell), VSCode (with help of (Azure Terraform - extension), but we need the credentials of the cloud), Windows PowerShell.

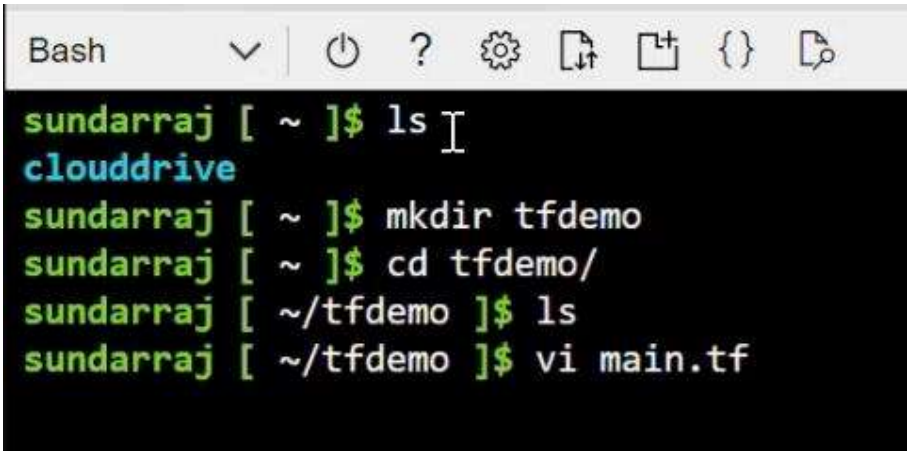
How to check version of terraform:

terraform --version

To check list of terraform commands available:

Terraform

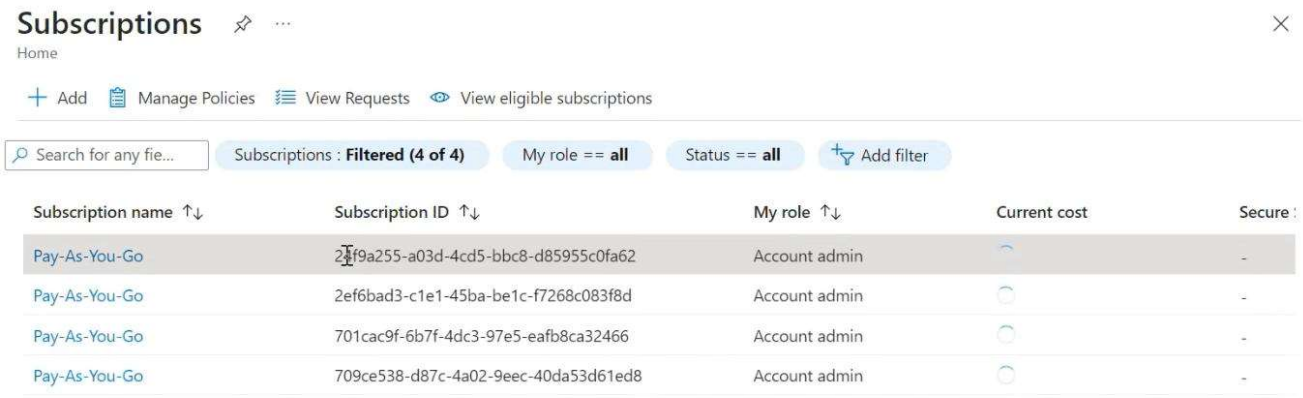
Now as we know that , we want to reuse the config files to create resources , which should be same as the other infrastructure , here the main benefit of terraform is to design an infrastructure in an automated way. And that config file , we must store somewhere in the system , similar to this ....



And , in the configuration file , we need to mention the cloud service provider that we wanted to automate , we are using azure as the cloud provider for us , so , using azurerm , which is the keyword for the azure service provider to mention in terraform config files. And we have a subtag called features , where we can mention our subscription id's and security tokens etc , for now , we are not demonstrating them , so leaving that as empty.

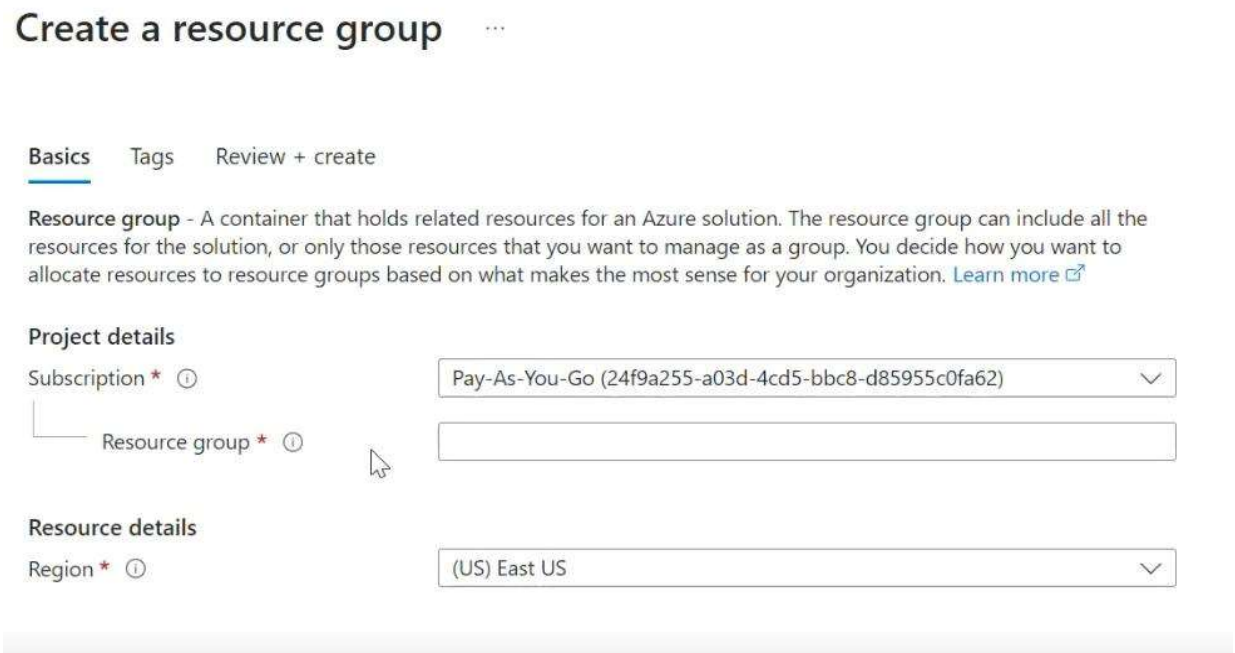
```
provider "azurerm" {
features {}
}
```

Terraform is basically used for automating the infrastructure. As said subscription id's will be present under subscriptions page , as follows ,



We can also pass configuration information like path , we can pass through this features section.

Coming to creating a resource group using this cloud shell scripts , we need to be sure , of what all attributes we use to create a resource group , while we do manual creation of resource.



So , we can even do this by using configuration files. By passing the subscription id, resource group , and region.

```
sundarraj [ ~ ]$ ls
clouddrive
sundarraj [ ~ ]$ mkdir tfdemo
sundarraj [ ~ ]$ cd tfdemo/
sundarraj [ ~/tfdemo ]$ ls
sundarraj [ ~/tfdemo ]$ vi main.tf
sundarraj [ ~/tfdemo ]$ cat main.tf
provider "azurerm" {
features {}
}
resource "azurerm_resource_group" "niqrg" {
name="myniqrg"
location="eastus"
}


sundarraj [ ~/tfdemo ]$ terraform init
```

Here , in the above , image as you see , that we are using configuration script to create a resource group , by passing the name of the resource group that we want to create and the region of the resource group that we want to create.

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.82.0...
- Installed hashicorp/azurerm v3.82.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
sundarraj [ ~/tfdemo ]$
```

Here , you can see that , after executing the above command , we will first start the backend installation process and install the necessary plugins , which are needed to create a aazure resource through configuration files / terraform files. Now we will create a lock file , which

have the details of what resources we installed through this execution , so we can use this script anywhere to ensure terraform to install the same infrastructure.

In the above configuration script , we can see something like “azurerm” , and where while declaring the resource groups that we need to install in the infrastructure , we need to define those resources as “azurerm_<resource_name>” and the resource group name and it’s location needs to be declared down to this.

```
sundarraj [ ~/tfdemo ]$ terraform validate
Success! The configuration is valid.

sundarraj [ ~/tfdemo ]$
```

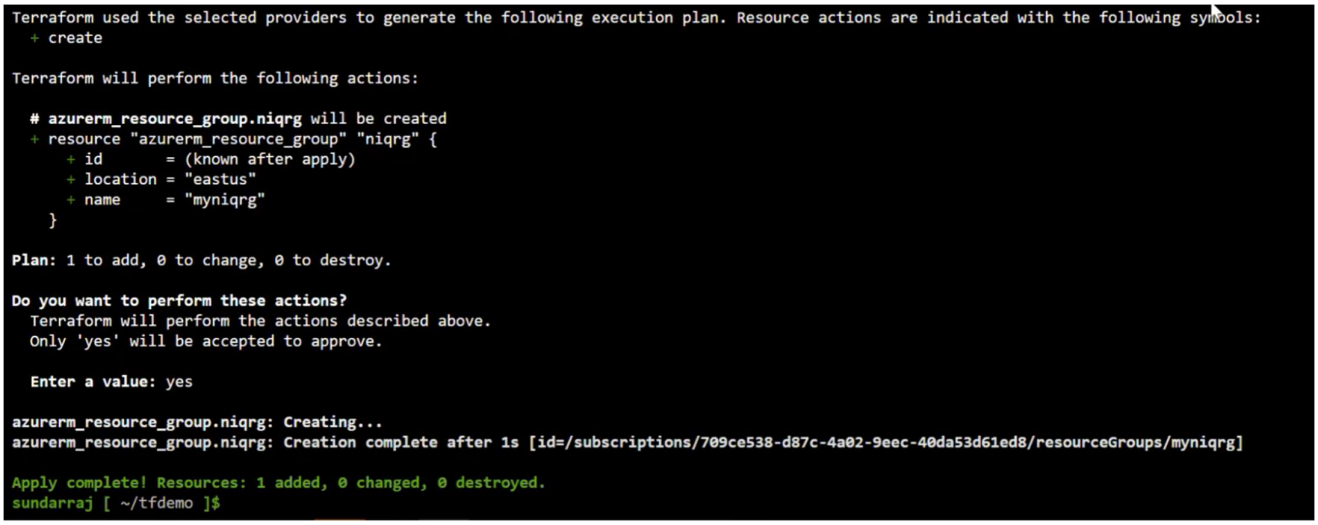
To validate the terraform configuration file , that we written , we need to use terraform validate ,and to download all the plugins that are need to run the terraform script or to create resources with help of terraform script , we need to execute terraform script. So , for that , we need to download some plugins , which are helpful to create a infrastructure , for that we need to use ‘terraform init’.

```
sundarraj [ ~/tfdemo ]$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.niqrg will be created
  + resource "azurerm_resource_group" "niqrg" {
      + id       = (known after apply)
      + location = "eastus"
      + name     = "myniqrg"
    }

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

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply"
now.
```

If an unknown person come to the team and understand what current workspace lock file and the configuration does , he can use the teraform plan , which give all the details of the infrastructure that the terraform gonna create with the request by the user.

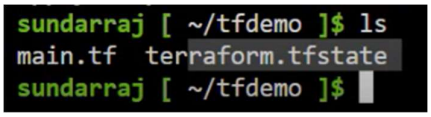
```
sundarraj [ ~/tfdemo ]$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.niqrg will be created
  + resource "azurerm_resource_group" "niqrg" {
      + id       = (known after apply)
      + location = "eastus"
      + name     = "myniqrg"
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Using terraform apply , we can apply the configuration to the cloud resource , we can use this command to change the configuration and re apply and even to destroy a resource or a configuration.



You can see the subscription id , in here on which we created all the resources on.

terraform apply command will create the following state file



Terraform state file looks like this

Terraform state file consists the details of version number , a unique id for state file called ‘lineage’ and the resources that we created.

In here , we see the subscription details where our resources are created and the region and some basic details of them.

In Ansible , we won't be having any state related information related to the resources of what we are having , where as terraform have this statefile , which describes the state of teh resource.

For example , if we have made some changes in the resource or in the infrastructure manually through the cloud service provider GUI , those changes won't be reflected on the local machines' terraform statefiles , so to keep them align with the remote server , we need to use the terraform refresh.

If we don't want any confirmation from the user to apply the modified plan , we can directly apply "terraform apply –auto-approve" , but it will ask you for sure about , do you want to modify the configuration or delete a resource.

Terraform refresh typically refresh the state file , but not the configuration file.

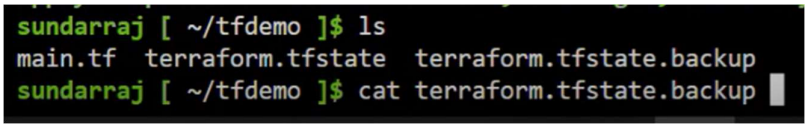
For installing the basic plugins needed for terraform , we use "terraform init"

For validating the syntax of terraform configuration file , we will perform the "terraform validate"

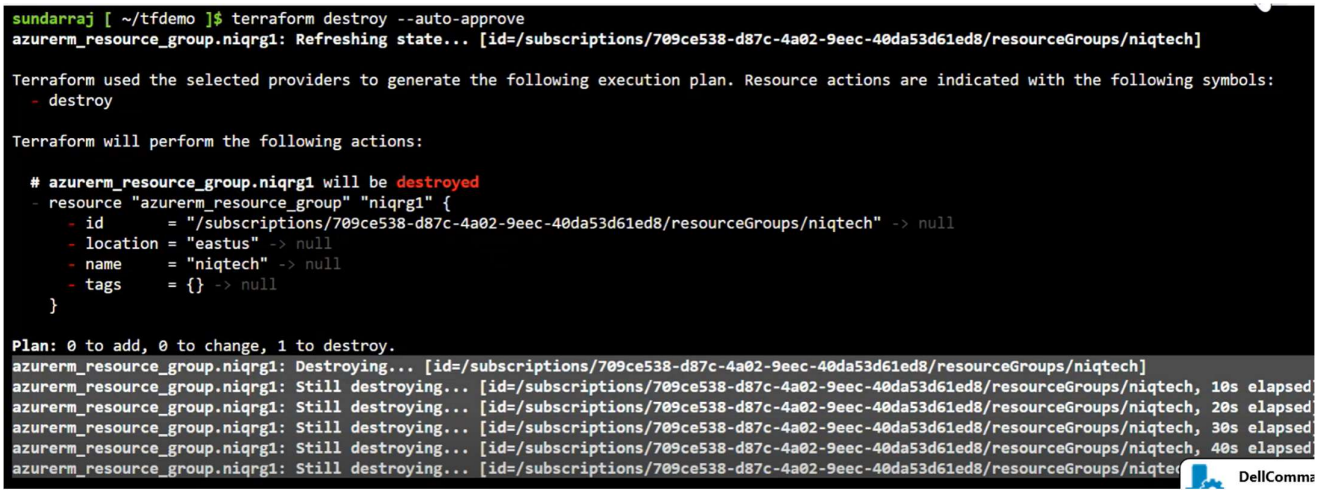
What ever resources that I want to automate or get created on the cloud by terraform configuration files , Can be checked by this command "terraform plan" and that plan will be executed on the cloud by using "terraform apply" , but this prompts us for confirmation , so to skip that , we can use "terraform apply –auto-approve"

“terraform apply –auto-apporve” applies the configuration on to the cloud , without any approval or confirmation , It is similar to force apply. When we made a change in the same terraform configuration file , then old resource will be destroyed and new one will be created and state file’s would change. But , if we create any other terraform state file , and re execute that , it will result in creating another statefile with another resource.

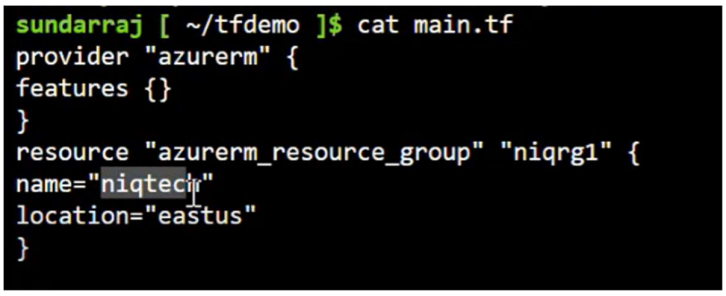
Whenever , we try to destroy or modify a resource by terraform through configuration fles , terraform automatically creates a backup for the state file which will be named as “terraform.tfstate.backup”



If we want to delete the resource group , that we created using terraform configuration files , then , we can simply use “terraform destroy –auto-approve” , where it deletes the resource without any approval from the cloud.



Previously , I had a doubt , where this is that



Here , you can see “niqrg1” which is the resource group block name , which we can use as internal referrence for terraform to recognise this resource , where as “niqtech” is the original name , which we name the resource as , which will be also same in the cloud resource.

While creating any infrastructure thorugh terraform files , there are two things in common everytime , resource provider block which is azurerm in our case and resource group block which is either VM or resource group or any other resource that we want to use.

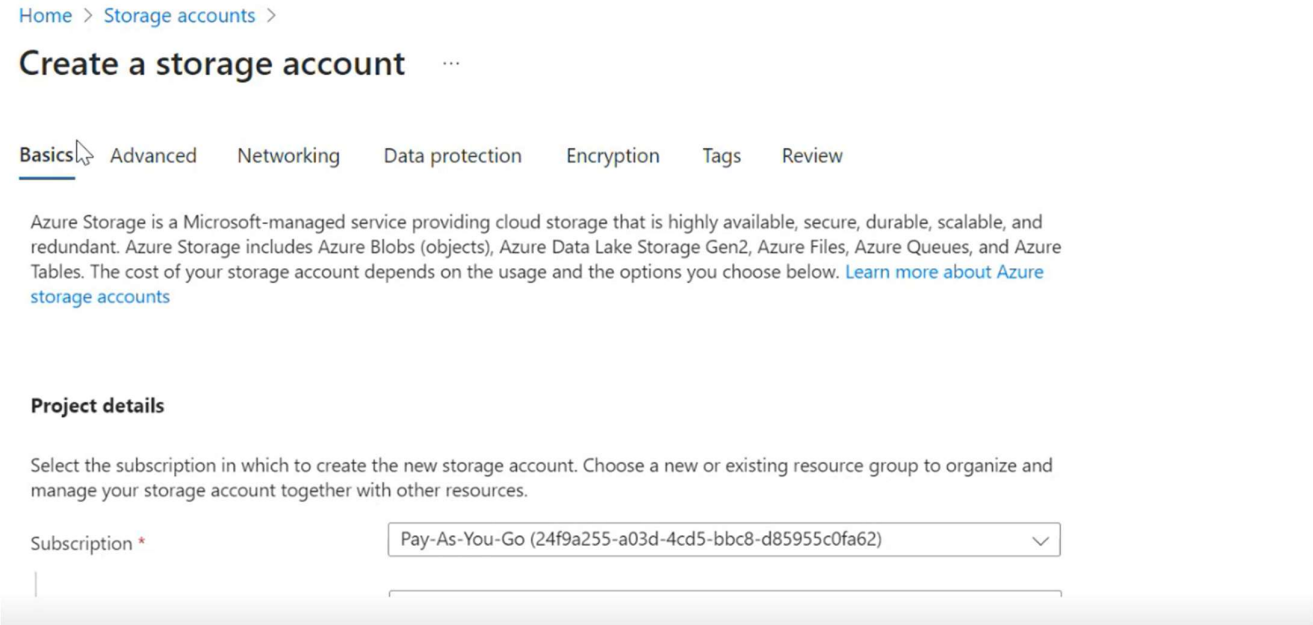
If we want to create virtual machine using resource block , we need to use "azurerm_resource_group" resource group block to create one.

You can refer this document , for creating an virtual machine https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/virtual_machine

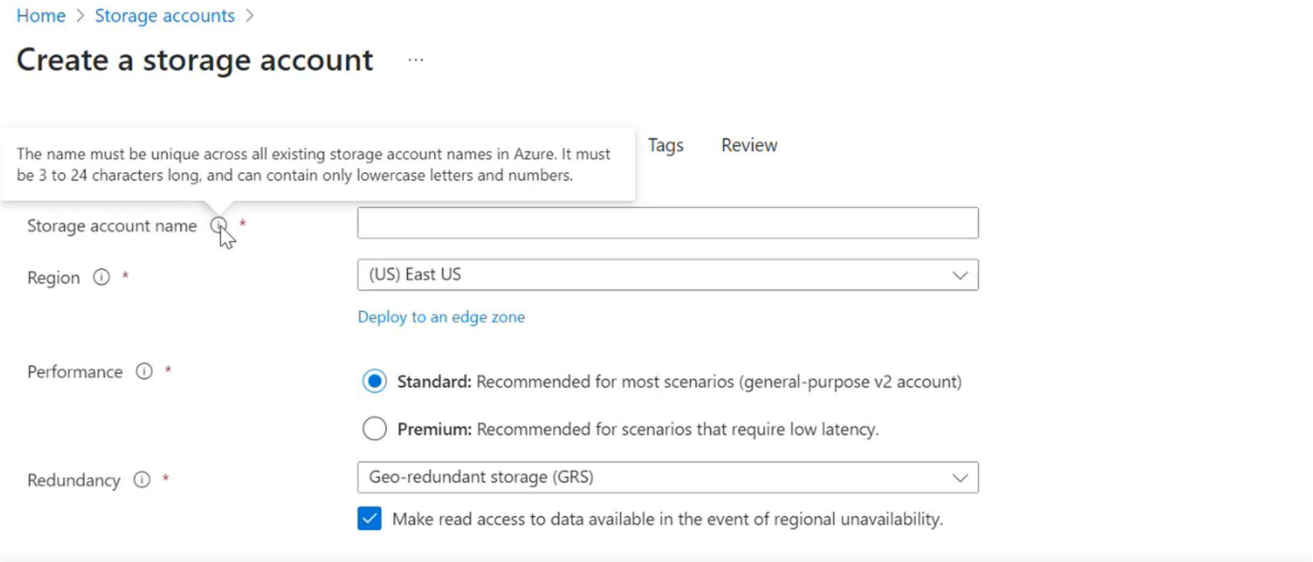
Creating infrastructure using terraform is easy , when we are having a clear idea on how the resource group or resources are getting created and what all options that we need to declare for those , when we create them manualy on the azure web application.

Check the above link , there we are creating a virtual machine , and in ther , first we are giving the basic details like name and location of the resource group , and later we need to create virtual network , subnet , network interface , ip configuration , virtual machine size , whether it must be a large one or is it a dynamic type of allocation or dedicated vm or a common vm which will be allocated for us based on the usage. And for the virtual machine , we need a base image on which we need to build the machine. OS disk information , some basic details like computer username , admin_username and admin_password by applying all these properties to a terraform file , we can automate it's creation irrespective of manual selection method , we can use the configuration file and use for some other team if needed to develop the same environment. If we are going to choose , private_iip_address_allocation to static , then we must also choose what IP that we wanted to point , if it is dynamic , it will be changing everytime when we establish a new connection.

The main usage of terraform is , we would have details of all the resources , which we created , and we can modify them and delete them in an organised manner , but when we do the same approch using the service provider UI , we may forget to delete some resources , which will increase our recurring cost , because , we have left a resource behind , we need to pay for that , even we don't use it.



While we want to create a storage account , we must ensure that , the location of the storage account must be same as the subscripton under which we are giving, because , storage account is something confined to the region , but not the global.



The name of the storage account must be unique across the world under a cloud providor. It means the storage account name is unique acrss the globe under that service provider.

azurerm_storage_account

How to create storage account using terraform configuration files , is mentioned in the following link : https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/data-sources/storage_account

To ensure the security of the storage account , we can choose multiple storage types , they are

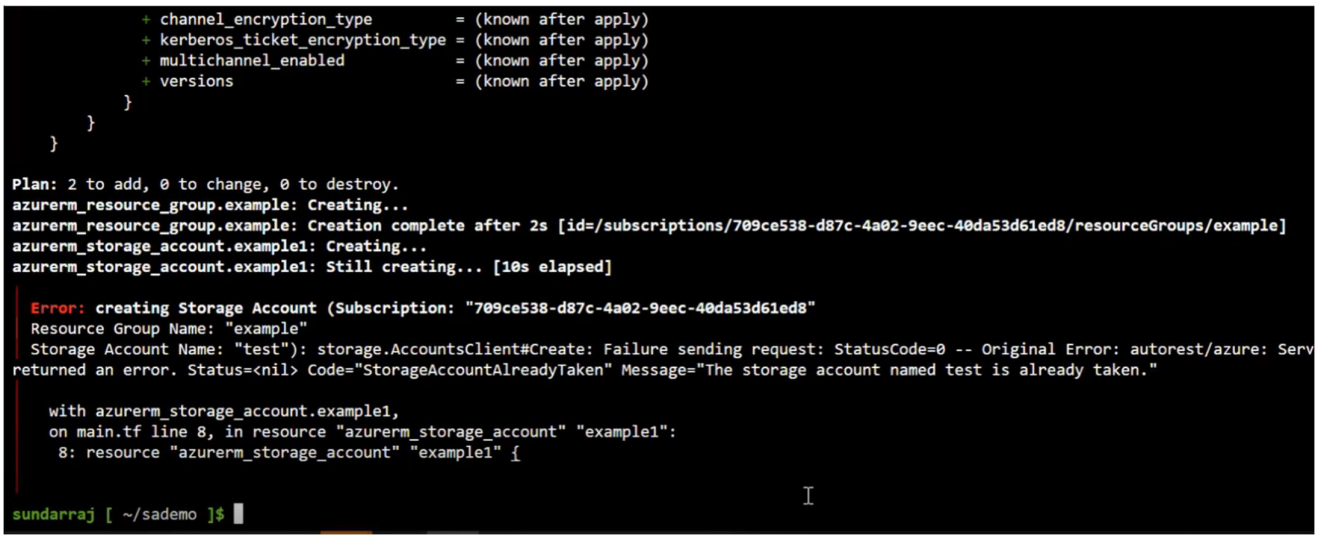
GRS => Multiple Regions (Data is copied across multiple regions for data backup)

LRS => Single Region Multiple AZ (Data centers) (Data is copied across different data centers of same region)

GZRS => Multiple Region as well as Multiple AZ

ZRS => Multiple AZ (Data centers)

When we try to apply , a configuration for creating the storage account , with already globally existing name , we will get an error at the stage of “terraform apply” , not before , as we are not having a unique storage account name for the resource.



How to create virtual networks in Microsoft azure using terraform?

Virtual networks are to securely connect our Azure resources to each other. Connect your virtual network to your on-premises network using an Azure VPN Gateway or Express route.

VIRTUAL NETWORKS:

A virtual network is a way to create an isolated network in Azure that allows you to securely connect and communicate with your cloud resources. You can also use virtual networks to connect to other virtual networks or to on-premises networks1.

Some of the common use cases for virtual networks using Azure are:

Creating topology and connectivity: You can use virtual networks to create different network topologies based on your needs, such as mesh, hub and spoke, or hub and spoke with direct connectivity2. You can also use virtual network peering to connect virtual networks across regions or subscriptions3.

Securing network traffic: You can use network security groups, application security groups, and virtual network service endpoints to filter and restrict network traffic between your virtual machines and other Azure services4.

Routing network traffic: You can use route tables to customize how network traffic is routed within or between your virtual networks5.

Creating IP addresses: You can use public IP addresses, public IP address prefixes, and IPv6 addresses to assign IP addresses to your virtual network resources.

One real time example of using virtual networks in Azure is to deploy a web application with a three-tier architecture. You can create a virtual network with three subnets: one for the web tier, one for the application tier, and one for the data tier. You can then deploy your web servers, application servers, and database servers in the respective subnets. You can also use network security groups to control the inbound and outbound traffic for each subnet, and use service endpoints to secure the communication between your web servers and Azure Storage. This way, you can create a secure and scalable web application in Azure.

As discussed , we can have multiple subnets in a single virtual network and for that , we need to declare the set of IP addresses that we can use to assign to the set of virtual machines which are under that. And under that we can create subnets IP address range.

We will check on how to create virtual network through terraform files.

Resource group should be the common for all the services that we want to automate.

Only in Azure , we have the concept of resource group, in AWS , we don't have that.

In terraform files , under features block , we would have authentication information like secret tokens to login , and if we have multiple subscription id's and configuration information , we can store them in here.

```
provider "azurerm" {
features {}
}
resource "azurerm_resource_group" "example" {
  name     = "vnetresources"
  location = "West Europe"
}

resource "azurerm_network_security_group" "example" {
  name                = "example-security-group"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
}

resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  address_space       = ["10.0.0.0/16"]
  dns_servers         = ["10.0.0.4", "10.0.0.5"]

  subnet {
    name           = "subnet1"
```

```
resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  address_space       = ["10.0.0.0/16"]
  dns_servers         = ["10.0.0.4", "10.0.0.5"]

  subnet {
    name           = "subnet1"
    address_prefix = "10.0.1.0/24"
  }

  subnet {
    name           = "subnet2"
    address_prefix = "10.0.2.0/24"
    security_group = azurerm_network_security_group.example.id
  }

  tags = {
    environment = "Production"
```

First in here , for creating a virtual network securely , we can create a security group, where we have to provide basic details like name , location and resource_group_name , and where as next , we are creating a virtual nework with with an address space and two subnets under that.

Then , to execute this , we need to follow casual procedure

1) terraform init
    a. terraform validate
    b. terraform plan

2) terraform apply –auto-approve

if we want to destroy the resource

1) terraform destroy –auto-approve

After successful execution , we can see the virtual network resource will be created in the GUI.

But as you see , we have done so many hardcoding's of ip addresses , terraform configuration files , so this makes us difficult in managing the configuration files. So , to escape this overhead , we have the concept of terraform variables which increases the reusability of variables , and helps us in changing the configuration files a lot of easier.

Terraform variables:

1) Input variables (User gives input to terraform files)

2) Output variables (Terraform files generate output details)

```
variable "resource_group_name" {
default="niqrg"
type=string
description="this is niq resource group"
}

variable "resource_group_location" {
default="eastus"
type=string
description="this is niq resource group location"
}
```

Here , this is the syntax of declaration of variables , which we can use in the configuration files , here you can see two variable blocks , and why is that is , the type of the variable block is declared as string , so only one value will be accepted for a resource variable block , and where as we can declare multiple values in a same variable block , when the type of the resource block is declared as list or we can even store map

type in a variable , we also see two variables which stores default value for a variable and description for variable resource block.

How to use these variables in configuation files:

```
var.resource_group_name
var.resource_group_location
```

This helps us in , whenever anyone wants to change the properties in the configuration files , they would change the respective variable files rather than the configuration files , because there won't be anything in the configuration files , no hardcoding will be present in there.

And , if we change any properties like resource group name or resource group location details in these variable files , then the related resources which are created earlier using these variable files , probably these will be present in the same folder as the resources , will be getting deleted and re creating them with modifying properties.

```
variable "resource_group_name" {
default="niqappservice"
typ=string
description="this is rg for app service"
}

variable "resource_group_location" {
default="eastus"
type=string
description"this is region for the rg"
}

variable "app_service_plan" {
default="niqappservice"
type=string
description="this is app servcie plan"
}
```

```
variable "resource_group_name" {
default="niqappservice"
typ=string
description="this is rg for app service"
}

variable "resource_group_location" {
default="eastus"
type=string
description="this is region for the rg"
}

variable "app_service_plan" {
default="niqappservice"
type=string
description="this is app servcie plan"
}

variable "app_service_name" {
default="niqdemo"
type=string
description="name of the app service"
}
-- INSERT --
```

Here , you can see in the above image , we are declaring variables for resource group name and location , and app service name and plan name , here a point to note that app service name shold be globally unique as the storage account name.

App services basically provides Platform as a service , there are many details , we can send the details through variables for configuration files. App service plan is noting but the pricing plan for the app service that we are creating.

Store these variables into a single terraform file. Name that as var.tf

```
provider "azurerm" {
features{}
}
resource "azurerm_resource_group" "example" {
  name     = var.resource_group_name
  location = var.resource_group_location
}
resource "azurerm_app_service_plan" "example" {
  name                =var.app_service_plan_name
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  sku {
    tier = "Standard"
    size = "S1"
  }
}
```

```
resource "azurerm_app_service" "example" {
  name                = var.app_service_name
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  app_service_plan_id = azurerm_app_service_plan.example.id

  site_config {
    dotnet_framework_version = "v4.0"
    scm_type                 = "LocalGit"
  }

  app_settings = {
    "SOME_KEY" = "some-value"
  }

  connection_string {
    name  = "Database"
    type  = "SQLServer"
```

Later as usual , we can initialize terraform using “terraform init” , now the basic terraform plugins and dependencies will be downloaded. “terraform validate” will validate the terraform files. “terraform plan” will plan 1 resource group , 1 app service plan , 1 app service.

terraform apply –auto-approve will be creating the above resources in my subscription.

Earlier , while we are writing the dynamic config files , we are using variales instead of hardcoding , but in the variable creation level , we are yet hardcoding the resource values in there , so if we use this pattern , we must create multiple variable files for multiple users. So , to prevent , even this hardcoding , we can create another file , which only consists the resource values for the variables , we need to surely name that as

terraform.tfvars

And that must be looking like something similar to this

```
resource_group_name="niqappservice"
resource_group_location="eastus"
app_service_plan_name="niqappservice"
app_service_name="niqdemo"
~
~
~
~
~
~
~
~
```

And as seen in the above image , we have created the default variables for resources through another file , so , we no need to again mention in the resource variable declarations.

```
variable "resource_group_name" {
    type        = string
  description = "this is rg for app service"
}

variable "resource_group_location" {
    type        = string
  description = "this is region for the rg"
}

variable "app_service_plan_name" {
    type        = string
  description = "this is app servcie plan"
}

variable "app_service_name" {
  type        = string
  description = "name of the app service"
}
```

For now , we have created the following structure , we have four files in total ,

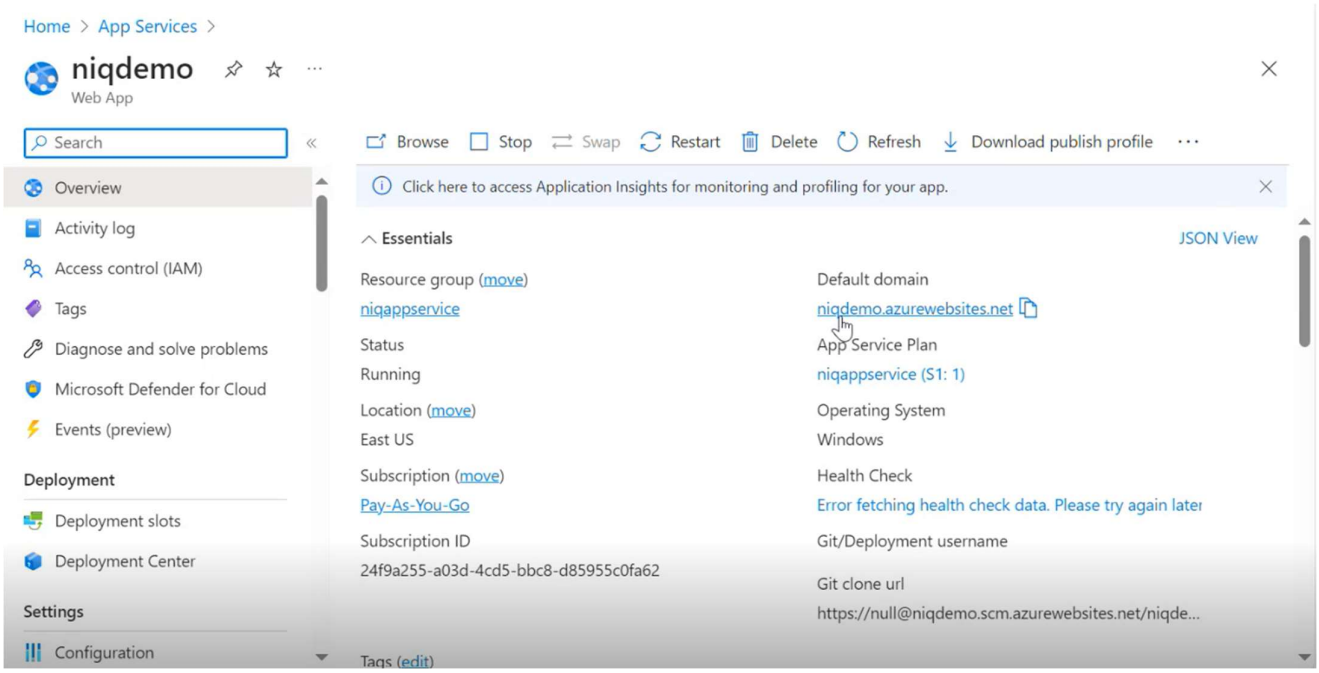
main.tf => main configuration file , for terraform

var.tf => This is the file main.tf depends on external variables declaration

terraform.tfvars => this is the file , where we declare all the default resource variables values into a single file , this file is referred by var.tf

```
sundarraj [ ~/vnet/demovar ]$ ls
main.tf  terraform.tfstate  terraform.tfvars  var.tf
sundarraj [ ~/vnet/demovar ]$ cat terraform.tfvars
resource_group_name="niqappservice"
resource_group_location="eastus"
app_service_plan_name="niqappservice"
app_service_name="niqdemo"
```

After creating resources , we want to get some details , to deploy an application , consider we application , so to get that end point details , instead of going to UI and check the domain name endpoint under app services , we can get them through the terminal through terraform , so we can deploy the application through the terminal itself , by getting the necessary details through terraform code and use them for deployment of web application.



We can get those details through output files in terraform.

https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/app_service

```
output "webapp_url" {
value=azurerm_app_service.example.default_site_hostname
}
output "webapp_ips" {
value=azurerm_app_service.example.outbound_ip_addresses
}
```

We should name the terraform outpout file as 'output.tf' , where we can mention some output variables , with the following syntax as in the above image , azure_app_service is the resource name that we created , example is the resource block name , we created in local , and the right side of example block is the property of that resource , for instance , we are using two of them , one is default_site_hostname and outbound_ip_address , there are many like this , which you can found in the above mentioned link.

Later , we can validate and apply the services through terraform commands

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

webapp_ips = "20.241.146.75,20.241.146.91,20.241.146.140,20.241.146.146,20.241.146.155,20.241.146.176,20.119.16.28"
webapp_url = "niqdemo.azurewebsites.net"
```

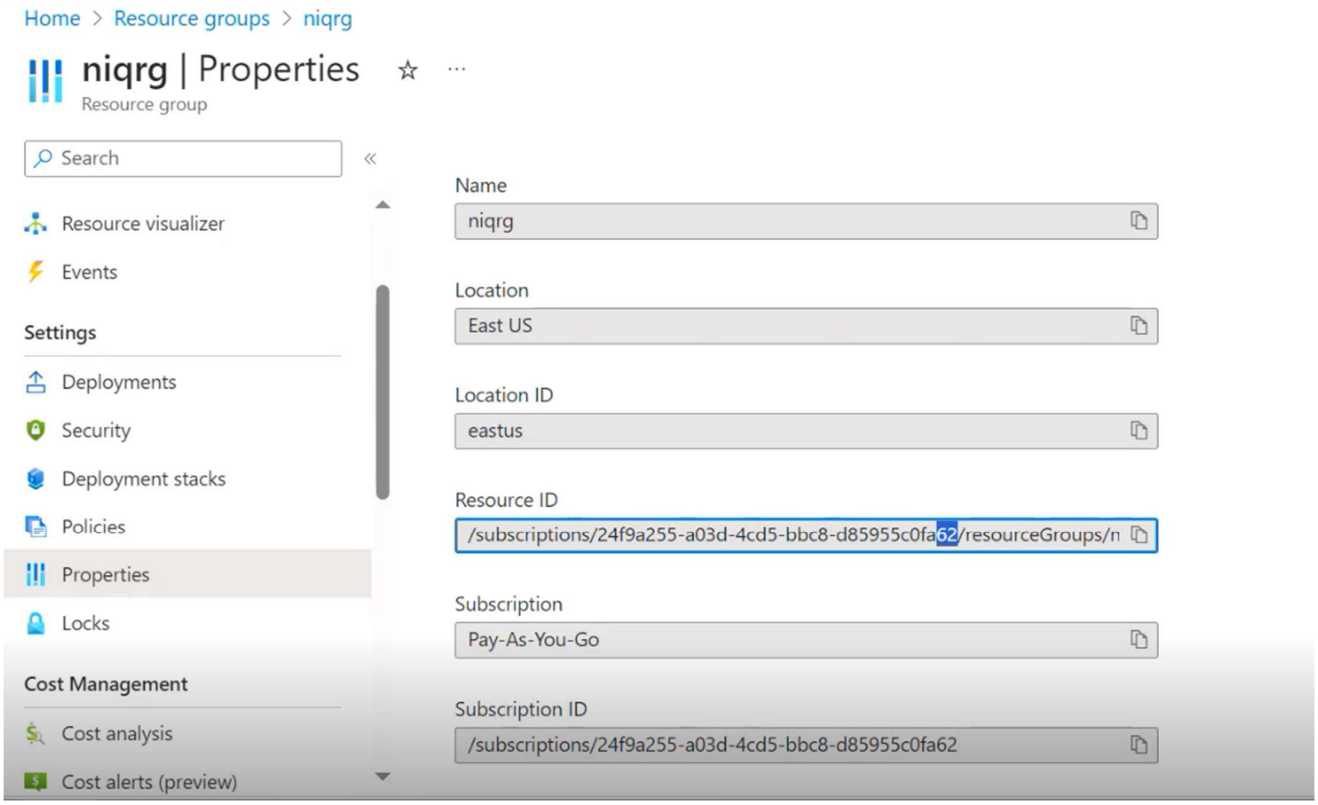
Here , we can see , after validating the terraform files , we had through 'terraform validate' , we apply the services through 'terraform apply' , we would get output of what we written in output.tf , in here , as you can see , webapp_ips & webapp_url output in the image above.

To destroy all the resources that we created through terraform , we can use a single command to destroy , that is 'terraform destroy –auto-approve'

Sometimes , there will be a situation , where we have 100 VM's where 50 VM's are created by the terraform scripts and other 50 were created through the Azure UI. So, if we want to integrate the resources which are already created through GUI to terraform , we must import those resources to terraform. So , for that , first we need to create a resource file (main.tf) which have the exact information about the resource that we have created through GUI , then when we try to do 'terraform apply' , we would get an error , because , we can't create another resource , when it already exists in the Azure cloud remotely.

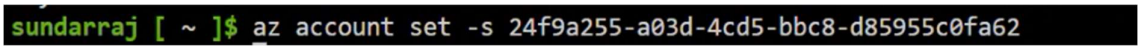
```
provider "azurerm" {
features {}
}
resource "azurerm_resource_group" "myrg" {
name="niqrg"
location="eastus"
}

resource "azurerm_storage_account" "example" {
  name                     = "niqdemo123"
  resource_group_name      = azurerm_resource_group.myrg.name
  location                 = azurerm_resource_group.myrg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  }
```

So , instead we have to import those resources from remote to the local , first for that , we want to check the subscription id of the resource which is present on the remote , as in the below picture.



To get all the subscription details of what we are having in azure , we can check that using terraform using ‘az account list -o table’ , where we can see the name , cloud name , subscription id , tenant id , state.

If we want to select any specific subscription , and get into that environment , we need to use the following command.



By using ‘terraform import’ , we can import only one resource at a time.

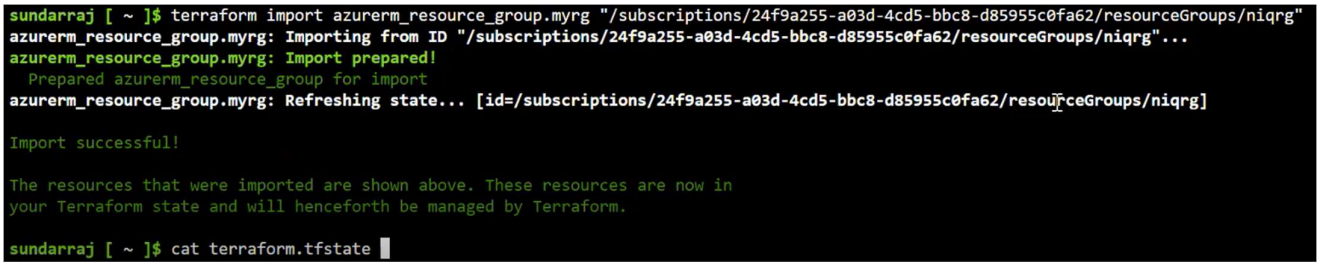
For example,

Import resource group first:

terraform import <resource_group_name>.<resource_block_name> “<resource_id>”

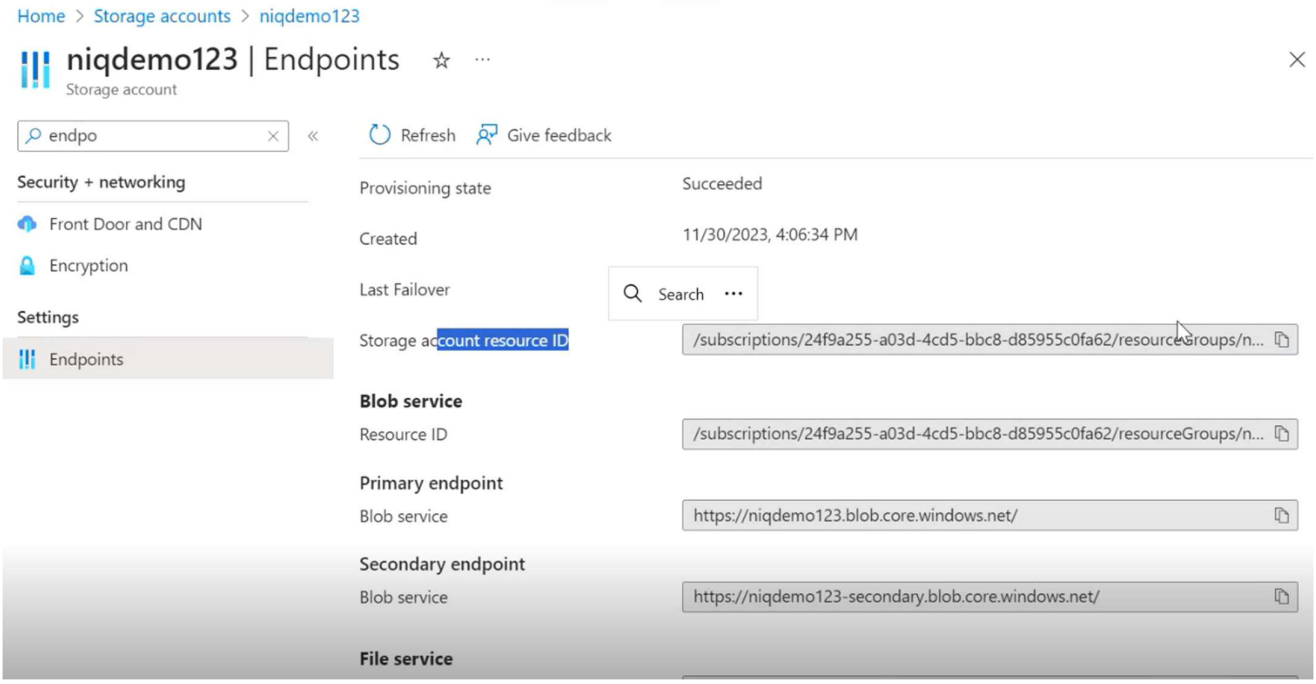
By using that , we can import the resource from the remote, after importing , we will be having one , state file.

The result , will be looking something like this ,



And we will be having a state file for that.

Storage accounts end point will be present somewhere here as in the image. The subscription Id for storage accounts will be present under endpoints, in the name of storage account resource ID.



We can import storage account from remote to the local by using the following method,

<< terraform method 2 >>

We will be having the following files in local , where we will be having two state files , where one which is created earlier as state file while we got when we used 'terraform import' for the resource group , and the same file is overwritten and stored as backup , when we executed the second import using terraform import for the storage account , in this manner , we will be having two versions of state file.

If we are creating a new resource by using the configuration file , then we need to use the apply , else if we are trying to write a configuration file for the resource which is already exists , we must write the configuration file , but we shouldn't apply the configuration file using terraform through 'terraform apply' , we need to import the resource from the remote to local system , by using the subscription ID , as discussed.

To format terraform files , we can use 'terraform fmt'

If we want to open , terraform console then use the command 'terraform console' , you will be opened a new shell.

How to maintain same infrastructure for different environment=> Terraform workspaces

How to recreate the resource => Terraform Taint and Retaint

Terraform taints:

Terraform taints are a way of marking a resources as degraded or damaged , so that Terraform destroy and recreate it in the next apply operation , because , if we didn't do this operation , and just perform terraform apply' command , we would get an error of resource already exists. This can be useful when a resource is in an undesirable or unexpected state , but it's configuration hasn't changed , but we want to re-create the resource.

For example , if a virtual machine becomes corrupted or infected by malware , you can use terraform taint to force Terraform to replace that resource with a new one.

However , the terraform taint command is deprecated since Terrraform version 0.15.2 , instead , we can use the -replace option with terraform apply to achieve the same behaviour. This option allows you to see the effects of replacing a resource in the terraform plan , before we take any action. This can be used when we want to replace the infrastructure due to any errors in the infra level for the resources.

```
sundarraj [ ~/demo ]$ terraform state list
azurerm_linux_virtual_machine.example
azurerm_network_interface.example
azurerm_resource_group.example
azurerm_subnet.example
azurerm_virtual_network.example
sundarraj [ ~/demo ]$
```

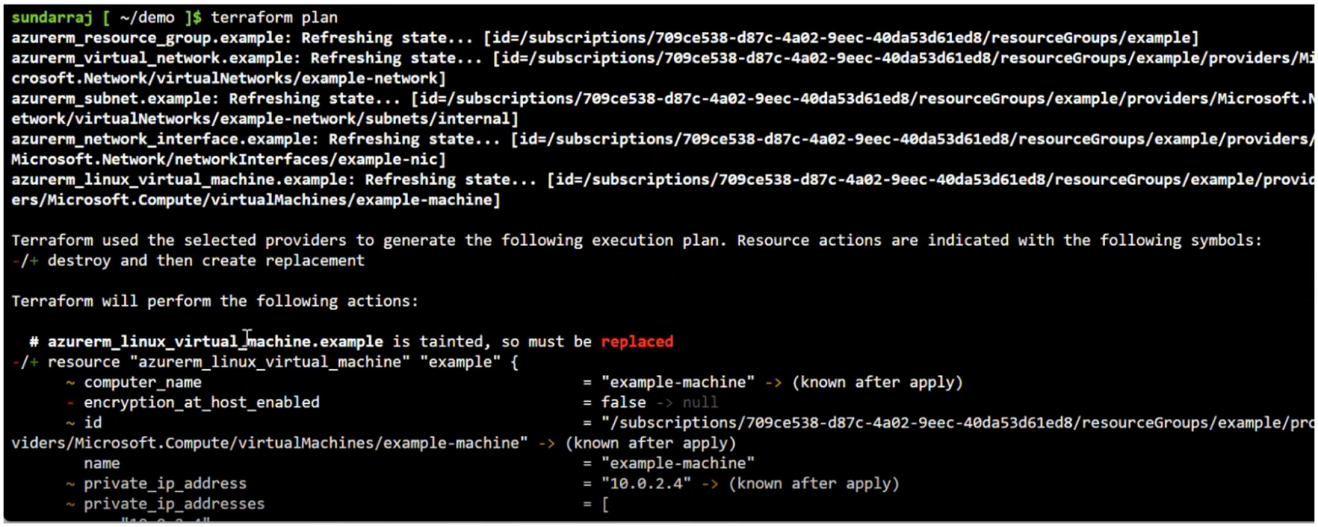
We can check the information about the resources and what resource statefiles we have on our system. Using "terraform state list"

And , later , if we want to re-create already existing resource through CLI , we have to taint that resource , it mean , we have to tell azure that this particular resource is out-dated , by using terraform taint <<resource_state_file>>

```
sundarraj [ ~/demo ]$ terraform taint azurerm_linux_virtual_machine.example
Resource instance azurerm_linux_virtual_machine.example has been marked as tainted.
```

So, whenever , we are applying the resource , the earlier resource will be destroyed , and create another resource with modified properties , or it can be even same properties , if we want a resource to just reset.

You see in here , when we are performing “terraform plan” , it is marking that our tainted resource will be replaced and create another resource.



If we want to untaint any resource that we marked as taint earlier , we need to use “terraform untaint” as following

```
sundarraj [ ~/demo ]$ terraform untaint azurerm_linux_virtual_machine.example
Resource instance azurerm_linux_virtual_machine.example has been successfully untainted.
```

If we taint some resource , and do “terraform apply” then , we will be destroying the earlier and recreating the resource that we tainted earlier.

This tainting method is useful , when we want to modify the infrastructure through the code.

Terraform modules

If we want to create a set of resources and create a infrastructure by using the configuration files , where those can exists either on local or on the cloud , in a single shot , that method or concept is called terraform modules. These terraform modules can be created either on Local Path , Git , BitBucket , S3 , GCS , Registry and should be able to retrieve other configuration files and execute them at a time using terraform modules.

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

variable "ami" {
  type = string
  default = "ami-0c55b159cbfafe1f0"
}

resource "aws_instance" "example" {
  ami = var.ami
  instance_type = var.instance_type
}
```

To use this module, you can create another
for the variables:

```
module "ec2" {
  source = "./modules/ec2"
  instance_type = "t3.small"
  ami = "ami-0a58e22c727337c51"
}
```

In terraform modules , we will be mentioning source of all the resources configuration files , wherever it is , either in local , git , bitbucket , s3 , GCS , Registry or anywhere. Check the example , which was shown above.

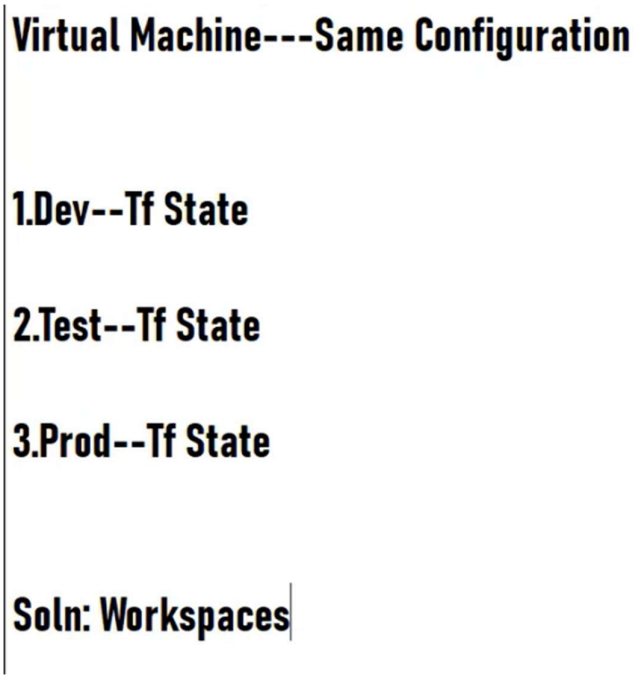
As said , we need to mention source of configuration files for the resource, in the below case , we written virtual machine configuration in demo.tf and storage account creation in demo1.tf , so we created module for each resource , by specifying the source location of the configuration file.

```
sundarraj [ ~ ]$ ls
clouddrive  demo  demo1
sundarraj [ ~ ]$ vi provider.tf
sundarraj [ ~ ]$ cat provider.tf
module "vmcreation" {
source="./demo"
}
module "sacreation" {
source="./demo1"
}

sundarraj [ ~ ]$
```

And if we perform ‘terraform apply –auto-approve’ , we will be able to create all the mentioned resources which are mentioned under provider.tf file that we mentioned ,

which is also having the information about the modules and the configuration files , that they must be referring to.

Sometimes , we will be having a condition like , we have same configuration files for the resources to build the infrastructure on all the dev , test and production environment, but , if we want to do that operation , we will be creating a single state file using the terraform on the local , for all the dev , test , and prod because terraform don't know the difference between the environments to create different state files , for this to be resolved , we have the concept of workspaces , where , we can build the infrastructure based on the workspace we are in. Concept is "Terraform Workspaces".



Going in a process to see what is workspace , we are creating a demo infrastructure , which consists of resource_group and storage_account , and , if we gonna execute them through terraform's we will be generating statefiles , and to store those state files of different environment in an organized manner , we are using azure storage's container , which stores file types in it. Following is the example to create an azure storage container which is part of azure storage account through code.

```
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "example" {
  name     = "myrg"
  location = "eastus"
}
resource "azurerm_storage_account" "example1" {
  name                     = "niqsa123"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "GRS"

}
resource "azurerm_storage_container" "example" {
  name                  = "niqctr"
  storage_account_name  = azurerm_storage_account.example.name
  container_access_type = "private"
}
```

If we do create any resource manually on the cloud to import that to the local machine , we need to check the subscription id of the resource that we want to import and later we need to move to that subscription id in the local

```
sundarraj [ ~/demo ]$ az account list -o table
Name          CloudName    SubscriptionId                        TenantId                              State    IsDefault
------------  -----------  ------------------------------------  ------------------------------------  -------  -----------
Pay-As-You-Go  AzureCloud   709ce538-d87c-4a02-9eec-40da53d61ed8  4b2faf64-9e45-4375-a6fa-976d8bd730a3  Enabled  True
Pay-As-You-Go  AzureCloud   701cac9f-6b7f-4dc3-97e5-eafb8ca32466  4b2faf64-9e45-4375-a6fa-976d8bd730a3  Enabled  False
Pay-As-You-Go  AzureCloud   2ef6bad3-c1e1-45ba-be1c-f7268c083f8d  4b2faf64-9e45-4375-a6fa-976d8bd730a3  Enabled  False
Pay-As-You-Go  AzureCloud   24f9a255-a03d-4cd5-bbc8-d85955c0fa62  4b2faf64-9e45-4375-a6fa-976d8bd730a3  Enabled  False
sundarraj [ ~/demo ]$ az account set -s 24f9a255-a03d-4cd5-bbc8-d85955c0fa62
```

This is the main file we had , currently

```
sundarraj [ ~/demo ]$ cat main1.tf

provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "example" {
  name     = "myrg"
  location = "eastus"
}
resource "azurerm_storage_account" "example1" {
  name                     = "niqsa123"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "GRS"

}
resource "azurerm_storage_container" "example" {
  name                  = "niqctr"
  storage_account_name  = azurerm_storage_account.example.name
  container_access_type = "private"
}
```

If We need the resources which are in the cloud to be used on local machine we need import the resources from the cloud to local machine , in the following manner

```
sundarraj [ ~/demo ]$ az account set -s 24f9a255-a03d-4cd5-bbc8-d85955c0fa62
sundarraj [ ~/demo ]$ terraform import azurerm_resource_group.example "/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/myrg"
azurerm_resource_group.example: Importing from ID "/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/myrg"...
azurerm_resource_group.example: Import prepared!
  Prepared azurerm_resource_group for import
azurerm_resource_group.example: Refreshing state... [id=/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/myrg]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.

sundarraj [ ~/demo ]$ az account set -s 24f9a255-a03d-4cd5-bbc8-d85955c0fa62
sundarraj [ ~/demo ]$ terraform import azurerm_storage_account.example1 "/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/myr
/providers/Microsoft.Storage/storageAccounts/niqsa123"
```

Following are the possible terraform commands

```
sundarraj [ ~/demo ]$ terraform workspace -h
Usage: terraform [global options] workspace

  new, list, show, select and delete Terraform workspaces.

Subcommands:
    delete    Delete a workspace
    list      List Workspaces
    new       Create a new workspace
    select    Select a workspace
    show      Show the name of the current workspace
```

To check list of workspaces available on the system ,

```
sundarraj [ ~/demo ]$ terraform workspace list
* default
```

To create a new workspace , we need to use the following command

```
sundarraj [ ~/demo ]$ terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

To show in what workspace we are on currently

```
sundarraj [ ~/demo ]$ terraform workspace show
dev
```

And if we want to create and select a workspace , we can do by the following

```
sundarraj [ ~/demo ]$ terraform workspace list
  default
* test

sundarraj [ ~/demo ]$ terarform workspace new prod
bash: terarform: command not found
sundarraj [ ~/demo ]$ terraform workspace new prod
Created and switched to workspace "prod"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
sundarraj [ ~/demo ]$ terraform workspace select test
Switched to workspace "test".
sundarraj [ ~/demo ]$
```

After creating different workspaces and initialization on them using “terraform init” , we will be able to generate two different folders under a folder called “terraform.tfstate.d/” in current directory . And under that , we can see two different folders are there , which will be having state files under them.

```
sundarraj [ ~/demo ]$ terraform workspace select test
Switched to workspace "test".
sundarraj [ ~/demo ]$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed hashicorp/azurerm v3.83.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
sundarraj [ ~/demo ]$ ls
main1.tf  main2.tf  main.tf  terraform.tfstate  terraform.tfstate.backup  terraform.tfstate.d
sundarraj [ ~/demo ]$ cd terraform.tfstate.d/
sundarraj [ ~/demo/terraform.tfstate.d ]$ ls
dev  prod
sundarraj [ ~/demo/terraform.tfstate.d ]$
```

When , we are in a workspace and a process going on , we can’t initiate another process from the same workspace or the other. We will be having a process lock until the completion of the process started in a workspace.

Now , if we want to maintain these statefile's irrespective of the workspace we are in , on the cloud , we need to use terraform backend to make sync on to the cloud, we are using container resource of azure for this purpose. Backend is just to maintain sync with the cloud.

```
terraform {
  backend "azurerm" {
    resource_group_name  = "myrg"
    storage_account_name = "niqsa123"
    container_name       = "niqctr"
    key                  = "prod.terraform.tfstate"
  }
}

resource "azurerm_resource_group" "example1" {
  name     = "example"
  location = "eastus"
}
resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
}

resource "azurerm_subnet" "example" {
  name                 = "internal"
  resource_group_name  = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
-- INSERT --
```

We can delete the terraform workspace as "terraform workspace delete <workspace-name>", we can't delete current workspace , we are on currently , we need to move to another workspace and delete the other workspace which we wanted to.

```
sundarraj [ ~/demo ]$ terraform workspace delete dev
Workspace "dev" doesn't exist.

You can create this workspace with the "new" subcommand.
sundarraj [ ~/demo ]$ terraform workspace delete prod
Acquiring state lock. This may take a few moments...
Releasing state lock. This may take a few moments...
Deleted workspace "prod"!
sundarraj [ ~/demo ]$
```

If we want to create multiple resources at a time , by passing different set of details for each resource , using a for_each map , which will be having set of resource details , which will be refered later as "each.value.<attr>" .

```
provider "azurerm" {
features {}
}
resource "azurerm_resource_group" "eitrg" {
for_each={
"rg1"={name="cdar",location="eastus"}
"rg2"={name="ogrds",location="eastus"}
"rg3"={name="cip",location="eastus"}
}
name=each.value.name
location=each.value.location
}
```

And later to this , it will be normal terraform initialization , validation and plan that using it's respecitve terraform commands. Terraform plan shows , number of resource groups that we tend to create using the main.tf file

To destroy the resources of terraform which we created by using main.tf file , we need to use the "terraform destroy –auto-approve"

```
sundarraj [ ~/demo1 ]$ terraform destroy --auto-approve
azurerm_resource_group.eitrg["rg2"]: Refreshing state... [id=/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/ogrds]
azurerm_resource_group.eitrg["rg1"]: Refreshing state... [id=/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/cdar]
azurerm_resource_group.eitrg["rg3"]: Refreshing state... [id=/subscriptions/24f9a255-a03d-4cd5-bbc8-d85955c0fa62/resourceGroups/cip]
```

We are having cloud for terraform , which is called as "terraform cloud" , we can login to terraform cloud through "terraform login"