



Ashoka Chakravarti Marpu



acmarpu@gmail.com



<http://www.linkedin.com/in/acmarpu>



<https://github.com/acmarpu>



Google



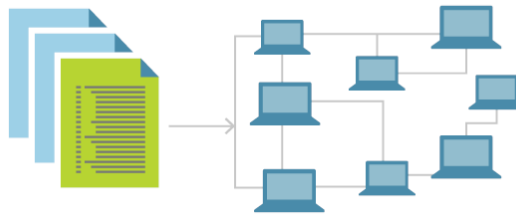
TERRAFORM NOTES

Contents

What is Terraform?	2
Terraform installation	3
Terraform setup in VS Code	3
Authenticating to Azure	3
Provider	4
Resource	5
Resource dependencies	6
Data Source	6
Terraform Commands	7
terraform init	7
terraform validate	7
terraform plan	8
terraform apply	9
Terraform Plan file create for Azure resource execution	10
Variable	11
Input variables:	12
String:	12
number:	12
bool:	12
list (or tuple):	12
map (or object):	12
Count variable:	12
length Variable:	13
Map With lookup	14
Prefix, random_id & Output	15
Merge Function	15
format function	16
distinct and setunion function	17
Environment variable	17
Output variables:	18
Locals variables:	18
Terraform State(terraform.tfstate)	19
Terraform state(terraform.tfstate.backup)	20
Terraform file organization	20
Terraform outputs	20
Terraform Modules	20
Terraform backends	20
Terraform Provisioning	21

What is Terraform?

- ✧ Terraform is an open source “Infrastructure as Code” tool, created by Hashicorp. help with multi-cloud by having one workflow for all clouds
- ✧ Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can Infrastructure as Code (IaC) is the managing and provisioning of infrastructure through code instead of through manual processes
- ✧ Infrastructure as code (IaC) uses DevOps methodology and versioning with a descriptive model to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies. Just as the same source code always generates the same binary, an IaC model generates the same environment every time it deploys.



- ✧ From terraform, infrastructure can be created in almost all the **virtual environments** (Azure, AWS, Google, VMware, VirtualBox etc).
- ✧ Terraform is developed in **Go language** (Go, also called Golang or Go language, is an open-source programming language that Google developed)
- ✧ Terraform is single **executable**
- ✧ Terraform template are written in custom **DSL** (Domain-Specific Languages). This DSL mostly looks like JSON, A domain-specific language (DSL) is a computer language specialized to a particular application domain. This contrasts with a general-purpose language (GPL), which is broadly applicable across domains.
- ✧ Terraform template ends with **mani.tf** extension.
- ✧ Provisioners: Terraform support various provisioners like **shell, chef and PowerShell**
- ✧ Terraform takes folder as an input

Terraform installation

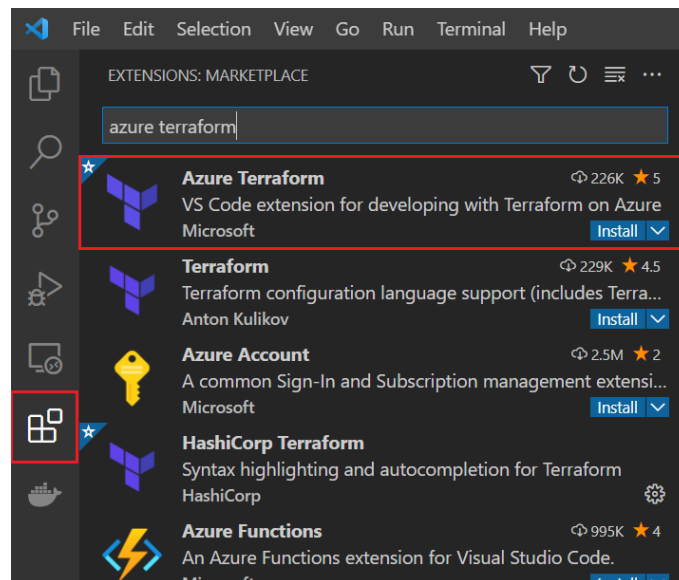
Windows:

- ✈ Download Terraform. Navigate to the Terraform download page (<https://www.terraform.io/downloads.html>).
- ✈ From the download, extract the executable to a directory of your choosing (for example, c:\terraform).
- ✈ Update your system's global path to the executable (In System Variables, select Path > edit > new > Enter the location of the Terraform .exe, eg C:\Terraform then click OK)
- ✈ Open a terminal window.
- ✈ Verify the global path configuration with the terraform command.
- ✈ terraform -version

Linux:

- ✈ Use the wget tool to download the file
- ✈ `wget https://releases.hashicorp.com/terraform/1.3.5/terraform_1.3.5_linux_amd64.zip`
- ✈ Unzip the file to the directory you chose. Use the full file name with the extension when extracting the archive.
- ✈ `sudo unzip terraform_1.3.5_linux_amd64.zip -d /usr/local/terraform`
- ✈ To verify if the Terraform directory is in the selected location, use the list command. For example:
`ls -l /usr/local/bin`
- ✈ terraform -version

Terraform setup in VS Code



Authenticating to Azure

- ✈ Authenticating to Azure using the Azure CLI
- ✈ `Az account set --subscription "subscription ID"` # if you must work on multiple subscription you need to set this command
- ✈ Authenticating to Azure using Managed Service Identity
- ✈ Authenticating to Azure using a Service Principal and a Client Secret and Client Certificate

Provider

Providers are the plugins that Terraform uses to manage those resources. Every supported service or infrastructure platform has a provider that defines which resources are available and performs API calls to manage those resources.

The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with

- ✧ Provider is used to create infrastructure in particular virtual environment.
 - ✧ The Azure Provider can be used to configure infrastructure in Microsoft Azure using the Azure Resource Manager **API's**.
 - ✧ Terraform providers help terraform to know which api to execute
 - ✧ Can be configured with version, alias.
 - ✧ Terraform has list of providers supported each providers provide **resources** and **data sources**
- This below block specifies the configuration for the Azure provider that Terraform will use to interact with Azure resources.
 - features {}: This is an empty block that can be used to enable optional features offered by the Azure provider. Currently, it's empty (no features enabled).

```
# Configure the Azure provider
provider "azurerm" {
  features {}
  client_id = "XXXX-XXXXXX-XXXXXX"
  client_secret = "XXXXXX-XXXXXX-XXXXXX-XXX"
  subscription_id = "XXXXXX-XXXXXX-XXXXXX-XXX"
  tenant_id = "XXXXXX-XXXXXX-XXXXXX-XXX"
}

# Configure the Azure provider with using variables file
provider "azurerm" {
  features {}
  client_id = var.client_id
  client_secret = var.client_secret
  subscription_id = var.subscription_id
  tenant_id = var.tenant_id
}

#This block defines the required provider(s) and their versions that this Terraform
configuration expects to use.

terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "~> 2.46.0"
    }
  }
}
```

```
# Configure the AWS provider
```

```
=====
provider "aws" {
  region      = "us-west-2"
  access_key  = "my-access-key"
  secret_key  = "my-secret-key"
}
```

Resource

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

- ✈ Resource is part of the infrastructure create on the provider.
- ✈ Every provider provides resource.
- ✈ Represent your infrastructure in code.
- ✈ Let terraform know which resource to create/modify/destroy.
- ✈ Expression

```
resource "azurerm_resource_group" "terraform-RG" {
  name      = " terraform -resources"
  location  = "East US2"
}

resource "azurerm_storage_account" "terraform-storage" {
  name                        = "storageaccountname"
  resource_group_name        = azurerm_resource_group.terraform-RG.name
  location                   = azurerm_resource_group.terraform-RG.location
  account_tier               = "Standard"
  account_replication_type   = "GRS"
}
```

Resource Type: "azurerm_resource_group" is a resource type provided by the AzureRM Terraform provider. It is used to manage Azure resource groups.

Resource Block Name: "terraform-RG" is the name given to this particular instance or resource block of azurerm_resource_group. This name is used as an identifier within the Terraform configuration to reference this specific resource group instance.

resource_group_name: azurerm_resource_group.terraform-RG.name references the name attribute of the "azurerm_resource_group" resource block named "terraform-RG".

```
=====
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-unique-bucket-name"
  tags   = {
    Name      = "MyS3Bucket"
    Environment = "Production"
  }
}
```

Resource dependencies

- ✧ When resource A requires resource B to present (or already exist) this is called as Dependency. In terraform terms you must create B before resource A
- ✧ Dependencies help Terraform understand the order in which resources should be created, updated, or deleted, based on relationships defined in your configuration.

Note: Inputs in terraform are called as argument and outputs are called as attributes

To create resource dependencies, use the following expression

"\${<resource-type>.<resource-name>.<attributes-name>}"

"\$azurerm_virtualnetwork.myvnet.id"

```
resource "azurerm_network_interface" "example-nic" {
  name            = "example-nic"
  location        = "East US"
  resource_group_name = azurerm_resource_group.terraform-RG.name
  depends_on = [
    azurerm_virtual_network.example-vnet,
    azurerm_subnet.example-subnet
  ]
}
```

Data Source

- ✧ In Terraform, a **data source** in the context of Azure (AzureRM provider) allows you to fetch existing information from Azure resources and use that information within your Terraform configuration
- ✧ Data sources enable you to query existing Azure resources, such as virtual networks, storage accounts, or resource groups, and use attributes from those resources in other parts of your Terraform configuration.

✧ data.<type>.<name>.<attribute>

```
data "azurerm_resource_group" "example_rg" {
  name = "existing-resource-group"
}

resource "azurerm_virtual_network" "example_vnet" {
  name            = "example-vnet"
  location        = data.azurerm_resource_group.example_rg.location
  resource_group_name = data.azurerm_resource_group.example_rg.name
  address_space   = ["10.0.0.0/16"]
}
```

```
=====

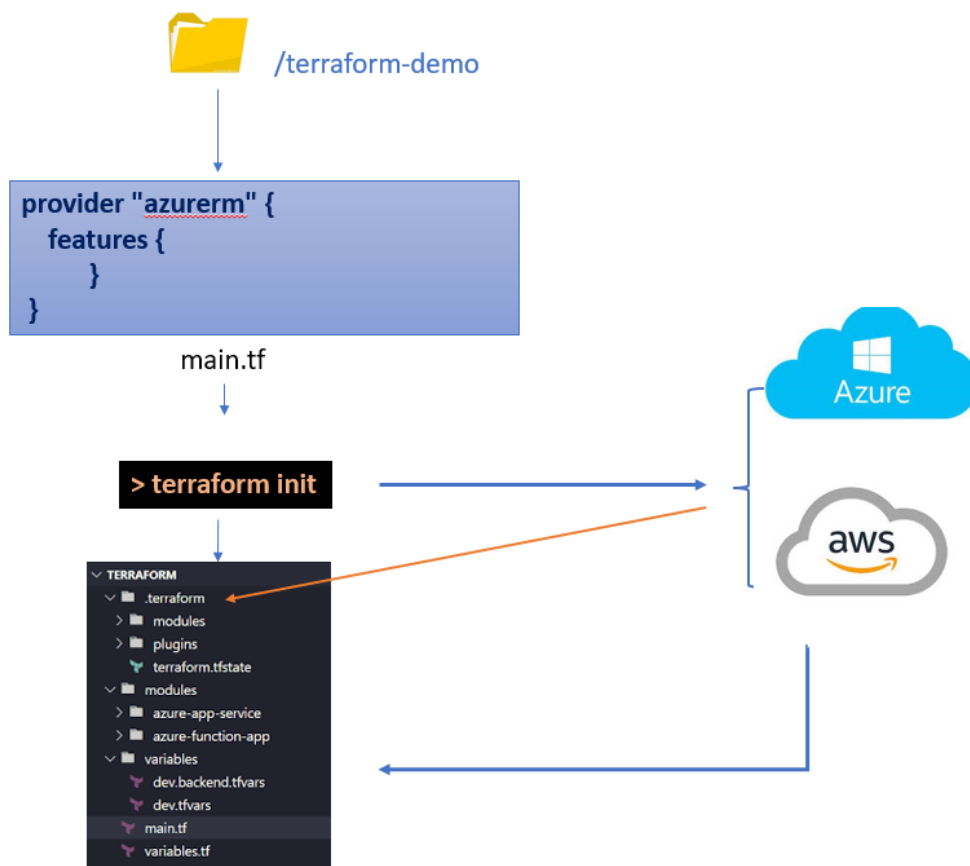
data "aws_s3_bucket" "example_bucket" {
  bucket = "my-existing-bucket"
}

resource "aws_s3_bucket_object" "example_object" {
  bucket = data.aws_s3_bucket.example_bucket.bucket
  key    = "example.txt"
  source = "path/to/local/file.txt"
}
```

Terraform Commands

terraform init .

- ✧ The terraform init command initializes a working directory containing Terraform configuration files.
- ✧ "Init command is just preparing your local environment by downloading all the required dependencies based on the provider which you specify since we have specified the provider as a azure so it has downloaded all the required dependencies for azure cloud platform and in case if I have specified over here aws then it is going to download all the required dependencies which is needed for communicating with the aws".

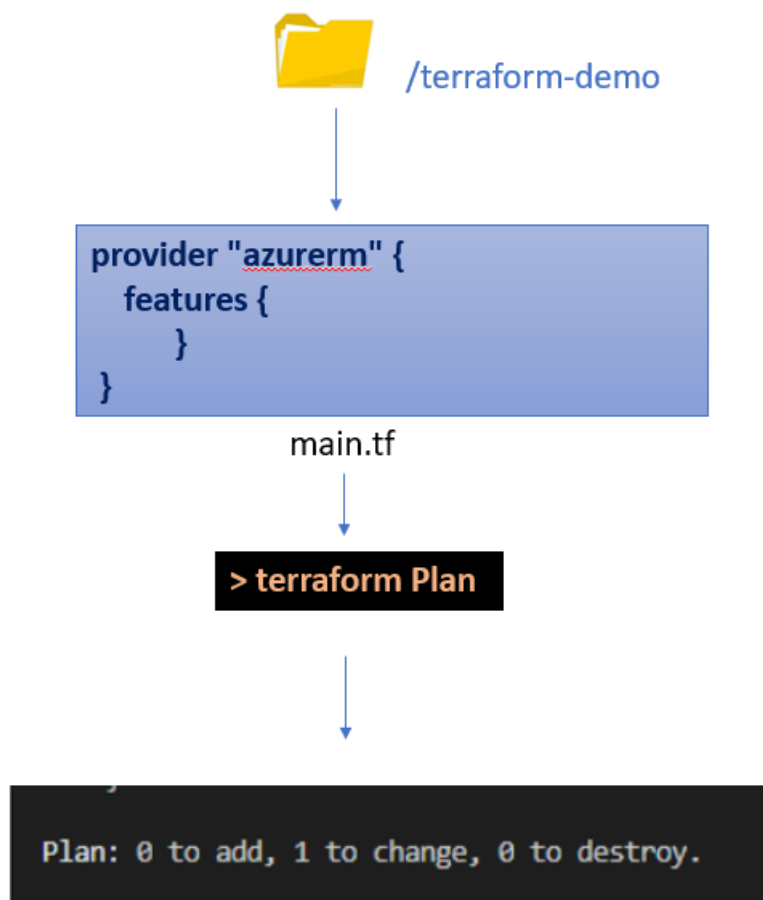


terraform validate .

- ✧ validate the terraform files
- ✧ Terraform Validate checks whether a configuration is **syntactically** valid. It can check things like unsupported arguments, undeclared variables, and others:

terraform plan .

- once you run the terraform plan command then you will see following message at the end of your terminal and this message is going to tell how many resources you have added how many resources you have changed or updated and how many resources you have destroyed this command
- Plan is not responsible for creating any resource on cloud platform
- Creates an execution plan, compares the desired state (defined in the configuration with the actual state of the infrastructure), and outputs the changes to be made.



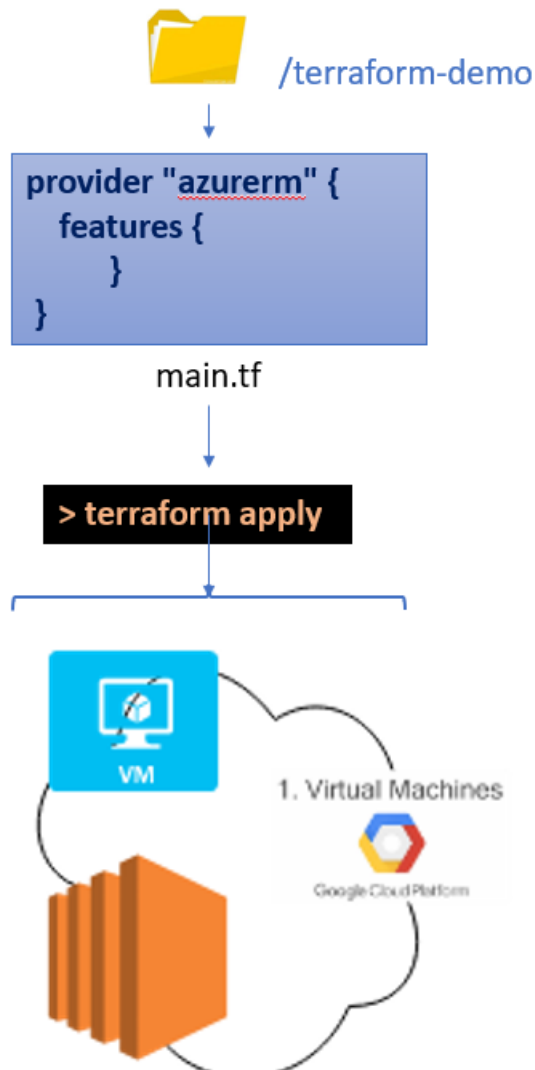
What is going to happen

terraform plan . # Plan execution command

terraform plan -out "add-object" # Create a saved plan for the new configuration.

terraform apply .

- ✚ Build or changes infrastructure
- ✚ makes the changes defined by your plan to create, update, or destroy resources.



terraform apply -auto-approve will immediately start applying any changes as per the configuration, without waiting for user input.

terraform apply -refresh-only This option instructs Terraform to refresh the state file to match the real-world resources without applying any configuration changes.

When you suspect that the actual infrastructure might be out of sync with your state file, running **terraform apply -refresh-only** will update the state file with any changes it detects (like manually deleted or externally altered resources).

terraform apply -lock-timeout command is used to specify a duration for which Terraform will wait for a state lock before it gives up and returns an error. The duration syntax is a number followed by a time unit, such as 5s for five seconds.

terraform apply -input=false is set, Terraform will not ask for input and will instead use the default values for any variables that have not been otherwise specified. If any required inputs are not provided and do not have default values, Terraform will produce an error and exit.

Let us look at some key differences between Terraform plan and apply.

	terraform apply	terraform plan
Purpose	Makes the actual changes in infrastructure.	Shows what changes Terraform will make. Acts as a "dry run".
Usage in workflow	Used after you're satisfied with the plan and ready to make actual changes to your infrastructure.	Used for creating an execution plan for reviewing and understanding the infrastructure changes.
State File	Updates the state file to reflect infrastructure changes.	Doesn't modify the state file.
Safety	Needs to be used with caution as it can alter your infrastructure state.	Safe to run, as it does not alter the infrastructure state.

terraform destroy . # once you are done resource can be removed using destroy

terraform apply -target=aws_vpc.default . This command tells Terraform to apply changes specifically to the aws_vpc.default resource defined in your Terraform configuration files.

terraform plan -help

Terraform Plan file create for Azure resource execution

- ✚ **terraform apply -auto-approve**
- ✚ **terraform plan -out="azure.plan" . # Save plan**
- ✚ **terraform apply "azure.plan"**
- ✚ **terraform plan -var "id=xxx" -var"seckey=xxx" -var "ass=xxxx" -var="asss=xxxxxx" -out="azure.plan" .**
- ✚ **terraform apply -var-file="auth.tfvars" -var-file="network.tfvars"**
- ✚ **terraform plan -out="azure.plan" . (not passing the variable, it will ask while executing)**
- ✚ **terraform refresh -var "ass=xxxx" -var "ass=xxxx"**
- ✚ **terraform taint <resource-type>.<resource-name>**
- ✚ **terraform apply -var "id=xxx" -var"seckey=xxx" -var "ass=xxxx" -var="asss=xxxxxx" -auto-approve .**

This command takes the input value and execute only .tfvar files

- ✚ **terraform plan -var-file="./dev.tfvars"**

Terraform Plan file create for AWS

- ✈ `terraform plan -out="aws.plan"`
- ✈ `terraform apply "aws.plan".`
- ✈ `terraform plan -var 'accesskey=xxxxxxxxxxxxxxxxxx' -var 'secretkey=xxxxxxxxxxxxxxxxxx' -out='aws.plan' .`
- ✈ `Terraform taint aws.subnet_subnet3`
- ✈ `Terraform untaint aws.subnet_subnet3`

Variable

- ✈ Since writing every thing in one file looks complex and unreadable, terraform allows you to write resource, provider etc in multiple .tf files
- ✈ To give options for the user to enter different values to resources. (hardcode)
- ✈ 3 types of variables in terraform
 - Input variables
 - Local variables
 - Output variables

Note- Inputs in terraform are called as **arguments** and outputs are called as **attributes**.

```
# Add this values in var file
client_id="-10dd- "
client_secret="2-a2e2-"
tenant_id=" -b437-4e73f12824e5"
subscription_id="-441d-8dbc-57af5"
location="westus"

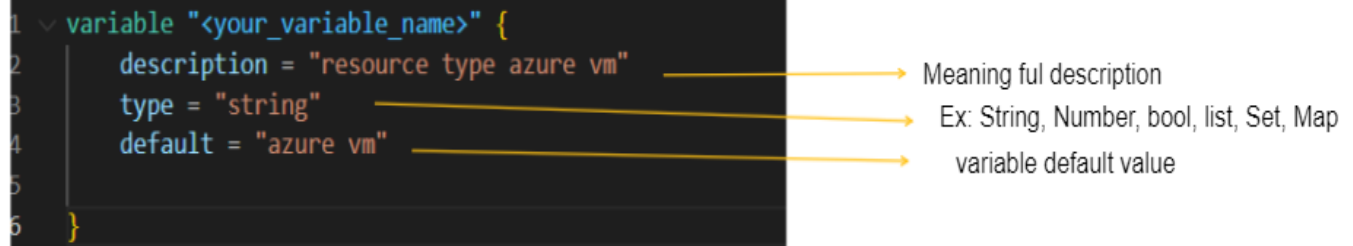
# Calling the values from var file and assign in main.tf file
provider "azurerm" {

    client_id      = "${var.client_id}"
    client_secret  = "${var.client_secret}"
    tenant_id      = "${var.tenant_id}"
    subscription_id = "${var.subscription_id}"

}
```

```
provider "azurerm" {
    features {
    }
    client_id = var.client_id
    client_secret = var.client_secret
    tenant_id = var.tenant_id
    subscription_id = var.subscription_id
}
```

<resource type>.<name>.<attribute>



```

1 variable "<your_variable_name>" {
2     description = "resource type azure vm"
3     type = "string"
4     default = "azure vm"
5 }
6

```

Meaning ful description

Ex: String, Number, bool, list, Set, Map

variable default value

Input variables:

In Terraform, input variables allow you to parameterize your infrastructure configurations, making them flexible and reusable across different environments or scenarios

String: a sequence of Unicode characters representing some text, like "hello".

```

variable "resource_group_name" {
    type = string
    default = "rg"
    description = "this is for resource group"
}

```

number: a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.

bool: a Boolean value, either **true** or **false**. bool values can be used in conditional logic.

list (or tuple): a sequence of values, like **["us-west-1a", "us-west-1c"]**. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.

map (or object): a group of values identified by named labels, like {name = "Mabel", age = 52}.

Count variable: We can write different types of count variable and terraform allows you like below.

```

# in variable file
variable "count" {
    default = "2"
}

```

```

resource_group_name = azurerm_resource_group.example.name
location             = azurerm_resource_group.example.location
virtual_network_name = azurerm_virtual_network.tf-vnet.name
location             = azurerm_resource_group.example.location
resource "azurerm_application_insights" "name" {
    count = 3
    name = format("%s-ins-%s", local.rg_name, count.index)
}

```

```
resource "azurerm_resource_group" "test" {
  name = "${var.resourcegroupname} -${count.index +1}"
  location = "${var.location}"
  count = 2
}
```

```
resource "azurerm_resource_group" "test" {
  name = "${var.resourcegroupname} -${count.index +1}"
  location = "${var.location}"
  count = "${var.count}"
}
```

```
resource "azurerm_application_insights" "name" {
  count = 3
  name = format("%s-ins-%s",local.rg_name,count.index)
}
```

- ✧ The var.number is used for how many same resources we need. If we need to provision multiple virtual machines or multiple IP addresses, then the count can declare as "var.numbercount" (note: you can use any name here except count, it is a deserved word).
- ✧ The "count.index" used for increment, for example, to provision two virtual machines we can mention the variable var.numbercount=2 then terraform need two names this can be achieved by count.index

```
variable "numbercount" {
  type      = number
  default   = 1
}

resource "azurerm_public_ip" "vmip" {
  count            = var.number
  name             = "vm-ip-${count.index}"
  resource_group_name = var.rgname
  allocation_method = "Static"
  location         = var.location
}
```

Note: Default terraform will create count from 0, if you want from 1 add +1

length Variable:

```
variable "subnet_names" {
  description = "A list of public subnets in the vnet"
  default = ["sub1", "sub2", "sub2"]
}

variable "subnet_prefixes" {
  description = "The address prefix to use for the subnet"
  default = ["10.0.0.0", "10.10.10.0", "11.012.0.11"]
}
```

```

resource "azurerm_subnet" "sub" {
  name                = "${var.sub_names[count.index]}"
  virtual_network_name = azurerm_virtual_network.vnet.name
  resource_group_name = azurerm_resource_group.rg.name
  address_prefixes     = "${var.subnet.prefixes[counr.index]}"
  count                = "${length(var.subnet_names)}"
}

```

Map With lookup

```

## in the variable section
variable "env" {
  description = "env = dev or prod or test"
}

variable "resource_group" {
  type = map
  default = {
    dev = "dev-rg"
    prod = "prod-rg"
    test = "prod1-rg"
  }
}

variable "location" {
  type = map
  default = {
    dev = "Centralindia"
    prod = "Southindia"
    test = "Centralindia"
  }
}

## in the Main tf
resource "azurerm_resource_group" "map_rg" {
  name = "${lookup (var.resource_group, var.env)}"
  location = "${lookup (var.location, var.env)}"
}

```

```

variable "age" {
  type = number
  default = 15
}

```

```
variable "age" {
  type = list
  default = ["15", "1"]
}
```

Prefix, random_id & Output

```
## in the Main.tf file
resource "random_id" "random" {
  byte_length = 2
}
```

```
resource "azurerm_resource_group" "rg" {
  name = "${var.prefix}-RG"
  location = var.location
}

resource "azurerm_storage_account" "example" {
  name                        = "${var.project}${random_id.random.dec}"
  resource_group_name        = azurerm_resource_group.rg.name
  location                   = azurerm_resource_group.rg.location
  account_tier                = "Standard"
  account_replication_type    = "GRS"
}
```

```
## in variable.tf file
variable "prefix" {
  default = "TF"
}
```

Merge Function

```
variable "project_tags" {
  type = map
  description = "(optional) describe your value"
  default = {
    Costcode = "Bas11"
    Owner = "Rakesh"
    Project = "Demo"
  }
}
```



```
variable "global_tags" {
  type = map
  description = "(optional) describe your value"
  default = {
    Environment = "dev"
    application = "prod"
  }
}

output "tags_out" {
  value = merge(var.project_tags, var.global_tags)
}
```

format function

```
variable "company_name" {
  default = "bee"
}

variable "environment_name" {
  default = "dev"
}

variable "project_name" {
  default = "demo"
}

locals {
  rg_name = "${var.company_name}-${var.project.name}-${var.project_name}"
  rg_name_1 = format("%s-%s-%s", var.company_name, var.project.name, var.project_name )
}

output "rg_name" {
  value = local.rg_name
}

output "rg_name" {
  value = local.rg_name_1
}
```

distinct and setunion function

```
variable "collection_1" {
  type = list(string)
  default = [ "pen", "pencil", "rule", "skatchpen" ]
}

variable "collection_2" {
  type = list(string)
  default = [ "pen", "pencil", "highliter" ]
}

locals {
  single_collections = contact(var.collection_1,var.collection_2)
}

output "single" {
  value = local.single_collections
}

output "single" {
  value = distinct(local.single_collections)
}
```

Environment variable

\$Env:TF_VAR_resource_group_name "dev-name"

```
variable "location" {
  type = "string"
  default = "westus"
}

variable "require_name" {
  type = "bool"
  default = "false"
}

variable "age" {
  type = "number"
  default = ["15","1"]
}

variable "tags" {
  type = "map"
  default = {
    environment = "prod"
    project = "eesl"
  }
}
```

```

    }
}
variable "persion" {
  type = "object"
  default = {
    name = "ashok"
    project = "eesl"
    age = "25"
    hobbies =["playing, 'music"]
    additional = {
      like_food = "true"
      need_more = "drink"
    }
  }
}

```

Output variables:

```

output "rg-name" {
  value = var.resource_group_name
}

output "rg-name" {
  value = resource_group_name.name.id
}

```

Locals variables:

```

locals {
  rg_name = format("rg-%", var.resource_group_name)
  //rg-demo-99
}

resource "azurerm_resource_group" "name" {
  name = local.rg_name
  location = var.location
}

```

```

format("%s-%s-%s-%s",
var.org_name,var.project_name,var.environment_name,var.evnironment_instance)

```

```

locals{
resource_group_name = format("%s-%s-%s-%s",
var.org_name,var.project_name,var.environment_name,var.environment_instance)
}
name = local.resource_group_name

```

```

variable subscription_id {
  type      = string
}
variable tenant_id {
  type      = string
}
variable client_secret {
  type      = string
}
variable client_id {
  type      = string
}
variable location {
  type      = string
}
variable "count" {
  default = "2"
}

```

Terraform State(terraform.tfstate)

-----Current state file -----

- ✚ Terraform state is used to reconcile deployed resources with Terraform configurations.
- ✚ State allows Terraform to know what Azure resources to add, update, or delete. By default, terraform state is stored locally when you run the terraform apply command. This configuration is not ideal for the following reasons:
 - Local state does not work well in a team or collaborative environment.
 - Terraform state can include sensitive information.
 - Storing state locally increases the chance of inadvertent deletion.
- ✚ Terraform will get the current status from provider (aws/Azure)
- ✚ It compares the status of provider with the tfstate file, if any differences are found they will be added for execution
- ✚ During apply plan will be created and compared to current status, if plan is not matching the current state, then the changes will be added to terraforms execution

Terraform state(terraform.tfstate.backup)

- ✈ Last successfully configuration file.

Terraform file organization

- ✈ Since writing everything in one file looks complex and unreadable, terraform allows you to write resources, providers etc in multiple.tf file.

Terraform outputs

- ✈ Output is result of the infra provisioning which can be shared.

To create outputs, create new file called as output.tf with following content

```
output "vpc-id" {
  value = "${aws_vpc.myvpc..id}"
}
output "subnet-id" {
  value = "${aws_subnet.subnet1..id}"
}
```

Terraform Modules

- ✈ Module is reusable terraform configuration
- ✈ Terraform community shares many modules for reuse in registry
- ✈ <https://registry.terraform.io>
- ✈ registry.terraform.io/modules

```
module "<name>" {
  source = "../hello-tf"
  accesskey = ""
  secretkey = ""
}
```

```
module "name" {
  source = "../modules/appinsight"
}
```

Terraform backends

- ✈ Two TF developers have the same terraform script and they have applied terraform, it creates two different resources. As the state file is stored on individual developers' laptop.
- ✈ Now if you want to restrict these two developers in such way, whenever they execute terraform, it should not create two different but one resource.
- ✈ Terraform support backend to store state file remotely and terraform also support locking feature to avoid simultaneous access to terraform state.
- ✈ **Remote shared state:** The state is shared and accessible to all authorized team members in an organization.
- ✈ **Security:** Remote backends are secured and encrypted to protect sensitive information in state files.

- ✈ State locking: Many remote backends support state locking, preventing conflicting operations from being applied simultaneously.
- ✈ this command takes the input value and execute only .tfvar files
- ✈ **terraform plan -var-file="./dev.tfvars"**

```
terraform {
  backend "azurerm" {
    resource_group_name = "ra-aad-b2c"
    storage_account_name = "storageacc99"
    container_name = "tfstate"
    key = "dev.terraform.tfstate"
  }
}
```

Terraform Provisioning

- ✈ Provisioning in terraform is majorly used to do configuration management
- ✈ Terraform supports many provisioners, some of the are
 - Remote-exec
 - Local-exec
 - Chef

For ansible based CM use remote-exec to install ansible and execute ansible-playbook with localhost in inventory

To use provisioners you should also use connections

Provisioners are written in terraform resources

```
provider "azurerm" {
  features {}
  client_id = var.client_id
  client_secret = var.client_secret
  subscription_id = var.subscription_id
  tenant_id = var.tenant_id
}

resource "azurerm_virtual_network" "teraform-vnet" {
  vm_details
  key_name="packer"

  connection {
    type = "ssh"
    user = "ec2-user"
    host = "${aws_instance.apache.publicip}"
    private_key = "${file("./packer.pem")}"
  }

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

Use Terraform Workspaces

Terraform Workspaces are essential for managing multiple distinct states within the same set of Terraform files.

This is particularly useful when managing different environments (like development, staging, and production). Benefits of workspaces include:

- **Environment isolation:** Workspaces allow you to keep state and Terraform configuration file data separate for each environment.
- **Resource efficiency:** Workspaces allow for the efficient use of resources by leveraging the same working copy of a Terraform configuration file and the same plugin and module caches across different infrastructure sets.