**Read Generate Modify Terraform Configuration**

**Terraform Variables:**

Terraform can store the values directly in the main.tf file with the argument in a key value pair.

Name = “value of the name”

Location = “location name”

There is a code-reusability approach when we make this aligned in the variables.tf file.

Example:

variable "example" {}

From the above sample terraform accepts the input variable called example.

Now if we need to use this in the code then we need to call this variable using

Var.example

# variables.tf

# Variable for VM instance type

variable "instance\_type" {

description = "Type of virtual machine instance"

type = string

}

# Variable for GitHub repository

variable "github\_repo" {

description = "GitHub repository name"

type = string

}

# Additional variables related to storage accounts or other resources can be added here

# For example:

# variable "storage\_account\_name" {

# description = "Name of the Azure storage account"

# type = string

# }

In this example:

We’ve declared two variables: instance\_type and github\_repo.

The type attribute specifies the data type of the variable (in this case, a string).

You can add more variables as needed, such as for storage account names or other resource-specific settings.

Variable TF file

* Purpose: The variables.tf file is where you declare all your variables. It acts as a central hub for defining variable names, types, and optional default values.
* Usage:
  + Declare variables with their data types and descriptions.
  + Optionally set default values.
  + Refer to these variables in your Terraform code.

Scenarios to check

terraform.tfvars vs variable.tf (which has highest precedence)

Variable.tf vs Env variable (which has highest precedence)

dev.tfvars vs variable.tf vs terraform.tfvars (which has highest precedence)

* + Scenario: terraform.tfvars vs variables.tf:
    - terraform.tfvars:
      * Highest Precedence: Values set in terraform.tfvars take precedence over other sources.
      * This file allows you to assign specific values to variables.
      * Useful for customizing settings without modifying the code.
    - variables.tf:
      * Lower Precedence: Declarations in variables.tf only define the structure of variables.
      * They don’t set actual values.
      * Used to declare variable names, types, and optional default values.
  + Scenario: variables.tf vs Environment Variables:
    - variables.tf:
      * Higher Precedence: Declarations in variables.tf take precedence over environment variables.
      * These declarations define the variable structure.
      * Actual values are assigned elsewhere (e.g., in terraform.tfvars).
    - Environment Variables (TF\_VAR\_name):
      * Lower Precedence: Environment variables are checked after variables.tf.
      * Set at the operating system level.
      * Useful for dynamic or secret values.
  + Scenario: dev.tfvars vs variables.tf vs terraform.tfvars:
    - dev.tfvars:
      * Highest Precedence: If you have a dev.tfvars file, it takes precedence over other sources.
      * Useful for environment-specific overrides.
    - variables.tf:
      * Middle Precedence: Declarations in variables.tf define the structure.
      * No actual values are set here.
    - terraform.tfvars:

**Lower Precedence**: Values set in terraform.tfvars are considered.

* + - Automatically loaded without any additional options.

New Scenario:

Lets say:

**Variable**

variable "location"

{ default = "west Europe” }

**terraform.tfvars:**

location = West US

**env**

TF\_LOCATION = East US

**dev.tfvars**

location = north europe which value terraform will it use in this scenario

1. Order of precedence:  
   **Environment Variables**:
   * The highest precedence is given to environment variables.
   * If you have set an environment variable named TF\_LOCATION with the value “East US,” Terraform will use that value.
2. terraform.tfvars**File**:
   * Next, Terraform checks the terraform.tfvars file.
   * If the file contains a value for location, it will be considered.
   * In your case, the terraform.tfvars file specifies location = West US.
3. dev.tfvars**File**:
   * Finally, Terraform looks at the dev.tfvars file.
   * If it contains a value for location, that value will be used.
   * In your case, the dev.tfvars file specifies location = north europe.

Variables cannot use the following names:

Source / count / for\_each / version / provider / lifecycles / locals / depends\_on

Terraform Output:

Terraform output values will return values of any data related to a resource provisioned in a provider to the CLI after a terraform apply is completed.

**Read documentation carefully**

Terraform plan -refresh-only

Terraform apply -refresh-only

Why we need refresh?

**terraform refresh** is a Terraform command that is used to reconcile the Terraform state with the real-world infrastructure. It queries the current state of the resources managed by Terraform providers and updates the state file with the latest information. Here are some reasons why you might need to use **terraform refresh**:

1. **State Synchronization:**
   * Over time, the actual state of your infrastructure might change due to manual modifications or changes made outside of Terraform. Running **terraform refresh** ensures that the Terraform state file accurately reflects the current state of your infrastructure.
2. **Detect Drift:**
   * Drift refers to the variance between the declared state (what's in your Terraform configuration) and the actual state (the real-world infrastructure). Running **terraform refresh** helps detect any drift and updates the state file accordingly.
3. **Identify Changes:**
   * Before making any modifications to your infrastructure, it's a good practice to run **terraform refresh** to identify any changes that might have occurred outside of Terraform. This helps you understand the existing state before planning and applying changes.
4. **Provider Updates:**
   * Terraform providers often receive updates that introduce changes to resource behavior or attributes. Running **terraform refresh** ensures that Terraform has the latest information about resource states, which is crucial for accurate planning and applying of changes.

Terraform refresh mainly focus on the drift that happens outside terraform may be created manually and updates the state file. This does not mean the state file is changed completely. If we run terraform plan and apply there is a in-memory refresh which will happen but it will change the state file back to normal. So refresh just checks and updates the state file to show you whats going to happen.

Once after running terraform refresh just check terraform show to view the state file.

Its very well used to view terraform output. For eg if you already added 5 outputs and then if you need to add one more, Then we need to update the output file and then execute terraform refresh instead of terraform apply, because this will check if there is any drift between the real world configuration and your state file and along with that it will also inject your new output configuration and brings back the result/

**Terraform Datasource**

In Terraform, a data source is used to fetch information or access data from an external system, such as an existing infrastructure, APIs, or other services, and make that data available for use within your Terraform configuration. Data sources enable you to incorporate external information into your Terraform configuration and use it to make decisions or set up dependencies.

All the syntax and code snippets are available in the documentation

We can use data source to fetch the resources that are already created through other tools, or if its created through manual creation and if we don’t want to import to terraform or even if its imported and if we wanted to use this resource in our configuration then we can use the data source to pull out the information and details and include in our code.

Eg: If VNET is created already and if wanted to include that during the creation of virtual machine then in this case we can use data source to pull the information and use it in our code

**Terraform Functions**

A block of organised, reusable code that meant to perform a single function

Eg: element / format / join / concat / format / substr

What is locals?

**Readability and Maintenance**

* Use **locals** to store intermediate or derived values in a clear and readable manner.
* Improve code maintainability by assigning meaningful names to complex expressions or calculations

**Reusability:**

* Define values in **locals** that are reused in multiple places throughout your configuration.
* Avoid duplicating complex expressions or calculations.

**Avoiding Repetition:**

* Use **locals** to store values that are used in multiple places, avoiding repetitive calculations or duplicate definitions.

**Testing and Debugging:**

* Use **locals** to isolate and test complex expressions or calculations.
* Debugging becomes easier as you can focus on specific local values.

**Terraform Functions:**

The idea of using terraform builtin functions is to enhance the code re-usability and at the same time logically enforcing some technical concepts to make it more flexible and feasible to understand and improvise the code for future use.

Some of the sample functions:

Format / join / length / subsctr / concat / lower

Please verify the examples mentioned in the code:

Lower function:

lower(local.truncateme) # Convert to lowercase

substr function:

#   truncateme = substr(var.storagename, 0, 11) # truncate to 11 characters

#   # substr(var.storagename, 0, 11) O is the starting position to extract the element and 11 is the number of characters to extract

Format function:

name                     = format("%s%s", var.resource\_prefix, var.environment)

  #%s is the placehodlers for the values

  #webserverprod

Concat function:

variable "tags" {

  type    = list(string)

  default = ["dev", "myapp"]

}

locals {

  fruits = ["apple", "banana", "cherry"]

}

output "concatenated\_fruits" {

  value = join(", ", local.fruits)

  #join(separator, list)

  #(", ") --> Seperator

  #local.fruits --> list

}

variable "networks" {

  type    = list(string)

  default = ["frontend", "backend"]

}

variable "zones" {

  type    = list(string)

  default = ["zone1", "zone2"]

}

locals {

  combined\_list = concat(var.networks, var.zones)

  #combined list = [frontend, backend, zone1, zone2]

}

Lenth function:

resource "azurerm\_virtual\_network" "example" {

  count               = length(local.combined\_list) # 4

  name                = "vnet-${local.combined\_list[count.index]}"

  address\_space       = ["10.0.0.0/16"]

  location            = "East US"

  resource\_group\_name = "functions"

  depends\_on = [azurerm\_resource\_group.function]

}

Join function:

resource "azurerm\_storage\_account" "format" {

  name                     = format("%s%s", var.resource\_prefix, var.environment)

  #%s is the placehodlers for the values

  #webserverprod

  resource\_group\_name      = "functions"

  location                 = "East US"

  account\_tier             = "Standard"

  account\_replication\_type = "LRS"

  tags = {

    description = join(", ", var.tags)

  }

  depends\_on = [azurerm\_resource\_group.function]

}

variable "tags" {

  type    = list(string)

  default = ["dev", "myapp"]

}

Join (“, “ 🡪 Seperator, var.tags 🡪 list)

Terraform Count 🡪 Meta-argument

Count parameter is used to created a fixed number of azure resources. It takes a numerical value and terraform takes the number and create the resources based on the count that was passed.

**It does not provide any kind of logical naming creation.**

Example:

variable "storage\_account\_names" {

default = ["images", "logs", "backups"]

}

variable "account\_tiers" {

default = ["Standard\_LRS", "Standard\_GRS", "Standard\_RAGRS"]

}

# Specify the count based on the length of storage\_account\_names

resource "azurerm\_storage\_account" "example" {

count = length(var.storage\_account\_names)

name = var.storage\_account\_names[count.index]

resource\_group\_name = azurerm\_resource\_group.example.name

location = azurerm\_resource\_group.example.location

account\_tier = var.account\_tiers[count.index]

account\_replication\_type = "LRS"

}

# Create an Azure resource group

resource "azurerm\_resource\_group" "example" {

name = "example-rg"

location = "East US"

}

**Terraform For\_each 🡪 Meta-argument**

locals {

  LeagueCup = ["Barcelona", "RealMadrid", "InterMiami","ManchesterCity"]

}

resource "null\_resource" "avengers" {

  for\_each = toset(local.LeagueCup)

  triggers = {

    name = each.value

  }

}

output "avengers" {

  value = null\_resource.avengers

}

In Terraform, when you use the **for\_each** argument with a list, Terraform automatically converts the list to a set to ensure that the keys used for identifying instances of resources are unique. This behavior is intentional and is designed to prevent naming conflicts and ensure that each element in the list results in a unique instance of the resource.

What is null resource here?

The **null\_resource** in Terraform is a resource that does nothing but can be used to trigger provisioners or perform other actions. It's often used when you need to perform some tasks that aren't supported by native Terraform resources or when you need to trigger actions based on changes in other resources. Here's an example of using **null\_resource**

In the above command what if I use without converting to set.

resource "null\_resource" "avengers" **{**

for\_each = local.avengers # instead of toset**(**local.avengers**)**

triggers = **{**

name = each.value

**}**

**}**

You will get the below error:

Error: Invalid for\_each argument

│

│ on main.tf line 60, in resource "null\_resource" "avengers":

│ 60: for\_each = local.LeagueCup # instead of toset(local.avengers)

│ ├────────────────

│ │ local.LeagueCup is tuple with 4 elements

│

│ The given "for\_each" argument value is unsuitable: the "for\_each" argument must be a map, or set of strings, and you have provided a value of

│ type tuple.

So from the above example it is clear that the for each can be applied to map and set of strings.

There’s a special each object that is assigned by Terraform.  The object has 2 attributes: each.key and .each.value

Terraform for-each example with map:

locals {

  strengthsofPlayers = {

    "LionelMessi"       = "Intelligence"

    "Cristiano Ronaldo" = "Physique"

    "Mbappe"            = "Dribbling"

    "Haaland"           = "Speed"

    "Luis Suarez"       = "Technical"

  }

}

resource "null\_resource" "strengths" {

  for\_each = local.strengthsofPlayers

  triggers = {

    name  = each.key

    power = each.value

  }

}

output "strengths" {

  value = null\_resource.strengths

}

**Terraform Count vs For-each**

**Use count when:**

* You have a fixed and known number of resource instances.
* Instances are identified using numeric indices.

**Use for\_each when:**

* You need to create instances dynamically based on a collection.
* Instances are identified by keys or values from a collection.
* The number of instances may vary or is not known in advance.

**count** is suitable for scenarios where the number of instances is fixed and known, while **for\_each** provides more flexibility when the number of instances may vary based on dynamic conditions or external data. The choice between them depends on the specific requirements of your infrastructure configuration.

**Terraform Dynamic Blocks**

Dynamic blocks are powerful features in terraform that allows to create multiple nested blocks within a Resource.

Benefits:

Speed: Make it quicker to process

Clarity: Code readability compared to repetitive blocks

Reusability: Avoid copy paste of the large blocks

Many Operations 🡪 🡪 pushed to dynamic block which will process it 🡪 sends output

Terraform Splat Operator:

A *splat expression* provides a more concise way to express a common operation that could otherwise be performed with a for expression.

If var.list is a list of objects that all have an attribute id, then a list of the ids could be produced with the following for expression:

[for o in var.list : o.id]

Copy

This is equivalent to the following *splat expression:*

var.list[\*].id

Terraform For Expression:

Eg:

lowercase\_rg\_names = [for name in var.resource\_group\_names : lower(name)]

Drill down:

1. **Variable Definition**:
   * You have defined a variable named resource\_group\_names with the type list(string).
   * The default value for this variable is a list of resource group names: ["MyRG1", "MyRG2", "MyRG3"].
2. **Local Variable Declaration**:
   * You create a local variable named lowercase\_rg\_names.
   * The value of lowercase\_rg\_names is determined by applying a transformation to each element in the resource\_group\_names list.
   * The transformation is done using the lower() function, which converts a string to lowercase.
3. **List Comprehension**:
   * The expression [for name in var.resource\_group\_names : lower(name)] is a list comprehension.
   * It iterates over each element (name) in the resource\_group\_names list.
   * For each name, it applies the lower() function to convert it to lowercase.
   * The result is a new list containing the lowercase versions of the original resource group names.
4. **Result**:
   * The lowercase\_rg\_names variable now holds the list ["myrg1", "myrg2", "myrg3"].

**Comparisition between For and Splat:**

**Comparison:**

1. **Purpose:**
   * **Splat:** Primarily used for extraction.
   * **For:** Primarily used for iteration and creation.
2. **Usage:**
   * **Splat:** Extracts existing values from lists or maps.
   * **For:** Creates new values or transforms existing ones based on iteration.
3. **Flexibility:**
   * **Splat:** Limited to extracting existing attributes.
   * **For:** Offers greater flexibility for dynamic generation and transformation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Construct** | **Type** | **Description** | **Use Case** |
| **Count** | Meta-Argument | Based on a count value | Resources you are provisioning are identical |
| **For\_each** | Meta-Argument | Based on a set of input values | Resources change between the different instances |
| **For** | Expression | Based on a set of input values | Transform a value |

**Terraform Conditional Expressions**

Conditional expressions in Terraform allow you to select one of two values based on the result of a boolean expression. They are commonly used for defining defaults, enabling/disabling features, and making decisions within your infrastructure code.

### Syntax

The syntax of a conditional expression is as follows:

condition ? true\_val : false\_val

* If condition is true, the result is true\_val.
* If condition is false, the result is false\_val.

**Terraform Dependencies:**  
  
Terraform manages two types of dependencies between the resources it orchestrates: **implicit dependencies** and **explicit dependencies**.

1. **Implicit Dependencies**:
   * As the name suggests, these dependencies are **automatically detected** by Terraform.
   * When one resource references another, Terraform inherently understands the relationship and establishes an implicit ordering requirement between them.
   * For instance, if a virtual machine (VM) depends on a network interface card (NIC), Terraform will automatically ensure that the NIC is created before the VM.
   * These implicit relationships are essential for maintaining the correct order of resource creation.
2. **Explicit Dependencies**:
   * Sometimes, a resource may depend on another without directly referencing it.
   * In such cases, we can create an explicit dependency using the depends\_on key within a Terraform resource.
   * By explicitly specifying dependencies, we ensure that Terraform follows the desired order during resource provisioning.
   * For example, if a security group rule depends on a network security group (NSG), we can explicitly define this relationship using depends\_on.

**Terraform Lifecycle**

Terraform lifecycle block helps us to take control on how the resource needs to be created or if we need to protect the resource from being destroyed or even to ignore the optional changes

**Create\_before\_destroy**

* By default, Terraform destroys the existing object and then creates a new replacement object with updated arguments.
* When create\_before\_destroy is set to true, the new replacement object is created first, and the prior object is destroyed after the replacement is created.
* Use this carefully, as some resource types have unique name requirements or other constraints.
* Note that destroy provisioners do not run if create\_before\_destroy is set to true.

**Prevent Destroy**

Setting prevent\_destroy to true is a protective measure to prevent **accidental deletion** of **critical resources**. If you attempt to destroy such a resource, Terraform will return an error and stop the operation. This can be useful when working with **Azure SQL Databases**, **Storage Accounts**, or any resource that holds important data.

**Ignore Changes**

This is more effective if we wanted to handle some attributes outside terraform. Eg: Tags or some configuration changes related to the application

**Terraform Graph**

Graph typically refers to Dependency Graph that represents the relationship the different resources in the infrastructure code. It will showcase the plan in which the terraform is going to proceed with the creation. It mainly implies the dependency strategies. Implicit Dependency & Explicit Dependency

Based on the dependency that we give to the terraform (Explicit dependency) and the implicit dependency happens in the resource creation for eg: Azure Resource Group needs to be created for all the remaining resources which means there is a implicit dependency created for Resource group

Understanding the dependency graph is crucial for managing complex infrastructure, as it ensures that resources are created and configured in the correct sequence. It also helps Terraform determine which resources can be created or updated concurrently to optimize the provisioning process.

The terraform graph command generates a DOT format representation of the dependency graph, and you can use tools like Graphviz to convert this representation into visual graphs.

What is terraform graph command?

1. **terraform graph command:**
   * When you run **terraform graph**, it produces a textual representation of the dependency graph of your Terraform configuration.
   * The output is in the DOT format, which is a simple text-based language for describing graphs.
2. **DOT Format:**
   * DOT is a plain text graph description language used by Graphviz, a powerful open-source graph visualization software.
   * The DOT format describes nodes, edges, and their attributes in a graph. It's human-readable and can be easily generated and processed.
3. **Graphviz:**
   * Graphviz is a suite of open-source tools for visualizing graph structures.
   * It includes a command-line tool called **dot** that reads DOT format files and generates visual representations of graphs in various formats (such as PNG, PDF, SVG).
4. **Conversion to Visual Graph:**
   * By using the **terraform graph** command, you obtain a DOT file representing the dependencies between your Terraform resources.
   * The subsequent part of the command (**| dot -Tpng > graph.png**) takes this DOT file and uses the **dot** command from Graphviz to convert it into a visual graph in PNG format.

**Terraform Parallelism**

Ability to create / Update / Delete multiple resources concurrently. Improving the overall efficiency of provisioning resources.

Concurrency : It specifies the level of parallelism with -parallelism tag.

Terraform by default runs with 10 parallelism. Meaning it can create the resources without any dependency in concurrent fashion.

It controls the number of resources that can be created parallely. However there are certain things to be noted like the dependency between the resources.

Command to set:  
terraform apply -parallelism=5

Advantages:

It speeds up the process

Spin and Go in a single shot

We can avail the complete benefits in a application which has lot of resource creation

Cons:

Need to be very careful as we need to check the dependency