**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)