spacelift

// The Most Flexible IaC Platform

Terraform
Loops & Conditionals

Loops and Conditionals take your Terraform code one step further, making it DRY and helping you offer full flexibility and genericity to it.

Count

- Used to create multiple resources of the same type
- Exposes an object called **count.index** that can be used as you would normally use an iterator
- Can be used on lists (based on their length) to create multiple resources of the same type but with different arguments -- This generates an issue when you are removing an element from the list if it's not the last one, because all the elements from that position onward will be recreated
- Instances of the resource are accessed with **resource_type.resource_name[index]**

```
# Simple count example
resource "azurerm_resource_group" "this" {
  count    = 3
  name     = format("resource_group%d", count.index)
  location = "westeurope"
}

# Terraform Plan Output
# azurerm_resource_group.this[0] will be created
+ resource "azurerm_resource_group" "this" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "resource_group0"
  }

 # azurerm_resource_group.this[1] will be created
+ resource "azurerm_resource_group" "this" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "resource_group1"
  }

 # azurerm_resource_group.this[2] will be created
+ resource "azurerm_resource_group" "this" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "resource_group2"
  }
```

For_each

- As count, it's used to create multiple resources of the same type
- Exposes an object called **each** and this object has a **key** and a **value**
- Keys are accessed with each.key
- You can define multiple fields inside of the the values and you can access them with **each.value.field**
- Can be used on maps and sets, but as a best practice, you should use maps. **Doesn't have the same limitation as count**, due to the fact that you are not using lists.
- Instances of the resource are accessed with **resource_type.resource_name[key]**

```
locals {
  # defining a variable that contains the details related to the resource groups
  rg_details = {
    rg1 = {
      location = "westeurope"
    }
    rg2 = {
      location = "eastus"
    }
    rg3 = {
      location = "westus"
    }

# One as a time, the keys in this example are: rg1, rg2 and rg3
# One at a time, the values in this example are the maps used after rg[123]
  }

  # defining a variable that contains the details related to the nsgs
  nsg_details = {
    nsg1 = {
      rg_name = "rg1"
    }
    nsg2 = {
      rg_name = "rg3"
    }
  }
}

resource "azurerm_resource_group" "this" {
  for_each = local.rg_details
  name     = each.key
  location = each.value.location
}

resource "azurerm_network_security_group" "this" {
  for_each            = local.nsg_details
  name                = each.key
  location            = azurerm_resource_group.this[each.value.rg_name].location # Giving the flexibility for each nsg to choose its resource group
  resource_group_name = azurerm_resource_group.this[each.value.rg_name].name     # Giving the flexibility for each nsg to choose its resource group
}

# Terraform Plan Output
  # azurerm_resource_group.this["rg1"] will be created
  + resource "azurerm_resource_group" "this" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "rg1"
    }

  # azurerm_resource_group.this["rg2"] will be created
  + resource "azurerm_resource_group" "this" {
      + id       = (known after apply)
      + location = "eastus"
      + name     = "rg2"
    }

  # azurerm_resource_group.this["rg3"] will be created
  + resource "azurerm_resource_group" "this" {
      + id       = (known after apply)
      + location = "westus"
      + name     = "rg3"
    }
```

Ternary Operator

- Used to verify a condition
- Syntax: condition ? val1 : val2 -- If the condition is true, the value assigned will be val1 otherwise it will be val2
- Nested conditions can be used and you can go as deep as you want
- Can be used with for_each or count to create | avoid creating resources

```
resource "azurerm_resource_group" "this" {
  count    = var.create_this_rg ? 1 : 0
  name     = "this_rg"
  location = "eastus"
}

variable "create_this_rg" {
  type    = bool
  default = true
}

locals {
  elem1              = 4
  elem2              = length("elem2")
  simple_condition = local.elem1 > local.elem2 ? local.elem2 : local.elem1
  nested_condition = local.elem1 > local.elem2 ? local.elem2 : local.elem1 == local.elem2 ? local.elem1 : 0
}

output "simple_condition" {
  value = local.simple_condition
}

output "nested_condition" {
  value = local.nested_condition
}

# Terraform Apply Output
simple_condition = 4
nested_condition = 0
```

For loop

- Used to verify a condition
- Syntax: condition ? val1 : val2 -- If the condition is true, the value assigned will be val1 otherwise it will be val2
- Nested conditions can be used and you can go as deep as you want
- Can be used with for_each or count to create | avoid creating resources

```
locals {
  a_list = ["Alice", "Bob", "Charlie"]
  a_map = {
    student1 = {
      name           = "Alice"
      age            = 20
      favourite_subjects = ["Math", "Physics"]
    }
    student2 = {
      name           = "Bob"
      age            = 19
      favourite_subjects = ["English", "Spanish"]
    }
    student3 = {
      name           = "Charlie"
      age            = 22
      favourite_subjects = ["Computer Science", "Marketing"]
    }
  }

  list_from_list = [for name in local.a_list : format("Hello %s!", name)]
  list_from_map  = [for student_id, student_details in local.a_map : student_details.name]
  map_from_list  = { for name in local.a_list : name => format("Hello %s!", name) }
  map_from_map   = { for student_id, student_details in local.a_map : student_id => student_details.name }
  nested_loop    = flatten([for student_id, student_details in local.a_map : [for subject in student_details.favourite_subjects : format("%s_%s", student_id, subject)]])
}

output "list_from_list" {
  value = local.list_from_list
}

output "list_from_map" {
  value = local.list_from_map
}

output "map_from_list" {
  value = local.map_from_list
}

output "map_from_map" {
  value = local.map_from_map
}

output "nested_loop" {
  value = local.nested_loop
}

# Terraform Apply Output
list_from_list = ["Hello Alice!", "Hello Bob!", "Hello Charlie!"]

list_from_map = ["Alice", "Bob", "Charlie"]

map_from_list = {
  "Alice" = "Hello Alice!"
  "Bob" = "Hello Bob!"
  "Charlie" = "Hello Charlie!"
}

map_from_map = {
  "student1" = "Alice"
  "student2" = "Bob"
  "student3" = "Charlie"
}

nested_loop = ["student1_Math", "student1_Physics", "student2_English", "student2_Spanish", "student3_Computer Science", "student3_Marketing"]
```

If Block

- The if block exists only in a for loop, for everything else you need to use a ternary operator
- Behaves as an if block in any other programming language
- You can use and (&&) / or (||) logical operators

```
locals {
  list_of_numbers                 = [10, 11, 87, 39, 22, 4]
  an_even_number_list             = [for i in local.list_of_numbers : i if i % 2 == 0]
  an_even_number_list_greater_than_9 = [for i in local.list_of_numbers : i if i % 2 == 0 && i > 9]
  an_odd_number_list_plus_numbers_greater_than_10 = [for i in local.list_of_numbers : i if i % 2 != 0 || i > 10]
}

output "an_even_number_list" {
  value = local.an_even_number_list
}

output "an_even_number_list_greater_than_9" {
  value = local.an_even_number_list_greater_than_9
}

output "an_odd_number_list_plus_numbers_greater_than_10" {
  value = local.an_odd_number_list_plus_numbers_greater_than_10
}

# Terraform Apply Output
an_even_number_list = [10, 22, 4]

an_even_number_list_greater_than_9 = [10, 22]

an_odd_number_list_plus_numbers_greater_than_10 = [11, 87, 39, 22]
```