whitehorses®
member of the human network

WHITEBOOK

A clear understanding of loops in Terraform

Januari 2022

Auteur:
Wilbert Spaanderman
INTEGRATIESPECIALIST

Introduction

Since version 0.13 Terraform, created by HashiCorp, supports loops in resources and modules. Especially, there was a high demand for the feature ‘loop over modules’. This will generate high code developing advantages. Possible use cases for this are for example:

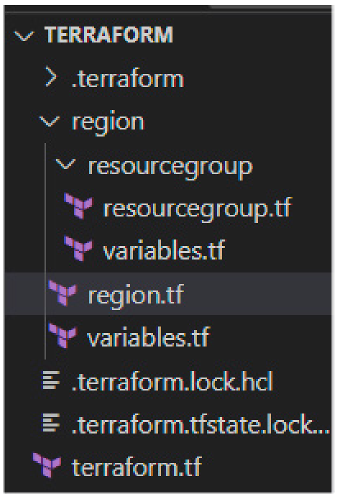
- Creating the exact same infrastructure over multiple regions in a cloud.
- Increasing or decreasing the amount of virtual machines in the infrastructure with just changing a single number.
- Updating virtual machines in a high available manner becomes really simple. Just add some new machines in the infrastructure, update those new machines, update the old machines and remove the last added machines again.

However, I noticed that the Terraform loops can also be really fragile. It can completely destroy the Terraform plans. Furthermore it costs some time to have a complete understanding of how HashiCorp implemented these looping features. This Whitebook will help to understand how HashiCorp implement loops in Terraform and especially how not to use them. It contains the following topics:

1. Loop over a singular resource
2. Difference between count and for_each
3. Chaining loops over chained resources
4. Loop over a singular module
5. Loop over nested modules using the for-loop
6. Loop over a set of objects
7. Only use 'known values' in for_each loops

Before we start first some information about the structure that is used in Terraform. It all starts with provide 'azurerm' in the terraform.tf file. The provider mainly will be used to create several resource-groups in Microsoft Azure Cloud. The terraform.tf file also contains the main module. This module will call another module called 'region'. Region is the equivalent of the regions used in Microsoft Azure, for example West-Europe and North-Europe. Eventually, region will call the module 'resourcegroup' which will be the module to create resource-groups in Microsoft Azure.

```
|— provider[registry.terraform.io/hashicorp/azurerm] 2.84.0
|— module.main (terraform.tf)
|   |— module.region (region.tf)
|   |   └— module.resourcegroup (resourcegroup.tf)
```



```
TERRAFORM
  .terraform
  region
    resourcegroup
      resourcegroup.tf
      variables.tf
    region.tf
    variables.tf
  .terraform.lock.hcl
  .terraform.tfstate.lock...
  terraform.tf
```

Loop over a singular resource

As mentioned in the introduction we will first loop over a single resource to create multiple resource-groups in Microsoft Azure. Terraform has two ways to create loops: ‘for_each’ and ‘count’. The example below make use of the for_each command for creating multiple resource-groups called ‘rg_MyResourceGroups’. It is pretty straight forward. The for_each count the amount of names defined in variable ‘names’ and will create that number of resource-groups. In the example three names are defined in variable ‘names’, so it will create three resource-groups in Microsoft Azure.

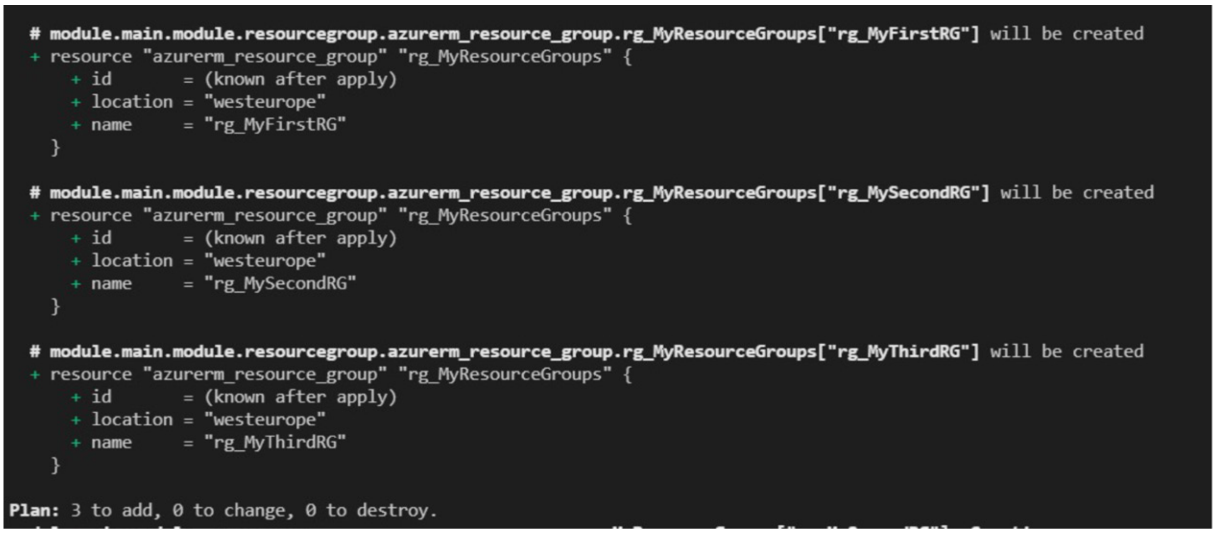
```
variable “names” {
    type    = set(string)
    default = [“rg_MyFirstRG”,”rg_MySecondRG”,”rg_MyThirdRG”]
}

resource “azurerm_resource_group” “rg_MyResourceGroups” {
    for_each = var.names
    name     = each.key
    location = var.region
}
```

Code block 1: for_each loop over a singular resource.

The each.key is an object provided by Terraform and contains the corresponding value of the running instance. So in this case, the first resource-group will get the first name from variable ‘names’, the second will contain the second name, etc. Because ‘names’ is defined as a set both, each.key and each.value could be used here.
When a variable is of type map, for example: { name = “myName”, age = “30” }. the each.key corresponds to the key of the map and each.value corresponds to the value of the map.

Plan result:



```
# module.main.module.resourcegroup.azurerm_resource_group.rg_MyResourceGroups["rg_MyFirstRG"] will be created
+ resource "azurerm_resource_group" "rg_MyResourceGroups" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "rg_MyFirstRG"
  }

# module.main.module.resourcegroup.azurerm_resource_group.rg_MyResourceGroups["rg_MySecondRG"] will be created
+ resource "azurerm_resource_group" "rg_MyResourceGroups" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "rg_MySecondRG"
  }

# module.main.module.resourcegroup.azurerm_resource_group.rg_MyResourceGroups["rg_MyThirdRG"] will be created
+ resource "azurerm_resource_group" "rg_MyResourceGroups" {
    + id       = (known after apply)
    + location = "westeurope"
    + name     = "rg_MyThirdRG"
  }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Plan Result 1: Three resources will be created.

Important: the for_each command only works unindexed types. In Terraform this means either a set or a map of strings! For indexed types like lists or objects the 'count' command has to be used.

Difference between count and for_each

Although count and the for_each looks pretty the same, there are some differences between them. As mentioned in the previous chapter, one of the differences between count and for each is that count is always indexed. When something is changed in the Terraform code, Terraform should only apply the change that is part of the code changed. However, because the count is indexed, changing the count loop can cause more apply changes in Terraform than expected beforehand. This chapter explains that count should only be used when for_ each is no option anymore.

Changing a for_each loop

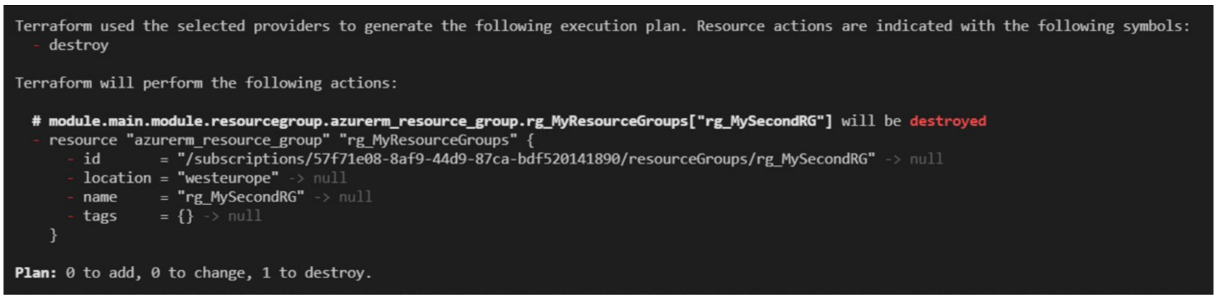
Back to the for_each example from the previous chapter. There, three resource-groups were created: ‘rg_MyFirstRG’, ‘rg_MySecondRG’, ‘rg_MyThirdRG’. But what will happen if we remove the ‘rg_MySecondRG’ in the code? Expected is that the Terraform plan will say that ‘rg_MySecondRG’ will be destroyed:

```
variable “names” {
    type    = set(string)
    default = [“rg_MyFirstRG”,”rg_MyThirdRG”]
}

resource “azurerm_resource_group” “rg_MyResourceGroups” {
    for_each = var.names
    name     = each.key
    location = var.region
}
```

Code block 2: rg_MySecondRG is removed from the code

Result of the Terraform plan:



```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # module.main.module.resourcegroup.azurerm_resource_group.rg_MyResourceGroups["rg_MySecondRG"] will be destroyed
  - resource "azurerm_resource_group" "rg_MyResourceGroups" {
      - id       = "/subscriptions/57f71e08-8af9-44d9-87ca-bdf520141890/resourceGroups/rg_MySecondRG" -> null
      - location = "westeurope" -> null
      - name     = "rg_MySecondRG" -> null
      - tags     = {} -> null
    }

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

Plan Result 2: Terraform destroys the resource-group ‘rg_MySecondRG’

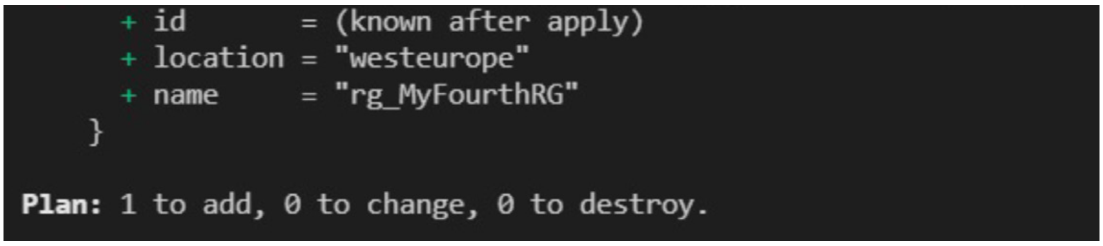
Terraform destroys as expected the second resource-group ‘rg_MySecondRG’. Also when a new resource-group, ‘rg_MyFourthRG’, is added, Terraform reacts as expected:

```
variable “names” {
    type    = set(string)
    default = [“rg_MyFirstRG”,”rg_MySecondRG”, “rg_MyFourthRG”, “rg_
MyThirdRG”]
}
```

Code block 3: rg_MyFourthRG is added to the code

Note: In the previous example only a terraform plan ran, so the second resource-group is still there.

Result of the plan:



```
       + id       = (known after apply)
       + location = "westeurope"
       + name     = "rg_MyFourthRG"
     }

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

Plan Result 3: Resource-group ‘rg_MyFourthRG’ is added to the plan

The new resource ‘rg_MyFourthRG’ is put on the third spot (index position [2]) on purpose. Although the ‘rg_MyThirdRG’ received a new index position it is not recreated again. As desired only the new resource-group ‘rg_MyFourthRG’ will be added.

Changing a count loop

Below is an example of how to perform the same code but then with a count instead of for_each:

```
variable “names” {
    type     = list(string)
    default = [“rg_MyFirstRG”, “rg_MySecondRG”, “rg_MyThirdRG”]
}

resource “azurerm_resource_group” “rg_MyResourceGroups” {
    count    = length(var.names)
    name     = var.names[count.index]
    location = var.region
}
```

Code block 4: Example of a (bad) count loop

It almost looks the same right? Almost! This is what is changed:

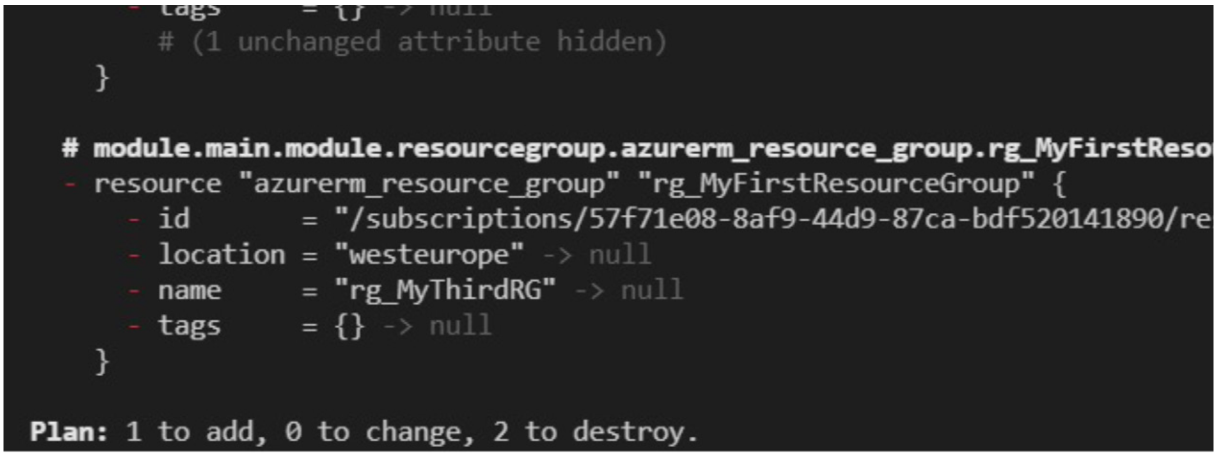
1. The type of variable name is changed from a set(string) to a list(string). List is an indexed array in Terraform.
2. For_each is changed to count. Because var.names is not a countable value, the length function is added too.
3. 'name' contains a count.index. This works because the list var.names contains an index now.

Although it looks the same as the for_each, Terraform behave differently after removing 'rg_MySecondRG':

```
variable “names” {
    type     = list(string)
    default = [“rg_MyFirstRG”, “rg_MyThirdRG”]
}
```

Code block 5: rg_MySecondRG is removed from the code

Result of the plan:



```
        - tags     = {} -> null
          # (1 unchanged attribute hidden)
      }

    # module.main.module.resourcegroup.azurerm_resource_group.rg_MyFirstReso
    - resource "azurerm_resource_group" "rg_MyFirstResourceGroup" {
        - id       = "/subscriptions/57f71e08-8af9-44d9-87ca-bdf520141890/re
        - location = "westeurope" -> null
        - name     = "rg_MyThirdRG" -> null
        - tags     = {} -> null
      }

  Plan: 1 to add, 0 to change, 2 to destroy.
```

Plan Result 4: rg_MySecondRG is destroyed and rg_MyThirdRG is destroyed and created again.

Unless only 'rg_MySecondRG' is removed from the code, Terraform both destroys 'rg_MySecondRG' and recreates 'rg_MyThirdRG'. Although the value of 'rg_MyThirdRG' is not changed, his index is changed. Therefore, Terraform destroys the 'rg_MyThirdRG' first and creates it again.

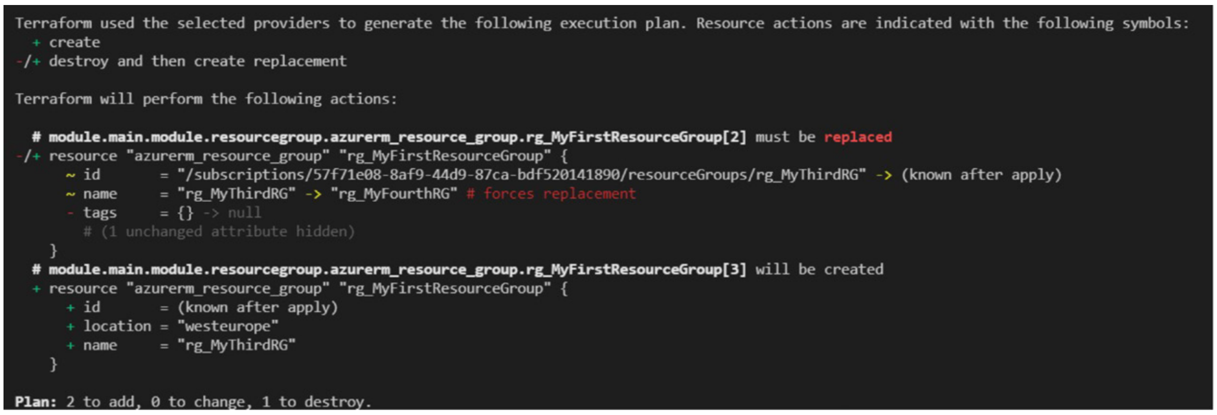
Another downside is that when the time between destroying and creating the same resource-group is too short, Terraform will fail. This is still a bug inside Terraform. The workaround is to run the ‘terraform apply’ command another time.

The same counts for adding a fourth resource-group and place that on the third spot, like:

```
variable “names” {
    type     = list(string)
    default = [“rg_MyFirstRG”, “rg_MySecondRG”, “rg_MyFourthRG”, “rg_
MyThirdRG”]
}
```

Code block 6: rg_MyFourthRG is added to the code

The plan result:



```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # module.main.module.resourcegroup.azurerm_resource_group.rg_MyFirstResourceGroup[2] must be replaced
-/+ resource "azurerm_resource_group" "rg_MyFirstResourceGroup" {
      ~ id       = "/subscriptions/57f71e08-8af9-44d9-87ca-bdf520141890/resourceGroups/rg_MyThirdRG" -> (known after apply)
      ~ name     = "rg_MyThirdRG" -> "rg_MyFourthRG" # forces replacement
      - tags     = {} -> null
        # (1 unchanged attribute hidden)
    }
  # module.main.module.resourcegroup.azurerm_resource_group.rg_MyFirstResourceGroup[3] will be created
  + resource "azurerm_resource_group" "rg_MyFirstResourceGroup" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "rg_MyThirdRG"
    }

Plan: 2 to add, 0 to change, 1 to destroy.
```

Plan Result 5: Resource-group ‘rg_MyFourthRG’ is added to the plan, but also ‘rg_MyThirdRG’ is destroyed and created again.

In the example above, Terraform adds the ‘rg_MyFourthRG’. Because that resource-group name is placed on the third spot (index place [2]), also the index position of ‘rg_MyThirdRG’ is changed. Therefore, ‘rg_MyThirdRG’ will be destroyed and recreated again. The examples of the for_each and count loops proves that for this situation the for_each is a much better option.

Chaining loops over chained resources

Each resource-group in Microsoft Azure can contain numbers of other resources. But how to give each resource-group its own resources when the resource-groups are created in a for loop. This is a so called chained resource. After creating the resource-groups a new loop have to be created to add new resources inside each resource-group. The code below is a nice example of how to do this.

Note: If the resource-group is created with a 'for_each' the output of the resource is a map. This means that also the next loop must be a 'for_each'. If the resource-group is created with a 'count' the output will be a tuple (is indexed). This means that also the next loop must be a 'count'.

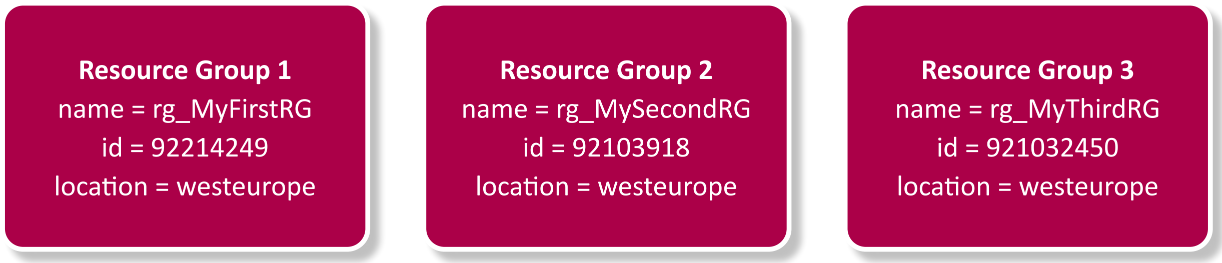
```
variable “names” {
    type     = set(string)
    default = [“rg_MyFirstRG”, “rg_MySecondRG”, “rg_MyThirdRG”]
}

resource “azurerm_resource_group” “rg_MyResourceGroups” {
    for_each = var.names
    name     = each.key
    location = var.region
}

resource “azurerm_virtual_network” “apim_vnet” {
    for_each            =  azurerm_resource_group.rg_MyResourceGroups
    name                = “vnet-of-${each.value.name}”
    location            = var.region
    resource_group_name = each.value.name
    address_space       = [“10.0.0.0/20”]
}
```

Code block 6:Chained resources created with loops.

The example above creates a virtual network for every resource-group. To do this, it loops over the output of the resource block ‘rg_MyResourceGroups’. The output will return a map with three objects. Every object (i.e.: every resource-group) contains several key value pairs, like name and id. The virtual network needs the name of the respective resource-group to know to whom it belongs too. The name is received with the ‘each.value.name’ function.



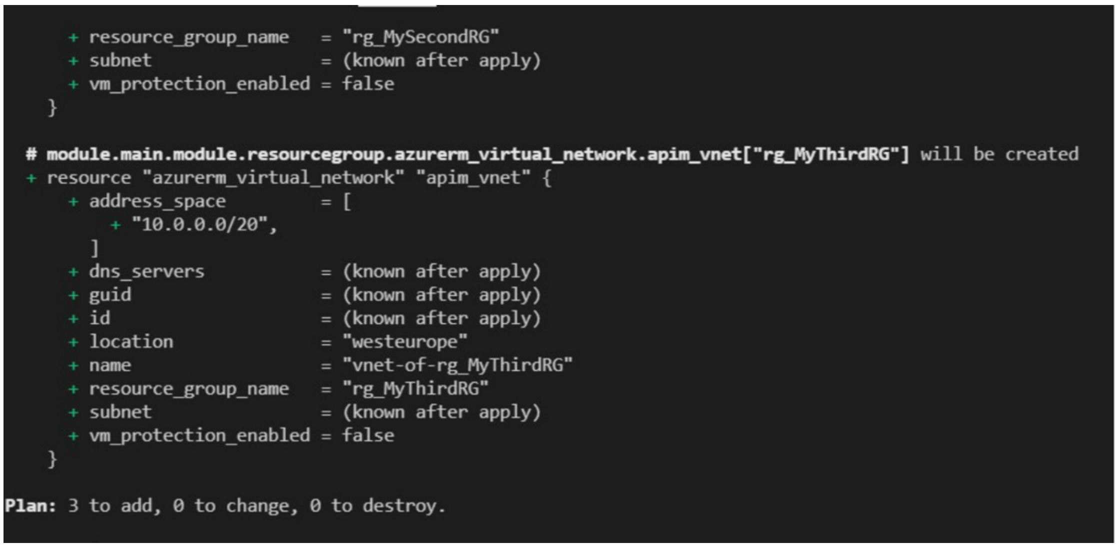
Resource Group 1
name = rg_MyFirstRG
id = 92214249
location = westeurope

Resource Group 2
name = rg_MySecondRG
id = 92103918
location = westeurope

Resource Group 3
name = rg_MyThirdRG
id = 921032450
location = westeurope

Figure 1: The output of the resource azurerm_resource_group.rg_MyResourceGroups visual represented.

A snippet of the plan result is:



```
      + resource_group_name    = "rg_MySecondRG"
      + subnet                 = (known after apply)
      + vm_protection_enabled  = false
    }

  # module.main.module.resourcegroup.azurerm_virtual_network.apim_vnet["rg_MyThirdRG"] will be created
  + resource "azurerm_virtual_network" "apim_vnet" {
      + address_space          = [
          + "10.0.0.0/20",
        ]
      + dns_servers            = (known after apply)
      + guid                   = (known after apply)
      + id                     = (known after apply)
      + location               = "westeurope"
      + name                   = "vnet-of-rg_MyThirdRG"
      + resource_group_name    = "rg_MyThirdRG"
      + subnet                 = (known after apply)
      + vm_protection_enabled  = false
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Plan Result 6: snippet of the outcome. Three virtual networks are created. Each belonging to one resource-group.

Although this concept of using multiple loops over chained resources works, it is not ideal. Now the resource-group contains only a virtual network. Next to the virtual network the resource-group also needs other resources like Virtual Machines, Gateways, etcetera. Furthermore, each Virtual Network can contain a loop of subnets, the subnet securities, etc. Eventually this will result in an obscure, unreadable code with tightly coupled loops.

The solution for this: loop over modules.

Loop over a singular module

A module is a container for multiple resources that are used together. This can be handy in our example. In the module ‘resourcegroup’ only one resource code will be created now. This resource-group will get its own virtual network. This time, the resource to create resource-groups is not created multiple times by a loop. For that particular instance the creation of the resource-group and the virtual network is a one-o-one relation. A direct link between the two can be created now. This makes the Terraform code less obscure and better readable.

```
variable “name” {
    type = string
}

resource “azurerm_resource_group” “rg_MyResourceGroups” {
    name     = var.name
    location = var.region
}

resource “azurerm_virtual_network” “apim_vnet” {
    name                = “vnet-of-${azurerm_resource_group.rg_
MyResourceGroups.name}”
    location            = var.region
    resource_group_name = azurerm_resource_group.rg_MyResourceGroups.name
    address_space       = [“10.0.0.0/20”]
}
```

Code block 7:the resource-group module without loops.

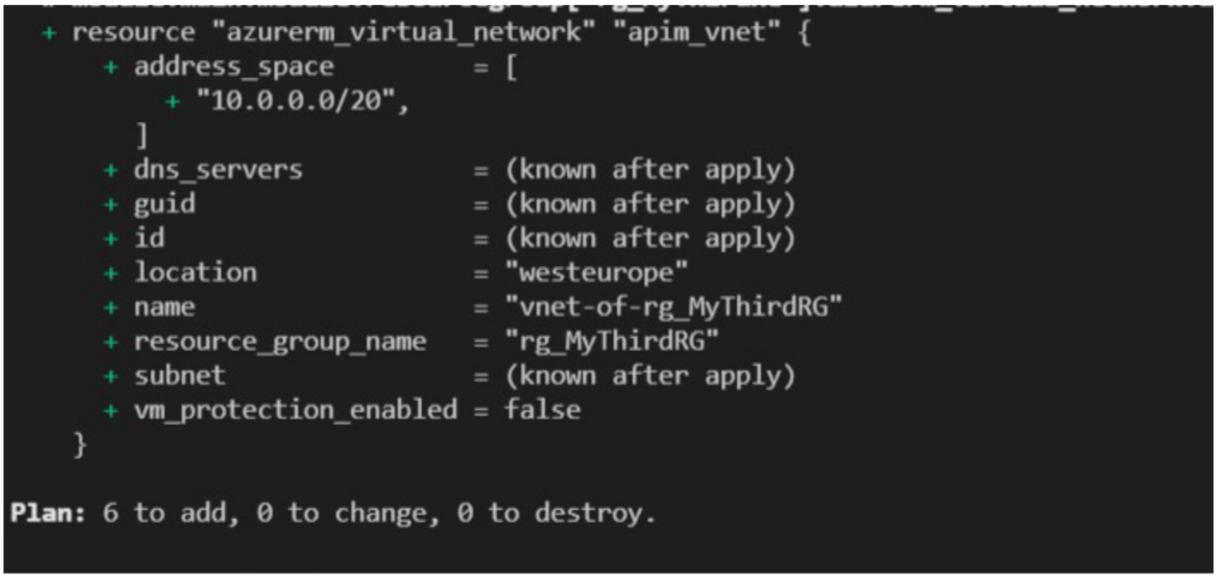
The for_each loop is implemented over the module now.

```
variable “names” {
    type     = set(string)
    default = [“rg_MyFirstRG”, “rg_MySecondRG”, “rg_MyThirdRG”]
}

module “resourcegroup” {
    for_each   = var.names
    source     = “./resourcegroup”
    region     = var.region
    name       = each.value
}
```

Code block 8: the region.tf file calling the resourcegroup module.

A snippet of the plan result is:



```
  + resource "azurerm_virtual_network" "apim_vnet" {
      + address_space          = [
          + "10.0.0.0/20",
        ]
      + dns_servers            = (known after apply)
      + guid                   = (known after apply)
      + id                     = (known after apply)
      + location               = "westeurope"
      + name                   = "vnet-of-rg_MyThirdRG"
      + resource_group_name    = "rg_MyThirdRG"
      + subnet                 = (known after apply)
      + vm_protection_enabled = false
    }

Plan: 6 to add, 0 to change, 0 to destroy.
```

Plan Result 7: Snippet of the plan: three resource-groups and three vnets created.

The snippet of the plan shows that there are three resource-groups created. Each resource-group contains its own virtual network.

Share data among modules

A module will not return any output data unless this is specified explicitly. Sometimes you want to share data among modules. With an output block it is possible to set any particular value. For example, another module needs the name of the resource-groups. Then set an output containing the name of the resource-group in the resourcegroup.tf.

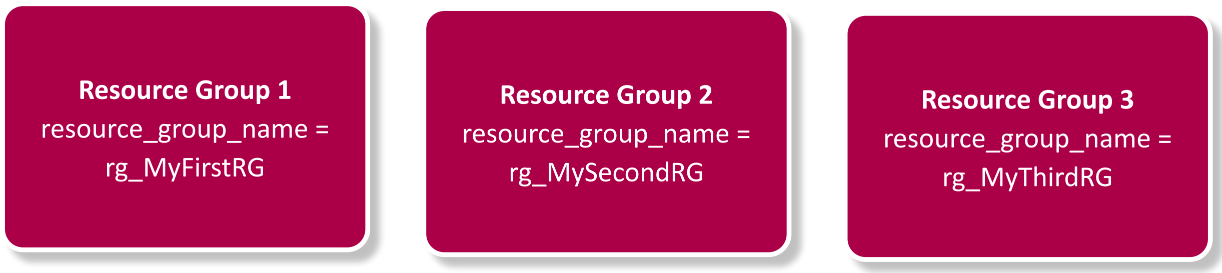
```
resource “azurerm_resource_group” “rg_MyResourceGroups” {
    name     = var.name
    location = var.region
}

resource “azurerm_virtual_network” “apim_vnet” {
    name                = “vnet-of-${azurerm_resource_group.rg_
MyResourceGroups.name}”
    location            = var.region
    resource_group_name = azurerm_resource_group.rg_MyResourceGroups.name
    address_space       = [“10.0.0.0/20”]
}
```

```
output “resource_group_name” {
    value = azurerm_resource_group.rg_MyResourceGroups.name
}
```

Code block 9: added an output to the resourcegroup.tf file

Because the resourcegroup module is called three times, Terraform will create three resourcegroup maps containing the resource_group_name as key. Visualized it looks like this:


Resource Group 1
resource_group_name = rg_MyFirstRG
Resource Group 2
resource_group_name = rg_MySecondRG
Resource Group 3
resource_group_name = rg_MyThirdRG

Another module, defined on the same level, can use the output of the resourcegroup module and loop again over it.

```
module “resourcegroup” {
    for_each    = var.names
    source      = “./resourcegroup”
    region      = var.region
    name        = each.value
}

module “resourcegroup2” {
    for_each    = module.resourcegroup
    source      = “./resourcegroup2”
    region      = var.region
    name        = “copy_of_resourcegroup:${each.value.resource_group_name}”
}
```

Code block 9: for each resourcegroup also a copy will be created.

Notice that name contains the expression ‘each.value.name’. An module output is always a map in case of a for_each. (a tuple in case of a count). So, also the key ‘resource_group_name’ needs to be specified.

Loop over nested modules using the for-loop

Create a list of outputs

Back to the originally tree structure of the code. The terraform.tf file calls module 'region' presented in folder 'region'. Region calls the module 'resourcegroup' multiple times in folder 'resourcegroup'.

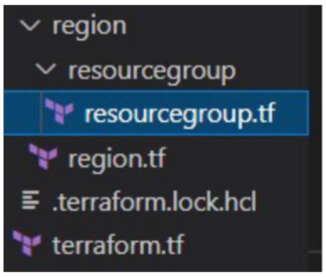


Figure 2: terraform.tf ➞ region.tf ➞ resourcegroup.tf

The region module is responsible for calling the resourcegroup three times. What to do if the output of the resourcegroup is not only needed in another module of the same level, but also in a module situated on an higher level. In our case the level of terraform.tf.

Terraform.tf calls the module region. As mentioned before, a module only returns an output if there are outputs declared inside the module. This implicits that at the region level, an output must be declared containing the values of the three resource-group names. However, a for_each cannot be used in an output declaration. For this purposes, Terraform created another way of looping: the **for-expression**.

```
module “resourcegroup” {
    for_each    = var.names
    source      = “./resourcegroup”
    region      = var.region
    name        = each.value
}

output “set_of_resource_group_names” {
    value = toset([ for rg in module.resourcegroup : rg.resource_group_name
])
}
```

Code block 10: for expression is used to gather all resource-groups names

In region.tf an output is created containing all the names of the created resource-groups. This is done with the for expression. It works as follows:

The for expression loops over each resourcegroup object and put that in a temp object called ‘rg’. The ‘rg’ object is just a temporary name that will be removed when the loop is done. From the object ‘rg’, it gets the resource_group_name and put that in a list. The toset() function converts the list into a set. When the loop is done a set of resource_group_names is created that can be used in terraform.tf.

Loops over loops over lo…

The previous example can be even more expanded. What if we want the same resource-groups in multiple regions. In practice infrastructures are shared among multiple regions. Most of the time these are exact copies, also called mirrors, of each other. This is pretty straightforward now with loops:

```
variable “regions” {
    type = set(string)
    default = [“westeurope”, “northeurope”]
}

module “main” {
    for_each                          = var.regions
    source                            = “./region”
    region                            = each.value
}
```

Code block 11: loop over regions in terraform.tf

You will see it already coming. How to pass now all the resource-group names of each region to a new module. Again the for expression is our saviour. The for expression can also be nested in Terraform. So a for loop can contain a for loop and that can contain a for loop… and so on:

```
variable “regions” {
    type = set(string)
    default = [“westeurope”, “northeurope”]
}

module “main” {
    for_each                            = var.regions
    source                              = “./region”
    region                              = each.value
}

module “otherResource” {
    source           = “./otherResource”
    set_of_rgnames = flatten([
                     for region in module.main : [
                        for rgname in region.set_of_resource_group_names :
rgname
                            ]
                        ])
}
```

Code block 12: loop over loops

A summary of the steps that happens for the set_of_rgnames:

1. To create a flat set of string values a new expression ‘flatten’ is used. This expression ‘flattens’ the values of both loops into one single set of values.
2. Next, the code loops over the regions. For each region it gets the ‘set of group names’.
3. It ends with the loop over the set of group names and returning the value of each group name.

Loop over a set of objects

As mentioned in the first chapter 'Loop over a singular resource', a for_each loop can only be used for sets/maps of strings. But what if you don't have a set of strings but only a set of objects. Then the set of objects must be converted to a set of maps that contains string keys and object values. This will be more clear in the following example:

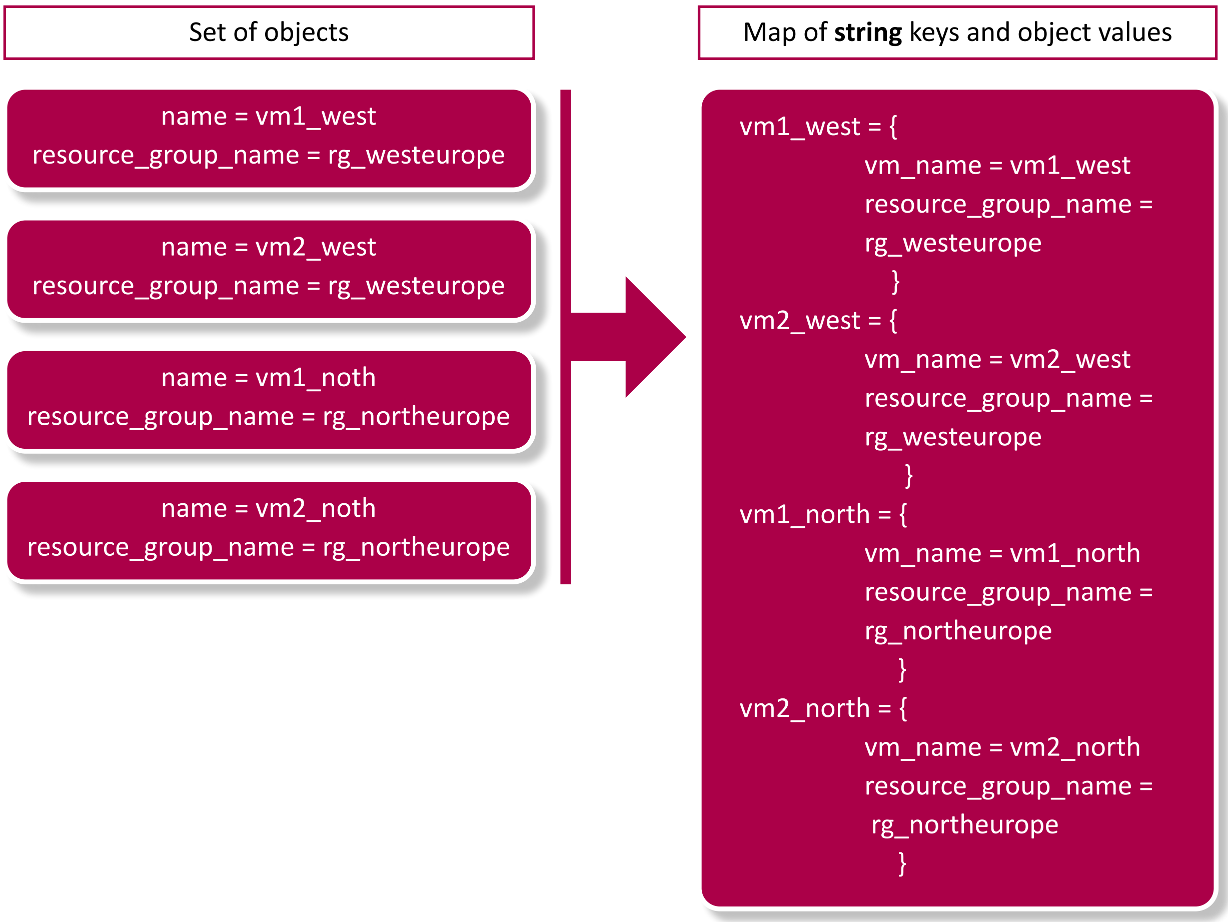
```
data "azurerm_virtual_machine" "get_vm_principalIDs" {
  for_each            = { for vm_object in var.vm_objects : vm_object.vm_
name => vm_object }
  name                = each.value.vm_name
  resource_group_name = each.value.resource_group_name
}

variable "vm_objects" {
    type = set(object({
      resource_group_name = string
      vm_name             = string
    }))
}
```

Code block 13: loop over a set of objects

The code block above shows a variable that contains a set of objects. The object contains two keys 'resource_group_name' and 'vm_name'. This variable represents the virtual machine names created created in both regions west- and north-Europe. For each virtual machine the principal id, also known as object id, will be tracked.

In the data block the magic happens. Here the for expression loops over every object. It get the vm_name of the object and set this vm_name as the key of the new created map. The value of the key is then, surprise…. the object itself. Because the value contains the object again, both vm_name and resource_group_name can be used in the data block after defining 'each.value'. Visualized it looks like the following:



Only use ‘known values’ in for_each loops

Terraform mentioned in the limitation page of the for each the following *‘The keys of the map (or all the values in the case of a set of strings) must be known values, or you will get an error message that for_each has dependencies that cannot be determined before apply, and a -target may be needed.’*

The above sentence actually applies a lot. Resources can need id’s of other resources. For example, to associate an virtual machine to an application gateway, the id’s of both resources are needed. However, the id of a resource is not an ‘**always known**’ value. Namely, the id of a resource will be determined after the creation of the resource and is not known before. Using id’s in a for_each will make the Terraform plan unstable. It depends on the existing of a resource how Terraform reacts.

For example, when there are three virtual machines already created and I use those id’s in a for_each. The Terraform code will succeed. The virtual machines exist already so Terraform is able to track down the id’s of these virtual machines. However, when a new fourth virtual machine is added to the code, the plan will fail. The fourth machine still has to be created. Therefore, Terraform cannot track down the id of that virtual machine.

Instead of id’s, it’s better to use the names of the resources. Names are already predefined by the user beforehand. Unless the virtual machine is created or not, Terraform will not fail because the name of the virtual machine is already known before creation. With a ‘**data**’ object Terraform can search for corresponding id of the resource. See the example below:

```
data “azurerm_virtual_machine” “get_vm_principalIDs” {
  for_each            = { for vm_object in var.vm_objects : vm_object.vm_
name => vm_object }
  name                = each.value.vm_name
  resource_group_name = each.value.resource_group_name
}

resource “azurerm_role_assignment” “apics_role_assignment” {
  for_each             = data.azurerm_virtual_machine.get_vm_principalIDs
  scope                =          azurerm_storage_container.apim_storage_
container.resource_manager_id
  role_definition_name = “Storage Blob Data Reader”
  principal_id         = each.value.identity[0].principal_id
}
```

```
variable “vm_objects” {
    type = set(object({
      resource_group_name = string
      vm_name             = string
    }))
}
```

Code block 14: loop over names, not id’s! The corresponding id’s you can get with the data object.

The above snippet of code tries to associate the principal_id’s of virtual machines to the container of the storage account. With this association our virtual machines can access the data inside the storage account. Instead of a set of principal_id’s a set of objects is given to the module. Each object contains the name of the virtual machine and the resource-group name. For each virtual machine the id is tracked down in the data block.

Conclusion

Loops are very powerful in Terraform. For example, looping over modules can created whole infrastructures with just a single line of code. However, because of their fragility, they must be well thought before using them in the code. When loops are implemented in the wrong way, resources can be easily destroyed and recreated again and because of that, the infrastructure will lose its high availability. The most important fist rule to use is 'always use for_each before count'. Use count only when there is no other possibility left. Furthermore, only loop over values that can be determined beforehand. When this is not the case, Terraform will show errors when new unknown resources are created in the future. Keeping these rules in mind, the loops will reduce lines of code in your Terraform code and make it much better readable.