TERRAFORM

SUMMARY

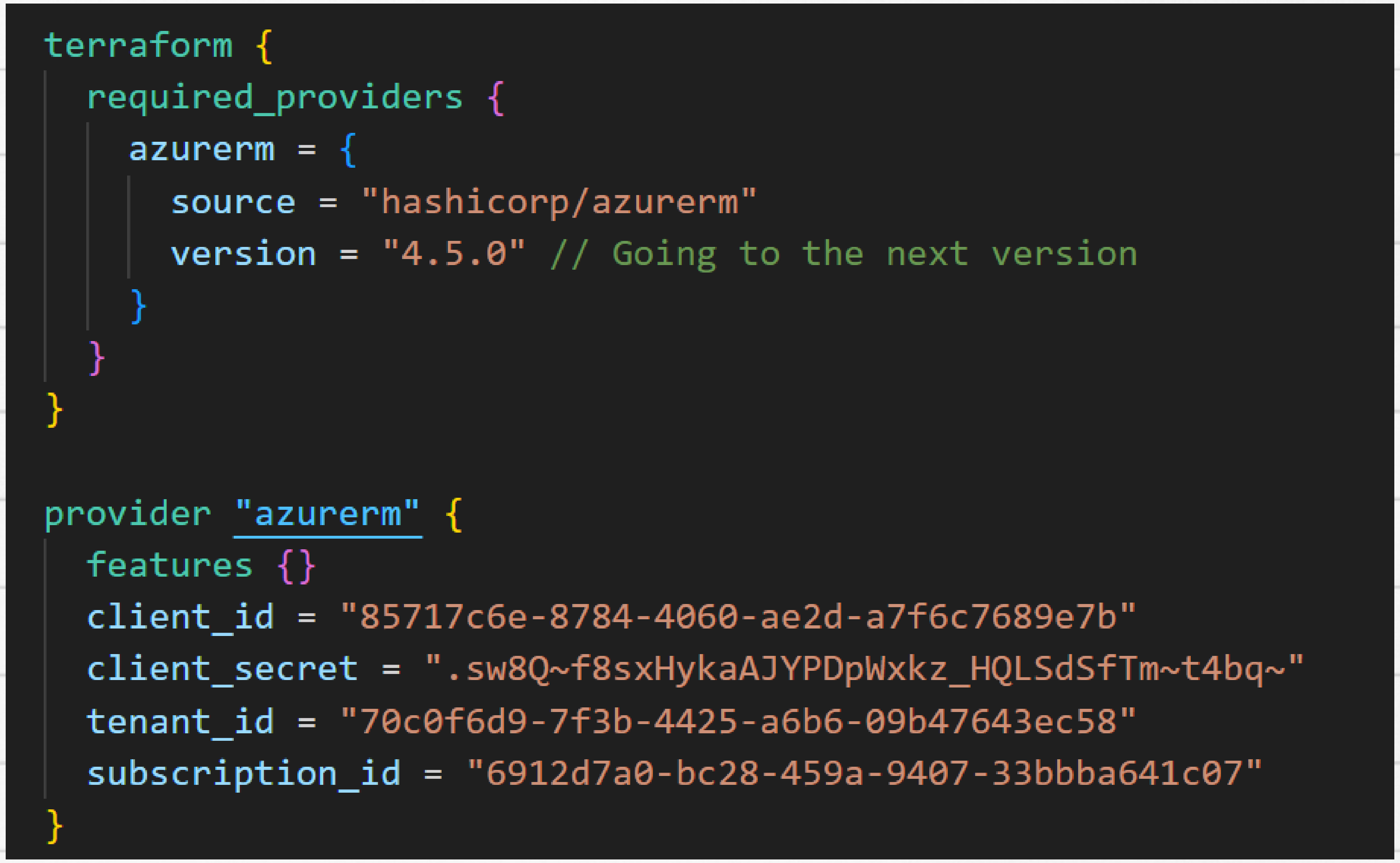
Terraform requires plugins that are called providers. This helps Terraform configuration files to interact with cloud providers and other Software as a service providers.

PROVIDERS

Terraform locates and installs the providers when you initialize a working directory.

The Terraform registry has a host of providers. Each provider is updated on a regular basis and has its own version number.

PROVIDERS



```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "4.5.0" // Going to the next version
    }
  }
}

provider "azurerm" {
  features {}
  client_id = "85717c6e-8784-4060-ae2d-a7f6c7689e7b"
  client_secret = ".sw8Q~f8sxHykaAJYPDpWxkz_HQLSdSfTm~t4bq~"
  tenant_id = "70c0f6d9-7f3b-4425-a6b6-09b47643ec58"
  subscription_id = "6912d7a0-bc28-459a-9407-33bbba641c07"
}
```

If the Terraform module requires any provider then it must be defined in the required_providers block.

Each provider will have its own configuration settings.

We can have multiple providers defined in the Terraform configuration file.

PROVIDERS

1) When you perform an initialization of your terraform directory via the terraform init command, terraform creates a dependency lock file named .terraform.lock.hcl .

2) This stores the dependencies of your terraform resources based on the installed version of the terraform providers.

3) It always good to include this file in version control because then you know what versions the configuration is dependent on.

4) If you want to upgrade the versions of the providers you can issue the command of terraform init -upgrade.

TERRAFORM COMMA

TERRAFORM INIT

1) This initializes the working directory for the Terraform configuration files .

TERRAFORM PLAN

1) This reads the existing objects on the target system, to see if they match the Terraform state .

2) It also compares the current configuration to the state and gives actions it will perform . It does not carry out any changes .

TERRAFORM APPLY

1) This executes the actions based on the terraform plan .

2) If you don't pass a saved plan file to the terraform apply command it will create a new execution plan .

Notes

1) You can use the Terraform init -upgrade command to upgrade to newer versions of your provider.

2) You can use the Terraform plan -refresh -only flag helps to just refresh the state files based on changes made to your infrastructure on your target system.

3) If you just execute the Terraform apply command without mentioning a plan file ,Terraform apply will first create a plan and then execute the plan.

TERRAFORM COMMA

TERRAFORM DESTROY

1) This destroys objects in the target system that is managed by the Terraform configuration .

2) You can run the terraform destroy command in plan mode to see the proposed changes .

```
terraform plan destroy
```

TERRAFORM VALIDATE

1) This only validates the configuration files in the directory . It does not make changes to the remote system .

TERRAFORM FOR

1) This is used to rewrite the Terraform configuration files to a canonical format and style.

2) The following options are present with the Terraform format command

a. -write=false - This does not overwrite the input files.
b. -diff - This displays diffs of formatting changes.
c. -check - Checks if the input is formatted.
d. -recursive - This also processes the files in the subdirectories

TYPES AND VALUES

string

This is a sequence of characters - "Azure"

number

This is used to represent a numerical value.

bool

This is used to represent a boolean value of either true or false.

list

This is a sequence of values ["North Europe","UK South"]

set

This is a collection of unique values that don't have secondary identifiers or ordering.

map

This is a map or object which is a group of values identified by labels {location="UK South"}

VARIABLES

1) Variables can be used to submit input values to Terraform configuration files. This makes the configuration code more reusable.

2) The input variable needs to be defined using a variable block.

3) For a variable you define the following
a. default - a default value for the variable
b. type - The type for the value
c. description - documentation for the input variable
d. validation - You can define rules to validate the input

4) The values for the variables are loaded in the following order
a. Environment variables
b. The terraform.tfvars or terraform.tfvars.json file
c. Any *. auto.tfvars or *. auto.tfvars.json file
d. Any -var and -var -file option specified in the command line

OUTPUT VALU

1) This makes information available in the command line about infrastructure in the target system.

2) The output value must be defined using an output block.

3) You can also suppress sensitive values in the CLI output.

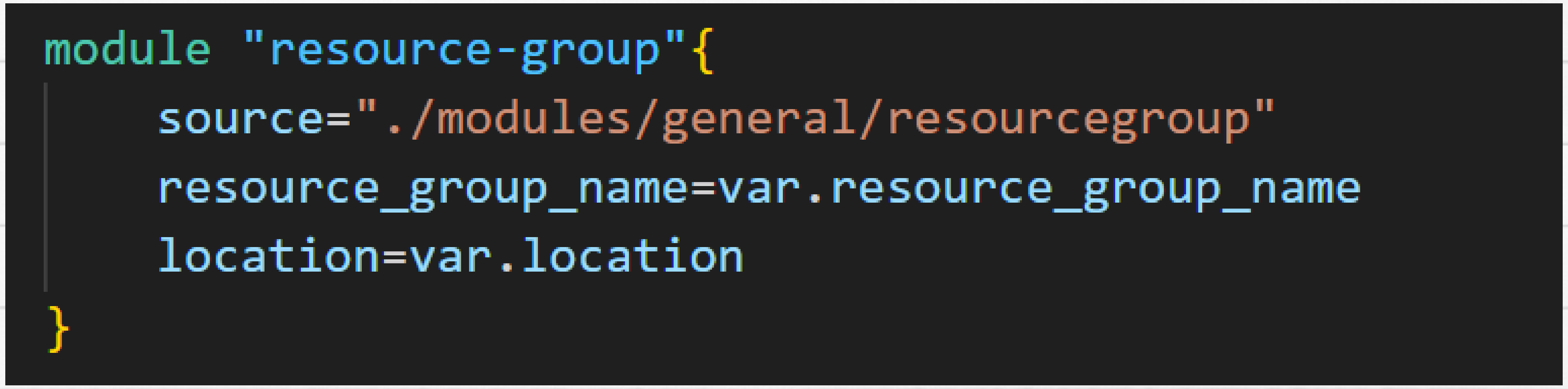
MODULES

1) These are containers for multiple resources and they are used together. A module consists of . tf or . tf.json files that are kept together in a directory

2) You can either store modules on your local machine. Or you can publish and then load modules from a public or private registry.

3) There are many modules available in the Terraform Registry.

4) When calling a child module, you specify the source of the module. When calling modules from a registry, you specify the version.



```
module "resource-group"{
    source="./modules/general/resourcegroup"
    resource_group_name=var.resource_group_name
    location=var.location
}
```

5) You then supply values for the input variables to the module.

MODULES

6) When you add, remove or modify terraform modules, you need to run terraform init so that terraform can use the modules.

7) You can use the count and for_each meta argument to create multiple instances of a module.

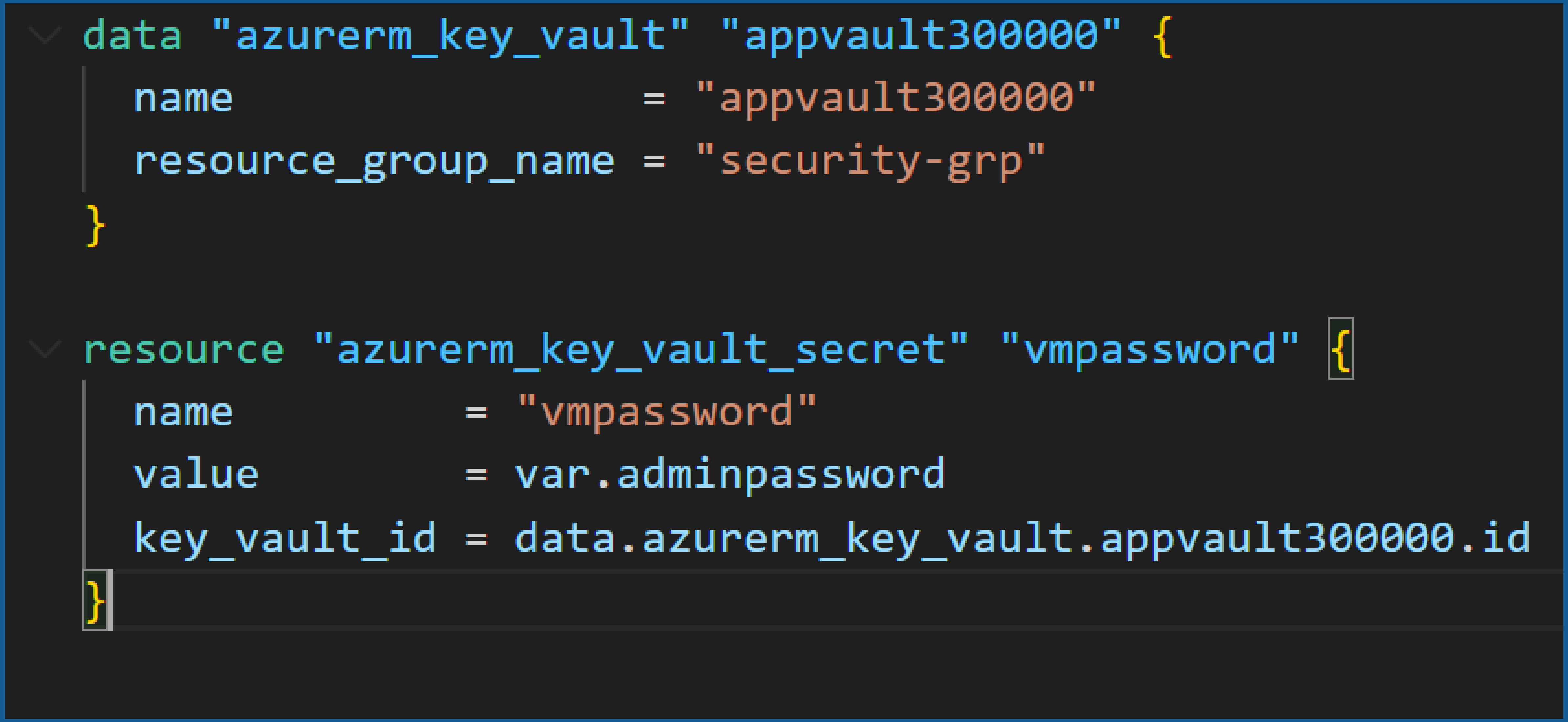
8) You can also use the depends_on clause to explicitly specify dependencies across modules.

9) Remember that the child modules cannot access the data from the calling modules and vice versa.

10) The calling modules should pass data to the child modules via the use of input variables. And the child module can pass data to the calling module via the use of Output values.

DATA SOURCES

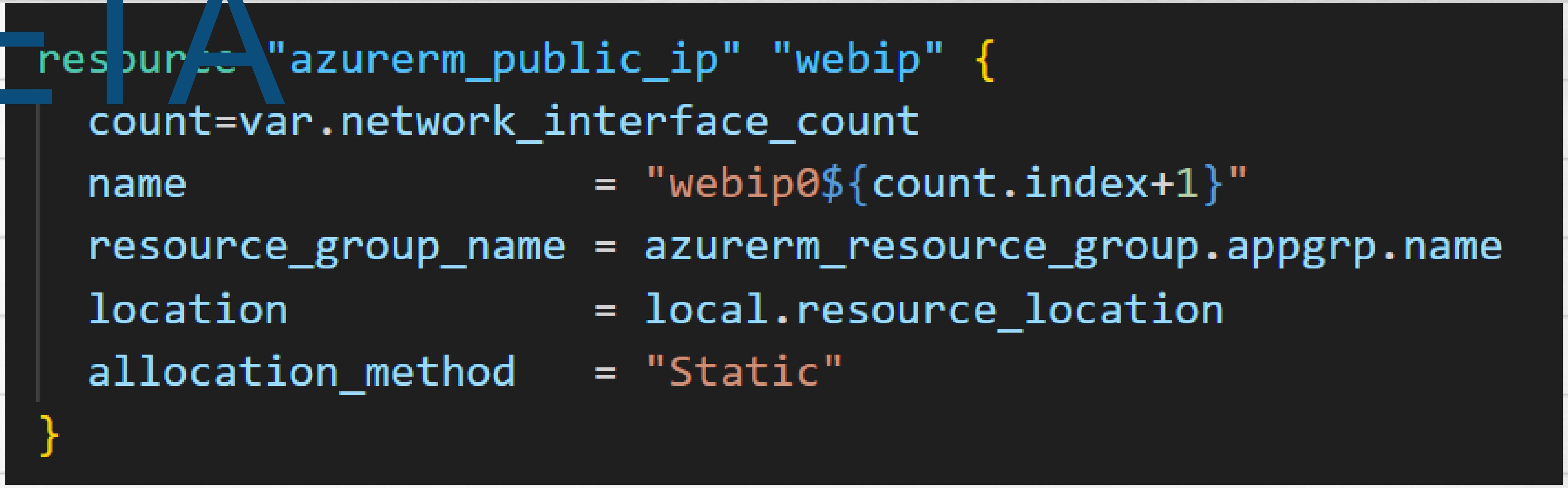
Data Sources help to use the information how resources in the target system that are not managed by Terraform.



```
data "azurerm_key_vault" "appvault300000" {
  name                = "appvault300000"
  resource_group_name = "security-grp"
}

resource "azurerm_key_vault_secret" "vmpassword" {
  name         = "vmpassword"
  value        = var.adminpassword
  key_vault_id = data.azurerm_key_vault.appvault300000.id
}
```

COUNT META ARGUEMENT



```
resource "azurerm_public_ip" "webip" {
  count=var.network_interface_count
  name                          = "webip0${count.index+1}"
  resource_group_name = azurerm_resource_group.appgrp.name
  location                      = local.resource_location
  allocation_method     = "Static"
}
```

1) The count meta argument can be used to create many instances of a resource or a module.

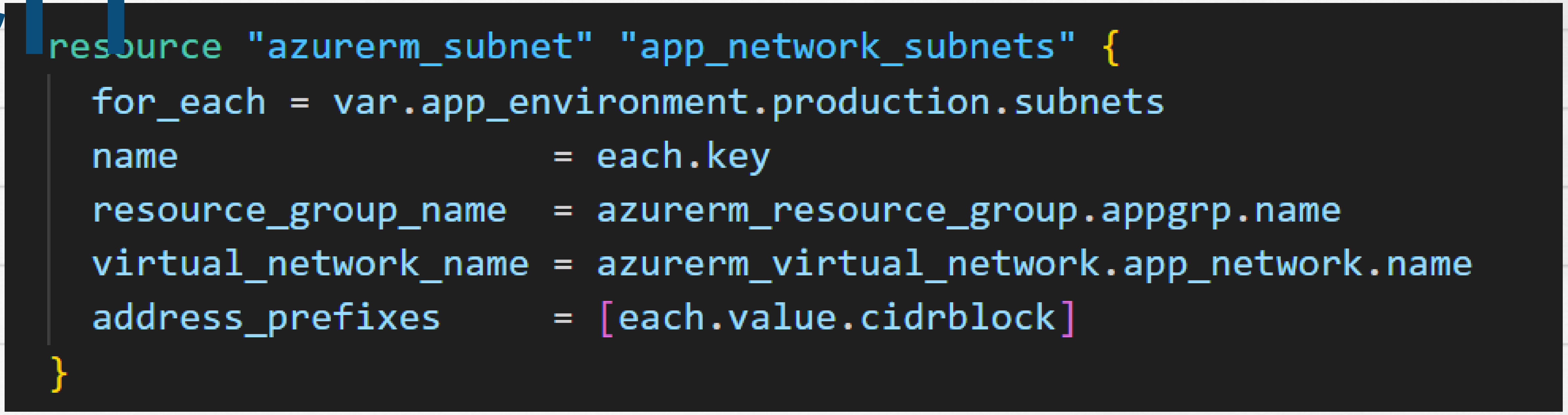
2) The count meta -argument accepts a number. This number needs to be known by Terraform before -hand. It cannot be based on a property of another resource.

3) The count.index expression is available to use within your resource block.

4) To access a Terraform resource, we then need to access it via the Index number.

E.g - azurerm_public_ip.webip [0], azurerm_public_ip.webip [1]

FOR_EACH META ARGUEMENT



```
resource "azurerm_subnet" "app_network_subnets" {
  for_each = var.app_environment.production.subnets
  name                 = each.key
  resource_group_name  = azurerm_resource_group.appgrp.name
  virtual_network_name = azurerm_virtual_network.app_network.name
  address_prefixes     = [each.value.cidrblock]
}
```

1) The for_each meta argument can be used to create many instances of a resource or a module.

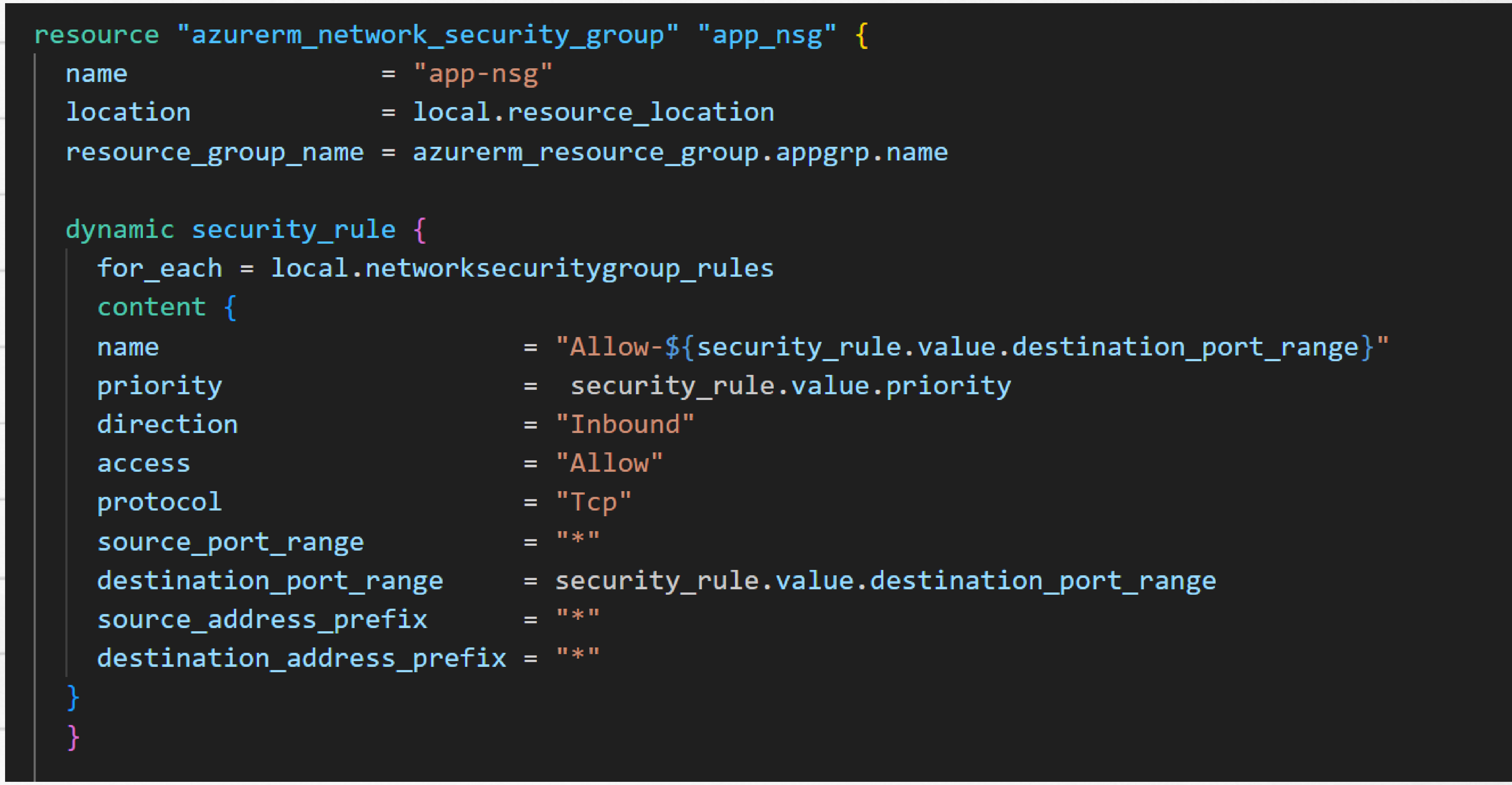
2) This accepts a map or a set of strings. It then creates an instance for each item in the map or set.

3) The element key and their values can be accessed via the expression - each.key and each.value .

4) To access a Terraform resource, we then need to access it via the key.

E.g - azurerm_subnet.app -network_subnets ["subnet 01 "], azurerm_subnet.app - network_subnets ["subnet 02 "]

DYNAMIC BLOCKS



```
resource "azurerm_network_security_group" "app_nsg" {
  name                = "app-nsg"
  location            = local.resource_location
  resource_group_name = azurerm_resource_group.appgrp.name

  dynamic security_rule {
    for_each = local.networksecuritygroup_rules
    content {
    name                        = "Allow-${security_rule.value.destination_port_range}"
    priority                    =  security_rule.value.priority
    direction                   = "Inbound"
    access                      = "Allow"
    protocol                    = "Tcp"
    source_port_range           = "*"
    destination_port_range      = security_rule.value.destination_port_range
    source_address_prefix       = "*"
    destination_address_prefix = "*"
  }
  }
```

1) If you have a repeatable argument setting in your resource block, you can iterate through the different configuration values using a dynamic block.

2) You can use the for_each meta -argument to process and iterate a complex value and substitute the elements in the dynamic block with those values.