|  |
| --- |
|  |
| Terraform  Learn to automate infrastructure with terraform |
| |  |  |  | | --- | --- | --- | | Rajagopalan | 11/24/19 | IAAS | |

Contents

[Author 4](#_Toc36639181)

[Erradata 4](#_Toc36639182)

[Disclaimer 4](#_Toc36639183)

[Foreword 4](#_Toc36639184)

[Infrastructure as Code 4](#_Toc36639185)

[Terraform 5](#_Toc36639186)

[Getting Started 5](#_Toc36639187)

[Installing in Windows 5](#_Toc36639188)

[Installing in Linux 5](#_Toc36639189)

[Verify Installation 6](#_Toc36639190)

[Let’s Build an Infrastructure 6](#_Toc36639191)

[Configuring AWS Provider 6](#_Toc36639192)

[Using AWS Credentials File 6](#_Toc36639193)

[ASSUME ROLES 9](#_Toc36639194)

[Environment Variables 10](#_Toc36639195)

[So, what are the input variables? 10](#_Toc36639196)

[Understanding Variables 11](#_Toc36639197)

[String 11](#_Toc36639198)

[List 11](#_Toc36639199)

[Map 11](#_Toc36639200)

[Booleans 12](#_Toc36639201)

[Example 12](#_Toc36639202)

[Environment Variables 12](#_Toc36639203)

[What are Locals ? 12](#_Toc36639204)

[Summary 12](#_Toc36639205)

[User-Defined Modules 13](#_Toc36639206)

[State File 15](#_Toc36639207)

[Purpose 15](#_Toc36639208)

[Mapping to real world 15](#_Toc36639209)

[Metadata 15](#_Toc36639210)

[Sensitive Data 16](#_Toc36639211)

[Backend and Remote State 16](#_Toc36639212)

[Here are the benefits of backend 16](#_Toc36639213)

[Configuring backend 16](#_Toc36639214)

[Terraform Cloud 16](#_Toc36639215)

[Configuration File Syntax 17](#_Toc36639216)

[AMAZON S3 17](#_Toc36639217)

[Workspaces 18](#_Toc36639218)

[Data Sources and Resources 19](#_Toc36639219)

[Interpolation Syntax 20](#_Toc36639220)

[Commands – CLI 20](#_Toc36639221)

[Apply 21](#_Toc36639222)

[Plan 22](#_Toc36639223)

[Now let us run a plan to delete the vpc, 24](#_Toc36639224)

[Terraform will perform the following actions: 26](#_Toc36639225)

[Taint 27](#_Toc36639226)

[Workspace 29](#_Toc36639227)

[Useful Functions 29](#_Toc36639228)

[Examples 31](#_Toc36639229)

[AWS VPC 31](#_Toc36639230)

[ACM import and base64decode function 33](#_Toc36639231)

[Provisioner 34](#_Toc36639232)

[chef 34](#_Toc36639233)

[file 36](#_Toc36639234)

[remote-exec 36](#_Toc36639235)

[Policy as Code 37](#_Toc36639236)

[Benefits 37](#_Toc36639237)

[Sentinel and Policy as Code 37](#_Toc36639238)

[Imports 38](#_Toc36639239)

[Enforcement Levels 38](#_Toc36639240)

[Example 38](#_Toc36639241)

[Mandatory Tags 38](#_Toc36639242)

# 

# Author

The author is an author and engineer, currently practising as devops engineer. Have been working as infra specialist for many projects.

# Erradata

Please email any erradata to [rajagopalanrg@outlook.com](mailto:rajagopalanrg@outlook.com)

# Disclaimer

This book is presented solely for educational purposes. The author is not offering it as legal, accounting, other professional services advice. While best efforts have been used in preparing this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein.

# Foreword

This book is primarily for beginners in terraform. It extensively uses AWS environment to demonstrate the examples. All the examples provided in this book are executed and tested, thereby users can also try them. Unlike other terraform learning sources, in this book I tried to cover the untouched concepts in terraform like terraform cloud and sentinel policies briefly. I will be explaining in another book on using terraform for other providers and sentinel policies in detail with more examples. Please support me with your valuable feedback to improvise myself. I had read the documentation of terraform from <https://www.terraform.io/docs/index.html> to understand and practiced all the examples provided in this documentation. For detailed information, please refer to the terraform documentation.

# Infrastructure as Code

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

## Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

# Getting Started

## Installing in Windows

To install Terraform, go to the link <https://www.terraform.io/downloads.html> and download

file for your workstation. Download the file respective for you operating system.

After downloading Terraform, unzip the package. Terraform runs as a single binary named terraform. Any other files in the package can be safely removed and Terraform will still function.

The final step is to make sure that the terraform binary is available on the PATH. Follow the instructions given below.

1. Open windows explorer and right click on “This PC”
2. Select Properties
3. Select Advanced options
4. Click on Environment Variables
5. Click “Path” and Select “New”
6. Copy the path where terraform binary is located
7. Paste and save it

## Installing in Linux

Copy the terraform zip file to linux box and unzip it using unzip command. The unzip utility, if not available by default, can be downloaded and installed with repository command, apt or yum

apt install unzip

yum install unzip

Once the unzip utility is installed, unzip the downloaded terraform package and change permissions for the terraform executable.

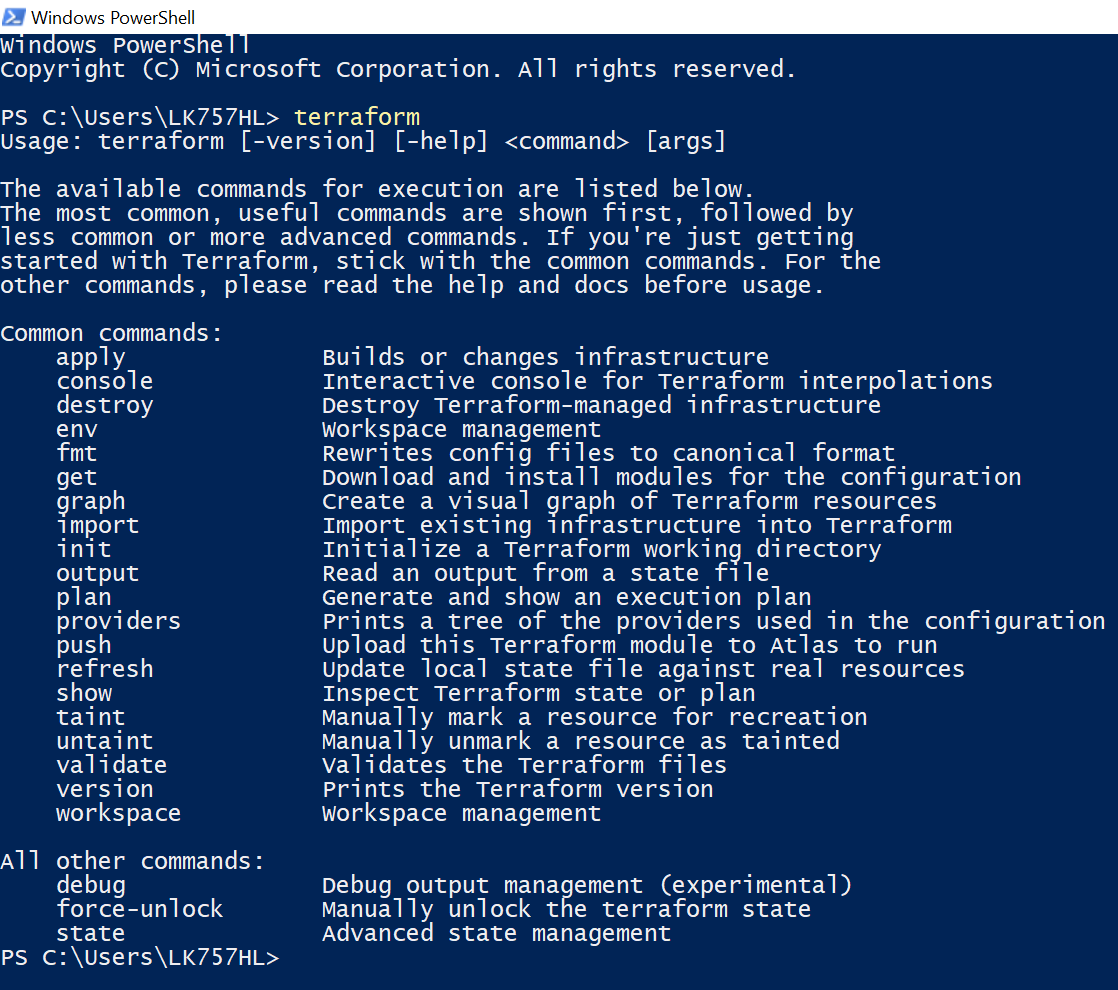
Chmod +x terraform

Copy the terraform executable to the /usr/local/sbin folder.

Cp terraform /usr/local/sbin/

## Verify Installation

After installing Terraform, verify the installation worked by opening a new cmd or Powershell prompt and checking that terraform is available. By executing terraform you should see help output similar to this:



# Let’s Build an Infrastructure

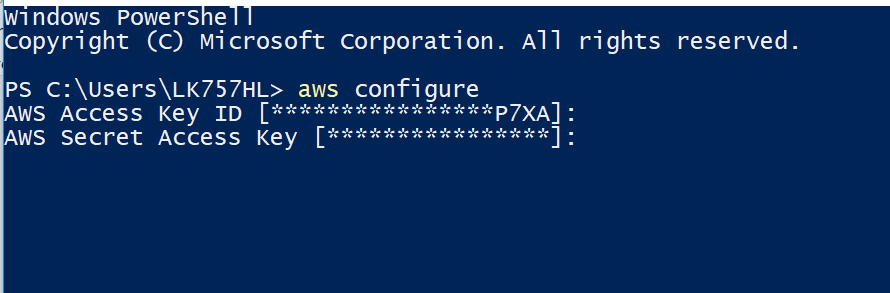
## Configuring AWS Provider

#### Using AWS Credentials File

For the training purpose, students are required to create an aws free-tier account. Once the account is created, download the awscli . Follow this link to get awscli setup in your computer, <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>

Once awscli is installed, configure your environment with “aws configure”. This will save your aws access key and secret access key in the path, %UserProfile%\.aws\credentials.

The below picture is taken with an existing aws configuration. A fresh configuration will not have any aws access key and secret key stored.



Terraform files have an extension “.tf”. Save the following content to a file and save it as “filename.tf” in your local directory.

variable "subnet" {

  type="string"

}

variable "ami" {

  type="string"

}

variable "sgID" {

  type="string"

}

variable "vmName" {

  type ="string"

  default = "linuxami"

}

variable "key" {

  type="string"

}

provider "aws" {

    shared\_credentials\_file = "%aws\_cred%"

    region = "us-east-1"

    version = "~> 2.0"

}

resource "aws\_instance" "name" {

  ami           = var.ami

  instance\_type = "t2.micro"

  subnet\_id     = var.subnet

  security\_groups = [var.sgID]

  associate\_public\_ip\_address = true

  key\_name = var.key

  tags  {

    Name = var.vmName

  }

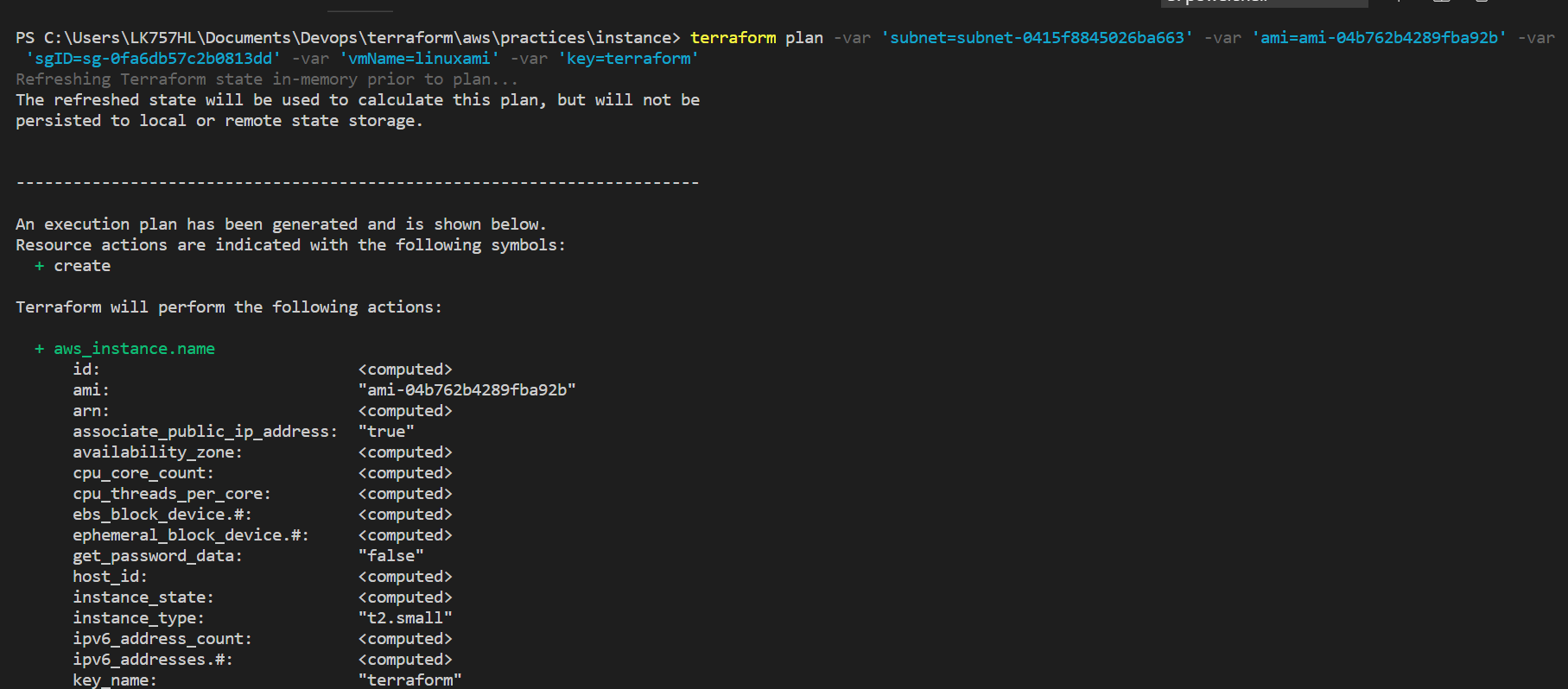
}

The credentials to the aws account are referred by the variable %aws\_cred%, which read it from %UserProfile%\.aws\credentials

The “provider” makes connection with the aws account and “resource” is the basic building block of all resources in any cloud provider. We are going to build an ec2 instance using the “aws\_instance” resource. The minimum required parameters to create instance (VM) are the ami ID, the instance type, the subnet ID, security groups, key\_name. tags and associate\_public\_ip\_address.

Once the file is saved, open powershell window and set current working directory to the path where you save the file. Run “terraform init” so terraform will download the binary required for aws provider into the current location and initialize the environment.

Run “terraform plan” to see that terraform builds a plan of what are all the resources that are going to be created and associated specifications of that resource.



The parameters can be passed through a parameter file as given below

subnet = "subnet-0415f8845026ba663"

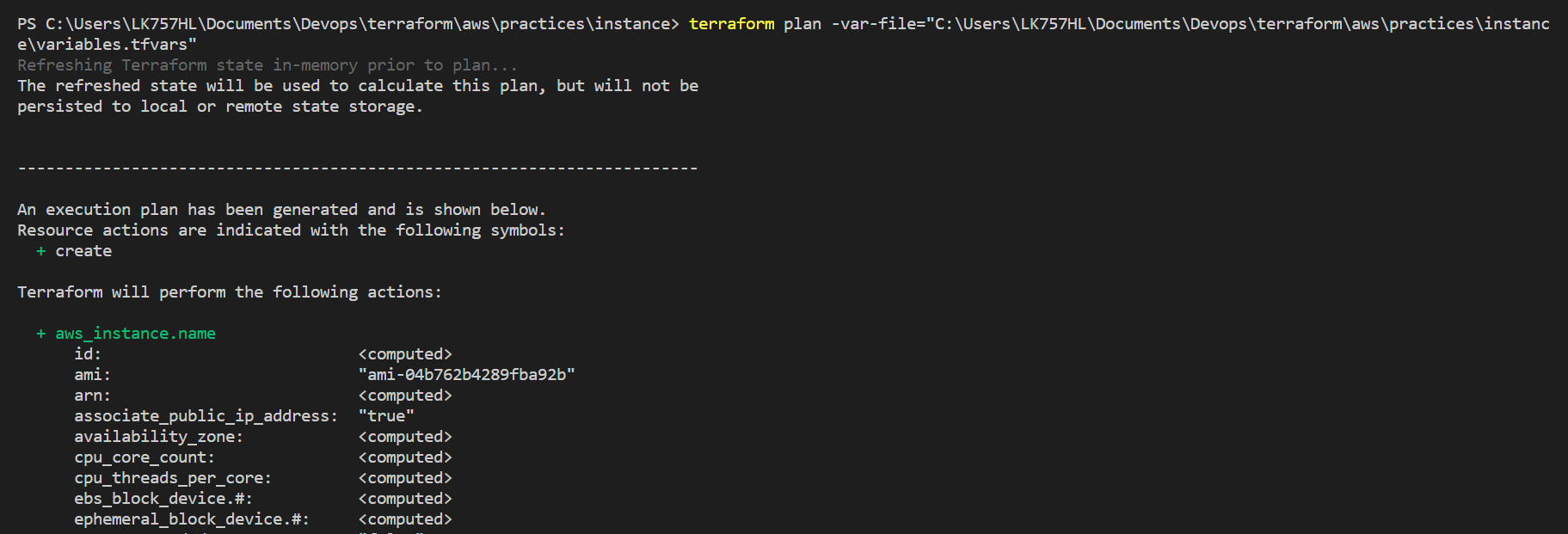
ami = "ami-04b762b4289fba92b"

sgID = "sg-0fa6db57c2b0813dd"

vmName = "linuxami"

key = "terraform"

Save the above contents in a file called variables.tfvars in the same working directory and then execute terraform plan with option, -var-file=”path”



#### ASSUME ROLES

If provided with a role ARN, Terraform will attempt to assume this role using the supplied credentials. This is the recommended method of all available methods of configuring aws provider and the policy must contain, sts:AssumeRole. To make this work, create a user in AWS IAM and attach to him a policy that control access to the intended services. Configure the awscli with access key and secret access key of the new user created in IAM. Create a role with trust policy and using the aws sts assume-role, assume the role to the IAM user configured in the workstation.

Follow the below steps:

aws iam create-user --user-name terraformuser

create a policy similar to this

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": “ec2:\*”

"Resource": "\*"

}

]

}

Attach the policy to the user created

1. aws iam create-policy --policy-name ec2-policy --policy-document <file://C:\documents\policy\ec2policy.json>
2. aws iam attach-user-policy --user-name Bob --policy-arn "arn:aws:iam::123456789012:policy/ec2-policy"

IN the second command, replace the account number 123456789012 with your account number.

Create a trust policy as below

{

"Version": "2012-10-17",

"Statement": {

"Effect": "Allow",

"Principal": { "AWS": "arn:aws:iam::123456789012:root" },

"Action": "sts:AssumeRole"

}

}

Create a role and attach any policy to the role

1. aws iam create-role --role-name ec2-role --assume-role-policy-document <file://C:\documents\policy\ec2trustpolicy.json>
2. aws iam attach-role-policy --role-name example-role --policy-arn "arn:aws:iam::aws:policy/AmazonRDSReadOnlyAccess"

Configure terraformuser access\_key and secret\_access\_key as described in earlier sections

Run aws sts get-caller-identity and verify the commands executed from this workstation is using terraformuser

Coming back to terraform, configure your provider as below:

provider "aws" {

  region     = "us-east-1"

  assume\_role {

    role\_arn = "arn:aws:iam::123456789012:role/ec2-role"

    session\_name = "ec2-role"

  }

}

The assume\_role uses the aws sts service and assume the role with admin access to the IAM user configured in the system.Always ensure the aws credentials is configured in the workstation before assume role.

Simply, when awscli is configured in your workstation, just a provider block is enough.

provider "aws" {

  region     = "us-east-1"

}

#### Environment Variables

This is a more simple approach. Configure your aws credentials as environment variables as below

export AWS\_ACCESS\_KEY\_ID="anaccesskey"

export AWS\_SECRET\_ACCESS\_KEY="asecretkey"

export AWS\_DEFAULT\_REGION="us-west-2"

# So, what are the input variables?

Generally, input values for terraform are supplied through variables. For a root module, configuration variables can be set from CLI or environment variables. For child modules, they can be passed on from parent to child.

Example for a variable declaration:

variable "key" {

  type="string"

  default = "terraform"

}

# Understanding Variables

Each variable block configures a single input. The name of the variable is given in the header and the name can be any valid identifier except the names source, version and providers.

The body of the variable block is the actual configuration of it. The following are the arguments found inside the body

1. type: It defines the type of input and the valid values are string, list and map. If the type is not specified, the type of the default value will be inferred. If the default value is also not present, then the type will be a string.
2. default: This sets a default value of a variable. While running terraform plan or apply, if no value is supplied by user for the variable, then the default value will be taken. The type of default value should match with the type specified.
3. description: A human readable description of the variable.

## String

variable "subnet" {

  type="string"

  default = "subnet-0415f8845026ba663"

}

## List

variable "subnetID" {

  type = "list"

default = [{

    "cidr" = "10.8.0.0/28"

    "name" = "sbn01"

},

{

    "cidr" = "10.8.0.16/28"

    "name" = "sbn02"

}]

}

## Map

variable "type" {

  type = "map"

  default = {

      dev = "t2.micro"

      prod = "t2.small"

  }

}

## Booleans

Although Terraform can automatically convert between boolean and string values, there are some subtle implications of these conversions that should be completely understood when using boolean values with input variables.

It is recommended for now to specify boolean values for variables as the strings "true" and "false", to avoid some caveats in the conversion process.

## Example

variable "active" {

default = false

}

## Environment Variables

Environment variables can be used to set values for terraform variables in the root module. The name of the variable should be TF\_VAR\_<variable name> .

For example, if there is a variable,

variable “myimage” then environment variable can be set as TF\_VAR\_myimage=just

# What are Locals ?

Locals are like variables inside a function in a programming language, that can be used several times within a module.

locals {

  common\_tags = {

    purpose = "cloudLego"

  }

}

resource "azurerm\_local\_network\_gateway" "lngw1" {

  name                = "lngw1"

  resource\_group\_name = var.resourceGroupName

  tags                = local.common\_tags

}

#### Summary

**Variables** are in other words, the **parameters** for the terraform template and **locals** are the **variables** used inside the template.

# User-Defined Modules

Let us begin with an example

provider "aws" {

    shared\_credentials\_file = "%aws\_cred%"

    region = "us-east-1"

    version = "~> 2.0"

}

terraform{

    backend "s3" {

        bucket = "terraformbackendskillforge"

        key    = "demo/terraform.tfstate"

        region = "us-east-1"

    }

}

module "vpc" {

  source = "./modules/vpc"

  subnet\_cidr\_block = var.subnet\_cidr\_block

  vpc\_cidr\_block = var.vpc\_cidr\_block

  azs = var.azs

  region = var.region

  vpcname = var.vpcname

}

output "vpcID" {

  value = module.vpc.vpcID

}

output "dev\_subnet\_id" {

  value = module.vpc.subnetIDs

}

output "dev\_sg\_id" {

  value = module.vpc.sgID

}

Variables

vpcname = "skillforgeVPC"

vpc\_cidr\_block = "10.8.0.0/24"

subnet\_cidr\_block = [{

    "cidr" = "10.8.0.0/28"

    "name" = "sbn01"

},

{

    "cidr" = "10.8.0.16/28"

    "name" = "sbn02"

}]

azs = "us-east-1a"

**Shown below is a partial output of the plan, to illustrate how modules are being called.**

PS C:\Users\username\Documents\Devops\terraform\aws> terraform plan -out=C:\Users\USERNAME\Documents\Devops\terraform\aws\localplan.tfstate -var-file=C:\Users\USERNAME\Documents\Devops\terraform\aws\variables.tfvars

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

------------------------------------------------------------------------

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

+ **module.vpc.aws\_internet\_gateway.int\_gtw**

id: <computed>

owner\_id: <computed>

tags.%: "1"

tags.Name: "Devops Gtw"

vpc\_id: aws\_vpc.main.id

+ module.vpc.aws\_route\_table.route\_table\_for\_public\_subnet

id: <computed>

owner\_id: <computed>

propagating\_vgws.#: <computed>

route.#: "1"

tags.Name: "Devops Route Table"

vpc\_id: aws\_vpc.main.id

+ module.vpc.aws\_route\_table\_association.route\_table\_subnet[0]

id: <computed>

route\_table\_id: aws\_route\_table.route\_table\_for\_public\_subnet.id

subnet\_id: element(aws\_subnet.public.\*.id, count.index)

+ module.vpc.aws\_route\_table\_association.route\_table\_subnet[1]

id: <computed>

route\_table\_id: aws\_route\_table.route\_table\_for\_public\_subnet.id

subnet\_id: element(aws\_subnet.public.\*.id, count.index)

Plan: 8 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

This plan was saved to: C:\Users\USERNAME\Documents\Devops\terraform\aws\localplan.tfstate

***To perform exactly these actions, run the following command to apply:***

***terraform apply "C:\\Users\\USERNAME\\Documents\\Devops\\terraform\\aws\\localplan.tfstate"***

**When terraform plan is run with the -out=<filename> option, the plan is stored in local storage, which can then be used to apply the changes.**

PS C:\Users\USERNAME\Documents\Devops\terraform\aws> terraform apply C:\Users\USERNAME\Documents\Devops\terraform\aws\localplan.tfstate

You can open the terraform state file to see what are all the resources configured and their current state.

# State File

Terraform stores information about the infrastructure being provisioned or planned to provision in a file called the state file. This state file is then used by terraform to map the actual infra with the infra template to make sure you are not deviating and to avoid executing same code blocks when the resource is already present. It will only do the modifications if there any updated in the template instead of building the infra again.

This state file is stored as terraform.tfstate or any filename.tfstate in your local disk. But can also be stored remotely which helps in working with a team.

## Purpose

#### Mapping to real world

Terraform uses the state file to map the configuration with the real world. For example, if you are creating an aws instance then terraform should be able to identify this instance with the instance id created in AWS. For this purpose terraform uses its own state file which actually serves as a database for it.

#### Metadata

Alongside mapping to real world, terraform must also store metadata such as resource dependencies. For example, when you create a VM in any cloud environment, you would first require a Virtual network and a subnet. So terraform stores this information in the state file and is used when cleaning up the resources. To delete a subnet, it should know that, first the VM should be deleted and then proceed with deletion of subnet. This dependency priority is achieved with the help of state file. Also when working with multiple cloud providers, for example when you setup a vpn between azure and aws, you will write codes for both cloud providers. At this stage terraform must know the priority between the providers. For this purpose, the state file is being used.

#### Sensitive Data

When state information is stored, it may contain sensitive information like DB passwords, which when exposed may cause a potential threat. Local state files are plain json files which exposes sensitive information to public. To prevent this happen, define how you want to use your sensitive information. Recommended practice is to use a remote state, where the remote state is only held in memory and persisted to local disk. The encryption of data in remote state depends on the backend providers, like S3, azure storage account and terraform cloud.

# Backend and Remote State

Backend in terraform denotes where the state file is loaded and how the operation such as *apply*  is executed. By default, terraform uses the “local” backend but you can choose your backend to be I any of the supported cloud platform.

## Here are the benefits of backend

1. when working in a team, backend helps achieve lock over the state when multiple users are working to avoid multiple writes and hence avoid corruption. Also some backends support versioning such as S3, terraform cloud.
2. When sensitive information is contained in the code, it is better to use remote backend to keep them off the disk
3. Some of the backends supports remote operations when operations can take a longer time to execute. Backend paired with locking helps in team environments.

# Configuring backend

Backends can be configured in terraform directly in the terraform section. Once it is done run *terraform init* to initialize it.

Let’s see configuring backend in two different providers, Terraform cloud and AWS S3.

## Terraform Cloud

The terraform cloud supports versioning of the state file and also remote execution. The url to access, app.terraform.io

terraform {

  backend "remote" {

    hostname = "app.terraform.io"

    organization = "my-org"

    workspaces {

      name = "my-workspace"

    }

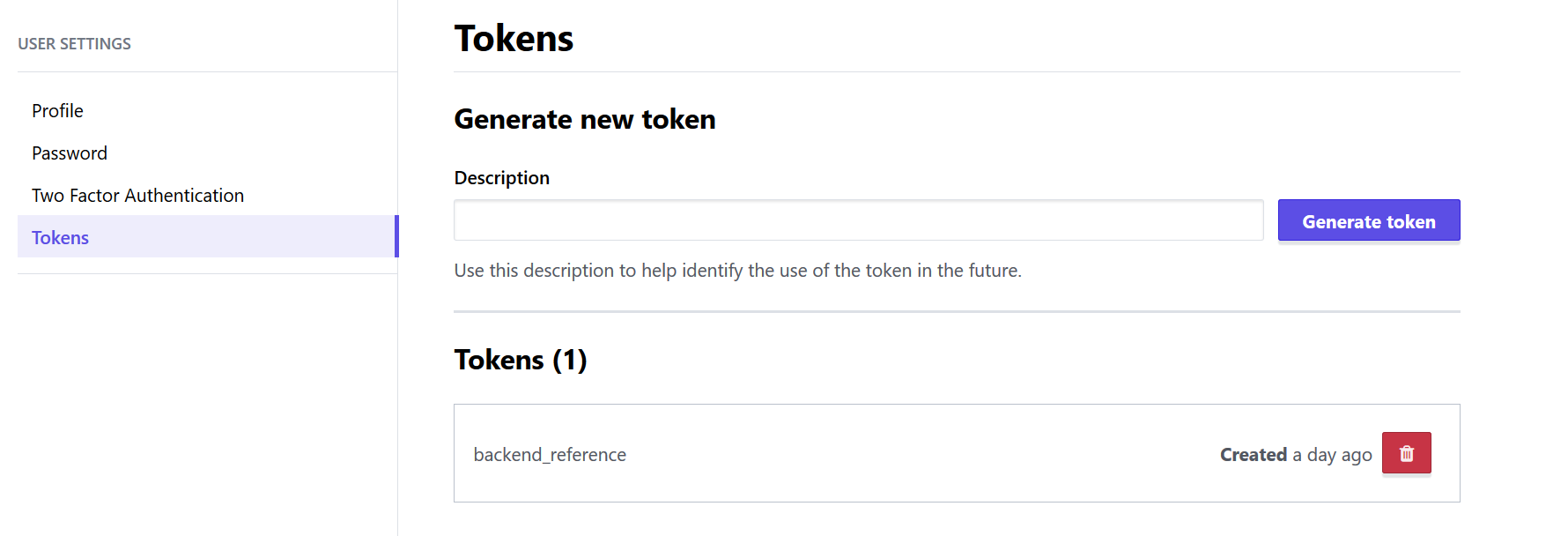
  }

}

When backend is configured as *remote* it implies either terraform cloud or terraform Enterprise.

The mandatory information required to configure are the organization, token and workspaces. In the terraform cloud everything resides inside an organization. The organization will contain multiple workspaces which will contain its own state files and remote runs.

To authenticate your terraform with the backend, you have to generate a token from the user settings, as given below.



This token can be provided in the terraform file or as in a CLI configuration file(terraform.rc or .terraformrc). On windows the file name should be terraform.rc and stored in %APPDATA% location of the current user. To identify this location, run the command, *$env:APPDATA*  in powershell.

On all other systems, the file must be named .terraformrc (note the leading period) and placed directly in the home directory of the relevant user.

#### Configuration File Syntax

disable\_checkpoint = true

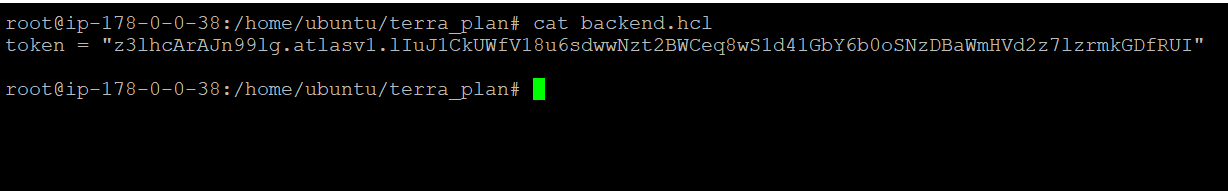
credentials "app.terraform.io" {

token = "xxxxxx.atlasv1.zzzzzzzzzzzzz"

}

Once it is added, run *terraform init*  to initialize the backend.

The credentials can also be stored in a backend file called, backend.hcl as below,



You can retrieve the attributes of the resources from the remote state using the *data* option.

data "terraform\_remote\_state" "foo" {

  backend = "remote"

  config = {

    organization = "CloudLego"

    workspaces = {

      name = "sitetosite-dev"

    }

  }

}

## AMAZON S3

terraform{

    backend "s3" {

        bucket = "terraformbackendraja"

        key    = "demo/terraform.tfstate"

        region = "us-east-1"

    }

}

# 

# Workspaces

For example, if you have a requirement to create two instances for dev and prod environment with different instance types for each environment. Lets say, dev needs a micro instance and prod needs a large instance, how can this be handled ? A good idea is to use the workspace. A workspace is a logical division to store the state of your infrastructure.

1. Create two workspaces called dev and prod
2. Create a variable of type map and have values for both env
3. Use the lookup function

Eg.,

terraform workspace new prod

terraform workspace new dev

To select the required workspace, use

Terraform workspace select prod

The default workspace for terraform is “default”

variable "type" {

  type = "map"

  default = {

      dev = "t2.micro"

      prod = "t2.small"

  }

}

The values can then be accessed using the lookup function

resource "aws\_instance" "name" {

  ami           = var.ami

  instance\_type = lookup(var.type, terraform.workspace)

  subnet\_id     = var.subnet

  security\_groups = [var.sgID]

  associate\_public\_ip\_address = true

  key\_name = var.key

  tags  {

    Name = var.vmName

  }

}

Then execute terraform plan and apply to get your expected result.

# Data Sources and Resources

If we go through the link, <https://www.terraform.io/docs/providers/aws/index.html>, we come to know that each component in aws has two links to follow, *data sources* and *resources.* The data sources are basically the search query with which you can find the component and its details from the cloud.

For example,

data "aws\_ami" "ubuntu" {

    most\_recent = true

    filter {

        name   = "name"

        values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-\*"]

    }

    filter {

        name   = "virtualization-type"

        values = ["hvm"]

    }

    owners = ["065045723570"] # Canonical

}

The above filter finds the ami id of an ubuntu server.

Similarly you can use this *data*  to get other resources or components of every cloud. The syntax for this code block is

data <resourcename> <identifier>{

Attributes

}

The data block for each resource will have its own attributes, so you have to look into the documentation before writing the code block for your requirement.

The *resourcename*  will be the same name used for *data* and *resource* blocks. The terraform documentation for each resource block also can be seen with a difference for both code blocks as follows

<https://www.terraform.io/docs/providers/aws/d/ami.html> - data

<https://www.terraform.io/docs/providers/aws/r/ami.html> - resource

On the other hand *resource* block is used to create the resource, for which we have seen a lot of examples while we were discussing the previous topics. To know more about the blocks for every resources, you can check in the terraform documentation similar to the above provided links.

# Interpolation Syntax

It is nothing but how you are calling or interpolate other values in your code, either in json or terraform format. You can refer a variable, attributes of resources, call functions, etc.

**Examples**

String Variable: ${var.variable}.

Map Variable: ${var.amis[“us-east-11”]}

List Variable: ${var.subnets} or ${var.subnets[index]}

Attributes of other resources: ${aws\_instance.instance1.id}

Attributes of own resource: ${self.private\_ip}, i.e., to access attribute of a resource within its block.

Function: ${length(var.subnetID)}

Attributes of data sources: ${data.aws\_ami.ami.id}

Output of other module: ${module.outputname.value}

Count Information: ${count.index}

Path Information: ${path.type}, i.e., ${path.root}, ${path.module}, ${path.cwd}, where cwd – current working directory, module – path of the module, root – path of the root module.

Terraform Meta Information: These are basically the meta data of the terraform environment, like env, workspace and so on. ${terraform.workspace}

Conditionals: ${var.env == "production" ? var.prod\_subnet : var.dev\_subnet}

# Commands – CLI

Find below the list of common commands of terraform cli

apply Builds or changes infrastructure

console Interactive console for Terraform interpolations

destroy Destroy Terraform-managed infrastructure

env Workspace management

fmt Rewrites config files to canonical format

get Download and install modules for the configuration

graph Create a visual graph of Terraform resources

import Import existing infrastructure into Terraform

init Initialize a Terraform working directory

output Read an output from a state file

plan Generate and show an execution plan

providers Prints a tree of the providers used in the configuration

refresh Update local state file against real resources

show Inspect Terraform state or plan

taint Manually mark a resource for recreation

untaint Manually unmark a resource as tainted

validate Validates the Terraform files

version Prints the Terraform version

workspace Workspace management

Most of the commands are self-explanatory and we will discuss in detail on a few important commands

## Apply

This command will apply the changes that are supposed to be performed on the environment. The command can be run either with the template file or with the plan which we obtain from *plan* command.

The attributes that belong to apply are,

Options:

-backup=path Path to backup the existing state file before

modifying. Defaults to the "-state-out" path with

".backup" extension. Set to "-" to disable backup.

-auto-approve Skip interactive approval of plan before applying.

-lock=true Lock the state file when locking is supported.

-lock-timeout=0s Duration to retry a state lock.

-input=true Ask for input for variables if not directly set.

-no-color If specified, output won't contain any color.

-parallelism=n Limit the number of parallel resource operations.

Defaults to 10.

-refresh=true Update state prior to checking for differences. This

has no effect if a plan file is given to apply.

-state=path Path to read and save state (unless state-out

is specified). Defaults to "terraform.tfstate".

-state-out=path Path to write state to that is different than

"-state". This can be used to preserve the old

state.

-target=resource Resource to target. Operation will be limited to this

resource and its dependencies. This flag can be used

multiple times.

-var 'foo=bar' Set a variable in the Terraform configuration. This

flag can be set multiple times.

-var-file=foo Set variables in the Terraform configuration from

a file. If "terraform.tfvars" or any ".auto.tfvars"

files are present, they will be automatically loaded.

## Plan

The command which will read the template and gives an outline of what is going to be deployed which includes creation, deletion and modification of resources.

The attributes of this command are,

Options:

-destroy If set, a plan will be generated to destroy all resources

managed by the given configuration and state.

-detailed-exitcode Return detailed exit codes when the command exits. This

will change the meaning of exit codes to:

0 - Succeeded, diff is empty (no changes)

1 - Errored

2 - Succeeded, there is a diff

-input=true Ask for input for variables if not directly set.

-lock=true Lock the state file when locking is supported.

-lock-timeout=0s Duration to retry a state lock.

-no-color If specified, output won't contain any color.

-out=path Write a plan file to the given path. This can be used as

input to the "apply" command.

-parallelism=n Limit the number of concurrent operations. Defaults to 10.

-refresh=true Update state prior to checking for differences.

-state=statefile Path to a Terraform state file to use to look

up Terraform-managed resources. By default it will

use the state "terraform.tfstate" if it exists.

-target=resource Resource to target. Operation will be limited to this

resource and its dependencies. This flag can be used

multiple times.

-var 'foo=bar' Set a variable in the Terraform configuration. This

flag can be set multiple times.

-var-file=foo Set variables in the Terraform configuration from

a file. If "terraform.tfvars" or any ".auto.tfvars"

files are present, they will be automatically loaded.

Both *apply* and *plan*  have almost same attributes except few. Plan can also be done for creation and deletion

Example *plan* for creation is already discussed in previous pages, so let us see one example to create a plan for deletion.

The below example is written in terraform version > 0.12

variable "azureSpnID" {

  type = string

  default = "783619fa-4f5f-4502-ae27-49309ded97c8"

}

variable "azureSpnSecret"{

  type = string

  default = "s6b5/]AWQsl0wFurngl0s@cGpeNyija@"

}

variable "aws\_access\_key\_id" {

  type = string

}

variable "aws\_secret\_access\_key" {

  type = string

}

variable "vpcCIDR" {

  type    = string

  default = "10.10.0.0/16"

}

provider "aws" {

  shared\_credentials\_file = "%aws\_cred%"

  region                  = "us-east-1"

}

resource "aws\_vpc" "vpc1" {

  cidr\_block = var.vpcCIDR

  tags = {

    Name = "awsvpc"

  }

  enable\_dns\_support   = true

  enable\_dns\_hostnames = true

}

First , let us know the resources from the state file using the **State command.**

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform state list

aws\_vpc.vpc1

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform state show aws\_vpc.vpc1

# aws\_vpc.vpc1:

resource "aws\_vpc" "vpc1" {

arn = "arn:aws:ec2:us-east-1:065045723570:vpc/vpc-0d0fcc3b5e1244588"

assign\_generated\_ipv6\_cidr\_block = false

cidr\_block = "10.10.0.0/16"

default\_network\_acl\_id = "acl-07661563026b77bc7"

default\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8"

default\_security\_group\_id = "sg-0d71f21166447ea4a"

dhcp\_options\_id = "dopt-208bc45b"

enable\_classiclink = false

enable\_classiclink\_dns\_support = false

enable\_dns\_hostnames = true

enable\_dns\_support = true

id = "vpc-0d0fcc3b5e1244588"

instance\_tenancy = "default"

main\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8"

owner\_id = "065045723570"

tags = {

"Name" = "awsvpc"

}

}

#### Now let us run a plan to delete the vpc,

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform plan -destroy

There are some problems with the CLI configuration:

Error: Error reading C:\Users\LK757HL\AppData: read C:\Users\LK757HL\AppData: The handle is invalid.

As a result of the above problems, Terraform may not behave as intended.

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

aws\_vpc.vpc1: Refreshing state... [id=vpc-0d0fcc3b5e1244588]

------------------------------------------------------------------------

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

# aws\_vpc.vpc1 will be destroyed

- resource "aws\_vpc" "vpc1" {

- arn = "arn:aws:ec2:us-east-1:00000000000:vpc/vpc-0d0fcc3b5e1244588" -> null

- assign\_generated\_ipv6\_cidr\_block = false -> null

- cidr\_block = "10.10.0.0/16" -> null

- default\_network\_acl\_id = "acl-07661563026b77bc7" -> null

- default\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> null

- default\_security\_group\_id = "sg-0d71f21166447ea4a" -> null

- dhcp\_options\_id = "dopt-208bc45b" -> null

- enable\_classiclink = false -> null

- enable\_classiclink\_dns\_support = false -> null

- enable\_dns\_hostnames = true -> null

- enable\_dns\_support = true -> null

- id = "vpc-0d0fcc3b5e1244588" -> null

- instance\_tenancy = "default" -> null

- main\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> null

- owner\_id = "065045723570" -> null

- tags = {

- "Name" = "awsvpc"

} -> null

}

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

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform

can't guarantee that exactly these actions will be performed if

"terraform apply" is subsequently run.

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform plan -destroy

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

aws\_vpc.vpc1: Refreshing state... [id=vpc-0d0fcc3b5e1244588]

------------------------------------------------------------------------

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

- destroy

#### Terraform will perform the following actions:

# aws\_vpc.vpc1 will be destroyed

- resource "aws\_vpc" "vpc1" {

- arn = "arn:aws:ec2:us-east-1:065045723570:vpc/vpc-0d0fcc3b5e1244588" -> null

- assign\_generated\_ipv6\_cidr\_block = false -> null

- cidr\_block = "10.10.0.0/16" -> null

- default\_network\_acl\_id = "acl-07661563026b77bc7" -> null

- default\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> null

- default\_security\_group\_id = "sg-0d71f21166447ea4a" -> null

- dhcp\_options\_id = "dopt-208bc45b" -> null

- enable\_classiclink = false -> null

- enable\_classiclink\_dns\_support = false -> null

- enable\_dns\_hostnames = true -> null

- enable\_dns\_support = true -> null

- id = "vpc-0d0fcc3b5e1244588" -> null

- instance\_tenancy = "default" -> null

- main\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> null

- owner\_id = "065045723570" -> null

- tags = {

- "Name" = "awsvpc"

} -> null

}

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

## Taint

Manually mark a resource as tainted, forcing a destroy and recreate

on the next plan/apply.

This will not modify your infrastructure. This command changes your

state to mark a resource as tainted so that during the next plan or

apply that resource will be destroyed and recreated. This command on

its own will not modify infrastructure. This command can be undone

using the "terraform untaint" command with the same address.

The address is in the usual resource address syntax, as shown in

the output from other commands, such as:

aws\_instance.foo

aws\_instance.bar[1]

module.foo.module.bar.aws\_instance.baz

Options:

-allow-missing If specified, the command will succeed (exit code 0)

even if the resource is missing.

-backup=path Path to backup the existing state file before

modifying. Defaults to the "-state-out" path with

".backup" extension. Set to "-" to disable backup.

-lock=true Lock the state file when locking is supported.

-lock-timeout=0s Duration to retry a state lock.

-state=path Path to read and save state (unless state-out

is specified). Defaults to "terraform.tfstate".

-state-out=path Path to write updated state file. By default, the

"-state" path will be used.

**Example**

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform taint aws\_vpc.vpc1

Resource instance aws\_vpc.vpc1 has been marked as tainted.

PS C:\Users\LK757HL\Documents\Devops\terraform\awsvpn\planDel> terraform plan

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

aws\_vpc.vpc1: Refreshing state... [id=vpc-0d0fcc3b5e1244588]

------------------------------------------------------------------------

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

-/+ destroy and then create replacement

Terraform will perform the following actions:

# aws\_vpc.vpc1 is tainted, so must be replaced

-/+ resource "aws\_vpc" "vpc1" {

~ arn = "arn:aws:ec2:us-east-1:0000000000:vpc/vpc-0d0fcc3b5e1244588" -> (known after apply)

assign\_generated\_ipv6\_cidr\_block = false

cidr\_block = "10.10.0.0/16"

~ default\_network\_acl\_id = "acl-07661563026b77bc7" -> (known after apply)

~ default\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> (known after apply)

~ default\_security\_group\_id = "sg-0d71f21166447ea4a" -> (known after apply)

~ dhcp\_options\_id = "dopt-208bc45b" -> (known after apply)

~ enable\_classiclink = false -> (known after apply)

~ enable\_classiclink\_dns\_support = false -> (known after apply)

enable\_dns\_hostnames = true

enable\_dns\_support = true

~ id = "vpc-0d0fcc3b5e1244588" -> (known after apply)

instance\_tenancy = "default"

+ ipv6\_association\_id = (known after apply)

+ ipv6\_cidr\_block = (known after apply)

~ main\_route\_table\_id = "rtb-0a6ba8c1cc02ff7d8" -> (known after apply)

~ owner\_id = "065045723570" -> (known after apply)

tags = {

"Name" = "awsvpc"

}

}

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

## Workspace

As we know well this command is used to create required workspace in the environment.

Usage: terraform workspace

New, list, show, select and delete Terraform workspaces.

Subcommands:

delete Delete a workspace

list List Workspaces

new Create a new workspace

select Select a workspace

show Show the name of the current workspace

**Example**

terraform workspace create newdev

# Useful Functions

We will give a list of useful functions,

• index(list, elem) - Finds the index of a given element in a list. This function only works on flat lists. Example: index(aws\_instance.foo.\*.tags.Name, "foo-test")

• join(delim, list) - Joins the list with the delimiter for a resultant string. This function works only on flat lists. Examples:

• join(",", aws\_instance.foo.\*.id)

• join(",", var.ami\_list)

• jsonencode(value) - Returns a JSON-encoded representation of the given value, which can contain arbitrarily-nested lists and maps. Note that if the value is a string then its value will be placed in quotes.

• keys(map) - Returns a lexically sorted list of the map keys

• length(list) - Returns the number of members in a given list or map, or the number of characters in a given string.

• ${length(split(",", "a,b,c"))} = 3

• ${length("a,b,c")} = 5

• ${length(map("key", "val"))} = 1

• list(items, ...) - Returns a list consisting of the arguments to the function. This function provides a way of representing list literals in interpolation.

• ${list("a", "b", "c")} returns a list of "a", "b", "c".

• ${list()} returns an empty list.

• lookup(map, key, [default]) - Performs a dynamic lookup into a map variable. The map parameter should be another variable, such as var.amis. If key does not exist in map, the interpolation will fail unless you specify a third argument, default, which should be a string value to return if no key is found in map. This function only works on flat maps and will return an error for maps that include nested lists or maps.

• lower(string) - Returns a copy of the string with all Unicode letters mapped to their lower case.

• map(key, value, ...) - Returns a map consisting of the key/value pairs specified as arguments. Every odd argument must be a string key, and every even argument must have the same type as the other values specified. Duplicate keys are not allowed. Examples:

• map("hello", "world")

• map("us-east", list("a", "b", "c"), "us-west", list("b", "c", "d"))

• merge(map1, map2, ...) - Returns the union of 2 or more maps. The maps are consumed in the order provided, and duplicate keys overwrite previous entries.

• ${merge(map("a", "b"), map("c", "d"))} returns {"a": "b", "c": "d"}

• md5(string) - Returns a (conventional) hexadecimal representation of the MD5 hash of the given string.

• concat(list1, list2, ...) - Combines two or more lists into a single list. Example: concat(aws\_instance.db.\*.tags.Name, aws\_instance.web.\*.tags.Name)

• contains(list, element) - Returns true if a list contains the given element and returns false otherwise. Examples: contains(var.list\_of\_strings, "an\_element")

• dirname(path) - Returns all but the last element of path, typically the path's directory.

• distinct(list) - Removes duplicate items from a list. Keeps the first occurrence of each element, and removes subsequent occurrences. This function is only valid for flat lists. Example: distinct(var.usernames)

• element(list, index) - Returns a single element from a list at the given index. If the index is greater than the number of elements, this function will wrap using a standard mod algorithm. This function only works on flat lists. Examples:

• element(aws\_subnet.foo.\*.id, count.index)

• element(var.list\_of\_strings, 2)

• file(path) - Reads the contents of a file into the string. Variables in this file are not interpolated. The contents of the file are read as-is. The path is interpreted relative to the working directory. Path variables can be used to reference paths relative to other base locations. For example, when using file() from inside a module, you generally want to make the path relative to the module base, like this: file(path.module}/file").

# Examples

AWS VPC

As you can see in the above arch diagram, what are all the components that are created and linked with each other, when creating a vpc in aws. Let me list the components as below.

1. VPC
2. NACL
3. Route Table
4. Internet Gateway
5. Subnet
6. Route
7. Security groups

##### VPC

variable "vpcname" {

    type = "string"

    default = "devopsVPC"

}

variable "vpc\_cidr\_block" {

  type = "string"

}

resource "aws\_vpc" "main" {

    cidr\_block           = var.vpc\_cidr\_block

    enable\_dns\_hostnames = true

    enable\_dns\_support   = true

    instance\_tenancy     = "default"

    tags  {

      Name = var.vpcname

    }

}

##### Subnet, Internet gateway, Route Table and Routes

variable "subnet\_cidr\_block" {

  type = "list"

}

variable "azs" {

  type = "string"

}

resource  "aws\_subnet" "public" {

    count             = length(var.subnet\_cidr\_block)

    vpc\_id            = aws\_vpc.main.id

    cidr\_block        = lookup(var.subnet\_cidr\_block[count.index], "cidr")

    availability\_zone = var.azs

    map\_public\_ip\_on\_launch = true

    depends\_on = ["aws\_vpc.main"]

    tags = {

        "Name" = lookup(var.subnet\_cidr\_block[count.index], "name")

    }

}

resource "aws\_internet\_gateway" "int\_gtw" {

  vpc\_id = aws\_vpc.main.id

  tags {

      Name = "Devops Gtw"

  }

}

resource "aws\_route\_table" "route\_table\_for\_public\_subnet" {

  vpc\_id = aws\_vpc.main.id

  route {

      cidr\_block = "0.0.0.0/0"

      gateway\_id = aws\_internet\_gateway.int\_gtw.id

  }

  tags  {

      Name = "Devops Route Table"

  }

}

resource "aws\_main\_route\_table\_association" "a1" {

  vpc\_id         = aws\_vpc.main.id

route\_table\_id = aws\_route\_table.route\_table\_for\_public\_subnet.id

  depends\_on     = "{aws\_route\_table.route\_table\_for\_public\_subnet]

}

resource "aws\_route\_table\_association" "route\_table\_subnet" {

  count = length(var.subnet\_cidr\_block)

  subnet\_id = element(aws\_subnet.public.\*.id, count.index)

  route\_table\_id = aws\_route\_table.route\_table\_for\_public\_subnet.id

}

##### Create multiple AWS ec2 instances

resource "aws\_instance" "web" {

  count = length(var.vmNames)

  ami           = "ami-0a313d6098716f372"

  instance\_type = var.instance\_type

  subnet\_id     = element(var.subnetID, count.index)

  security\_groups = [var.sgID]

  associate\_public\_ip\_address = true

  key\_name = var.key

  tags = {

    Name = element(var.vmNames, count.index)

  }

}

To create a VM the required parameters are

1. Ami
2. Instance\_type
3. Subnet\_id
4. Security\_groups
5. Key\_name

There are many optional parameters like count, tags, associate\_public\_ip\_address. With the count variable, we can iterate the instance block to create *n* number of instances. The *tags* are the regular tags parameter to name a resource with meaningful name for user identification.

#### ACM import and base64decode function

In this example, we use terraform “aws\_acm\_certificate” resource to import a self-signed certificate into AWS ACM. For this we are going to generate the cert using opevpn following the below steps

1. git clone <https://github.com/OpenVPN/easy-rsa.git>
2. cd easy-rsa/easyrsa3
3. ./easyrsa init-pki
4. ./easyrsa build-ca nopass
5. ./easyrsa build-server-full example.com nopass
6. mkdir /home/ec2-user/custom\_folder
7. cp pki/ca.crt /home/ec2-user/custom\_folder/
8. cp pki/issued/example.com.crt /home/ec2-user/custom\_folder/
9. cp pki/private/example.com.key /home/ec2-user/custom\_folder/
10. chown -R ec2-user:ec2-user /home/ec2-user/custom\_folder
11. cd /home/ec2-user/custom\_folder
12. cat ca.crt | base64
13. cat example.com.crt | base64
14. cat example.com.key | base64

Now, we are having base64 encoded files of certificate chain, certificate data and private key

When running steps 12 to 14, the base64 value will be printed in the screen. This value can either be put in variables.tfvars file or put it in “default” in variable declaration in template file. Next we use the base64decode function and decode these data and store them in the “locals”.

To know more about this, follow the below example or you can also watch the demo in the below link



# Provisioner

#### chef

We all know that chef is one of the popular configuration management tool. It operates in a pull-model and whenever a VM is provisioned, it is bootstrapped to install chef-client inside it. The chef-client will later pull configuration updates from chef-server. This bootstrap process is usually done from inside a workstation which is our local workstation or an instance in cloud. Terraform leverages infrastructure provisioning with provisioners to bootstrap the created VM with chef-client. We will look through an example template and the arguments it requires.

resource "aws\_instance" "web" {

  ami                         = var.ami

  instance\_type               = var.instance\_type

  subnet\_id                   = var.subnetID

  security\_groups             = [var.securityGroups]

  associate\_public\_ip\_address = true

  key\_name                    = var.instance\_key

  tags = {

    Name = "chefclient"

  }

  provisioner "chef" {

    connection {

      type        = "ssh"

      user        = "ubuntu"

      host        = aws\_instance.web.public\_ip

      private\_key = file("/home/ubuntu/chefprovision/cptest.pem")

    }

    client\_options          = ["chef\_license 'accept'"]

    environment             = "\_default"

    log\_to\_file             = true

    fetch\_chef\_certificates = true

    node\_name               = "client1"

    server\_url              = "https://chefurl/organizations/cloudlego"

    recreate\_client         = true

    run\_list                = [""]

    user\_name               = "legoadmin"

    user\_key                = file("/home/ubuntu/chefprovision/legoadmin.pem")

    ssl\_verify\_mode         = ":verify\_none"

  }

}

The chef provisioner is usually called from the aws\_instance resource block. The following are the key parameters for a chef-provisioner

1. client\_options
2. environment
3. log\_to\_file
4. fetch\_chef\_certificates
5. node\_name
6. server\_url
7. recreate\_client
8. run\_list
9. user\_name
10. user\_key
11. ssl\_verify\_mode

**client\_options:** This is a mandatory option to accept chef-client EULA

**environment:** The environment of chef-client, it defaults to \_default

**log\_to\_file:** This option enables logging output to a file when running the chef-client installation

**fetch\_chef\_certificates:** To pull certificates from chef-server into the chef client, that authenticate chef client with chef server

**node\_name:** Name of the chef node

**server\_url:** The url for the chef server along with the organization name

**recreate\_client:** In case, the chef client is already installed in the VM, then this option delete the chef client from VM as well as chef server and installs again

**run\_list:** The list of recipes or cookbooks to run inside the chef node

**user\_name:** The username for chef client to connect with chef server

**user\_key:** The ssh key for the chef user

**ssl\_verify\_mode:** If the chef server uses a self-signed certificate, then this option should set value as :verify\_none

Another key note is, the terraform documentation don’t tell about the connection object. This is required, without which the terraform cannot login to the instance provisioned and installs chef client

#### file

The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. **Remember to use connection object in all provisioners as it is given in chef provisioner.**

resource "aws\_instance" "web" {

  # ...

  # Copies the myapp.conf file to /etc/myapp.conf

  provisioner "file" {

    source      = "conf/myapp.conf"

    destination = "/etc/myapp.conf"

  }

  # Copies the string in content into /tmp/file.log

  provisioner "file" {

    content     = "ami used: ${self.ami}"

    destination = "/tmp/file.log"

  }

  # Copies the configs.d folder to /etc/configs.d

  provisioner "file" {

    source      = "conf/configs.d"

    destination = "/etc"

  }

  # Copies all files and folders in apps/app1 to D:/IIS/webapp1

  provisioner "file" {

    source      = "apps/app1/"

    destination = "D:/IIS/webapp1"

  }

}

#### remote-exec

The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc

provisioner "remote-exec" {

    inline = [

      "chmod +x /tmp/script.sh",

      "/tmp/script.sh args",

    ]

}

# Policy as Code

Policy as code is the idea of writing code in a high-level language to manage and automate policies. By representing policies as code in text files, proven software development best practices can be adopted such as version control, automated testing, and automated deployment.

## Benefits

Policy as code provides a number of benefits:

* **Sandboxing.** Policies provide the guardrails for other automated systems. As the number of automated systems grow, there is also a growing need to protect those automated systems from performing dangerous actions. Manual verification is too slow; policies need to be represented as code to keep up with other automated systems.
* **Codification.** By representing policy logic as code, the information and logic about a policy is directly represented in code and can be augmented with comments rather than relying on oral tradition to learn about the reason for policies.
* **Version Control.** Policies are encouraged to be stored as simple text files managed by a version control system. This lets you gain all the benefits of a modern VCS such as history, diffs, pull requests, and more.
* **Testing.** Policies are just code. Their syntax and behavior can be easily validated with Sentinel. This also encourages automated testing such as through a CI. Paired with a VCS system, this allows a pull request workflow to verify that a policy keeps the system behavior as expected before merging.
* **Automation.** With all policies as code in simple text files, various automation tools can be used. For example, it is trivial to create tools to automatically deploy the policies into a system.

# Sentinel and Policy as Code

Terraform uses **Sentinel** to write policies to govern infrastructure to be or already provisioned. Sentinel works with terraform cloud or enterprise versions. It has its own **sentinel language** for development. Also, sentinel allows users to version the code and can be used with other hashicorp tools like **Nomad, Consul and Vault.** Most tools out there in market allows infrastructure to be evaluated after it is provisioned or test by provisioning and removing it.

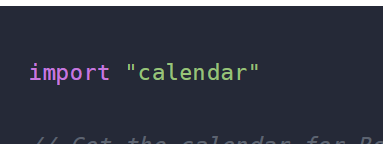
Also sentinel provides a cli to development and testing. This local CLI can be used to verify the test cases before executing them in enterprise.

Sentiel brought out its own language because it should be easy for non-programmers as well as dynamic like a programming language.

## Imports

Imports enable a Sentinel policy to access reusable libraries and external data and functions. Anyone can write their own [custom import](https://docs.hashicorp.com/sentinel/extending). Imports are what enable Sentinel policies to do more than look at only local context for making policy decisions.

Imports are extremely powerful. They enable policy decision based on arbitrary external information. For example, a behavior coming into a system can be allowed or denied based on information from a separate system where the two systems can be unaware of each other.



## Enforcement Levels

Sentinel has three enforcement levels:

* **Advisory:** The policy is allowed to fail. However, a warning should be shown to the user or logged.
* **Soft Mandatory:** The policy must pass unless an override is specified. The semantics of "override" are specific to each Sentinel-enabled application. The purpose of this level is to provide a level of privilege separation for a behavior. Additionally, the override provides non-repudiation since at least the primary actor was explicitly overriding a failed policy.
* **Hard Mandatory:** The policy must pass no matter what. The only way to override a hard mandatory policy is to explicitly remove the policy. Hard mandatory is the default enforcement level. It should be used in situations where an override is not possible.

In this document, we will not see in depth of the language. To learn further on this, visit <https://docs.hashicorp.com/sentinel/language/>

We will show you a small piece of policy written as code and executed through terraform cloud. To test the policy in your local, you need to have enterprise or generate mock data from terraform cloud and apply the policies.

The example here is to enforce tags mandatorily on all resources being created in AWS. To begin with terraform provides a repository of all common functions which can be reused with the required change. The repository is available at <https://github.com/hashicorp/terraform-guides/tree/master/governance/second-generation>

From this repository we take a code to check mandatory tags. The script has two different functions, one to get the list of all resources from the plan file and the other to evaluate the resource attribute, tags against the allowed values. The policy will validate true if the tag exists and contains one of the allowed values. In the “main” we call the “rule” to validate the given condition.

## Example

#### Mandatory Tags

The sentinel has given their own repository with common useful functions for different approaches. We can use it for config, plan or state. The plan or config or state file is imported using the import function and then the list of modules are traversed to get the address of each resource. The required condition is then executed after this.

import "tfplan"

import "strings"

import "types"

find\_resources\_from\_plan = func(type) {

  resources = {}

  for tfplan.module\_paths as path {

    # Iterate over the named resources of desired type in the module

    for tfplan.module(path).resources[type] else {} as name, instances {

      # Iterate over resource instances

      for instances as index, r {

        # Get the address of the instance

        if length(path) == 0 {

          # root module

          address = type + "." + name + "[" + string(index) + "]"

        } else {

          # non-root module

          address = "module." + strings.join(path, ".module.") + "." +

                    type + "." + name + "[" + string(index) + "]"

        }

        # Add the instance to resources map, setting the key to the address

        resources[address] = r

      }

    }

  }

  return resources

}

validate\_attribute\_contains\_list = func(type,attribute, required\_values) {

  validated = true

  resource\_instances = find\_resources\_from\_plan(type)

  for resource\_instances as address, r {

    #skip resources that are going to be destroyed

    if r.destroy and not r.requires\_new {

      print("Skipping resource", address, "that is being destroyed.")

      continue

    }

    # to validate instances that are computed

    if (r.diff[attribute + ".%"].computed else false) or

       (r.diff[attribute + ".#"].computed else false) {

      print("Resource", address, "has attribute", attribute,

            "that is computed.")

        #disable the below line incase you want the computed resources to make validated fail

        # validated = false

    } else {

      # Validate that each instance has allowed value

      if r.applied[attribute] else null is not null and

         (types.type\_of(r.applied[attribute]) is "list" or

          types.type\_of(r.applied[attribute]) is "map") {

        for required\_values as rv {

          if r.applied[attribute] not contains rv {

            print("Resource", address, "has attribute", attribute,

                  "that is missing required value", rv, "from the list:",

                  required\_values)

            validated = false

          }

        }

      } else {

        print("Resource", address, "is missing attribute", attribute,

              "or it is not a list or a map")

        validated = false

      }

    }

  }

  return validated

}

mandatory\_tags = [

  "Name",

  "owner",

]

resource\_types = [

    "aws\_acm\_certificate",

    "aws\_instance",

    "aws\_vpc",

    "aws\_subnet",

    "aws\_lb",

]

### Rules ###

tags\_validated = false

for resource\_types as type {

    tags\_validated = validate\_attribute\_contains\_list(type,"tags", mandatory\_tags)

}

# Call the validation function

#Main rule

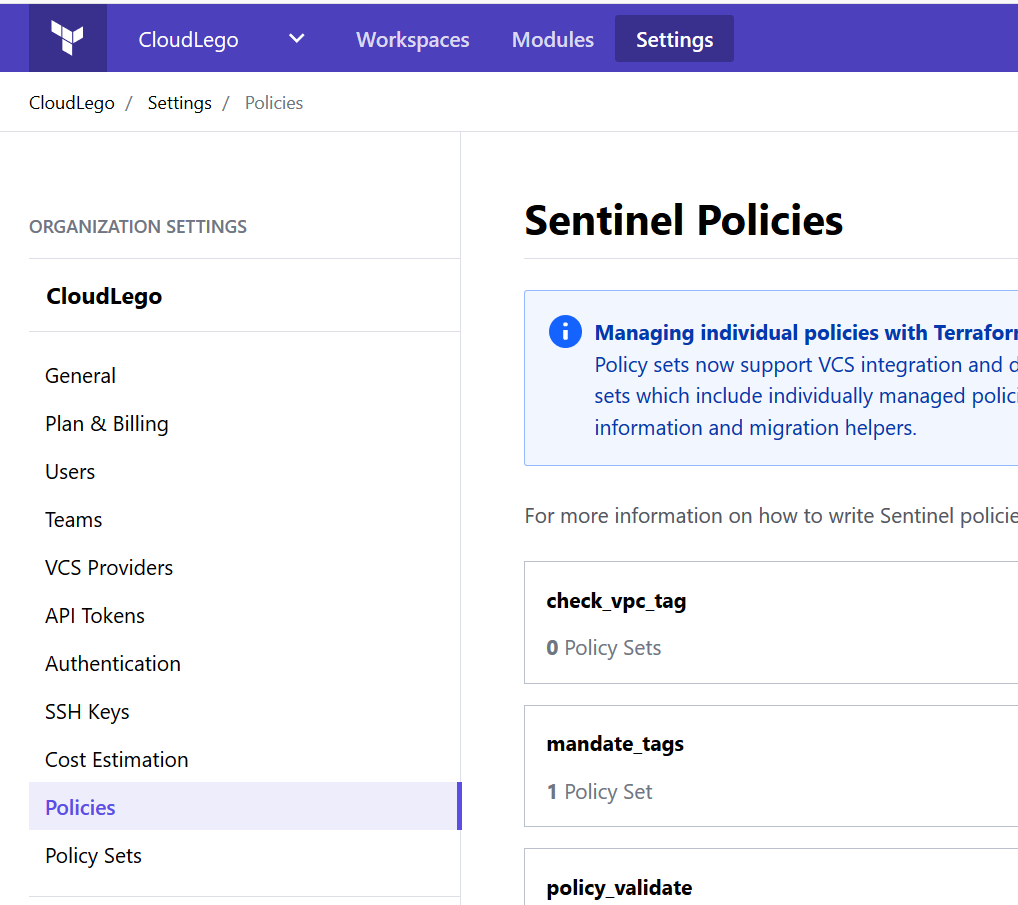
main = rule {

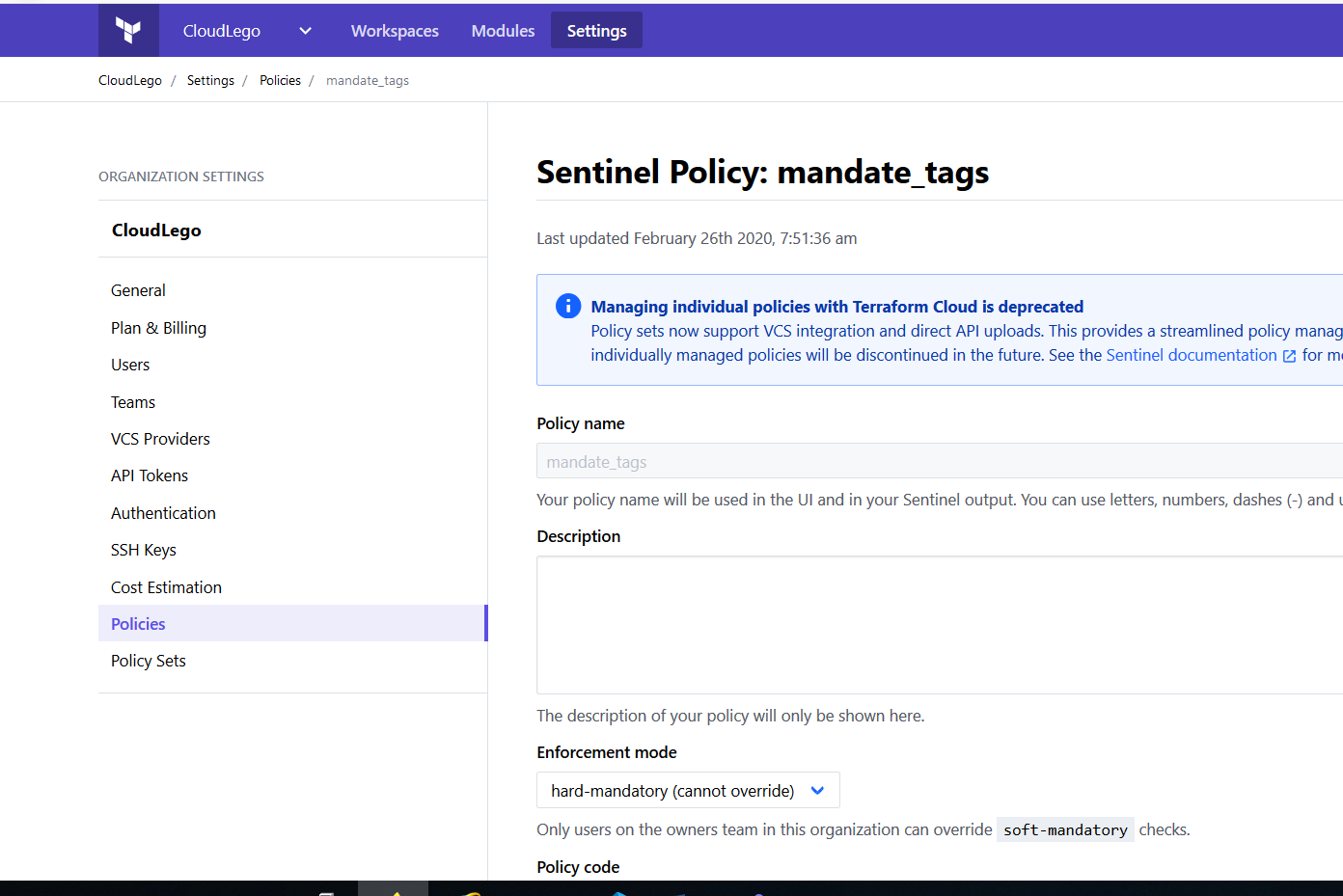
  tags\_validated

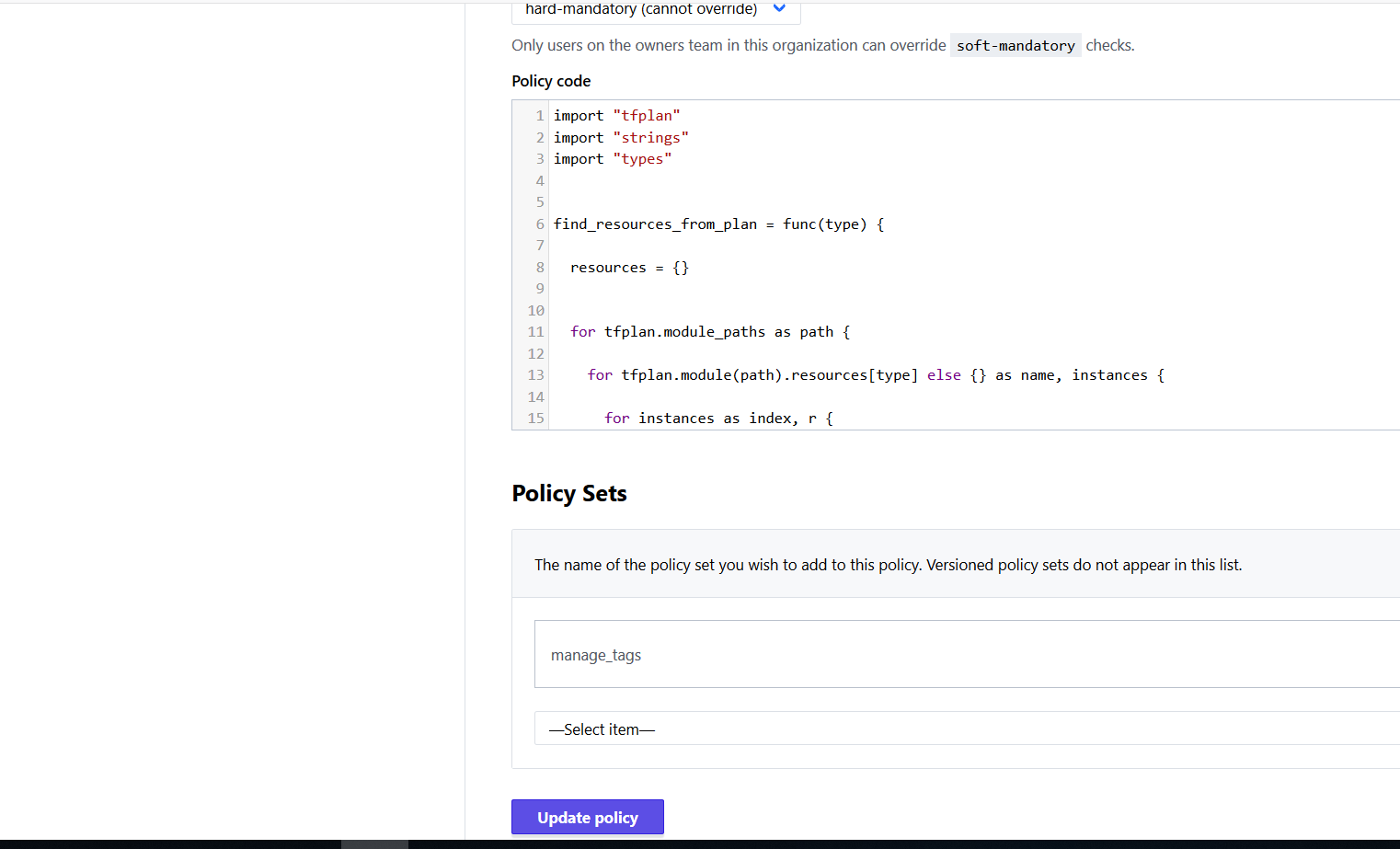
}

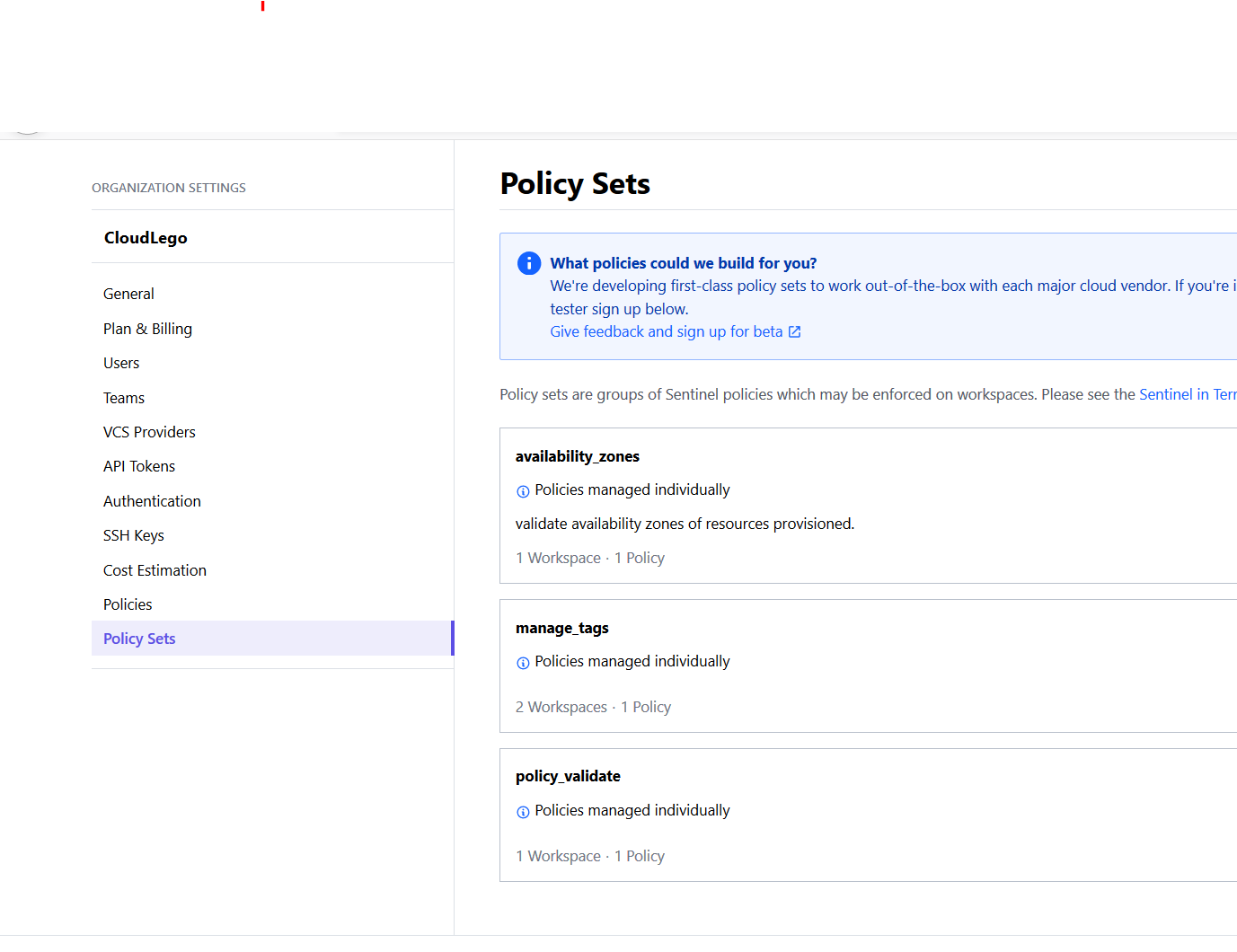
Once the policy is written and tested in local, it can be put into terraform UI. Go to “settings” of the organization and select “Policies”. Create a new policy and paste your code into that. Create a Policy Set from the “Policy Sets” and select the list of workspace on which it should execute the policy. Update the policy to which policy set it belongs to. Policy Set can also be linked to your VCS like how you sync your infra code. When the “Plan” is initiated in the workspace, after the plan is completed, the policy validation is started and if the test case pass through, then the plan can be applied or not.

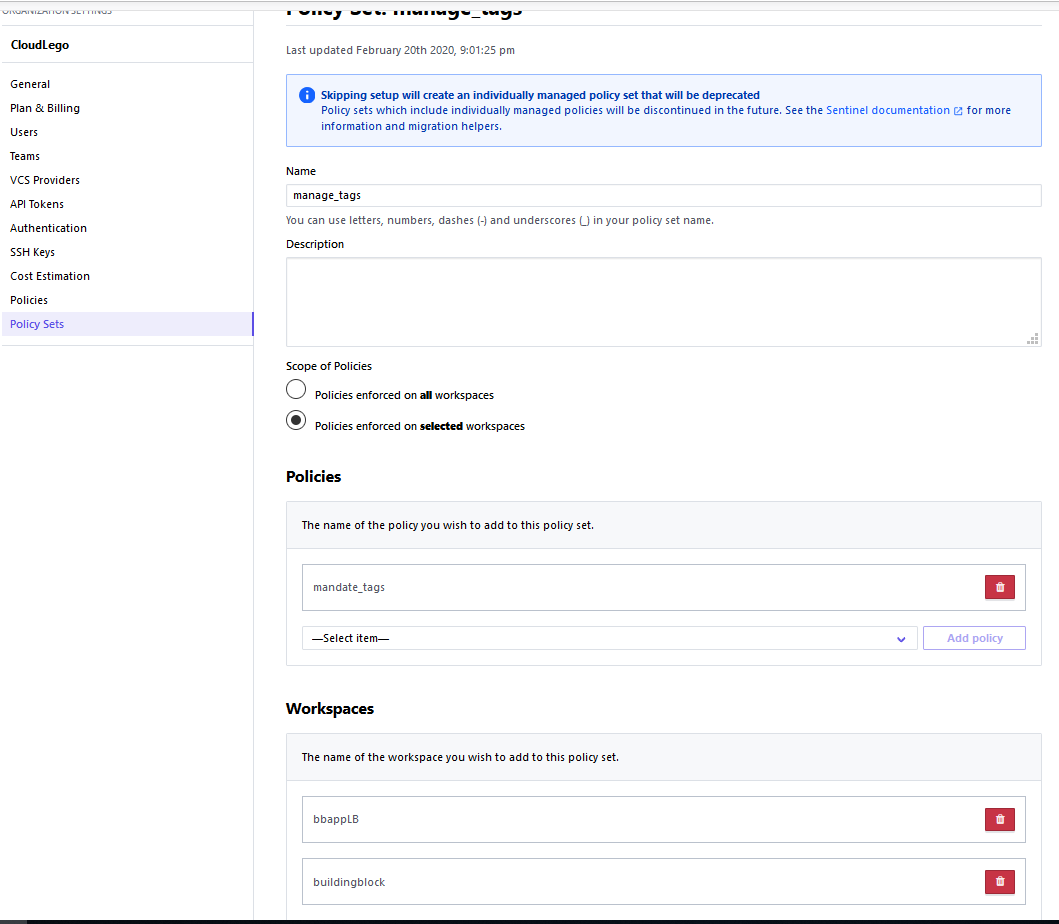
**Note**: Policies are available in terraform cloud only for a paid version or a free trail for one month.

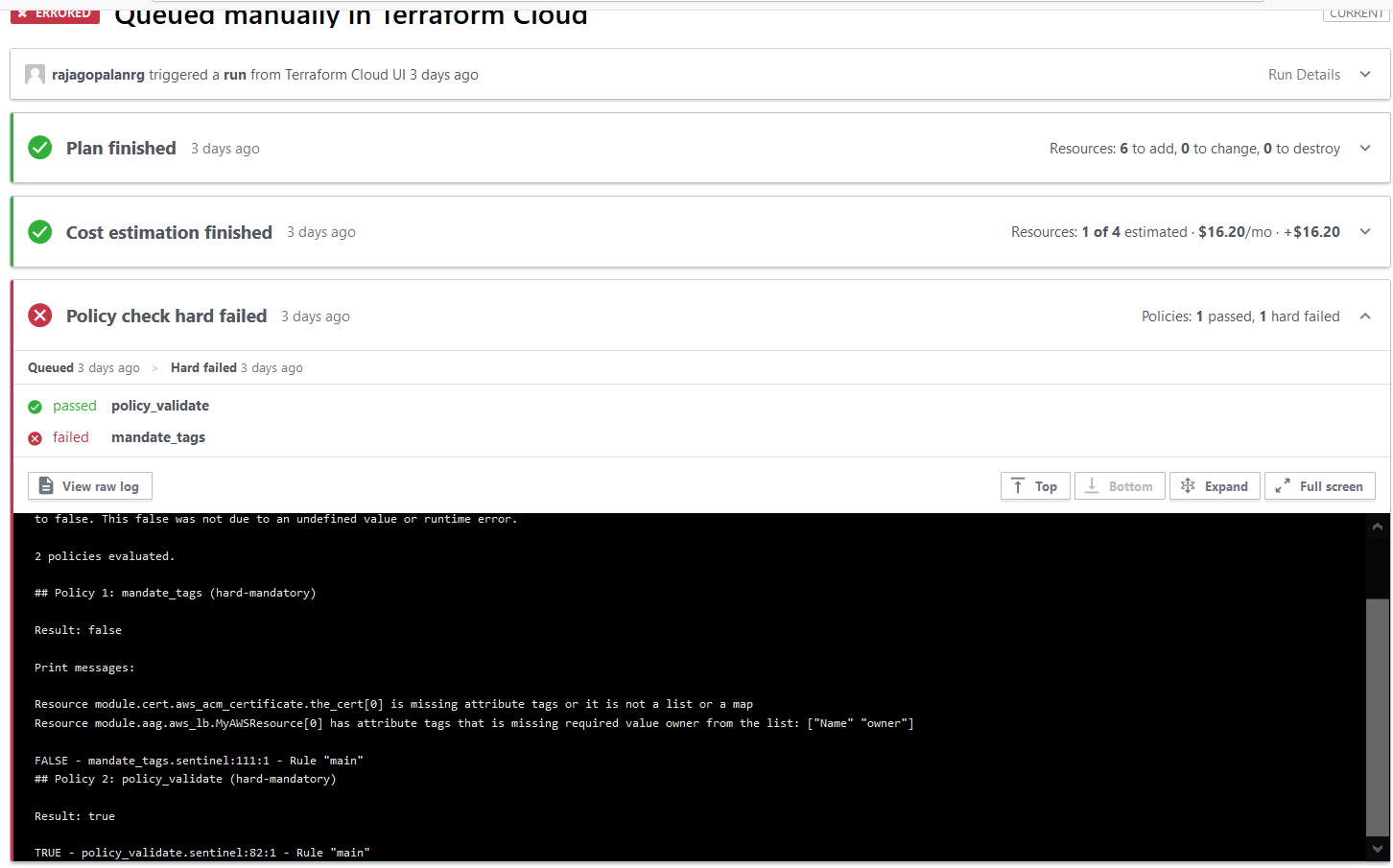












# Dynamic Blocks

In the terraform versions greater than 0.12, it is possible to iterate resource properties with the help of dynamic blocks.

variable "dataDisks" {  
 type = list  
 default = [127,60]  
}

resource "azurerm\_virtual\_machine" "mvm01" {  
 dynamic "storage\_data\_disk" {  
 for\_each = [for d in var.dataDisks :{  
 diskname = join("\_",[var.vm\_name,"datadisk",index(var.dataDisks,d )])  
 disklun = index(var.dataDisks,d )  
 disksizegb = d  
 createoption = "Empty"  
 }]  
 content {  
 create\_option = storage\_data\_disk.value.createoption  
 lun = storage\_data\_disk.value.disklun  
 name = storage\_data\_disk.value.diskname  
 disk\_size\_gb = storage\_data\_disk.value.disksizegb  
 }  
 }  
  
}

Or it can be a simple for-each as below

for\_each = var.storage\_data\_disks

content {

create\_option = storage\_data\_disk.value["createOption"]

lun = storage\_data\_disk.value["lun"]

name = storage\_data\_disk.value["name"]

disk\_size\_gb = storage\_data\_disk.value["diskSizeInGB"]

}

# Terraform in Json

Terraform templates can also be written in json syntax just like the below example.

{

  "module": {

    "azureVM": {

      "admin\_password": "",

      "admin\_username": "winuser",

      "location": "westeurope",

      "os\_data\_disk\_size\_in\_gb": 0,

      "resource\_group\_name": "Testing",

      "source": "",

      "storage\_data\_disks": [

        {

          "CreateOption": "Empty",

          "Lun": 0,

          "Name": "datadisk1",

          "DataDiskSizeInGB": 0

        }

      ],

      "subnet\_name": "subnet001",

      "version": "1.0.3",

      "vm\_name": "euwmvm01",

      "vm\_size": "Standard\_DS1\_v2",

      "vm\_sku": "2016-Datacenter",

      "vnet\_name": "vnet001"

    }

  },

  "provider": {

    "azurerm": {

      "client\_id": "${var.client\_id}",

      "client\_secret": "${var.client\_secret}",

      "feature": {},

      "subscription\_id": "${var.subscription\_id}",

      "tenant\_id": "${var.tenant\_id}",

      "version": "=2.4.0"

    }

  },

  "variable": {

    "client\_id": {

      "type": "string"

    },

    "client\_secret": {

      "type": "string"

    },

    "subscription\_id": {

      "type": "string"

    },

    "tenant\_id": {

      "type": "string"

    }

  }

}