Are you Ready for HashiCorp Certified Terraform Associate exam? Self-assess yourself with "[Whizlabs FREE TEST]"

HashiCorp Certified Terraform Associate WhizCard

Quick Bytes for you before the exam!

WHIZLABS

Index

WHIZLABS

Introduction to Terraform

What is Infrastructure as Code (IaC)?

A way of managing and provisioning IT infrastructure through code and automation. Typically, in the form of configuration files or scripts.

Why IaC?

Automation, Version control, Scalability, Consistency, and reduced human error, are easily implemented in CI/CD integration.

Well known IaC tools in the market:

- HashiCorp Terraform – Well known platform agnostic IaC tool.
- AWS CloudFormation – IaC tool from AWS and can only be used to build AWS resources.
- Azure Resource Manager – IaC tool from Azure and can only be used to build Azure resources.
- Ansible, Chef & Puppet, etc. – Configuration management tools used to manage large groups of computer systems.

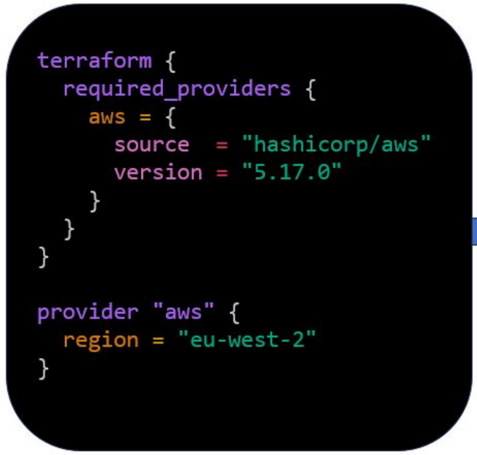
What is Terraform?

- Terraform is an infrastructure as a code tool that allows you to define both Cloud and On-prem resources in human-readable configuration files.
- HashiCorp Configuration Language (HCL) is used to define the resources as a code.
- Terraform is Cloud agnostic. It works with multiple cloud providers. Terraform enables developers to maintain the same workflow when provisioning resources across cloud/infrastructure providers.

Introduction to Terraform

WHIZLABS

- Below is the example of how terraform configuration looks like.

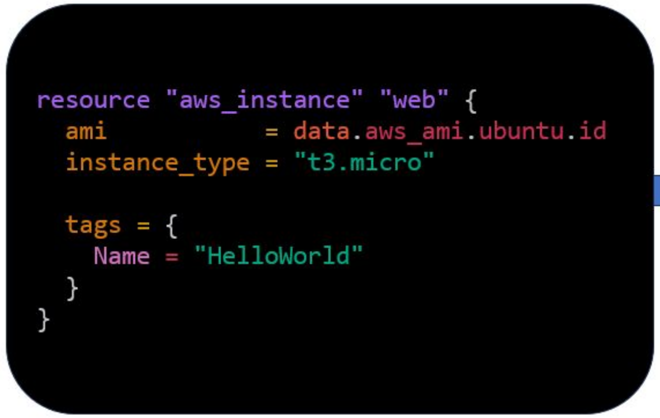


```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "5.17.0"
    }
  }
}

provider "aws" {
  region = "eu-west-2"
}
```

"provider" block installs the required providers and versions

- Below is the example of how resource configuration looks like.



```
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```

"resource" block used to define how the resources to be provisioned.

- Terraform vs Other IaC:

Terraform	**Other IaC tools (Ansible, Chef, Puppet etc.)**
Multi-Cloud support – Terraform can be used in any Cloud platforms and even in On-premises.	Other tools are platform specific.
Terraform is declarative language - you just describe the required end resource and terraform is intelligent to know how to achieve it.	Procedural - you have to define step by step what to do.
HCL – Terraform code is developed in HCL language which is easy to learn and implement.	Other tools language varies as per the requirement and complex compared to HCL.
Agentless – Terraform doesn't require any agents to be installed to run.	Tools like Ansible and Puppet needs agents to be installed.

Exam Tips:
Focus on the features Terraform offers over other IaC tools.

Terraform Providers

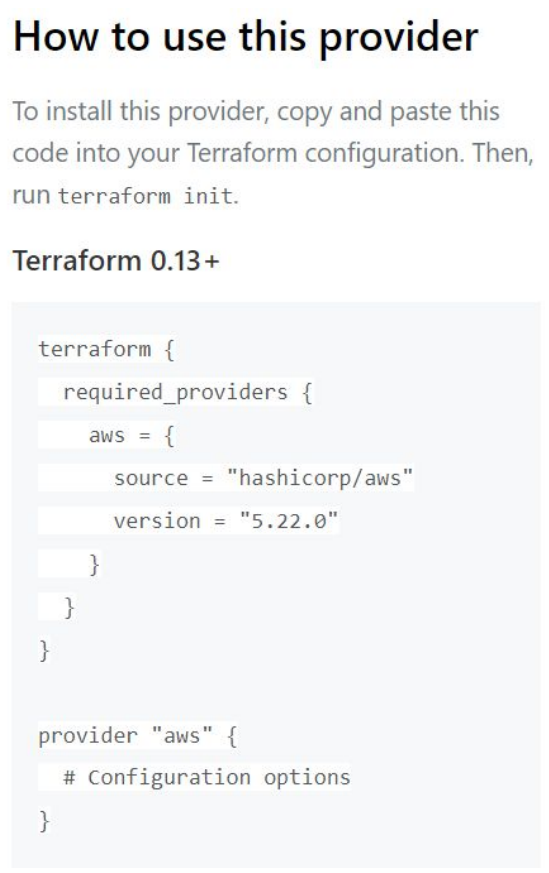
- In Terraform, "providers" are plugins or extensions that enable communication between Terraform and various infrastructure platforms or services. Providers uses API to interact.

- **Terraform supports more than 200 Providers.**
  - Few of them are – AWS, Azure, Google cloud, Kubernetes etc.
- **How to use Providers in your Terraform code?**
  - Terraform Providers official website - https://registry.terraform.io/browse/providers

  we have to copy the providers block to install all dependencies required to interact to the platform.

- Providers can be Official, Partner and Community supported.
- Official providers are owned and maintained by HashiCorp.
- Partner providers are written, maintained, validated and published by third-party companies against their own APIs.
- Community providers are published to the Terraform Registry by individual maintainers, groups of maintainers, or other members of the Terraform community.

- Below is the example of AWS provider block.



How to use this provider

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

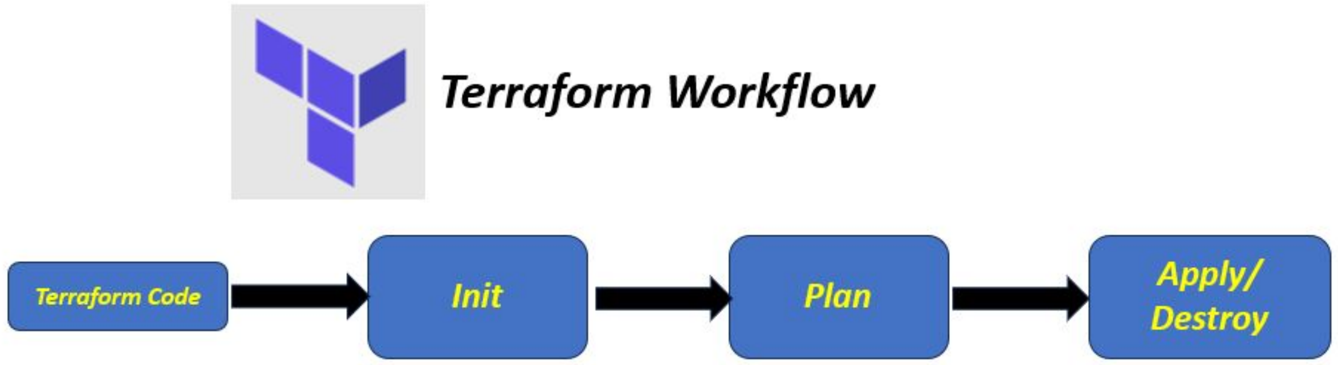
Terraform 0.13+

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.22.0"
    }
  }
}

provider "aws" {
  # Configuration options
}
```

- **Exam Tips:**
  - Make sure to practice and remember the syntax of Providers block.

Terraform Workflow

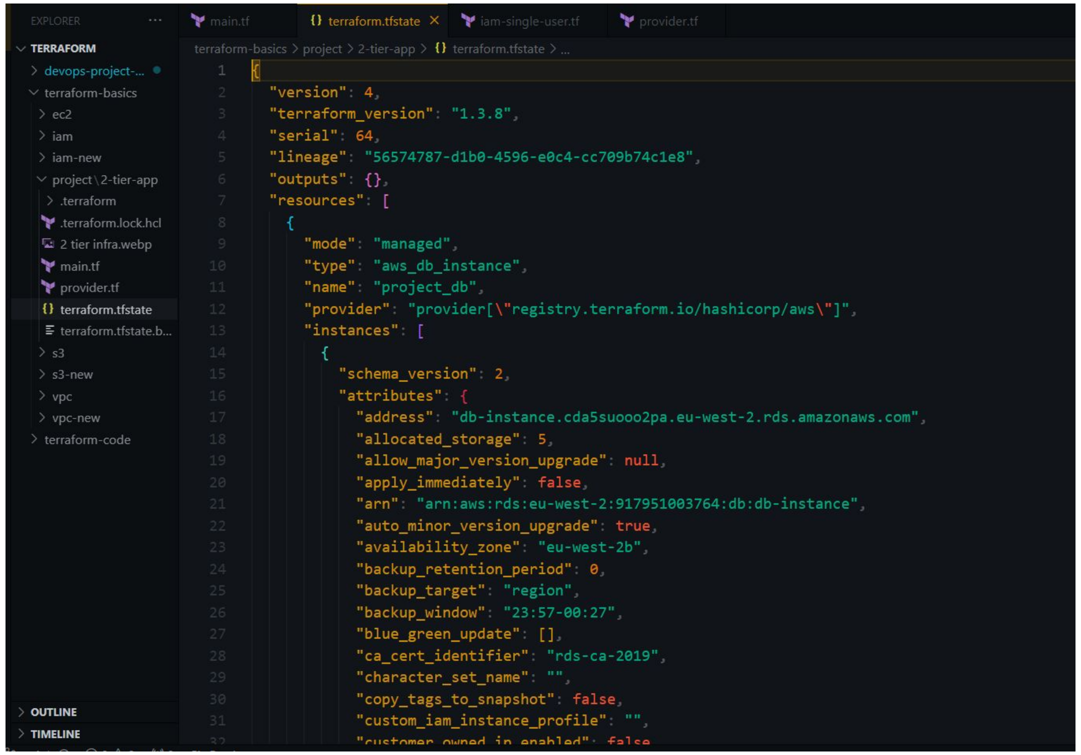


- Init – “terraform init” is a command you will run to initialize your workspace so Terraform can apply your configuration.
- Plan – “terraform plan” is a command that allows you to preview the changes Terraform will make before you apply them.
- Apply – “terraform apply” is a command that creates/modifies the actual resources as shown in the plan output.
- Destroy – “terraform destroy” is a command that destroys your all resources mentioned in the code.

- **Exam tips:**
  - ‘auto-approve’ - to apply/destroy Terraform configurations and automatically approve without user interaction, you can use the -auto-approve flag with the terraform apply command.
    - terraform apply -auto-approve
    - terraform destroy -auto-approve

Terraform State

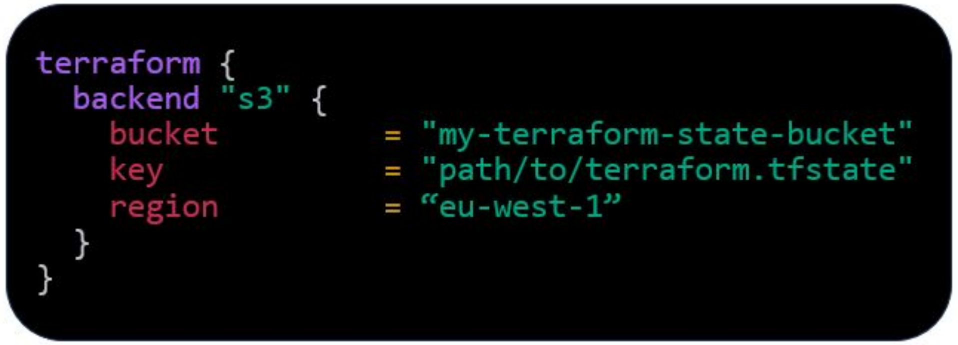
- **State file** – It is an HCL/JSON file that Terraform manages automatically to record the current state of your infrastructure. This file contains information about the resources that Terraform manages, their attributes, and their current configuration.
    - Example state file snippet.



```
{
  "version": 4,
  "terraform_version": "1.3.8",
  "serial": 64,
  "lineage": "56574787-d1b0-4596-e0c4-cc709b74c1e8",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_db_instance",
      "name": "project_db",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 2,
          "attributes": {
            "address": "db-instance.cda5suooo2pa.eu-west-2.rds.amazonaws.com",
            "allocated_storage": 5,
            "allow_major_version_upgrade": null,
            "apply_immediately": false,
            "arn": "arn:aws:rds:eu-west-2:917951003764:db:db-instance",
            "auto_minor_version_upgrade": true,
            "availability_zone": "eu-west-2b",
            "backup_retention_period": 0,
            "backup_target": "region",
            "backup_window": "23:57-00:27",
            "blue_green_update": [],
            "ca_cert_identifier": "rds-ca-2019",
            "character_set_name": "",
            "copy_tags_to_snapshot": false,
            "custom_iam_instance_profile": "",
            "customer_owned_ip_enabled": false
```

- State lock – Terraform locks your State file to prevent others from acquiring the lock and potentially corrupting it.

- State Backend – It determines where the State file is stored and how it is managed.
    - Local Backend – This is the default Backend configuration in which the State file is stored on the same machine Terraform runs.
    - Remote Backend – This allows you to store your State file on a remote server or Cloud storage service. A few of the remote backend options are Amazon S3, Azure Blob storage, etc.



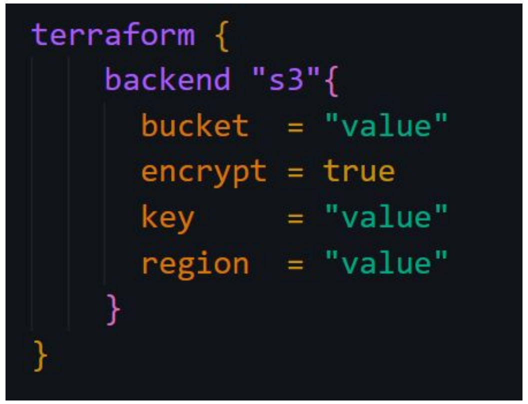
```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "path/to/terraform.tfstate"
    region         = “eu-west-1”
  }
}
```

- **Exam Tips:**
    - Remote backend is the secure way of storing your State files.
    - Always make sure not to store any sensitive data in the State file.

Terraform Backend Configuration

- Terraform Backend defines where and how the State file will be stored.
- Local Backend - This is the default Backend configuration in which the State file is stored on the same machine Terraform runs.
- Remote Backend - This allows you to store your State file on a remote server or Cloud storage service. A few remote backend options are Amazon S3, Azure Blob storage, etc.
- Local Backend Configuration – This is the default Backend configuration in Terraform. If the configuration includes no backend block, Terraform defaults to using the local backend, which stores the state as a plain file in the current working directory.
- Remote Backend Configuration – Remote Backend can store State files in different places such as AWS S3, Azure Blob storage, etc.

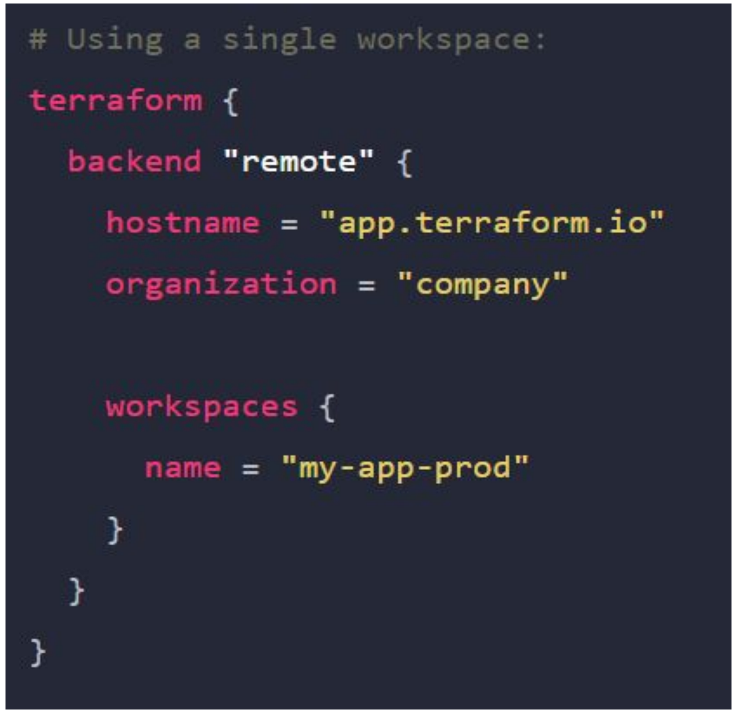
  a. Below is the Configuration block to store the State file in AWS S3.



```
terraform {
    backend "s3"{
      bucket  = "value"
      encrypt = true
      key     = "value"
      region  = "value"
    }
}
```

- **Exam Tips:**
  - Remote backend is the secure way of storing your State files.
  - Always make sure not to store any sensitive data in the State file.

- Remote Backend allows multiple people to access the State file and work collaboratively on the collection of resources.



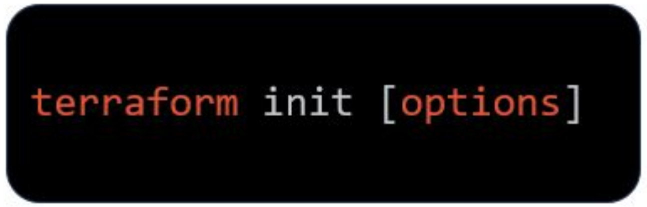
```
# Using a single workspace:
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Source: terraform backend

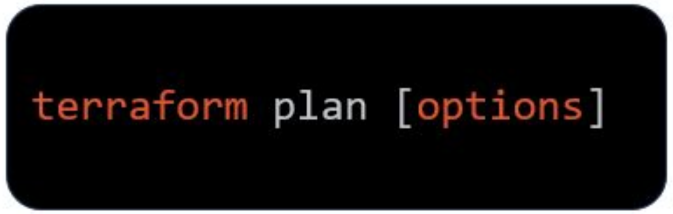
Terraform Commands

- **terraform init** - Initializes the working directory, installs required plugins.



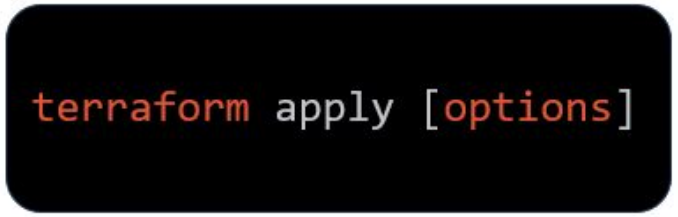
```
terraform init [options]
```

- **terraform plan** – Shows the execution plan without performing the actual actions.



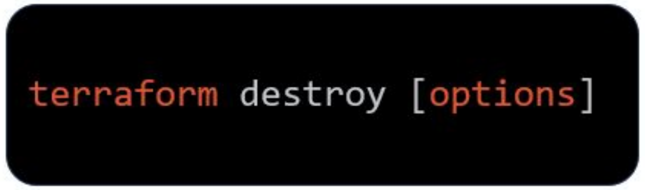
```
terraform plan [options]
```

- **terraform apply** – Creates or Updates the infrastructure based on the Configuration file.



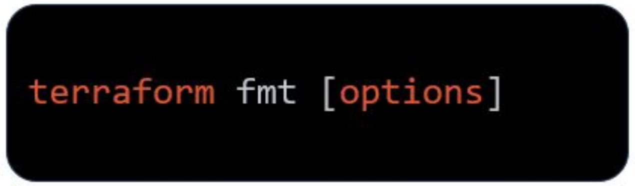
```
terraform apply [options]
```

- **terraform destroy** – Destroys all the resources mentioned in the Configuration file.



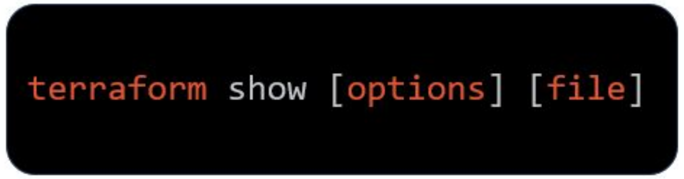
```
terraform destroy [options]
```

- **terraform fmt –** Formats your configuration to the HCL code standards.



```
terraform fmt [options]
```

- **terraform show** – Shows the State file in a human readable format.



```
terraform show [options] [file]
```

Terraform Commands

- **terraform state list** – Lists all the resources tracked in the current State file.

```
terraform state list [options] [address]
```

```
terraform state list aws_instance.example
```

- **terraform state rm –** Removes the specified resource from the State file.

```
terraform state rm “resource_name”
```

```
terraform state rm aws_instance.example
```

- **terraform refresh** – Refreshes the State file with as per the Configuration file.

```
terraform refresh [options]
```

- **terraform import** – Imports the resource into state file.
  - Let's say you want to import an existing AWS S3 bucket with the name "my-whizlabs-bucket" into your Terraform configuration.
  - Below is the format and example command used to import the bucket.

```
terraform import aws_s3_bucket.bucket bucket-name
```

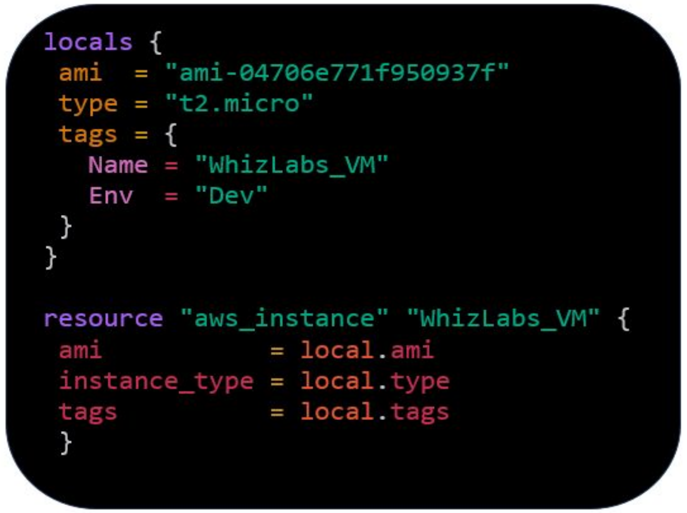
```
terraform import aws_s3_bucket.bucket my-whizlabs-bucket
```

Terraform Variables and Outputs

- In Terraform, **Variables** allow you to customize your values when you run your Terraform configuration. This feature of Terraform allows you to use and share the modules without altering the source code.

- **Local Variables** - Local variables are declared using the "**locals**" block. It is a group of key-value pair that can be used in the configuration.

  - Below example shows how to use local variable block for creating EC2 configuration.

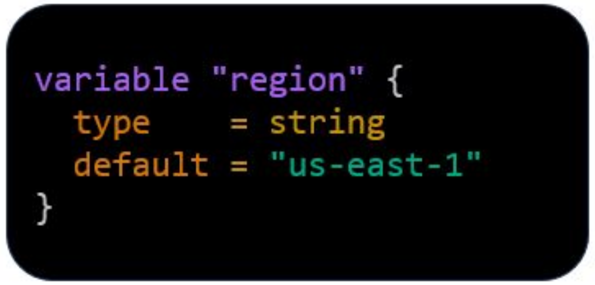


```
locals {
 ami  = "ami-04706e771f950937f"
 type = "t2.micro"
 tags = {
   Name = "WhizLabs_VM"
   Env  = "Dev"
 }
}

resource "aws_instance" "WhizLabs_VM" {
 ami           = local.ami
 instance_type = local.type
 tags          = local.tags
 }
```

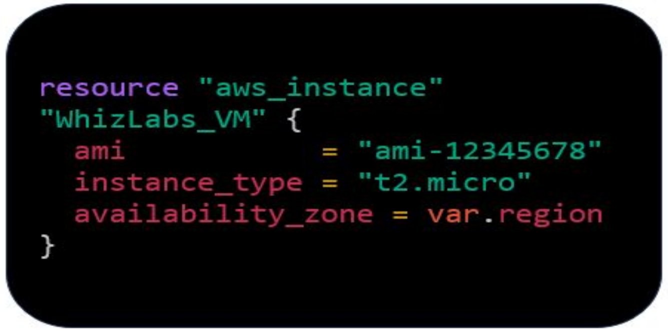
- **Input Variables** – Each input variable accepted by a module must be declared using a "**variable**" block.

  - In the below example, a variable named "region" of type "**string**" is declared with a default value of "ap-south-1".



```
variable "region" {
  type     = string
  default = "us-east-1"
}
```

- Below block shows how the variable will be called in the main configuration code.



```
resource "aws_instance"
"WhizLabs_VM" {
  ami               = "ami-12345678"
  instance_type = "t2.micro"
  availability_zone = var.region
}
```

Terraform Variables and Outputs

- The commonly used other types of Input variables are – **Number, Boolean, List, Map.**
  - Below is the example of variable type "**number**".

```
variable "number_type" {
 description = "This is a variable of type number"
 type        = number
 default     = 42
}
```

  - Below is the example of variable type "**boolean**".

```
variable "boolean_type" {
 description = "This is a variable of type bool"
 type        = bool
 default     = true
}
```

  - Below is the example of variable type "**list**".

```
variable "list_type" {
 description = "This is a variable of type list"
 type        = list(string)
 default     = ["string1", "string2", "string3"]
}
```

  - Below is the example of variable type "**map**".

```
variable "map_type" {
 description = "This is a variable of type map"
 type        = map(string)
 default     = {
   key1 = "value1"
   key2 = "value2"
 }
}
```

- **Environment Variables** - Input variable values can also be set using Terraform environment variables.
  - It can be done by setting the environment variable in the format TF_VAR_<variable name>.
  - The variable name part of the format is the same as the variables declared in the variables.tf file. For example, to set the ami variable run the below command to set its corresponding value.

```
export TF_VAR_ami=ami-0d26eb3972b7f8c96
```

Terraform Variables and Outputs

- **Passing variables through CLI and. tfvars** – In the above examples we have seen only default values being set.
  - o The default values can be overridden in two ways:
    - ▪ Passing the values in CLI as -var argument.
    - ▪ Using .tfvars file to set variable values explicitly.

```
terraform plan -var "ami=example" -var "type=t2.micro" -var "tags={\"name\":\"WhizLabs_VM\",\"env\":\"Dev\"}"
```

  - o Below example shows how to pass variables through .tfvars file.

```
ami   = “ami-0d26eb39724567c”
type = "t2.micro"
tags = {
  "name": “WhizLabs_VM”
  "env"  :  "Dev"
}
```

  - o The variables defined in .tfvars file can be used through plan command by passing the file name “**terraform plan -var-file values.tfvars**”.

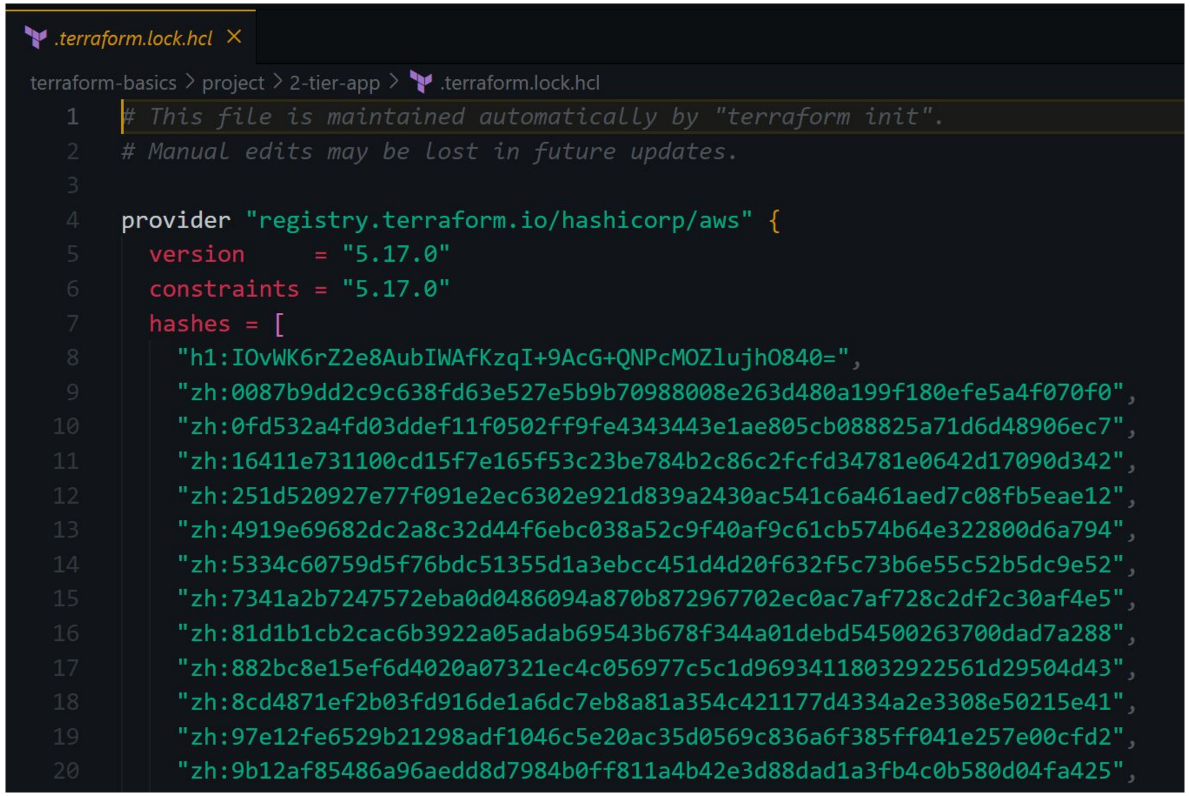
- **Precedence to variables** – Terraform allocate the precedence to know in which order the passed variables should be read.
  - o The precedence is shown below from first to last.
    - ▪ **cli arguments > .tfvars > default variables.**

- **Output variables** – When you deploy a large infrastructure using Terraform you would be required to read the values of some resources in other modules or will be required to print the values of provisioned resources and in that scenario, we will make use of “output” variable. This should be defined as a separate file “outputs.tf”.
  - o Below is the example block code how to use “**output**” variable.

```
output "instance_id" {
 value         = aws_instance.myvm.id
 description = "AWS EC2 instance ID"
 sensitive    = false
}
```

Terraform: Dependency Lock File

- Terraform 0.14 and above have introduced a feature that tracks dependency selections of your code.
- This Dependency lock file will be named "**.terraform.lock.hcl**".
- Terraform recommends including this file in your source code repository so that any changes to dependencies can be identified and updated.
- Dependency lock file can be updated by running "**terraform init**" command.



.terraform.lock.hcl

terraform-basics > project > 2-tier-app > .terraform.lock.hcl

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version     = "5.17.0"
  constraints = "5.17.0"
  hashes = [
    "h1:IOvWK6rZ2e8AubIWAfKzqI+9AcG+QNPcMOZlujhO840=",
    "zh:0087b9dd2c9c638fd63e527e5b9b70988008e263d480a199f180efe5a4f070f0",
    "zh:0fd532a4fd03ddef11f0502ff9fe4343443e1ae805cb088825a71d6d48906ec7",
    "zh:16411e731100cd15f7e165f53c23be784b2c86c2fcfd34781e0642d17090d342",
    "zh:251d520927e77f091e2ec6302e921d839a2430ac541c6a461aed7c08fb5eae12",
    "zh:4919e69682dc2a8c32d44f6ebc038a52c9f40af9c61cb574b64e322800d6a794",
    "zh:5334c60759d5f76bdc51355d1a3ebcc451d4d20f632f5c73b6e55c52b5dc9e52",
    "zh:7341a2b7247572eba0d0486094a870b872967702ec0ac7af728c2df2c30af4e5",
    "zh:81d1b1cb2cac6b3922a05adab69543b678f344a01debd54500263700dad7a288",
    "zh:882bc8e15ef6d4020a07321ec4c056977c5c1d96934118032922561d29504d43",
    "zh:8cd4871ef2b03fd916de1a6dc7eb8a81a354c421177d4334a2e3308e50215e41",
    "zh:97e12fe6529b21298adf1046c5e20ac35d0569c836a6f385ff041e257e00cfd2",
    "zh:9b12af85486a96aedd8d7984b0ff811a4b42e3d88dad1a3fb4c0b580d04fa425",
```

Terraform Functions

- Terraform functions are built-in, reusable code blocks that perform specific tasks within Terraform configurations. Functions make your code more dynamic.
  - Syntax example:

```
<FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>)
```

- Terraform as of now supports only built-in Functions. You cannot define your own Functions.
- Let's see different types of Terraform Functions and their example syntax.
  - **String** Functions – This allows you to perform various **String** operations such as format, indent, join, lower, split, etc.
    - Example code for "**join**" String function.

```
> join("-", ["t2.micro", "t2.nano", "t2.small"])
"t2.micro-t2.nano-t2.small"

> join(", ", ["t2.micro", "t2.nano", "t2.small"])
t2.micro, t2.nano, t2.small

> join(", ", ["t2.medium"])
t2.medium
```

  - **Numeric** Functions – This allows you to perform various **Numeric** operations such as min, max, floor, ceiling, etc.
    - Example code for "**min**" and "**max**" Numeric function.

```
> min(12, 54, 3)
3

> max(12, 54, 3)
54
```

  - **Collection** Functions – This allows you to perform various **Collection** operations such as lookup, list, map, concat and etc.
    - Example code for "**lookup**" Collection function.

```
> lookup({a="t2.micro", b="t2.nano"}, "a", "what?")
t2.micro

> lookup({a="aws", b="azure"}, "c", "google?")
google?
```

Terraform Functions

- **Filesystem** Functions – This allows you to perform various **Filesystem** operations such as dirname, basename, file, fileexists and etc.
  - Example code for “**file**” Filesystem function.

```
> file("${path.module}/hello.txt")
Hello World
```

- **Date and Time** Functions – This allows you to perform various **Date and Time** operations such as formatdate, timeadd, timestamp and etc.
  - Example code for “**formatdate**” Date and Time function.

```
> formatdate("DD MMM YYYY hh:mm ZZZ", "2023-11-04T23:12:01Z")
04 Nov 2023 23:12 UTC
```

- **Type Conversion** Functions – This allows you to perform various **Type Conversion** operations such as tobool, tolist, tomap, tonumber and etc.
  - Example code for “**tolist**” Type Conversion function.

```
> tolist(["t2.micro", "t2.nano", "t2.small"])
[
  "t2.micro",
  "t2.nano",
  "t2.small"
]
```

Terraform Registry

- Terraform **Registry** is a central repository for sharing, discovering, and using Terraform modules and provider plugins.
- Terraform's official registry website is found here https://registry.terraform.io/.
- Terraform **Registry** serves different purposes:
    - **Module sharing**: Terraform Registry allows you to share your Terraform code in the form of Terraform modules. These modules can be reused by others to provision common infrastructure components.
    - **Provider documentation**: Terraform providers are plugins that allow Terraform to interact with specific cloud and infrastructure platforms. Terraform Registry provides documentation and examples for such Terraform providers.
    - **Collaboration**: Terraform Registry allows collaboration between multiple users and organizations to contribute and maintain modules and providers.

Terraform Modules

What is a Terraform Module?

Terraform Module is a Container that holds multiple resource configuration files (.tf) and can be reused.

Module usage – Module "my-module" defined above can be referred/used like below.

```
module "my-module" {
   source =
"path/to/the/module"
   version = "1.0.0"

   argument_1 = var.test_1
   argument_1 = var.test_1
   argument_1 = var.test_1
}
```

- **Benefits of Terraform module:**
  - Reusability.
  - Keep the code DRY as much as possible.
  - Modules can be versioned to maintain consistency.
  - Collaboration across teams.

Module structure:

```
my-module/
|
|---main.tf          #main config file for the module
|---variables.tf     #input variable for the root module
|---outputs.tf       #output variable for the root module
|---README.d.        #usage instructions
```

- Sources - Terraform modules can be stored either **locally or remotely**. The `source` argument will change depending on their location.

- Version - Terraform modules can be versioned to manage the lifecycle and consistency of the modules you use.

- Root module – The root module is the entry point for your Terraform configuration.
  It's the main directory where your main Terraform files, normally named main.tf, variables.tf, and outputs.tf, are located.

- Child module - Child modules are reusable units of Terraform code that can be called from the root module or other child modules.
  Each child module can define its own set of variables, resources, and outputs.

Terraform Workspaces

- Terraform Workspaces allows you to manage resources in multiple environments using a single Terraform configuration code.

- For example, if you want to create an EC2 instance in 3 different environments say Dev, PPE, and Prod. Then you have to create three different Workspaces and switch between them and can use the same code to deploy EC2 in all 3 environments.

```
resource "aws_instance" "whizlabs-server" {
  ami           = "ami-0fb653ca2d321234"
  instance_type = "t2.micro"
  tags = {
    Name = "whizlabs-server"
  }
}
```

- Below is the code example of how you can use the same code and deploy the different types of EC2 instances based on the Workspace you are in.
  - The below code provisions “m4.large” EC2 if you are in the “prod” workspace and provisions “t2.micro” if you are in any other workspace.

```
resource "aws_instance" "whizlabs-server" {
  ami           = "ami-0fb653ca2d567gde"
  instance_type = terraform.workspace == "prod" ? "m4.large" : "t2.micro"
  tags = {
    Name = "whizlabs-server-${terraform.workspace}"
  }
}
```

- To check the current Workspace.

```
terraform workspace show
default
```

- To create a new Workspace and switch to it.

```
terraform workspace new whizlabs_workspace
Created and switched to workspace "whizlabs_workspace"!
```

- To list existing Workspaces.

```
terraform workspace list
* default
* dev
* ppe
* prod
```

Terraform Cloud Workspaces are nothing but Workspaces that will be created in Terraform Cloud rather than in your local machine.

Terraform Cloud

- **Terraform vs Terraform Cloud** – Terraform is an open-source platform that you need not pay anything for. Whereas, if you want to have extended features of Terraform such as Security, CI/CD, SSO, etc. you can make use of the Terraform Cloud and this comes with a charge.

- Terraform Cloud is a platform that helps manage your Terraform code in the Cloud rather than on your local machine. This allows teams to use Terraform together.

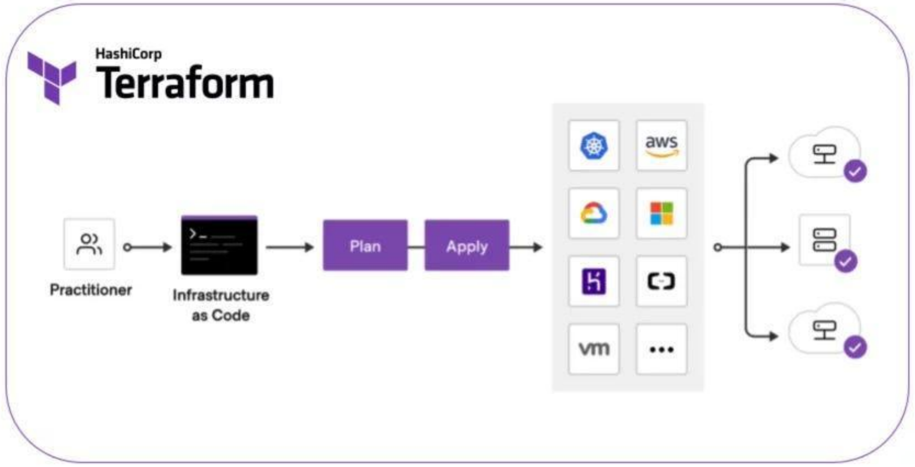


Image source: Terraform official document

- Terraform Cloud comes with a private registry that allows you to share your organization's Terraform modules and providers.

- Terraform Cloud comes with below features:

  - **Variables and Variable Set** - Terraform Cloud stores your variables securely, encrypting them at rest. Variable Sets,help you reuse variables in multiple workspaces, without having to declare them multiple times.

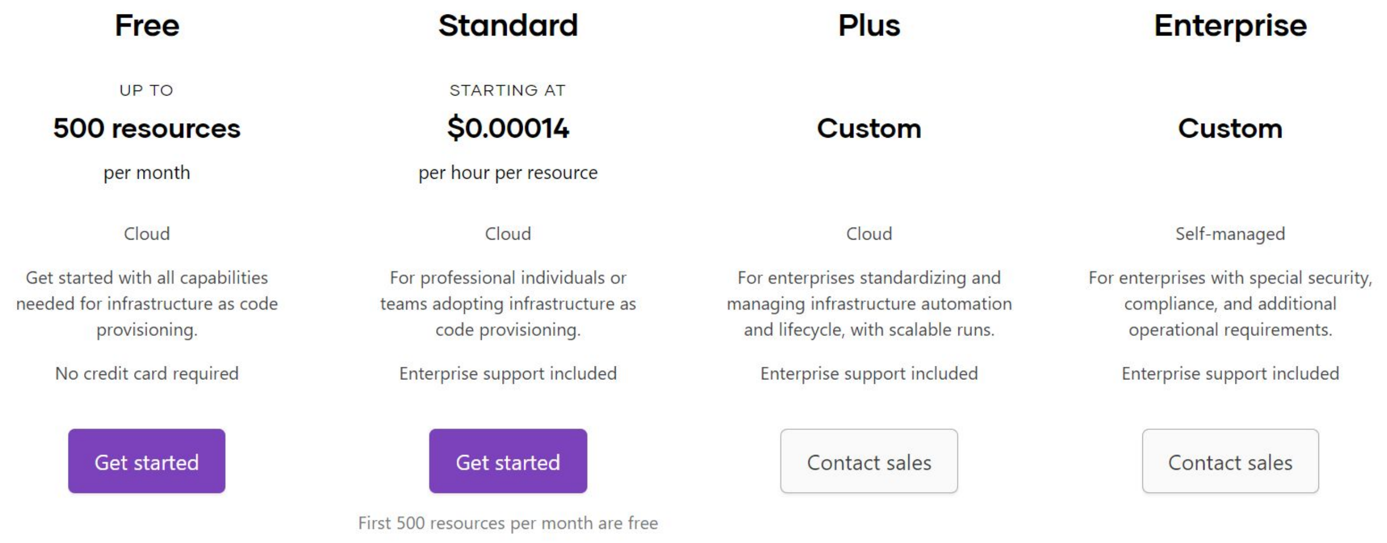
  - **Remote State** – Terraform Cloud comes with a feature to store your State file remotely and securely. This feature allows your team members to access the State file and work collaboratively.

  - **Policies** - With Terraform Cloud, you get Sentinel policies to enforce security practices and governance throughout your workflow.

Terraform Cloud

- o **Single Sign-On** - Terraform Cloud enables you to achieve SSO by using an identity provider such as Okta, SAML, or Microsoft Azure AD.
- o **Private Registry** - Private registry is feature for hosting your Terraform modules and providers.
- o **Drift detection** - Terraform Cloud comes with a feature that detects the drift in your Infrastructure. Drift you will be notified to you, and you will be able to quickly fix the issue.

- **Terraform Pricing** – The details of Terraform Cloud pricing can be found here at the official Terraform website https://www.hashicorp.com/products/terraform/pricing.



Free	Standard	Plus	Enterprise
UP TO	STARTING AT		
500 resources	$0.00014	Custom	Custom
per month	per hour per resource		
Cloud	Cloud	Cloud	Self-managed
Get started with all capabilities needed for infrastructure as code provisioning.	For professional individuals or teams adopting infrastructure as code provisioning.	For enterprises standardizing and managing infrastructure automation and lifecycle, with scalable runs.	For enterprises with special security, compliance, and additional operational requirements.
No credit card required	Enterprise support included	Enterprise support included	Enterprise support included

Image source: Terraform official document