What Why and How Terraform

Before getting into the Tutorial, we should understand Why Terraform is needed in the first place. Creating and managing application instances in Datacenters is a really crucial job and it cannot even afford a small mistake or misconfiguration. Wait! even application developers do mistakes with their application codes but how they are managing it to avoid big downtime or issue? Yes, They have everything coded and everyone has their own versions of code which can be reviewed and approved!. Likewise, creating code for Infrastructure will help misconfiguration, and most importantly, it will automate the way long infrastructure processes. Now, Terraform provide such a platform that will store the configuration of infrastructure with configuration, provision, and manage feature as code. We call that Infrastructure as Code (IaC).

Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc. Terraform can manage both existing service providers and custom in-house solutions.

Terraform's configuration language is declarative, meaning that it describes the desired end-state for your infrastructure, in contrast to procedural programming languages that require step-by-step instructions to perform tasks. Terraform providers automatically calculate dependencies between resources to create or destroy them in the correct order.

Terraform is HashiCorp's infrastructure as a code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure.

Terraform is an orchestration engine and language that enables you to safely and predictably create, change, and improve production infrastructure. It is an open-source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned.

What is Terraform used for?

* External resource management -- Terraform supports public and private cloud infrastructure, as well as network appliances and software as a service (SaaS) deployments.
* Multi-cloud deployment -- the software tool's native ability to support multiple cloud services helps increase fault tolerance.
* Multi-tier applications -- Terraform allows each resource collection to easily be scaled up or down as needed.
* Self-service clusters -- the registries make it easy for users to find prepackaged configurations that can be used as is or modified to meet a particular need.
* Software-defined networking (SDN) -- Terraform's readability makes it easy for network engineers to codify the configuration for an SDN.
* Resource scheduler -- Terraform modules can stop and start resources on AWS and allow Kubernetes to schedule Docker containers.
* Disposable environments -- modules can be used to create an ad hoc, throwaway test environment for code before it's put into production.

How does Terraform work?

Terraform allows users to define their entire infrastructure simply by using configuration files and version control. When a command is given to deploy and run a server, database or load balancer, Terraform parses the code and translates it into an application programming interface (API) call to the resource provider. Because Terraform is open source, developers are always able to extend the tool's usefulness by writing new plugins or compiling different versions of existing plugins.

Terraform has two important components: **Terraform Core** and **Terraform Plugins**.

**Terraform Core** oversees the reading and interpolation of resource plan executions, resource graphs, state management features and configuration files. Core is composed of compiled binaries written in the Go programming language. Each compiled binary acts as a command-line interface (CLI) for communicating with plugins through remote procedure calls (RPC).

**Terraform Plugins** are responsible for defining resources for specific services. This includes authenticating infrastructure providers and initializing the libraries used to make API calls. Terraform Plugins are written in Go as executable binaries that can either be used as a specific service or as a provisioner. (Provisioner plugins are used to execute commands for a designated resource.)

Features of Terraform

* **Reuse**: Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.
* **Execution Plans**: Terraform generates an execution plan describing what it will do and asks for your approval before making any infrastructure changes. This allows you to review changes before Terraform creates, updates, or destroys infrastructure.
* **Resource Graph**: Terraform builds a resource graph and creates or modifies non-dependent resources in parallel. This allows Terraform to build resources as efficiently as possible and gives you greater insight into your infrastructure.
* **Change Automation:** Terraform can apply complex changesets to your infrastructure with minimal human interaction. When you update configuration files, Terraform determines what changed and creates incremental execution plans that respect dependencies.
* Terraform can manage infrastructure on multiple cloud platforms.
* The human-readable configuration language helps you write infrastructure code quickly.
* Terraform's state allows you to track resource changes throughout your deployments.
* You can commit your configurations to version control to safely collaborate on infrastructure.
* **Manage any infrastructure:**Terraform plugins called providers let Terraform interact with cloud platforms and other services via their application programming interfaces (APIs). HashiCorp and the Terraform community have written over 1,000 providers to manage resources on Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, and DataDog, just to name a few. Find providers for many of the platforms and services you already use in the Terraform Registry.
* If you don't find the provider you're looking for you can write your own provider plugin.
* make incremental changes to resources.
* Standardize your deployment workflow Providers define individual units of infrastructure, for example compute instances or private networks, as resources. You can compose resources from different providers into reusable Terraform configurations called modules, and manage them with a consistent language and workflow.
* **Track your infrastructure:**Terraform keeps track of your real infrastructure in a state file, which acts as a source of truth for your environment. Terraform uses the state file to determine the changes to make to your infrastructure so that it will match your configuration.
* **Collaborate:** Terraform allows you to collaborate on your infrastructure with its remote state backends. When you use Terraform Cloud (free for up to five users), you can securely share your state with your teammates, provide a stable environment for Terraform to run in, and prevent race conditions when multiple people make configuration changes at once.
* You can also connect Terraform Cloud to version control systems (VCSs) like GitHub, GitLab, and others, allowing it to automatically propose infrastructure changes when you commit configuration changes to VCS. This lets you manage changes to your infrastructure through version control, as you would with application code.

Benefits of Using Terraform

Manual Needs many Human Resources which can be Managed by a single Terraform Platform.

Since we need fewer human resources, the Cost of Infrastructure management can be reduced drastically.

Misconfiguration and Manual errors can be taken away by Terraform

Scaling or High Availability of the Application stack within Infrastructure can be easily achieved with Terraform.

The Infrastructure setup is available as Configuration Files, It can be Documented and Reviewed every version of the changes.

No Blame Games in Infrastructure Configurations. It can be traced back and Identified easily.

# Terraform Installations

In this blog, We are going to install terraform in the Ubuntu operating system.

Ensure that your system is up to date, and you have the gnupg, software-properties-common, and curl packages installed. You will use these packages to verify HashiCorp's GPG signature, and install HashiCorp's Debian package repository.

$ sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl

Add the HashiCorp GPG key.

$ curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -

Add the official HashiCorp Linux repository.

$ sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb\_release -cs) main"

Update to add the repository, and install the Terraform CLI.

$ sudo apt-get update && sudo apt-get install terraform

## Verify the installation

Verify that the installation worked by opening a new terminal session and listing Terraform's available subcommands.

$ terraform -help

Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.

The most common, useful commands are shown first, followed by

less common or more advanced commands. If you're just getting

started with Terraform, stick with the common commands. For the

other commands, please read the help and docs before usage.

##...

Add any subcommand to terraform -help to learn more about what it does and available options.

$ terraform -help

# First Terraform Configuration

## Terraform Language

The Terraform language is Terraform's primary user interface. In every edition of Terraform, a configuration written in the Terraform language is always at the heart of the workflow.

The main purpose of the Terraform language is declaring resources, which represent infrastructure objects.

## ****Terraform configuration****

A Terraform configuration is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. A configuration can consist of multiple files and directories.

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {

# Block body

<IDENTIFIER> = <EXPRESSION> # Argument

}

## ****File Extension****

Code in the Terraform language is stored in plain text files with the**.tf**file extension. There is also a**JSON-based variant** of the language that is named with the **.tf.json** file extension.

## ****Text Encoding****

Configuration files must always use UTF-8 encoding, and by convention usually use Unix-style line endings (LF) rather than Windows-style line endings (CRLF), though both are accepted

let's create first terraform configurations

create a file first.tf in the present working directory with the below content

// this is comment

# this is comment

output hello1 {

    value = "Hello World 1234"

}

now let's run **terraform plan** command

output

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world]

└─$ terraform plan

Changes to Outputs:

  + hello1 = "Hello World 1234"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world]

└─$

# Terraform Configurations As JSON File

Code in the Terraform language is stored in plain text files with the**.tf**file extension. There is also a **JSON-based variant**of the language that is named with the**.tf.json**file extension.

Terraform also supports an alternative syntax that is JSON-compatible. This syntax is useful when generating portions of a configuration programmatically, since existing JSON libraries can be used to prepare the generated configuration files.

let's create our first.tf.json file in the present working directory with the below content

{

"output" : {

  "hello1": {

      "value": "Hello Gaurav"

     }

  }

}

Run the **terraform plan**command and see the output.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-json]

└─$ terraform plan

Changes to Outputs:

  + hello1 = "Hello Gaurav"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-json]

└─$

# Multiple Block in Single Terraform File

In the previous blogs, we used single-output block in a file. in this blog, we will demonstrate that we can use multiple blocks in the same file.

let create the **first.tf** file in the present working directory with the below content. here we can see that we are using multiple blogs in the same file. make source that label must be different.

output "firstoutputblock" {

        value = " this is first hello world block"

}

output "secondoutputblock" {

        value = "this is second hello world block"

}

output "thirdoutputblock" {

        value = "this is third hello world block"

}

now run **terraform plan**and see the output.

**Output:**

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-multi-block]

└─$ terraform plan

Changes to Outputs:

  + firstoutputblock  = " this is first hello world block"

  + secondoutputblock = "this is second hello world block"

  + thirdoutputblock  = "this is third hello world block"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

# Terraform Variables

So in last blog we see that we can put multiple files in a directory. and in this blogs we will use variables and print that variables. so let's start and create a file **first.tf** with below content and you can change the name of files as per your choice.

variable username {}

output printname {

    value = "Hello, ${var.username}"

}

In the above content we declare a variable **username** with the help of **variable block.** The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module.

## Using Input Variable Values

Within the module that declared a variable, its value can be accessed from within expressions as **var.<NAME>**, where <NAME> matches the label given in the declaration block (here is username)

so if we want to use a variable in terraform then we can use it using **var.variableName**and "${var.variableName}" inside string.

now lets run the terraform plan command and see the output.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-variable]

└─$ terraform plan

var.username

  Enter a value: Learning-Ocean

Changes to Outputs:

  + printname = "Hello, Learning-Ocean"

You can apply this plan to save these new output values to the Terraform state, without changing any

real infrastructure.

───────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly

these actions if you run "terraform apply" now.

Terraform will take value of username as input (we give it **Learning-Ocean**) and it will return the output as **Hello, Learning-Ocean.**

## Divide Terraform files

in the previous blogs, we have seen that we can put multiple terraform configurations files in same directory. let's implement it and divide the above first.tf file.

let create a files variable.tf with the below content, in this file we will only define the variables.

variable username {}

let create one more file hello-variable.tf with below content.

output printname {

    value = "Hello, ${var.username}"

}

now lets run the terraform plan command and see the output.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-variable]

└─$ ls

hello-variable.tf  variable.tf

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-variable]

└─$ terraform plan

var.username

  Enter a value: Learning-Ocean

Changes to Outputs:

  + printname = "Hello, Learning-Ocean"

You can apply this plan to save these new output values to the Terraform state, without changing any

real infrastructure.

# Pass Variable from Command Line

In last blogs we create two files. variable.tf and hello-variable.tf with the below content.

variable.tf contains the below content.

variable username {}

hello-variable.tf with below content.

output printname {

    value = "Hello, ${var.username}"

}

then we run **terraform plan**then terraform will ask us variable value that is**not defined.**

Now we want to supply the variable value from the command line. so that we can run terraform plan command in non-interactive node.

for that we will use below.

syntex:

terraform plan -var "<VARIABLE\_NAME>=<VARIABLE\_VALUE>"

let's run the below command and see the output

$ terraform plan -var "username=Learning-Ocean"

output:

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-variable]

└─$ terraform plan -var "username=Learning-Ocean"

Changes to Outputs:

  + printname = "Hello, Learning-Ocean"

You can apply this plan to save these new output values to the Terraform state, without changing any

real infrastructure.

───────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly

these actions if you run "terraform apply" now.

Terraform Variable Type

The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable. If no type constraint is set then a value of any type is accepted.

type constraints are optional but its recommended by terraform to specifying them.

Type constraints are created from a mixture of type keywords and type constructors. The supported type keywords are:

* string
* number
* bool

The type constructors allow you to specify complex types such as collections:

* list
* map

If both the type and default arguments are specified, the given default value must be convertible to the specified type.

Variable description

Because the input variables of a module are part of its user interface, you can briefly describe the purpose of each variable using the optional description argument.

let's create a file variable.tf in your current working directory. with the below content.

variable username {

type = string

default = "world"

}

variable age {

type = number

default = 23

}

and create one more file printvariable.tf with the below content

output "printvariable" {

value = "Hello ${var.username} and your age is ${var.age}"

}

you can pass the variable value from the command line using below syntax

terraform plan -var "variablename=variablevalue" -var "variable2name=varible2value"

now lets run terraform plan command in actions.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-variable]

└─$ terraform plan -var "username=Learning-Ocean" -var "age=23"

Changes to Outputs:

  + printname = "Hello, Learning-Ocean, your age is 23"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

if we pass age as String it will give an error.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/hello-variable]

└─$ terraform apply -var username="Gaurav" -var age=gaurav

╷

│ Error: Invalid value for input variable

│

│ The argument -var="age=..." does not contain a valid value for variable "age": a number is required.

╵

│ Error: Incorrect variable type

│

│   on variable.tf line 5:

│    5: variable "age" {

│

│ The resolved value of variable "age" is not appropriate: a number is required.

# List Variable

A **list variable** holds a list of values (for example, name of users) to be used. Each list variable specifies the order.

let see with example create a file variable.tf with below contents.

variable users {

    type = list

}

let's create one more file that will access user from the users variables. (list variable.)

output printfirst {

  value = "first user is ${(var.users[1])}"

}

in the above file we will accessing second element of the array. because indexing starting from the zero. let run the **terraform plan**and see the output.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/list-variable]

└─$ terraform plan

var.users

  Enter a value: [ "gaurav", "saurav", "ankit"]

Changes to Outputs:

  + printfirst = "first user is saurav"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

here we can see the second element of the array in the output.

we can add the default value using the below snippet

variable users {

  type = list

  default = ["gaurav","Saurav","anKit"]

}

and we can pass the list variable from the command line using below syntax.

terraform plan -var 'variablename=["valueone","valuetwo","value3"]'

let's run terraform apply and see it in action.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/list-variable]

└─$ terraform plan -var 'users=["gaurav","saurav","ankit"]'

Changes to Outputs:

  + printfirst = "first user is saurav"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/list-variable]

└─$

# Terraform Functions

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The general syntax for function calls is a function name followed by comma-separated arguments in parentheses

max(5, 12, 9)

The Terraform language**does not support user-defined functions**, and so only the functions built into the language are available for use.

## Numeric Functions

### **abs**

abs returns the absolute value of the given number. In other words, if the number is zero or positive then it is returned as-is, but if it is negative then it is multiplied by -1 to make it positive before returning it.

> abs(24)

23

> abs(0)

0

> abs(-24.4)

24.4

### **ceil**

ceil returns the closest whole number that is greater than or equal to the given value, which may be a fraction.

> ceil(5)

5

> ceil(5.1)

6

### **floor**

floor returns the closest whole number that is less than or equal to the given value, which may be a fraction.

> floor(5)

5

> floor(4.9)

4

### **log**

log returns the logarithm of a given number in a given base.

> log(50, 10)

1.6989700043360185

> log(16, 2)

4

### **max**

max takes one or more numbers and returns the greatest number from the set.

> max(12, 54, 3)

54

### **min**

min takes one or more numbers and returns the smallest number from the set.

> min(12, 54, 3)

3

### **pow**

pow calculates an exponent, by raising its first argument to the power of the second argument.

> pow(3, 2)

9

> pow(4, 0)

1

## String Functions

### **join**

join produces a string by concatenating together all elements of a given list of strings with the given delimiter.

> join(", ", ["foo", "bar", "baz"])

foo, bar, baz

> join(", ", ["foo"])

foo

### **upper**

upper converts all cased letters in the given string to uppercase.

> upper("hello")

HELLO

### **lower**

lower converts all cased letters in the given string to lowercase.

> lower("HELLO")

hello

### **title**

title converts the first letter of each word in the given string to uppercase.

> title("hello world")

Hello World

lets take an example of String functions

create a file variable.tf with below code:

variable users {

    type = list

    default = ["gaurav","Saurav","anKit"]

}

create a file first.tf (you can change the file name as per your convenient.

output printfirst {

        value = "${join("--->",var.users)}"

}

output helloworldupper {

        value = "${upper(var.users[0])}"

}

output helloworldlower {

        value = "${lower(var.users[1])}"

}

output helloworldtitle {

        value = "${title(var.users[2])}"

}

let run terraform apply and see the output:

└─$ terraform plan

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

helloworldlower = "saurav"

helloworldtitle = "AnKit"

helloworldupper = "GAURAV"

printfirst = "gaurav--->Saurav--->anKit"

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/functions]

└─$

**Below are the types of other functions with function names**:

**Numeric**Functions : abs, ceil, floor, log, max, min, parseint, pow, signum

**String Functions**: chomp, format, formalist, join, lower, regex, regexall, rep[lace, split, strrev, title, trim, trimprefix, trimsuffix, trimspace, upper

**Collection Functions**: alltrue, anytrue, chunklist, coalesce, coalescelist, compact, concat, contains, distinct, element, flatten, index, keys, length, list, lookup, map, matchkeys,merge, one, range, reverse, setintersection, setproduct, setsubsctract, setunion, slice, sort, sum, transpose, values, zipmap

**Encoding Functions**: base643ncode, base64decode, base64gzip, csvdecode, jsonencode, jsondecode, urlencode, yamlencode, yamldecode

**Filesystem functions**: absath, dirname, pathexpand, basename, file, fileexists, fileset, filebase64, templatefile

**Date & Time Functions**: formade, timeadd, timestamp

**Hash and Crypto Functions**: base64sha256, base64sha512, bcrypt, filebase64sha512, filemd5, filesha1, filesha256, filesha512, md5, rsadecrypt, sha, sha256, sha512, uuid, uuidv5

**IP Network Functions**: cidrhost, cidrnetmask, cidrsubnets

**Type Conversion functions**: can, defaults, nonsensative, sensitive, tobool, tolist, tomap, tonumber, toset, tostring, try

Reference : <https://www.terraform.io/language/functions>

# Terraform Map Variable

A map value is a lookup table from string keys to string values. This is useful for selecting a value based on some other provided value.

A common use of maps is to create a table of machine images per region, as follows:

variable "images" {

  type    = "map"

  default = {

    "us-east-1" = "image-1234"

    "us-west-2" = "image-4567"

  }

}

lets take an simple example of terraform map

create a file with tf extension.

variable "usersage" {

    type = map

    default = {

        gaurav = 20

        saurav = 19

    }

}

output "userage" {

    value = "my name is gaurav and my age is ${lookup(var.usersage, "gaurav")}"

}

now let's run terraform apply and see the output

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/map-variable]

└─$ terraform plan

Changes to Outputs:

  + userage = "my name is gaurav and my age is 20"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

userage = "my name is gaurav and my age is 20"

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/map-variable]

└─$

## How to Read values Dynamically from Map Variable:

lets create a file

variable "usersage" {

    type = map

    default = {

        gaurav = 20

        saurav = 19

    }

}

variable "username" {

  type = string

}

output "userage" {

    value = "my name is ${var.username} and my age is ${lookup(var.usersage, "${var.username}")}"

}

now lets run terraform apply

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/map-variable]

└─$ terraform plan -var username="gaurav"

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

userage = "my name is gaurav and my age is 20"

## How to pass Map Variable From Command line

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/map-variable]

└─$ terraform plan -var username="gaurav" -var usersage="{"gaurav"=22,"saurav"=23}"                                                                       1 ⨯

Changes to Outputs:

  + userage = "my name is gaurav and my age is 22"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

Terraform TFVARs Files

To set lots of variables, it is more convenient to specify their values in a *variable definitions file* (with a filename ending in either .tfvars or .tfvars.json).

Terraform automatically loads a number of variable definitions files if they are present:

1. Files named exactly terraform.tfvars or terraform.tfvars.json.
2. Any files with names ending in .auto.tfvars or .auto.tfvars.json.

let's take an example

create a file with tf extension

variable age {

    type = number

}

variable "username" {

  type = string

}

output printname {

        value = "Hello, ${var.username}, your age is ${var.age}"

}

create a file with **terraform.tfvars**

age=25

username="Gaurav Sharma"

now lets run command terraform plan or terraform apply

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/tf-var]

└─$ terraform plan

Changes to Outputs:

  + printname = "Hello, Gaurav Sharma, your age is 25"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

TFVARs File With Different Name

if you have .tfvars file with different name then need to specify the filename explicitly like the below command

terraform apply -var-file="testing.tfvars"

let's take an how can we use tfvars file with a different name other than terraform.tfvars

create a file with tf extension

variable age {

    type = number

}

variable "username" {

  type = string

}

output printname {

        value = "Hello, ${var.username}, your age is ${var.age}"

}

create a file with **development.tfvars**(you can change it as per your requirement)

age=25

username="Saurav Sharma"

now run the below command and specify the tfvar files explicity.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/tf-var-custom]

└─$ terraform plan --var-file development.tfvars

Changes to Outputs:

  + printname = "Hello, Saurav Sharma, your age is 25"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

# Terraform Environment Variable

Terraform searches the environment of its own process for environment variables named TF\_VAR\_ followed by the name of a declared variable.

This can be useful when running Terraform in automation, or when running a sequence of Terraform commands in succession with the same variables.

On operating systems where environment variable names are case-sensitive, Terraform matches the variable name exactly as given in configuration, and so the required environment variable name will usually have a mix of upper and lower case letters.

let's take an example create a terraform file

variable "username" {

  type = string

}

output printname {

        value = "Hello, ${var.username}"

}

now set the value of username variable in the environment using the below command

export TF\_VAR\_username=Hinal

now run terraform plan command and see the output.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/env-variable]

└─$ export TF\_VAR\_username=Hinal

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/env-variable]

└─$ terraform plan

Changes to Outputs:

  + printname = "Hello, Hinal"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

──────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

# Multiple Terraform File in the Same Directory

let's create the below files in the same directory

add the below content in the **first.tf** file

output "firstoutputblock" {

        value = " this is first hello world block"

}

add the below content in **second.tf** file

output "secondoutputblock" {

        value = "this is second hello world block"

}

add the below content in **third.tf** file

output "thirdoutputblock" {

        value = "this is third hello world block"

}

now we have four files in our current working directory lets list them using **ls** command.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$ ls

first.tf  second.tf  third.tf

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$

now let's run **terraform plan** command in the same directory.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$ terraform plan

Changes to Outputs:

  + firstoutputblock  = " this is first hello world block"

  + secondoutputblock = "this is second hello world block"

  + thirdoutputblock  = "this is third hello world block"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

so, when we run **terraform plan or apply**command then all the tf files that are present in the present working directory are loaded.

Now I have a questions for you guys,

## How these files will get loaded while running terraform plan?

so let's create one more file in **abc.tf** in the same directory with the below command

output "abc" {

        value = "this is abc hello world block"

}

now we have four files in our current working directory lets list them using **ls** command.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$ ls

abc.tf  first.tf  second.tf  third.tf

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$

let's run terraform plan command and check the output.

┌──(gaurav㉿learning-ocean)-[~/youtube-course/hello-world-file-destructure]

└─$ terraform plan

Changes to Outputs:

  + abc               = "this is abc hello world block"

  + firstoutputblock  = " this is first hello world block"

  + secondoutputblock = "this is second hello world block"

  + thirdoutputblock  = "this is third hello world block"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

so the answer is **all the files loaded in the alphabetic order.**

Terraform Variable Precedence

The above mechanisms for setting variables can be used together in any combination. If the same variable is assigned multiple values, Terraform uses the *last* value it finds, overriding any previous values. Note that the same variable cannot be assigned multiple values within a single source.

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

1. Environment variables
2. The terraform.tfvars file, if present.
3. The terraform.tfvars.json file, if present.
4. Any \*.auto.tfvars or \*.auto.tfvars.json files, processed in lexical order of their filenames.
5. Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

# Terraform Core And Plugins

Terraform is logically split into two main parts: **Terraform Core** and **Terraform Plugins**. Terraform Core uses remote procedure calls (RPC) to communicate with Terraform Plugins, and offers multiple ways to discover and load plugins to use. Terraform Plugins expose an implementation for a specific service, such as AWS, or provisioner, such as bash.

## Terraform Core

Terraform Core is a statically-compiled binary written in the Go programming language. The compiled binary is the command line tool (CLI) terraform, the entrypoint for anyone using Terraform.

## Responsibilities of terraform Core

* Infrastructure as code: reading and interpolating configuration files and modules
* Resource state management
* Plan execution
* Construction of the Resource Graph
* Communication with plugins over RPC

## Terraform Plugins

Terraform Plugins are written in Go and are executable binaries invoked by Terraform Core over RPC. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash.

All Providers and Provisioners used in Terraform configurations are plugins.

They are executed as a separate process and communicate with the main Terraform binary over an RPC interface.

Terraform has several Provisioners built-in, while Providers are discovered dynamically as needed.

Terraform Core provides a high-level framework that abstracts away the details of plugin discovery and RPC communication so developers do not need to manage either.

Terraform Plugins are responsible for the domain-specific implementation of their type.

## Responsibilities of Provider Plugins are:

* Initialization of any included libraries used to make API calls
* Authentication with the Infrastructure Provider
* Define Resources that map to specific Services

## Responsibilities of Provisioner Plugins are:

* Executing commands or scripts on the designated Resource after creation, or on destruction.

# Create Github Repo From Terraform

create a file in a folder with .tf extension with the below contents

provider "github" {

token="ghp\_lEvJ2GJxXgBnyZlVfQXXXXXXXXXjMUK"

}

resource "github\_repository" "terraform-first-repo" {

name = "first-repo-from-terraform"

description = "My First resource for my youtube viewers."

visibility = "public"

auto\_init = true

}

you can get token value from GitHub setting [pages](https://github.com/settings/tokens).

and change the name, and description as per your requirement.

## Terraform providers

now let's run **terraform providers** command and see the output.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/terraform-first-resource]

└─$ terraform providers

Providers required by configuration:

.

└── provider[registry.terraform.io/hashicorp/github]

Providers required by state:

    provider[registry.terraform.io/hashicorp/github]

The terraform providers command shows information about the provider requirements of the configuration in the current working directory, as an aid to understanding where each requirement was detected from.

## Terraform init

This command performs several different initialization steps in order to prepare the current working directory for use with Terraform.

This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration.

In order to prepare the working directory for use with Terraform, the terraform init command performs the following steps:

* Backend Initialization
* Child Module Installation
* Plugin Installation

we have very simple configurations as of now (we are not using the child module and backed till now but we will use it in future blogs.) so terraform init just install the plugin.

now let's run terraform init command see the output.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/terraform-first-resource]

└─$ terraform init                                                                                                                                        1 ⨯

Initializing the backend...

Initializing provider plugins...

- Reusing previous version of hashicorp/github from the dependency lock file

- Using previously-installed hashicorp/github v4.19.1

╷

│ Warning: Additional provider information from registry

│

│ The remote registry returned warnings for registry.terraform.io/hashicorp/github:

│ - For users on Terraform 0.13 or greater, this provider has moved to integrations/github. Please update your source in required\_providers.

╵

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see

any changes that are required for your infrastructure. All Terraform commands

should now work.

If you ever set or change modules or backend configuration for Terraform,

rerun this command to reinitialize your working directory. If you forget, other

commands will detect it and remind you to do so if necessary.

## Terraform Plan

This command looks in the current working directory for the root module configuration. The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. After a successful initialization of the working directory and the completion of the plugin download, we can create an execution plan using **terraform plan** command, this is a handy way to check whether the execution plan matches your expectations without making any changes to real resources or to the state. By default, when Terraform creates a plan it:

* Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
* Compares the current configuration to the prior state and noting any differences.
* Proposes a set of change actions that should, if applied, make the remote objects match the configuration.
* If Terraform detects that no changes are needed to resource instances or to root module output values, terraform plan will report that no actions need to be taken.
* If the Terraform discovers no changes to resources, then the terraform plan indicates that no changes are required to the real infrastructure.
* Terraform also helps to save the plan to a file for later execution with **terrafom apply,**which can be useful while applying automation with Terraform. This can be achieved by using **-out** argument.

now let's run the terraform plan command and see the output.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/terraform-first-resource]

└─$ terraform plan

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

  + create

Terraform will perform the following actions:

  # github\_repository.terraform-first-repo will be created

  + resource "github\_repository" "terraform-first-repo" {

      + allow\_auto\_merge       = false

      + allow\_merge\_commit     = true

      + allow\_rebase\_merge     = true

      + allow\_squash\_merge     = true

      + archived               = false

      + auto\_init              = true

      + branches               = (known after apply)

      + default\_branch         = (known after apply)

      + delete\_branch\_on\_merge = false

      + description            = "My First resource for my youtube viewers."

      + etag                   = (known after apply)

      + full\_name              = (known after apply)

      + git\_clone\_url          = (known after apply)

      + html\_url               = (known after apply)

      + http\_clone\_url         = (known after apply)

      + id                     = (known after apply)

      + name                   = "first-repo-from-terraform"

      + node\_id                = (known after apply)

      + private                = (known after apply)

      + repo\_id                = (known after apply)

      + ssh\_clone\_url          = (known after apply)

      + svn\_url                = (known after apply)

      + visibility             = "public"

    }

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

───────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

so here we can see what action our configurations will perform on the provider (github.)

now let's talk about **terraform apply** command

## Terraform apply

By default, apply scans the current directory for the configuration and applies the changes appropriately. However, a path to another configuration or an execution plan can be provided.

Terraform apply command is used to **create** or**introduce** changes to real infrastructure.

By default, apply scans the current working directory for the configuration and applies the changes appropriately.

However, you’ll optionally give the path to a saved plan file that was previously created with terraform plan.

If you do not provides a plan file on the instruction, terraform apply will create a replacement plan automatically then prompt for approval to use it. If the created plan does not include any changes to resources or to root module output values then terraform apply will exit immediately, without prompting.

┌──(gaurav㉿learning-ocean)-[~/terraform/youtube-course/terraform-first-resource]

└─$ terraform apply

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

  + create

Terraform will perform the following actions:

  # github\_repository.terraform-first-repo will be created

  + resource "github\_repository" "terraform-first-repo" {

      + allow\_auto\_merge       = false

      + allow\_merge\_commit     = true

      + allow\_rebase\_merge     = true

      + allow\_squash\_merge     = true

      + archived               = false

      + auto\_init              = true

      + branches               = (known after apply)

      + default\_branch         = (known after apply)

      + delete\_branch\_on\_merge = false

      + description            = "My First resource for my youtube viewers."

      + etag                   = (known after apply)

      + full\_name              = (known after apply)

      + git\_clone\_url          = (known after apply)

      + html\_url               = (known after apply)

      + http\_clone\_url         = (known after apply)

      + id                     = (known after apply)

      + name                   = "first-repo-from-terraform"

      + node\_id                = (known after apply)

      + private                = (known after apply)

      + repo\_id                = (known after apply)

      + ssh\_clone\_url          = (known after apply)

      + svn\_url                = (known after apply)

      + visibility             = "public"

    }

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

Do you want to perform these actions?

  Terraform will perform the actions described above.

  Only 'yes' will be accepted to approve.

  Enter a value: yes

github\_repository.terraform-first-repo: Creating...

github\_repository.terraform-first-repo: Creation complete after 6s [id=first-repo-from-terraform]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

now in Github, we can see a new repository created.