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.

# Terraform TFState File

When we build infrastructure with terraform configuration, a state file will be created automatically in the local workspace directory named “terraform.tfstate”. This tfstate file will have information about the provisioned infrastructure which terraform manage. Whenever we make changes to the configuration file, it will automatically determine which part of your configuration is already created. And, also it will determine which needs to be changed with the help of the state file.

you may have noticed that when you ran the terraform plan or terraform apply commands, Terraform was able to find the resources it created previously and update them accordingly. But how did Terraform know which resources it was supposed to manage? You could have all sorts of infrastructure in your AWS account deployed through a variety of mechanisms (some manually, some via Terraform, some via the CLI), so how does Terraform know which infrastructure it’s responsible for?

The answer is that Terraform records information about what infrastructure it created in a Terraform state file. By default, when you run Terraform in the folder, Terraform creates the file terraform.tfstate in that folder. This file contains a custom JSON format that records a mapping from the Terraform resources in your templates to the representation of those resources in the real world.

# Terraform Destroy

in the previous blog, we create a terraform resource we can delete it using terraform destroy command.

## Terraform Destroy

The terraform destroy is used to destroy infrastructure governed by terraform. To check the behavior of **terraform destroy**command at any time we can use **terraform plan -destroy** command.

.let's use terraform destroy command to delete the infra that we created in the last blog.

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

└─$ terraform destroy

github\_repository.terraform-first-repo: Refreshing state... [id=first-repo-from-terraform]

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

  - destroy

Terraform will perform the following actions:

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

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

      - allow\_auto\_merge       = false -> null

      - allow\_merge\_commit     = true -> null

      - allow\_rebase\_merge     = true -> null

      - allow\_squash\_merge     = true -> null

      - archived               = false -> null

      - auto\_init              = true -> null

      - branches               = [

          - {

              - name      = "main"

              - protected = false

            },

        ] -> null

      - default\_branch         = "main" -> null

      - delete\_branch\_on\_merge = false -> null

      - description            = "My First resource for my youtube viewers." -> null

      - etag                   = "W/\"f09b3b6b2064f4478f01a77d22ac102992fb35bc660e\"" -> null

      - full\_name              = "XXXXXXXX/first-repo-from-terraform" -> null

      - git\_clone\_url          = "git://github.com/XXXXXXXX/first-repo-from-terraform.git" -> null

      - has\_downloads          = false -> null

      - has\_issues             = false -> null

      - has\_projects           = false -> null

      - has\_wiki               = false -> null

      - html\_url               = "https://github.com/XXXXXXXX/first-repo-from-terraform" -> null

      - http\_clone\_url         = "https://github.com/XXXXXXXXv/first-repo-from-terraform.git" -> null

      - id                     = "first-repo-from-terraform" -> null

      - is\_template            = false -> null

      - name                   = "first-repo-from-terraform" -> null

      - node\_id                = "R\_kgDOHehMxxhw" -> null

      - private                = false -> null

      - repo\_id                = 501763207 -> null

      - ssh\_clone\_url          = "git@github.com:XXXXXXXXX/first-repo-from-terraform.git" -> null

      - svn\_url                = "https://github.com/XXXXXXXX/first-repo-from-terraform" -> null

      - visibility             = "public" -> null

      - vulnerability\_alerts   = false -> null

    }

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

Do you really want to destroy all resources?

  Terraform will destroy all your managed infrastructure, as shown above.

  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

github\_repository.terraform-first-repo: Destroying... [id=first-repo-from-terraform]

github\_repository.terraform-first-repo: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.

For the deletion of the particular resource and its dependencies **-target**flag can be used

example:

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

└─$ terraform destroy --target github\_repository.terraform-second-repo

Terraform Validate

Once you’ve initialized the directory, it’s good to run the validate command before you run the plan or apply. Validation will catch syntax errors, version errors, and other issues. One thing to note here is that you can’t run validate before you run the init command. You have to initialize the working directory before you can run the validation.

Validate works at the level of checking your code for soundness, including loading modules and ensuring that variables are correctly named.

If you only want the basic syntax of a local file, consider using terraform fmt -write=false. If the formatter is unable to parse the file, it will throw an error.

This command **does not** check formatting (e.g. tabs vs spaces, newlines, comments etc.).

The following can be reported:

* invalid HCL syntax (e.g. missing trailing quote or equal sign)
* invalid HCL references (e.g. variable name or attribute which doesn't exist)
* same provider declared multiple times
* same module declared multiple times
* same resource declared multiple times
* invalid module name
* interpolation used in places where it's unsupported (e.g. variable, depends\_on, module.source, provider)
* missing value for a variable (none of -var foo=... flag, -var-file=foo.vars flag, TF\_VAR\_foo environment variable, terraform.tfvars, or default value in the configuration)

now let's run terraform validate command in the same folder that we create in our previous blogs (create first resource)

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

└─$ terraform validate

Success! The configuration is valid.

# Terraform Refresh

The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure but does modify the state file. If the state is changed, this may cause changes to occur during the next plan or apply.

By default, refresh requires no flags and looks in the current directory for the configuration and state file to refresh.

now let's run terraform refresh command in the folder that we created in last videos.

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

└─$ terraform refresh

github\_repository.terraform-first-repo: Refreshing state... [id=first-repo-from-terraform]

# Terraform Fmt

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.

Other Terraform commands that generate Terraform configuration will produce configuration files that conform to the style imposed by terraform fmt, so using this style in your own files will ensure consistency.

The canonical format may change in minor ways between Terraform versions, so after upgrading Terraform we recommend to proactively run terraform fmt on your modules along with any other changes you are making to adopt the new version.

By default, fmt scans the current directory for configuration files. If the dir argument is provided then it will scan that given directory instead. If dir is a single dash (-) then fmt will read from standard input (STDIN).

let's run terraform fmt command in our folder that we created in previous blogs.

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

└─$ terraform fmt

terraform.tf

Terraform Output

The terraform output command is used to extract the value of an output variable from the state file.

Usage: terraform output [options] [NAME]

With no additional arguments, output will display all the outputs for the root module. If an output NAME is specified, only the value of that output is printed.

The command-line flags are all optional. The list of available flags are:

* -json - If specified, the outputs are formatted as a JSON object, with a key per output. If NAME is specified, only the output specified will be returned. This can be piped into tools such as jq for further processing.
* -state=path - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state is used.
* -module=module\_name - The module path which has needed output. By default this is the root path. Other modules can be specified by a period-separated list. Example: "foo" would reference the module "foo" but "foo.bar" would reference the "bar" module in the "foo" module.

**Example**

let's create a file with tf extension with below content.

provider "github" {

  token = "XXXXXXXXXXXXXXXXXXX"

}

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

  name        = "first-repo-from-terraform"

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

  visibility  = "public"

  auto\_init   = true

}

output "terraform-repo-url" {

  value = github\_repository.terraform-first-repo.html\_url

}

now run terraform init, terraform apply after that let's try with terraform output.

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

└─$ terraform output terraform-repo-url

"https://github.com/crgaurav/first-repo-from-terraform"

# Terraform Console

The terraform console command provides an interactive console for evaluating expressions.

This command provides an interactive command-line console for evaluating and experimenting with expressions.

This is useful for testing interpolations before using them in configurations, and for interacting with any values currently saved in state.

If the current state is empty or has not yet been created, the console can be used to experiment with the expression syntax and built-in functions.

The dir argument specifies the directory of the root module to use. If a path is not specified, the current working directory is used.

The supported options are:

* -state=path - Path to a local state file. Expressions will be evaluated using values from this state file. If not specified, the state associated with the current [workspace](http://man.hubwiz.com/docset/Terraform.docset/Contents/Resources/Documents/docs/state/workspaces.html) is used.

You can close the console with the exit command or by pressing Control-C or Control-D.

# Create First AWS Resource

let's create a directory with name **aws-first-instance**

$ mkdir aws-first-instance

let's create a file with the name provider.tf (you can change the name of the file as per your requirement.) with the below content.

provider "aws" {

  region     = "us-east-1"

  access\_key = "YOUR\_ACCESS\_KEY\_HERE"

  secret\_key = "YOUR\_SECRET\_KEY\_HERE"

}

put your access key and secret key that you get from aws.

now we are going to create an instance on AWS so let's create one more file in the same directory named as instance.tf with the below content

resource "aws\_instance" "web" {

  ami                    = "ami-0b0ea68c435eb488d"

  instance\_type          = "t2.micro"

  tags = {

    Name = "first-tf-instance"

  }

}

now let's run terraform init command.

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

└─$ terraform init

Initializing the backend...

Initializing provider plugins...

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

- Using previously-installed hashicorp/aws v3.71.0

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.

now lets run terraform apply

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

└─$ terraform apply

and you can see on AWS console that one instance gets created.

# Create an SSH key in AWS using Terraform

In this blog, we are going to create an ssh key on AWS and assign it to a newly created AWS instance.

let's create an ssh keypair on our local system using the **ssh-keygen** command

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

└─$ ssh-keygen -t rsa                                                                                                                                     1 ⨯

Generating public/private rsa key pair.

Enter file in which to save the key (/home/kali/.ssh/id\_rsa): ./id\_rsa

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in ./id\_rsa

Your public key has been saved in ./id\_rsa.pub

The key fingerprint is:

SHA256:8CdVo44oVuPiN4AdAKt2sLTiYWpTqCzyF1s5YBcOvWo gaurav@learning-ocean

The key's randomart image is:

+---[RSA 3072]----+

|...  .      o    |

| . .. o    o .   |

|.o  .o+o  o      |

|o =oo++= +       |

|oB.+\*+o.S o      |

|O +oE++  o       |

|== ..+o.         |

|+.. o. .         |

|  ..             |

+----[SHA256]-----+

now we can see that **id\_rsa** and **id\_rsa.pub** files were created.

now let's create a file aws-kp.tf with the below content.

# creating ssh-key.

resource "aws\_key\_pair" "key-tf" {

  key\_name   = "key-tf"

  public\_key = file("${path.module}/id\_rsa.pub")

}

now let's run terraform plan and apply the command we can see that the new key pair create on aws.

$ terraform plan

$ terraform apply

# Assign SSH-key to AWS Instance Using Terraform

in the last blog we create a key pair, in this blog, we will assign that key to the AWS instance. let's modify the instance.tf file(that we created in the previous blog) with the below content.

and make sure in your current directory have**id\_rsa, id\_rsa.pub, provider.tf, instance.tf, aws-kp.tf** that we created in previous blogs.

resource "aws\_instance" "web" {

  ami                    = "ami-0b0ea68c435eb488d"

  instance\_type          = "t2.micro"

  key\_name               = aws\_key\_pair.key-tf.key\_name

  tags = {

    Name = "first-tf-instance"

  }

}

first, destroy the instance that we created in previous blogs using terraform destroy command.

and run terraform apply command and verify that new instance created and ssh key assigned to that instance.

you can ssh that instance with the help of that id\_rsa key.

example

ssh -i id\_rsa username@ipAddressOfInstance

# Terraform Dynamic Block

In this blog, we are going to create a security group and assign it to the instance.

## ****Terraform dynamic blocks****

**Terraform dynamic blocks** are a special Terraform block type that provides the functionality of a **for expression** by creating multiple nested blocks.

The need to create identical (or similar) infrastructure resources is common. A standard use case is multiple virtual server instances on a cloud platform like AWS or Azure Terraform provides routines such as for\_each and count to simplify deploying these resources, removing the requirement for large blocks of duplicate code.

Additionally, teams may need to configure multiple duplicate elements within a resource. In conjunction with a for\_each routine, dynamic blocks are used within an infrastructure resource to remove the need for multiple duplicate “blocks” of Terraform code.

## Benefits of dynamic blocks

The key benefits of Terraform dynamic blocks are:

* **Speed**– simplifying the code makes it much quicker to write and also for it to be processed and thus for the infrastructure to be deployed.
* **Reliability**– linked to clarity and re-use, errors are less likely to be made in simple, easy-to-read code.
* **Re-use**– copying, pasting, and amending large blocks of code is difficult and tedious. Combine dynamic blocks and variables/parameters to streamline this process.
* **Clarity**– in contrast to multiple blocks of repetitive code, it’s much easier to read and understand code written using dynamic blocks.

let's create a file aws-sg.tf with below content.

# creating security group

resource "aws\_security\_group" "allow\_tls" {

  name        = "allow\_tls"

  description = "Allow TLS inbound traffic"

  dynamic "ingress" {

    for\_each = [80,8080,443,9090,9000]

    iterator = port

    content {

      description = "TLS from VPC"

      from\_port   = port.value

      to\_port     = port.value

      protocol    = "tcp"

      cidr\_blocks = ["0.0.0.0/0"]

    }

  }

  egress {

    from\_port        = 0

    to\_port          = 0

    protocol         = "-1"

    cidr\_blocks      = ["0.0.0.0/0"]

    ipv6\_cidr\_blocks = ["::/0"]

  }

}

In the above code, we are using a dynamic block to open ports 80,8080,443,9090,9000. you change the values as per your requirement.

now let's modify the instance.tf file, with the below content.

# creating instance.

resource "aws\_instance" "web" {

  ami                    = data.aws\_ami.ubuntu.id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

}

let's destroy first the already create infrastructure using the terraform destroy command and then again run terraform apply command you can verify that an instance, security group, and ssh key pair are created and the key-pair and security group attached to that instance.

# Terraform Project Structure

In this blog, we going variablise the files that we created in previous blogs.

let's modify the provider.tf with the below content.

provider "aws" {

  region     = "us-east-1"

  access\_key = var.access\_key

  secret\_key = var.secret\_key

}

modify instance.tf with the below content.

# creating instance.

resource "aws\_instance" "web" {

  ami                    = var.image\_id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

}

modify aws-kp.tf with below content.

# creating ssh-key.

resource "aws\_key\_pair" "key-tf" {

  key\_name   = "key-tf"

  public\_key = file("${path.module}/id\_rsa.pub")

}

modify aws-sg.tf with below content

# creating security group

resource "aws\_security\_group" "allow\_tls" {

  name        = "allow\_tls"

  description = "Allow TLS inbound traffic"

  dynamic "ingress" {

    for\_each = var.ports

    iterator = port

    content {

      description = "TLS from VPC"

      from\_port   = port.value

      to\_port     = port.value

      protocol    = "tcp"

      cidr\_blocks = ["0.0.0.0/0"]

    }

  }

  egress {

    from\_port        = 0

    to\_port          = 0

    protocol         = "-1"

    cidr\_blocks      = ["0.0.0.0/0"]

    ipv6\_cidr\_blocks = ["::/0"]

  }

}

create a new file variable.tf with the below content

variable "ports" {

  type = list(number)

}

variable "instance\_type" {

  type = string

}

variable "access\_key" {

  type = string

}

variable "secret\_key" {

  type = string

}

variable "image\_name" {

  type = string

}

now create a file **terraform.tfvars** file with the below contents.

ports         = [22, 80, 443, 3306, 27017, 1080]

instance\_type = "t2.micro"

image\_id = "ami-0b0ea68c435eb488d"

access\_key = "YOUR\_AWS\_ACCESS\_KEY"

secret\_key = "YOUR\_AWS\_SECRET\_KEY"

# Terraform Taint

The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.

This command will not modify infrastructure but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource. For example: re-running provisioners will cause the node to be different or rebooting the machine from a base image will cause new startup scripts to run.

Note that tainting a resource for recreation may affect resources that depend on the newly tainted resource. For example, a DNS resource that uses the IP address of a server may need to be modified to reflect the potentially new IP address of a tainted server. The plan command will show this if this is the case.

## Example: Tainting a Resource

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

└─$ terraform taint aws\_security\_group.allow\_tls

# AWS UserData Using Terraform

AWS userdata is **the set of commands/data you can provide to a instance at launch time**. For example if you are launching an ec2 instance and want to have docker installed on the newly launched ec2, than you can provide set of bash commands in the userdata field of aws ec2 config page.

In this blog, we are going to assign user-data to AWS instance with the help of terraform.

lets create a file named script.sh with the below content.

#!/bin/bash

apt-get update

apt-get install nginx -y

echo "Hi Gaurav" >/var/www/html/index.nginx-debian.html

now modify instance.tf file with the below content, so we can use this script as user data.

resource "aws\_instance" "web" {

  ami                    = var.image\_id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

}

Provisioners

Provisioners are used to executing scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next terraform apply. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

You can change this behavior by setting the on\_failure attribute, which is covered in detail below.

example

resource "aws\_instance" "web" {

# ...

provisioner "local-exec" {

command = "echo ${self.private\_ip} > file.txt"

}

}

Destroy-Time Provisioners

If when = "destroy" is specified, the provisioner will run when the resource it is defined within is *destroyed*.

Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next terraform apply. Due to this behavior, care should be taken for destroy provisioners to be safe to run multiple times.

A destroy-time provisioner within a resource that is tainted *will not* run. This includes resources that are marked tainted from a failed creation-time provisioner or tainted manually using terraform taint.

resource "aws\_instance" "web" {

# ...

provisioner "local-exec" {

when = "destroy"

command = "echo 'Destroy-time provisioner'"

}

}

Multiple Provisioners

Multiple provisioners can be specified within a resource. Multiple provisioners are executed in the order they're defined in the configuration file.

You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run. Those valid provisioners will be run in the order they're defined in the configuration file.

resource "aws\_instance" "web" {

# ...

provisioner "local-exec" {

command = "echo first"

}

provisioner "local-exec" {

command = "echo second"

}

}

Failure Behavior

By default, provisioners that fail will also cause the Terraform apply itself to error. The on\_failure setting can be used to change this. The allowed values are:

* "continue" - Ignore the error and continue with creation or destruction.
* "fail" - Error (the default behavior). If this is a creation provisioner, taint the resource.

resource "aws\_instance" "web" {

# ...provisioner "local-exec" {

command = "echo ${self.private\_ip} > file.txt"

on\_failure = "continue"

}

}

# File Provisioner

The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both ssh and winrm type connections.

let's modify instance.tf file to use file provisioner. and also create a readme.md file with any content because we are going to copy that file to remote instance.

resource "aws\_instance" "web" {

  ami                    = data.aws\_ami.ubuntu.id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

  connection {

    type        = "ssh"

    user        = "ubuntu"

    private\_key = file("${path.module}/id\_rsa")

    host        = self.public\_ip

  }

  # file, local-exec, remote-exec

  provisioner "file" {

    source      = "readme.md"      # terraform machine

    destination = "/tmp/readme.md" # remote machine

  }

  provisioner "file" {

    content     = "this is test content" # terraform machine

    destination = "/tmp/content.md"      # remote machine

  }

}

## Directory Uploads

The file provisioner is also able to upload a complete directory to the remote machine. When uploading a directory, there are a few important things you should know.

First, when using the ssh connection type the destination directory must already exist. If you need to create it, use a remote-exec provisioner just prior to the file provisioner in order to create the directory.

If the source is /foo (no trailing slash), and the destination is /tmp, then the contents of /foo on the local machine will be uploaded to /tmp/foo on the remote machine. The foo directory on the remote machine will be created by Terraform.

If the source, however, is /foo/ (a trailing slash is present), and the destination is /tmp, then the contents of /foo will be uploaded directly into /tmp.

This behavior was adopted from the standard behavior of [rsync](https://linux.die.net/man/1/rsync" \t "_blank).

# Copy a directory

provisioner "file" {

source = "conf"

destination = "/home/ec2-user"

}

# Local-Exec Provisioner

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

let's use local-exec, modify instance.tf file with below content.

resource "aws\_instance" "web" {

  ami                    = data.aws\_ami.ubuntu.id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

  connection {

    type        = "ssh"

    user        = "ubuntu"

    private\_key = file("${path.module}/id\_rsa")

    host        = self.public\_ip

  }

provisioner "local-exec" {

    command = "echo testing > /tmp/test.txt"

  }

provisioner "local-exec" {

working\_dir = "/tmp"

    command = "echo ${self.public\_ip}>myip.txt"

  }

  provisioner "local-exec" {

interpreter = [

"/usr/bin/python3", "-c"

]

    command = "print('HelloWorld')"

  }

provisioner "local-exec" {

    command = "env>env.txt"

    environment = {

      envname = "envvalue"

    }

  }

}

# Remote-Exec Provisioner

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. The remote-exec provisioner supports both ssh and winrm type connections.

# creating instance.

resource "aws\_instance" "web" {

  ami                    = data.aws\_ami.ubuntu.id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

  connection {

    type        = "ssh"

    user        = "ubuntu"

    private\_key = file("${path.module}/id\_rsa")

    host        = self.public\_ip

  }

  provisioner "remote-exec" {

    inline = [

      "ifconfig > /tmp/ifconfig.output",

      "echo 'hello gaurav'>/tmp/test.txt"

    ]

  }

  provisioner "remote-exec" {

    script = "./testscript.sh"

  }

}

# Data Sources

Data sources allow Terraform to use the information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

A data source is accessed via a special kind of resource known as a data resource, declared using a data block.

data "aws\_ami" "example" {

most\_recent = trueowners = ["self"]

tags = {

Name = "app-server"Tested = "true"

}

}

A data block requests that Terraform read from a given data source ("aws\_ami") and export the result under the given local name ("example"). The name is used to refer to this resource from elsewhere in the same Terraform module, but has no significance outside of the scope of a module.

The data source and name together serve as an identifier for a given resource and so must be unique within a module.

Within the block body (between { and }) are query constraints defined by the data source. Most arguments in this section depend on the data source, and indeed in this example most\_recent, owners and tags are all arguments defined specifically for the aws\_ami data source.

When distinguishing from data resources, the primary kind of resource (as declared by a resource block) is known as a managed resource. Both kinds of resources take arguments and export attributes for use in configuration, but while managed resources cause Terraform to create, update, and delete infrastructure objects, data resources cause Terraform only to read objects. For brevity, managed resources are often referred to just as "resources" when the meaning is clear from context.

now let's modify our code to use data sources.

modify terraform.tfvars files with the below content.

ports         = [22, 80, 443, 3306, 27017, 1080]

instance\_type = "t2.micro"

image\_name    = "ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-\*"

now modify the instance.tf file.

data "aws\_ami" "ubuntu" {

  most\_recent = true

  owners      = ["099720109477"]

  filter {

    name   = "name"

    values = ["${var.image\_name}"]

  }

  filter {

    name   = "root-device-type"

    values = ["ebs"]

  }

  filter {

    name   = "virtualization-type"

    values = ["hvm"]

  }

}

# creating instance.

resource "aws\_instance" "web" {

  ami                    = data.aws\_ami.ubuntu.id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

  vpc\_security\_group\_ids = ["${aws\_security\_group.allow\_tls.id}"]

  tags = {

    Name = "first-tf-instance"

  }

  user\_data = file("${path.module}/script.sh")

  connection {

    type        = "ssh"

    user        = "ubuntu"

    private\_key = file("${path.module}/id\_rsa")

    host        = self.public\_ip

  }

  # file, local-exec, remote-exec

  provisioner "file" {

    source      = "readme.md"      # terraform machine

    destination = "/tmp/readme.md" # remote machine

  }

  provisioner "file" {

    content     = "this is test content" # terraform machine

    destination = "/tmp/content.md"      # remote machine

  }

  provisioner "local-exec" {

    command = "env>env.txt"

    environment = {

      envname = "envvalue"

    }

  }

  provisioner "local-exec" {

    command = "echo 'at Create'"

  }

  provisioner "local-exec" {

    when    = destroy

    command = "echo 'at delete'"

  }

  provisioner "remote-exec" {

    inline = [

      "ifconfig > /tmp/ifconfig.output",

      "echo 'hello gaurav'>/tmp/test.txt"

    ]

  }

  provisioner "remote-exec" {

    script = "./testscript.sh"

  }

}

kindly notice we replace image\_id with data source value.

now try to run terraform apply and see an instance is created with ubuntu-20.04 version.

Terraform Block

The special terraform configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration.

Each terraform block can contain a number of settings related to Terraform's behavior. Within a terraform block, only constant values can be used; arguments may not refer to named objects such as resources, input variables, etc, and may not use any of the Terraform language built-in functions.

Specifying a Required Terraform Version

The required\_version setting can be used to constrain which versions of the Terraform CLI can be used with your configuration. If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

The value for required\_version is a string containing a comma-separated list of constraints. Each constraint is an operator followed by a version number, such as > 0.12.0. The following constraint operators are allowed:

* = (or no operator): exact version equality
* !=: version not equal
* >, >=, <, <=: version comparison, where "greater than" is a larger version number
* ~>: pessimistic constraint operator, constraining both the oldest and newest version allowed. For example, ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4, is equivalent to >= 0.8.4, < 0.9

Specifying Required Provider Versions

The required\_providers setting is a map specifying a version constraint for each provider required by your configuration.

This is one of several ways to define provider version constraints and is particularly suited to re-usable modules that expect a provider configuration to be provided by their caller but still need to impose a minimum version for that provider.

let's use terraform block in our code. create a new file named terraform.tf with the below content

terraform {

  required\_version = "1.1.0"

  required\_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "3.71.0"

    }

  }

}

we can also specify backend in terraform block that we will use in future blogs.

# Terraform Graph

The terraform graph command is used to generate a visual representation of either a configuration or execution plan. The output is in the DOT format, which can be used by GraphViz to generate charts.

Outputs the visual dependency graph of Terraform resources according to configuration files in DIR (or the current directory if omitted).

The graph is outputted in DOT format. The typical program that can read this format is GraphViz, but many web services are also available to read this format.

The -type flag can be used to control the type of graph shown. Terraform creates different graphs for different operations. See the options below for the list of types supported. The default type is "plan" if a configuration is given, and "apply" if a plan file is passed as an argument.

The output of terraform graph is in the DOT format, which can easily be converted to an image by making use of dot provided by GraphViz:

$ terraform graph | dot -Tsvg > graph.svg

$ terraform graph | dot -Tpdf > graph.pdf

# Workspace

The terraform workspace command is used to manage workspaces.

## List

terraform workspace list command is used to list the workspaces. by default, **default** workspace is created.The current workspace is indicated using an asterisk (\*) marker.

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

└─$ terraform workspace list

\* default

## Create

The **terraform workspace new** command is used to create a new workspace.

This command will create a new workspace with the given name. A workspace with this name must not already exist

$ terraform workspace new dev

let's create a new workspace using below command

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

└─$ terraform workspace new test

Created and switched to workspace "test"!

You're now on a new, empty workspace. Workspaces isolate their state,

so if you run "terraform plan" Terraform will not see any existing state

for this configuration.

list all the created workspace

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

└─$ terraform workspace list

  default

\* test

we can see that test workspace created and selected.

## Show

**terraform workspace show** command will display the current workspace.

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

└─$ terraform workspace show

test

so right now you are in test workspace and if you run terraform apply then tf.state will be create in that workspace.

## Switch Workspace

If you want to switch workspace then you can use **terraform workspace select command.**

example:

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

└─$ terraform workspace list

  default

\* test

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

└─$ terraform workspace select default

Switched to workspace "default".

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

└─$ terraform workspace list

\* default

  test

## Delete Workspace

If you want to delete any existing workspace then you can use terraform delete command.

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

└─$ terraform workspace delete test

Deleted workspace "test"!

# Terraform Module

Up to this point, we've been configuring Terraform by editing Terraform configurations directly. As our infrastructure grows, this practice has a few key problems: a lack of organization, a lack of reusability, and difficulties in management for teams.

**Modules** in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components, improve organization, and to treat pieces of infrastructure as a black box.

A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

The .tf files in your working directory when you run terraform plan or terraform apply together form the root module. That module may call other modules and connect them together by passing output values from one to input values of another.

## Module structure

Re-usable modules are defined using all of the same configuration language concepts we use in root modules. Most commonly, modules use:

* Input variables to accept values from the calling module.
* Output values to return results to the calling module, which it can then use to populate arguments elsewhere.
* Resources to define one or more infrastructure objects that the module will manage.

To define a module, create a new directory for it and place one or more .tf files inside just as you would do for a root module. Terraform can load modules either from local relative paths or from remote repositories; if a module will be re-used by lots of configurations you may wish to place it in its own version control repository.

Modules can also call other modules using a module block, but we recommend keeping the module tree relatively flat and using [module composition](http://man.hubwiz.com/docset/Terraform.docset/Contents/Resources/Documents/docs/modules/composition.html) as an alternative to a deeply-nested tree of modules, because this makes the individual modules easier to re-use in different combinations.

## When to write a module

In principle, any combination of resources and other constructs can be factored out into a module, but over-using modules can make your overall Terraform configuration harder to understand and maintain, so we recommend moderation.

A good module should raise the level of abstraction by describing a new concept in your architecture that is constructed from resource types offered by providers.

Now let's create a module and use them. create a **modules/webserver**folder in your current working directory.

Now create a file /modules/webserver/resource.tf with below content.

# key

resource "aws\_key\_pair" "key-tf" {

  key\_name   = var.key\_name

  public\_key = var.key

}

# instance

resource "aws\_instance" "web" {

  ami                    = var.image\_id

  instance\_type          = var.instance\_type

  key\_name               = aws\_key\_pair.key-tf.key\_name

}

Create a file /modules/webserver/variables.tf with below content.

variable "instance\_type" {

  type = string

}

variable "image\_id" {

    type = string

}

variable "key" {

    type = string

}

variable "key\_name" {

  type= string

}

Create a file /modules/webserver/output.tf with the below content.

output publicIP {

    value = aws\_instance.web.public\_ip

}

now use this module, let's create a file in your current directory.

create variable.tf with the below content.

variable "access\_key" {

  type = string

}

variable "secret\_key" {

  type = string

}

variable "instance\_type" {

  type = string

}

variable "image\_id" {

    type = string

}

variable "key\_name"{

  type = string

}

create a file named resource.tf with the below content.

provider "aws" {

  region     = "us-east-1"

  access\_key = var.access\_key

  secret\_key = var.secret\_key

}

module "mywebserver" {

  source = "./modules/webserver"

  key = file("${path.module}/id\_rsa.pub")

  image\_id = "${var.image\_id}"

  instance\_type = "${var.instance\_type}"

  key\_name = "${var.key\_name}"

}

output mypublicIp {

  value = module.mywebserver.publicIP

}

create a file named terraform.tfvars with the below contents

instance\_type = "t2.micro"

image\_id   = "ami-01b996646377b6619"

key\_name= "mykey"

now let's run terraform init, and terraform apply and check instance is created in your AWS account.

Terraform Backend with Locking

A "backend" in Terraform determines how state is loaded and how an operation such as apply is executed. This abstraction enables non-local file state storage, remote execution, etc.

By default, Terraform uses the "local" backend, which is the normal behavior of Terraform you're used to.

Here are some of the benefits of backends:

* **Working in a team**: Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Enterprise even automatically store a history of all state revisions.
* **Keeping sensitive information off disk**: State is retrieved from backends on demand and only stored in memory. If you're using a backend such as Amazon S3, the only location the state ever is persisted is in S3.
* **Remote operations**: For larger infrastructures or certain changes, terraform apply can take a long, long time. Some backends support remote operations which enable the operation to execute remotely. You can then turn off your computer and your operation will still complete. Paired with remote state storage and locking above, this also helps in team environments.

**Backends are completely optional**. You can successfully use Terraform without ever having to learn or use backends. However, they do solve pain points that afflict teams at a certain scale. If you're an individual, you can likely get away with never using backends.

Even if you only intend to use the "local" backend, it may be useful to learn about backends since you can also change the behavior of the local backend.

In this blog we are using s3 as a backend.

let's create an S3 bucket (here I am creating a bucket with the name **gaurav-youtube-tf**). and create a dynamoDB table (here I am creating it with name **gaurav-youtube-tf-table**)

let's create a file with tf extension ( i am naming it as resource.tf )

terraform {

  backend "s3" {

    bucket = "gaurav-youtube-tf"

    region = "us-east-1"

    key="terraform.tfstate"

    dynamodb\_table = "gaurav-youtube-tf-table"

  }

}

variable "access\_key" {

  type = string

}

variable "secret\_key" {

  type = string

}

provider "aws" {

  region     = "us-east-1"

  access\_key = var.access\_key

  secret\_key = var.secret\_key

}

resource "aws\_instance" "web" {

  ami           = "ami-0e472ba40eb589f49"

  instance\_type = "t2.small"

}

now let's run terraform init command and after that terraform apply. you can see that terraform.tfstate file is not stored in your local system it will be stored in your s3 bucket that you created and configured. (gaurav-youtube-tf).

and one more thing two or more developers can not run terraform apply at the same time.