Terraform

Terraform is a tool for building, changing, versioning, and destroying infrastructure safely and efficiently.

Terraform can manage existing and popular service providers as well as a custom in-house solution which has been developed by HashiCorp.

A provider is responsible for understanding API interactions and exposing resources.

Most of the available providers correspond to one cloud or on-premises infrastructure platform and offers resource types that correspond to each of the features of that platform.

To make a provider available on Terraform, we need to make a terraform init, these commands download any plugins we need for our providers.

If for example, we need to copy the plugin directory manually, we can do it, moving the files to. terraform.d/plugins

If the plugin is already installed, terraform init will not download again unless to upgrade the version, run terraform init -upgrade.

Terraform useful commands:

```
$ terraform init
$ terraform validate
$ terraform plan
  + indicates resource creation
  - indicates resource deletion
  +/- indicates resource recreation
  ~ indicates resource modification
$ terraform apply -auto-approve
$ terraform show
$ terraform destroy -auto-approve
```

Terraform

Amar Gajula

Multiple Providers

You can optionally define multiple configurations for the same provider and select which one to use on a per-resource or per-module basis. Here, we are creating two Virtual Machines in two different regions.

```
#default configuration provider "aws" {
  region = "us-east-1"
}

# reference this as `aws.west`. provider "aws" {
          alias = "west"
  region = "us-west-2"
}
```

Versioning

The required version setting can be used to constrain which version 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.

Please check the version of the terraform and include the below mentioned script as per your requirement in the starting of the main.yml file.

```
terraform {
  required_version = ">= 0.12"
}
```

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 >
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

We can also specify a provider version requirement

```
provider "aws" {
          region = "us-east-1"
          version = ">= 2.9.0"
}
```

Show version: $ terraform version

Amar Gajula

Workspace

Terraform starts with a single workspace named “default”.

The workspace feature of Terraform allows users to switch between multiple instances of a single configuration with a unique state file.

For local states, terraform stores the workspace states in a directory called terraform.tfstate.d.
Workspace commands

The terraform workspace new command is used to create a new workspace and switched to a new workspace.

The terraform workspace list command is used to list all existing workspaces.

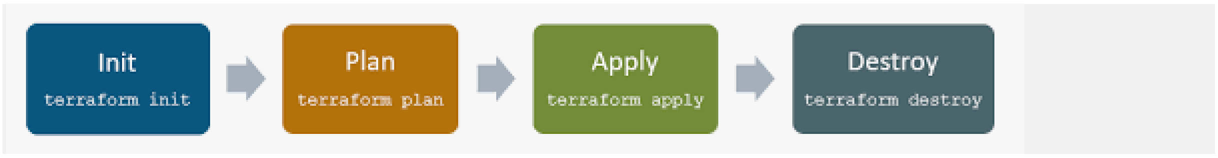
The terraform workspace select command is used to choose a different workspace to use for further operations.

The terraform workspace delete command is used to delete an existing workspace.

The terraform workspace show command is used to output the current workspace.

Note: Terraform Cloud and Terraform CLI both have features called “workspaces,” but they’re slightly different.

Terraform Workflow



Terraform Init

The terraform init command is used to initialize a working directory containing Terraform configuration files.
During init, the configuration is searched for module blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments. Terraform must initialize the provider before it can be used.
Initialization downloads and installs the provider’s plugin so that it can later be executed.
Initializes the backend configuration.
It will not create any sample files like example.tf

Init Terraform and don’t ask any input

$ terraform init input= false

Change backend configuration during the init

$ terraform init -backend-config=cfg/s3.dev.tf - reconfigure is used to tell terraform to not copy the existing state to the new remote state location.

Amar Gajula

Terraform plan

The terraform plan command is used to create an execution plan. It will not modify things in infrastructure.
Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files. This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.

$ terraform plan

Terraform Apply

The terraform apply command is used to apply the changes required to reach the desired state of the configuration.

Terraform apply will also write data to the terraform.tfstate file.

Once the application is completed, resources are immediately available.

$ terraform apply

Apply and auto approve

$ terraform apply -auto-approve

Apply and define new variables value

$ terraform apply -auto-approve -var tags-repository_url=${GIT_URL}
Apply only one module

$ terraform apply -target=module.s3

This -target option works with terraform plan too.

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 does not modify infrastructure but does modify the state file.

Terraform Destroy

The terraform destroy command is used to destroy the Terraform-managed infrastructure.
terraform destroy command is not the only command through which infrastructure can be destroyed.

$ terraform destroy

You can remove the resource block from the configuration and run terraform apply this way you can destroy the infrastructure.

A deletion plan can be created before:

$ terraform plan –destroy

Amar Gajula

•target options allow to destroy only one resource, for example a S3 bucket:

```
$ terraform destroy -target aws_s3_bucket-my_bucket
```

Terraform Validate

The terraform validate command validates the configuration files in a directory.
Validate runs checks that verify whether a configuration is syntactically valid and thus primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.
It is safe to run this command automatically, for example, as a post-save check in a text editor or as a test step for a reusable module in a CI system. It can run before the terraform plan.

Validation requires an initialized working directory with any referenced plugins and modules installed.

```
$ terraform validate
```

Provisioners

Provisioners can be used to model specific actions on the local machine or on a remote machine to prepare servers or other infrastructure objects for service.

Provisioners are inside the resource block.

Note: Provisioners should only be used as a last resort. For most common situations there are better alternatives.

file Provisioner

The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource.

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
          source = "conf/myapp.conf"
          destination = "/etc/myapp.conf"
  }
```

local-exec Provisioner

The local-exec provisioner requires no other configuration, but most other provisioners must connect to the remote system using SSH or WinRM.

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
          command = "echo the server's IP address is
${self.private_ip}"
  }
}
```

Amar Gajula

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.

```
resource "aws_instance" "web" {
  # ...

  provisioner "remote-exec" {
          inline = [
          "puppet apply",
          "consul join ${aws_instance.web.private_ip}",

          ]
  }
}
```

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 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.

Destroy-time Provisioners

if when = "destroy" is specified, the provisioner will run when the resource it is defined within is destroyed.Other Sub-commands Terraform Format

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style.

For use-case, where all configurations written by team members needs to have a proper style of code, terraform fmt can be used.

Terraform Taint

Taint means recreate but it is a forcible recreate.

When there are no changes in the main.tf or the resource, then the terraform will not show anything in the desired state.

But when you want a resource to be forcefully recreated, then you need to mark that resource as taint.

If a resource is marked as taint, then the terraform will recreate the resource on the next apply or the plan.

taint: Manually mark a resource for recreation
untaint: manually unmark a resource as tainted

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 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 application will implement this change.
For multiple sub-modules, the following syntax-based example can be used

```
terraform taint [options] <address>
```

Terraform Untaint

The terraform untaint command manually unmark a Terraform-managed resource as tainted, restoring it as the primary instance in state.

Terraform Import

Terraform can import existing infrastructure.

This allows you to take resources that you've created by some other means and bring them under Terraform management.

The current implementation of Terraform import can only import resources into the state. It does not generate a configuration.

Terraform manages the resources which are created through terraform.

If you have created a resource outside like in aws or docker without terraform intervention, then the terraform will not know anything about it as it doesn't have any data in its state file.

But you want to manage the resource using the terraform, then will go with the terraform import function.

Here you need to enter or inject the resource details into the terraform state file. That way terraform will consider it is a created resource and we will be able to manage it.

You can't directly edit the state file but you can give the inputs to the state file.

It is a two-step process.

* update your .tf files with the new resource block.

* inject the details of the existing resource.

We will create a new resource through terraform.

```
provider "aws" {
  region = "ap-south-1"
}

# Specify the EC2 details
resource "aws_instance" "example" {
  ami           = "ami-0c1a7f89451184c8b"

  instance_type = "t2.micro"
  count = 1
}
```

It will create an instance in aws and you need to create a new instance parallelly.

Later, go to the main.tf file and change the count = 2.

```
provider "aws" {
  region = "ap-south-1"
}

# Specify the EC2 details
resource "aws_instance" "example" {
  ami           = "ami-0c1a7f89451184c8b"
  instance_type = "t2.micro"
  count = 2
}
```

If you do the terraform plan now, it will show there's a new resource that needs to be created.

Instead of creating a resource, you can import the details of the instance which is created on AWS by you.

```
$ terraform import aws_instance.example[1] i-077b9c03601fb6ba9
```

Here aws_instance.example[1] is the name of our instance that we are injecting. The ID is an ami ID of the machine or instance that we created on AWS.

Terraform Show

The terraform show command is used to provide human-readable output from a state or plan file.

terraform show -json will show a JSON representation of the plan, configuration, and current state.

Terraform plan -destroy

The behavior of any terraform destroy command can be previewed at any time with an equivalent terraform plan -destroy command.

Variables

Terraform Variables:

* Variables can be defined by the terraform files and provided when executing a command.

* They give more flexibility to our configuration and let us deploy the same elements in different configurations.

* We can also provide a value at runtime by using an environment variable or a command-line parameter.

* Use interpolation to obtain the value of a variable using $(var.NAME)

* Each block declares a single variable.

** Type (optional): If set, this defines the type of the variable. Valid values are string, list, and map.

** default (optional): This sets a default value for the variable. If no default is provided, Terraform will raise an error if a value is not provided by the caller.

** Description (optional): A human-friendly description for the variable.

syntax:

```
variable [name] {

[option] = "{value]"

}
```

Types of variables:

-> String (default) - var.variablename - Any variable which is created is a string variable.
-> Numeric -If we are specifically creating or storing numbers, then we use numeric variables.
-> list/array - var.variablename[indexnumber] ex: var.image_name[1]
-> map/hash - var.variablename[keyname] ex: var.image_name["prod"]

String :

If there's a value that we wanted to use again and again, then you can create a variable name and put the value.
You need to use var.variablename - It will return the value whatever you have stored in it.

Ex:

```
Variable "myimg" {
default = "centos:latest"
}

variable "intport" {
default = "80"
}
```

Amar Gajula

```
variable "extport" {
default = "80"
}

# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}

# Download the latest Centos image
resource "docker_image" "myimg" {
  name = var.myimg
}

resource "docker_container" "mycontainer" {
  name  = "testng"
  image = docker_image.myimg.latest
  ports {
    internal = var.intport
    external = var.extport
  }
}
```

Save these blocks as main.tf and every time you refer to the variables the terraform will pick the variables from the above inputs.

You can edit/change the variables through the command line.

```
$terraform plan -var "myimg=nginx" -var "extport=8080"
$terraform apply -auto-approve -var "myimg=nginx" -var "export=8080"
$terraform destroy -var "myimg=centos:7" -var "export=8080"
```

You can create a vars.tfvars file and enter all the values you need and save it.

Later, when you need to update the details. You can simply refer to the file as shown below.

```
$terraform plan -var-file=vars.tfvars
$terraform apply -auto-approve -var-file=vars.tfvars
$terraform destroy -auto-approve -var-file=vars.tfvars
```

Array/List:

If we are storing one value, then it is called "scalar value or string value."

If we want to pass/store more values, then we will use array/list variables. Here you need to mention the type of the variable as a list as we are telling
terraform that we are no longer using string variables. So we need to specify which variable we are using here.

```
# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}

# Download the latest Centos image
resource "docker_image" "myimg" {
  name = var.image_name[0]
```

Amar Gajula

```
}

variable "image_name" {
    type    = list
    default = ["nginx:latest", "httpd:latest"]
}
```

Since we are using more than one variable, we need to tell terraform to select a value to execute. We use index number here.

You can see the above script we are using name =var.image_name[0]. Here the number '0' refers to the value [nginx:latest] as it is an image name.

The index number will start counting from the number '0.' If we want to execute 'httpd:latest,' then we will select '1'

Explained below:

```
# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}

# Download the latest Centos image
resource "docker_image" "myimg" {
  name = var.image_name[var.indexno]
}

variable "image_name" {
    type    = list
    default = ["nginx:latest", "httpd:latest"]
}



variable "indexno" {
   default = 0
}
```

Here we have added an extra variable 'indexno' and it is getting referred here name = var.image_name[var.indexno].

The indexno value is given '0.' Once you start executing, terraform will run the '0' or the nginx here as a default.

But if you want to change and select 'httpd:latest' here, then you can use the following command.

```
$terraform -var "indexno=1"
```

Map variable:

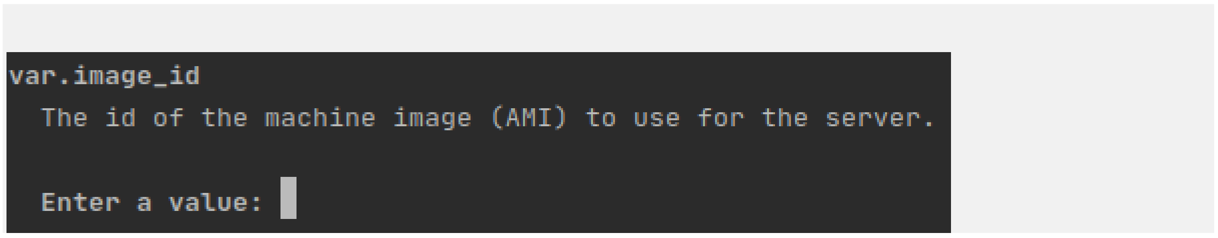
We use the map variable for storing a lot of numbers. Since the array or list variable can store multiple values as it is very difficult for us to count. Every time and choose the desired value.

In the map variable, every value will be given a nickname or a unique name. Here we use "key name" as indexno in the array/list variable.

```
# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}


# Download the latest Centos image
resource "docker_image" "myimg" {
  name = var.image_name[var.key]
}

variable "image_name" {
    type = map
    default = {
        "test"  = "nginx:latest"
        "prod" = "httpd:latest"
    }
}

variable "key" {
  default = "test"
}

$ terraform plan -var "key=prod"
```

When variables are declared in a module or configurations, they can be set in a number of ways.

Manually set a variable when we run Terraform plan If values are set, then it will ask at runtime.



```
var.image_id
  The id of the machine image (AMI) to use for the server.

  Enter a value:
```

CLI Variables

To specify individual variables on the command line, use the -var option when running the terraform plan and terraform apply commands:

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami- def456"]'
terraform apply -var='image_id_map={"us-east-1":"ami- abc123","us-east-2":"ami-def456"}'
```

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) and then specify that file on the command on the command line with -var-file:

Amar Gajula

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

Variable Definition Precedence

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

Environment variables

The terraform.tfvars file, if present.

The terraform.tfvars.json file, if present.

Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.

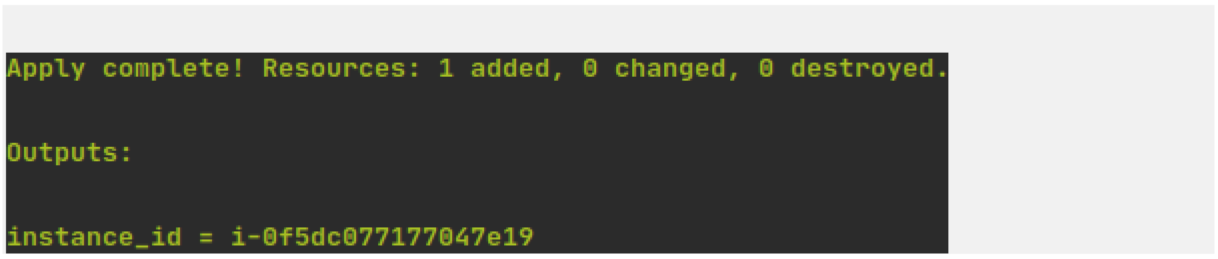
Any -var and -var-file options on the command line, in the order they are provided.

If the same variable is assigned multiple values, terraform uses the last value it finds.
Output Values

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

```
resource "aws_instance" "myec2" {
  ami     = var.image_id
  instance_type = "t2.micro"
}output "instance_id" {
          value = aws_instance.myec2.id
}
```

If we run a apply we can see the next message:



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

Outputs:

instance_id = i-0f5dc077177047e19
```

Terraform Output:

* Used to extract the value of a variable from the state file.

* Output variables can also be stored in their own output variables file named output.tf

* If you want to capture some property or attribute or resource then you use the output variable.

```
# Download the latest Centos image
resource "docker_image" "myimg" {
  name = "nginx:latest"
}

resource "docker_container" "mycontainer" {
  name  = "testng"
  image = docker_image.myimg.latest
```

Amar Gajula

```
  ports {
    internal = 80
    external = 80
  }
}

output "ip_address" {
  value       = docker_container.mycontainer.ip_address
  description = "The IP for the container."
}

# docker_container.mycontainer.command[0]
```

Assigning Values to Input Variables

Terraform Advanced :

Data Source:

This is a type of provider that we will be using for reading the data unlike the provider block and resource block in the terraform code.

This is for “Read Only” purposes. You can’t modify any data.

This resource should be available else the terraform plan will give you an error.

Ex: In the AWS, we can request the list of t2-micro machines list and it will provide us with the list. But if we make any errors while typing, then it’ll give you an error.

Syntax:

```
    DATA.DATASOURCETYPE.NAME.ATTRIBUTES
```

```
provider "SBI" {
  region = "KA-BNG-1"
}
```

```
data "SBI_ATM_availability_zones" "example" {
    state = "available"
}
```

For example: Let’s take an example of an ATMs of a certain bank. We’ll have a lot of ATMs in our city but only a few will be available.

To get a quick list of the active ATMs, we can find the available machines as mentioned in the code.

Once you run the code, the output data will be available inside the terraform.

You can fetch the data using **$terraform show**.

Amar Gajula

To get the data in the output variable. We can follow the below.

```
provider "aws" {
  region = "ap-south-1"
}

data "aws_availability_zones" "example" {
    state = "available"
}

output "azlist" {
    value = data.aws_availability_zones.example.names
}
```

If you give the names, then it picks up all the zones in ap-south-1. The output will give you the list of the available zones.

But if you want to pick a certain region then you can go as below. Here we use the array/list variable.

```
provider "aws" {
  region = "ap-south-1"
}

data "aws_availability_zones" "example" {
    state = "available"
}

output "azlist" {
    value = data.aws_availability_zones.example.names[0]
}
```

To get the list of all the aws machines that you have created under your account.

```
provider "aws" {
  region = "ap-south-1"
}

data "aws_availability_zones" "example" {
    state = "available"
}

output "azlist" {
    value = data.aws_availability_zones.example.names
}
```

This part will give you the list of the machines that are available within the selected zone.

But adding the below part to the terraform code will give you the output of all the machines/instances which you have created under your account.

We mostly use t2.micro and t2.small as these are the only types that are available for trial version/free version. You can choose any instance type.

```
data "aws_instances" "test" {
  filter {
    name = "instance-type"
    values = ["t2.micro","t2.small"]
  }
  instance_state_names = ["running", "stopped"]
}
```

We can use the above variable to list the machines that are running and terminated.

For example, you can add the below command to add the private ips of the machines.

```
output "machinelist" {
    value = data.aws_instances.test.private_ips
}
```

Dependencies

Explicitly specifying a dependency is only necessary when a resource relies on some other resource’s behaviour but doesn’t access any of that resource’s data in its arguments.

```
resource "aws_instance" "example" { ami      = "ami-a1b2c3d4" instance_type = "t2.micro"
depends_on = [aws_iam_role_policy.example]
}
```

Terraform State (Important) :

Whenever we run terraform plan and terraform apply, the terraform will create and store the data in the files called “terraform state.”

The terraform will store the actual state of the main.tf document or the directory in the state file.

Before you run the terraform plan for the first time, there is no state file available in the terraform.

Once you plan and apply the resources, the terraform will create the resources and store the entire data to the state file and it can be fetched by “terraform show”

Terraform will store the data in the same directory where you have saved the main.tf file.

When you run the plan for the next time, terraform will check the previous state and compare it with the new given state.

It will connect to the AWS and check the data. On every plan and apply command, the state file will get updated. This is the reason you need to create separate folders for different projects.

The plugins will also get stored in the same folder as the state file.

You can get to your folder and check there will be a file created - “**terraform.tfstate**”

Once you run the command “terraform plan”, it will read the tfstate file and it will get updated when running the command “terraform apply.”

Also, the terraform will have a backup file created “**terraform.tfstate.backup.**”

Once you apply the desired state of the file, the terraform will take a backup of the state file and make changes to the state file.

I wanted to get the same docker example folder as yesterday or a particular date after updating a lot of times, we will use the “Inmutable infra” method to get to the desired state.

If two people are working on the same project and work at different timings/regions, then it will give you an error while trying to update the state.

That’s why we use a distributed state or a remote state on some other machine.

The people need to upload the state file to the storage (git lab/aws s3 buckets/Azure Blobs) after the work is done.

So that once the second person pulls the code and checks, the latest state file will be available. We need to add a file “**backend.tf**”

How does a terraform know where to store it?

So we use the below resource block to store the state file on the cloud. Here I have used AWS as I have stored my credentials of AWS in terraforming.

You can use git, consul and the terraform enterprise.

You need to create a new S3 bucket and use the details of the bucket to backup your state.tf file. We have created a bucket “Reserved_S3_bucket_for_beckend” using terraform.

You need to create a file with any name but it needs to be matched with the below value as we have given the name “backend.”

We have created a file called backend.tf and we will do terraform init so that terraform will know we have a new resource block in the backend.tf.

```
 terraform {
  backend "s3" {
    bucket = "Reserved_S3_bucket_for_beckend"
    key = "default/terraform.tfstate"              # path & file which will hold the state #
    region = "ap-south-1"
  }
}
```

Once you run the terraform apply, then the state file will get created and moved to the S3 bucket rather than stored in the local folder.

<u>State Locking</u>

The state locking is an important practice.

What if two developers are working on the state file and pushes the new committed code to the backend. It will give you an error or a merge conflict.

Terraform introduced a feature called state locking. It is an automatic process.

Once the developer/DevOps engineer commits a code, then the AWS/Azure has a feature of locking the file till our commits updated.

Amar Gajula

Then, the lock will be removed automatically.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, terraform will not continue.

You can disable state locking in most commands with the -lock flag but it is not recommended.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

```
Syntax: terraform force-unlock [options] LOCK_ID [DIR]
```

Sensitive Data

Terraform state can contain sensitive data, e.g., database password, etc.

When using a remote state, the state is only ever held in memory when used by Terraform.

The S3 backend supports encryption at rest when the encrypt option is enabled. IAM policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.

Note: Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end of terraform apply.

However, output values are still recorded in the state and so will be visible to anyone who is able to access the state data.

```
output "db_password" {
  value = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive       = true
}
```

Backend Management

A backend in Terraform determines how state is loaded and how an operation such as apply is executed.

Terraform must initialize any configured backend before use.

Local:

By default, terraform uses the "local" backend. After running first terraform apply the terraform.tfstate file created in the same directory of main.tf terraform.tfstate file contains JSON data.

The local backend stores state on the local filesystem, locks the state using system APIs, and performs operations locally.

```
terraform {
  backend "local" {
          path = "relative/path/to/terraform.tfstate"
  }
}
```

Amar Gajula

Remote

When working with Terraform in a team, the use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

With a remote state, terraform writes the state data to a remote data store, which can then be shared between all members of a team.

```
terraform {
  backend "remote" {}
}
```

This is called partial configuration

When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others.

```
terraform init -backend-config=backend.hcl
```

Standard Backend Types AWS S3 bucket.

AWS S3 is typically the best bet as a remote backend for the following reason.

It's a managed service, so no need to manage infrastructure.

It supports encryption at rest.

It supports locking via DynamoDB.

It supports versioning, so you can roll back to an older version.

```
terraform {
  backend "s3" {
          bucket = "mybucket"
          key       = "path/to/my/key"
          region = "us-east-1"    dynamodb_table = "terraform-locks"
          encrypt            = true
  }
}
```

Terraform will automatically detect that you already have a state file locally and prompt you to copy it to the new S3 backend.

If you type in "yes," you should see:

```
Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.
```

After running this command, your Terraform state will be stored in the S3 bucket.

Note: GitHub is not supported as backend type

Terraform State commands

terraform state list: List resources within terraform state.

terraform state mv: Move items within terraform state. This will be used to resource renaming without destroy, apply command

terraform state pull: Manually download and output the state from the state file.

terraform state push: Manually upload a local state file to the remote state

terraform state rm: Remove items from the state. Items removed from the state are not physically destroyed. This item no longer managed by Terraform.

terraform state show: shows attributes of a single resource in the state.

Terraform Modules:

Modules refer to reusing a bunch of code in terraform again and again.

Usually, we take a bunch of terraform files and reuse them for a new resource is called a module in terraform.

For growing infra editing single configuration 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 for Terraform configurations that are managed as a group.

Modules are used to create reusable components, improve organization, and treat pieces of infrastructure as a black box.

Every terraform configuration has at least 1 module which is the “root” module.

The Terraform Registry includes a direction of ready-to-use modules for various common purposes, which can serve as large building blocks for your infrastructure.

Module Structure:

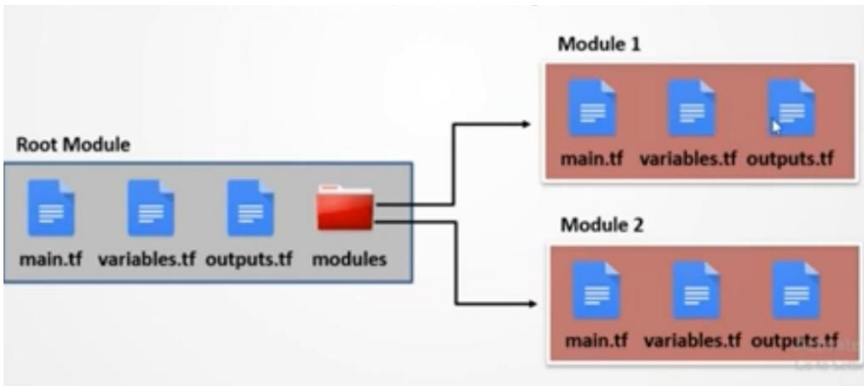
These are the recommended filenames for minimal module:

- main.tf
- variables.tf
- outputs.tf

-> main.tf should be the primary entry-point.

-> Nested modules should exist under the modules/subdirectory.

For example, we will have multiple folders/projects available in terraform. If you want to use a few commands from another folder/project you can access them through the “root module.”



Here, we create a new folder called "modules."

Inside that, you create separate folders for separate projects and inside that you should have the .tf files of other projects which you wanted to use the code.

The module folder is called a parent folder and the subfolders are called child modules.

The above picture shows that the root module files were created for a new project and to use the old code we have created a modules folder.

Here, we need to create the folders - Please refer to the above image.

We are creating the docker image folder separately and the docker container separately.

We need to create the main.tf file with the main attributes.

```
$ vi main.tf
# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}

resource "docker_image" "myimage" {
  name = var.image_name
}
```

Next, we need to create variable files.

```
$ vi variables.tf
variable "image_name" {

  description = "Image for container."
}
```

Later, we will create an output.tf file.

```
$ vi outputs.tf
output "image_out" {
  value       = docker_image.myimage.latest
}
```

Amar Gajula

We are using this method so that we can brush up on the previous classes rather than entering all the code in a single file. Also, we can reuse the modules.

Now, if you run terraform plan, the terraform will ask for the image input as you see we haven't given any input for the image that needs to be created.

So we will give the command " $ terraform plan -var "image_name=nginx."

You can choose any image like jenkins/jenkins, httpd, and centos.

Once you give the input, the terraform plan will pick up the details and shows the desired state.

We will create the main.tf file.

```
$ vi main.tf
# Specify the Docker host
provider "docker" {
  host = "unix:///var/run/docker.sock"
}

resource "docker_container" "mycontainer" {
  name  = var.container_name
  image = var.image
  ports {
    internal = var.int_port
    external = var.ext_port
  }
}
```

Variables file.

```
$ vi variables.tf

variable "image" {}
variable "container_name" {}
variable "int_port" {}
variable "ext_port" {}
```

Now, if you run terraform plan, the terraform will ask for the image input as you see we haven't given any input for the container name and port details that need to be created.

(Image will be picked from the image folder by giving the var file details.)

So we will give the commands:

```
$ terraform plan -out=tfplan -var 'container_name=wezva' -var 'image=nginx:latest' -var

'int_port=80' -var 'ext_port=80'

$ terraform apply –auto-approve tfplan


$ terraform destroy -var 'container_name=wezva' -var 'image=nginx:latest' -var 'int_port=80' -
var 'ext_port=80'
```

Amar Gajula

Now, we will create the root module files.

You need to create a main.tf file and give the commands to check the image and container folder for the details as given below.

```
$ vi main.tf
module "image" {
  source = "./modules/image"
  image_name = var.mod1_image
}

module "container" {
  source = "./modules/container"
  container_name = var.mod2_cname
  image = module.image.image_out
  int_port = var.mod2_intport
  ext_port = var.mod2_extport
}
```

Variables file.

```
$ vi variables.tf
variable "mod1_image" {}
variable "mod2_cname" {}
variable "mod2_intport" {}
variable "mod2_extport" {}
```

```
$ terraform plan -out=myplan -var "mod1_image=httpd" -var "mod2_cname=wezvatech" -var
"mod2_intport=80" -var "mod2_extport=80"
$ terraform apply -auto-approve myplan
$ terraform destroy -var "mod1_image=httpd" -var "mod2_cname=wezvatech" -var
"mod2_intport=80" -var "mod2_extport=80"
```

Terraform Loops:

When you want to create the same resource multiple times, you do not need to create the same resource code multiple times.

Instead of that, you can use the existing code multiple times.

Terraform offers several different looping constructs, each intended to use in a slightly different scenario:

-> Count parameter: loop over resources.
-> for_each expressions: loop over resources and inline blocks within a resource.
-> for expressions: loop over lists and maps.

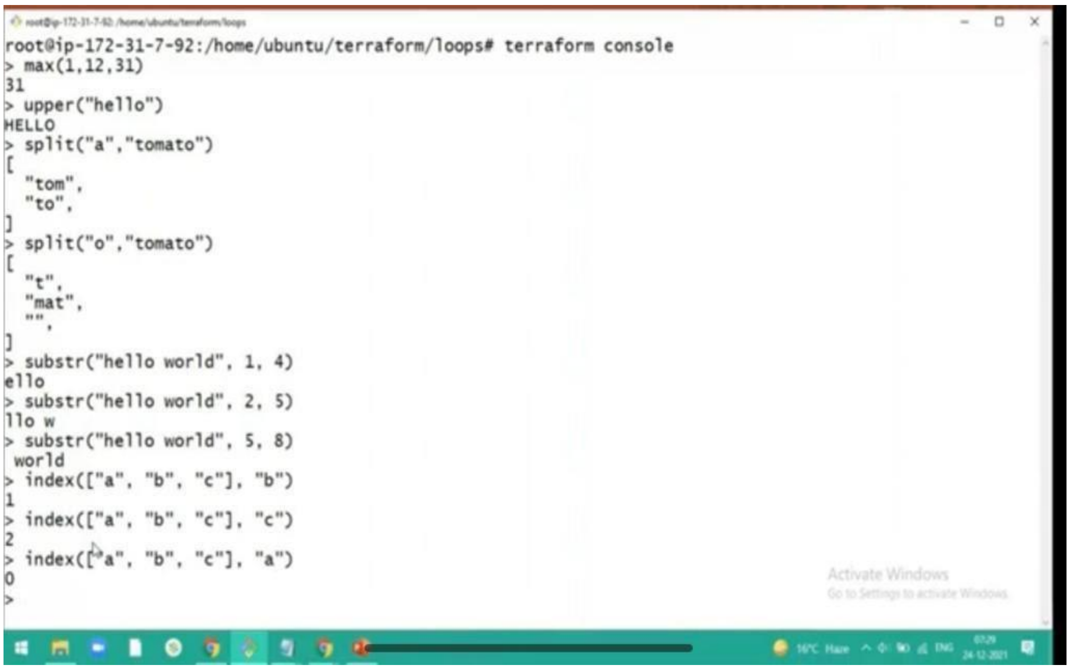
Terraform Built-in Function:
(It is available => Version 12 of Terraform)

- The Terraform language includes a number of built-in functions that you can call from within expression to transform and combine values.
- The Terraform language does not support user-defined functions, and so only the functions built into the language are available for use.
- You can experiment with the behaviour of Terraform’s built-in functions from the Terraform expression console.
- The general syntax for functions calls is a function name followed by a comma-separated argument in parameters.
-

Go to Terraform console:

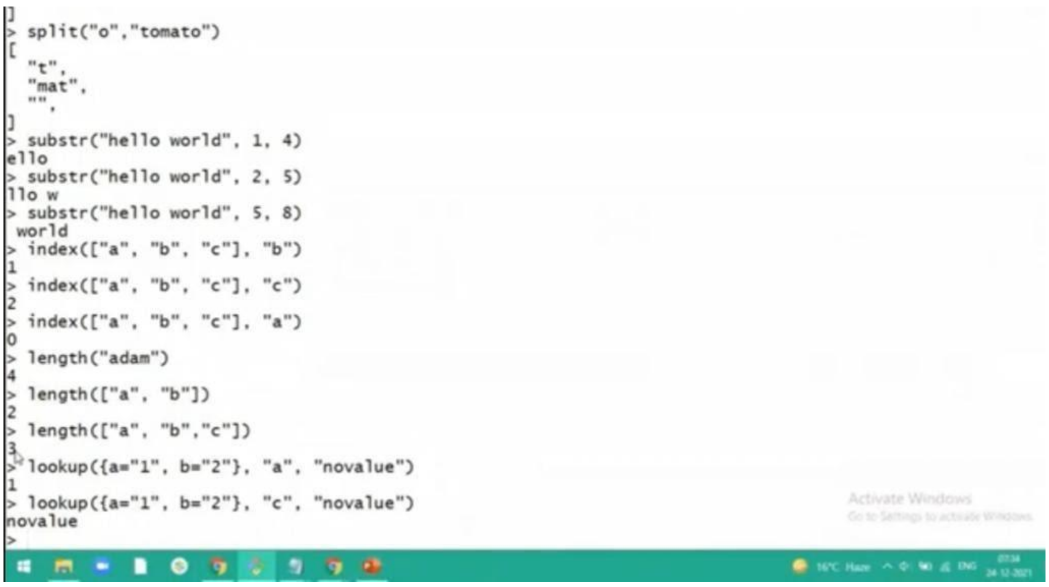
```
$ terraform console
```

```
max(1,31,12)
upper("hello")
split("a", "tomato")
substr("hello world", 1, 4)
index(["a", "b", "c"], "b")
length("adam")
length(["a", "b"])
lookup({a="1", b="2"}, "a", "novalue")
```



```
root@ip-172-31-7-92:/home/ubuntu/terraform/loops# terraform console
> max(1,12,31)
31
> upper("hello")
HELLO
> split("a","tomato")
[
  "tom",
  "to",
]
> split("o","tomato")
[
  "t",
  "mat",
  "",
]
> substr("hello world", 1, 4)
ello
> substr("hello world", 2, 5)
llo w
> substr("hello world", 5, 8)
 world
> index(["a", "b", "c"], "b")
1
> index(["a", "b", "c"], "c")
2
> index(["a", "b", "c"], "a")
0
>
```

Amar Gajula



```
]
> split("o","tomato")
[
  "t",
  "mat",
  "",
]
> substr("hello world", 1, 4)
ello
> substr("hello world", 2, 5)
llo w
> substr("hello world", 5, 8)
 world
> index(["a", "b", "c"], "b")
1
> index(["a", "b", "c"], "c")
2
> index(["a", "b", "c"], "a")
0
> length("adam")
4
> length(["a", "b"])
2
> length(["a", "b","c"])
3
> lookup({a="1", b="2"}, "a", "novalue")
1
> lookup({a="1", b="2"}, "c", "novalue")
novalue
>
```

```
provider "aws" {
  region = "ap-south-1"
}

variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["kiran","rakesh","veda","ram","chandra"]
}

resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

Explaining the last resource block:

Resource “aws_iam_user” “example”

We are referring to the service which we have chosen to create the users and “example” is just a name so that we can refer to.

“Count” is most important here. By count = length(var.user_names).

We are telling the terraform to count the user names given in the variable called “User_names” in the above block.

Since the count is an inbuilt feature of terraform for looping, it will also understand that the count of values or the names that need to execute.

name = var.user_names[count.index] - This command informs the terraform to divide the names mentioned in the user_names and refer the same to create users.

*count.index = This command helps the terraform to remember the count of values or names that are executed.

Amar Gajula

For example: If the terraform starts executing the variables, it will start with Kiran (as it is a list/array variable, it will assign the value ‘0’ to the name Kiran) and starts assigning the numbers (0,1,2,3) to the respective names.

Once you do the terraform plan, it will show how many names will get created. Then, you will understand the concept of a loop.

For each variable:

For each is also an inbuilt function of the terraform. Unlike the count function, the for each function uses the map variable to identify the values.

It tells the terraform to check the names and create the resources. It is a straight forward method.

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["ramesh","suresh","shubham","nagendra","shahin"]
}
resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Let me explain the second resource block.

resource "aws_iam_user" "example" - Here the resource name is aws_iam_user and the key value or the remembering value is “example.”

for_each = toset(var.user_names) - While using the function for each, we need to use the function called toset. A lot of times, the values will be not in order so the toset function will convert those values and send those to for each function.

Name = each.value - Every time we use the key word “each.value”, it will go through every value each value and store it and send the next value of execution.

On every execution, the loop consider the value as a map variable since the values were not in numerical and didn’t use the count.index function.

Terraform Conditionals:

For terraform whether to run or not to run a code is called condition.

Usually, after giving a command or a code, the terraform will create a resource if there is a resource that needs to be created. If the code or the commands matches the desired state, then it will not create a resource.

What if there’s a requirement to create a resource but you want the terraform to pick or decide to create the resource or not. We can give a condition to terraform to whether to pick up the certain resource or not.

For example, there’s a code in the main.tf which has the commands to create a user and an instance, but you want to execute the instance part of the resource only.

Amar Gajula

Here we use the condition function, we give conditions to each and every resource. Once the terraform plans, it will gather up the data which needs to be executed that met our condition.

If we have given the condition as true, then it will pick up.
If we have given the condition as false, then it will ignore the resource creation.

Here we can use the count variable to meet our desired state.

```
  provider “aws” {
 region “ap-south-1”
}
 resource “aws_iam_user” “example” {
count = 0
name = “Newname”
}
resource “aws_iam_user” “example2” {
name = “Newname2”
}
```

Here we have give added a function called count. Depends upon the count value, the terraform will create the resources. If the value has been given as ‘0’, then no resources will be created.

Ternary Function:

```
  provider “aws” {
   region = “ap-south-1”
}
varable “enable usercreation” {
 description = “enable or disable user creation”
}
resouce “aws_iam_user” “example” {
 count = “var.enable.usercreation ? 2 : 0  # expression ? <true_value> : <false_value>
 name = “amar”
}
```

```
$ terraform plan -var “enable_usernamecreation=1”
$ terraform plan -var “enable_usernamecreation=0”
```

Terraform Dependencies:

Implicit dependencies:

Terraformresource name, via interpolation syntax, allows us to reference any other resource it manages using the following syntax:

RESOURCE_TYPE.RESOURCE_NAME.ATTRIBUTE_NAME

By using this interpolation terraform can automatically infer when one resource depends on another.

When you add a reference from one resource to another, you create an implicit dependency.

Terraform parses these dependencies, builds a dependency graph from them, and users that to automatically figure out in what order it should create resources.

Explicit dependencies:

When a resource doesn't require any dependence on another resource, but we need to make a dependency between them, you rely on explicit dependency.

You need to use the keyword "depends_on" in the terraform script.

#Proving the instance provider

```
provider "aws" {
region = "ap-south-1"
}

resource = "aws_instance" "example" {
ami = "ami - ************"
instance type = "t2.micro"
}

resource = "aws_s3_bucket" "example" {
bucket = "Newcreation"
depends_on = "aws_instance.example"
}
```

Once you use the depends_on command, the S3 bucket is not at all dependent on instance. Here, we are making it as a dependent by using “depends_on.”

Terraform Cloud

Terraform Cloud (TFC) is a free to use, self-service SaaS platform that extends the capabilities of the open source Terraform CLI and adds collaboration and automation features.

The remote backend stores Terraform state and may be used to run operations in Terraform Cloud.

When using full remote operations, operations like terraform plan or terraform apply can be executed in Terraform Cloud’s run environment, with log output streaming to the local terminal.

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. Sentinel is a proactive service.

Note: Terraform Cloud always encrypts the state at rest and protects it with TLS in transit. Terraform Enterprise.

Terraform Enterprise provides several added advantages compared to Terraform Cloud.

Some of these include:

- Single Sign-On.
- Auditing
- Private Data Centre Networking
- Clustering
- Team & Governance features are not available for Terraform Cloud Free (Paid)

Amar Gajula

Miscellaneous

Terraform Cloud supports the following VCS providers: GitHub, Gitlab, Bitbucket, and Azure DevOps.

The existence of a provider plugin found locally in the working directory does not itself create a provider dependency. The plugin can exist without any reference to it in Terraform configuration.

The overuse of dynamic blocks can make configuration hard to read and maintain.
By default, provisioners that fail will also cause the terraform to apply itself to fail.

The on_failure=continue setting can be used to change this.

Terraform Enterprise requires a PostgresSQL for a clustered deployment.

Terraform can limit the number of concurrent operations as Terraform walks the graph using the -parallelism=n argument.

The default value for this setting is 10. This setting might be helpful if you're running into API rate limits.

terraform.tfstate and *.tfvars contains sensitive data, hence should be configured in .gitingore file.

Arbitrary Git repositories can be used by prefixing the address with the special git: prefix.
By default, terraform will clone and use the default branch (referenced by HEAD) in the selected repository.

You can override it with ref=version.

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

If Terraform state file is locked then,

Blocked action: - terraform destroy and terraform apply

Non-blocked action: - terraform fmt, terraform validate, terraform state list.

Amar Gajula