Terraform Documentation

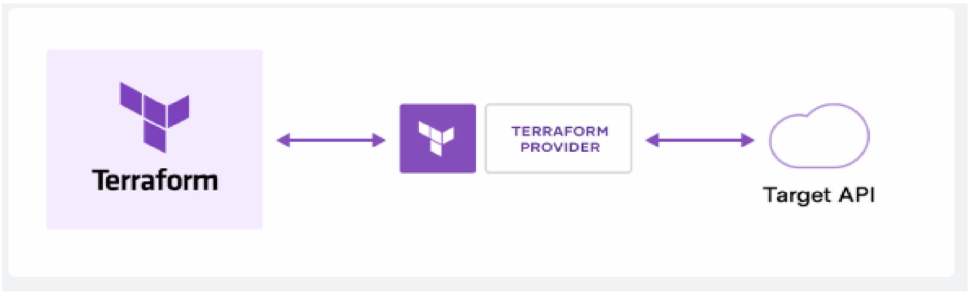
→ What is Terraform?

Terraform is very popular open-source tool used for infrastructure provisioning. It is developed by Hashi Corp (HCL). It will allow us to build, destroy, modify infrastructure in minutes. it supports lot of cloud and physical platforms.

- HashiCorp Terraform is an infrastructure as code tool (IAC)
- Terraform uses HCL language.
- All infastrucure resources can be define in .tf extension file.
- Terraform .tf file can be created using any text editor, notepad++, vi, vim or visual studio etc..
- HCL language uses Declarative and can be maintain in version control system.
- Every object terraform provisions is called as Resource.
- This is very widely used and huge demand in market.
- It’s very powerful
- Popular provider’s uses Terraform, Including major cloud providers like aws, azure, GCP and etc.…

→How does Terraform work?

- Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.



- HashiCorp and the Terraform community have already written **thousands of providers** to manage many different types of resources and services. You can find all publicly available providers on the Terraform Registry, including Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, and many more.

→ What is a Resource?

- Resource is a object that terraform manages.
- It can be any file in local machine, ec2, IAM, s3 and etc.

Key Characteristics of Resources in Terraform:

Type: The type of the resource, which specifies what kind of infrastructure it represents (e.g., aws_instance , google_storage_bucket, azurerm_virtual_network).

Name: A unique identifier within your Terraform configuration that allows you to reference the resource.
Attributes: Configuration settings that define the properties of the resource, such as instance types, storage sizes, or region.

Provider: A plugin that defines the cloud or infrastructure platform Terraform interacts with (e.g., AWS, Google Cloud, Azure).

→ What is Provider?

It is api call which is used to communicate with the resources, it can be aws, gcp, azure or local.

Providers are in three categories

1) Official --> provided by Hashicorp
2) Partner --> third party vendors
3) Community --> individual

→ Configuration Directory:

1. **Main.tf** -- main configuration file containing resource definition
2. **variables.tf** -- contains varibles declaration
3. **output.tf** -- contains outputs from resources
4. **provider.tf** -- contains the provider definition

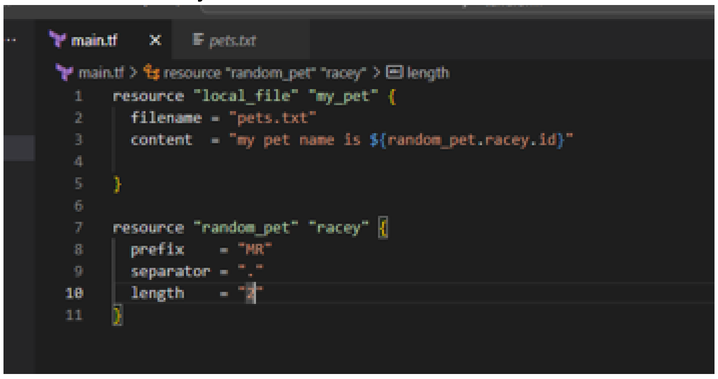
→ Multiple providers:

In Terraform, you can use **multiple providers** in a single configuration. This allows you to manage resources across different cloud platforms or services simultaneously. For example, you might want to create resources in both **AWS** and **Azure** within the same project.

Ex: Here we are using 2 providers **local** and **random**

```
resource "local_file" "my_pet" {
filename = "pets.txt"
content = "my pet name is racey"
          }

resource "random_pet" "my_pet" {
prefix = "MR"
separator = "."
length = "1"
          }
```



```
resource "local_file" "my_pet" {
  filename = "pets.txt"
  content  = "my pet name is ${random_pet.racey.id}"

}

resource "random_pet" "racey" {
  prefix    = "MR"
  separator = "."
  length    = "2"
}
```

→ Here are few commands:

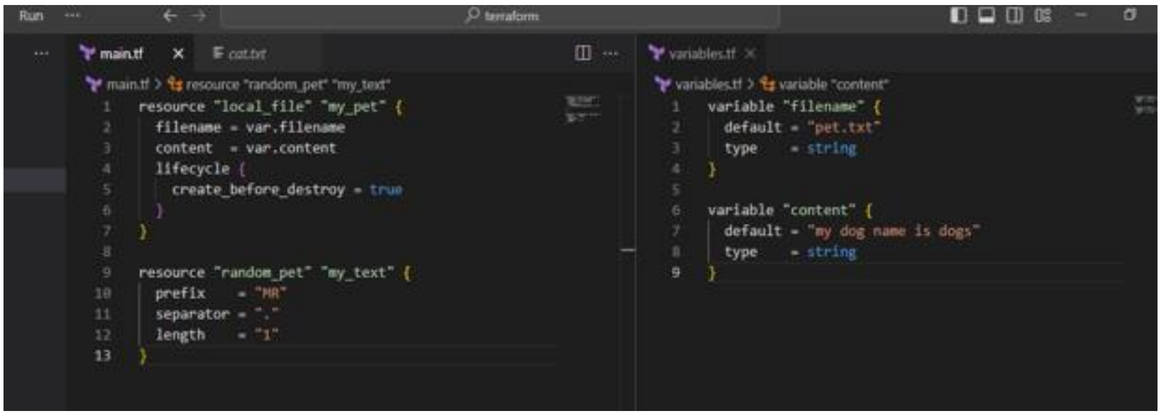
terraform init --> To initliaze the repository and download the dependencies
terraform plan --> dry run to check how it will be create the resource
(It will not actually create them)
terraform apply --> To execute and create the infra based on the configuration
terraform destroy --> To remove/delete the existing infra.
terraform show --> To show/get the details of the resources created

→ variables:

Instead of hardcoding all the variables in the same configuration file we can use variables so that it can be reuse again and again by only changing the variables file.

We need to create a file called variables.tf in the same directory

Ex:



main.tf

```
resource "local_file" "my_pet" {
  filename = var.filename
  content  = var.content
  lifecycle {
    create_before_destroy = true
  }
}

resource "random_pet" "my_text" {
  prefix    = "MR"
  separator = "."
  length    = "1"
}
```

variables.tf

```
variable "filename" {
  default = "pet.txt"
  type    = string
}

variable "content" {
  default = "my dog name is dogs"
  type    = string
}
```

→ Using of variables:

1) By using varibles.tf file

2) By using interactive mode (This will get activated if we don't pass default value in variable.tf file)

3) Command line flags

→terraform apply - var "filename=/root/pets.txt" -var "prefix=MR"

4) Environment variables

→ export TF_VAR_filename="/root.pets.txt"

→ export TF_VAR_prefix= "MR"

→ Set-Item -Path env:TF_VAR_filename -Value 'wild.txt' terraform apply

5) varibale definition file (Should be end with terraform.tfvars/terraform.tfvars.json)

→ for automatically loaded file name *.auto.tfvars/*.auto.tfvars.json

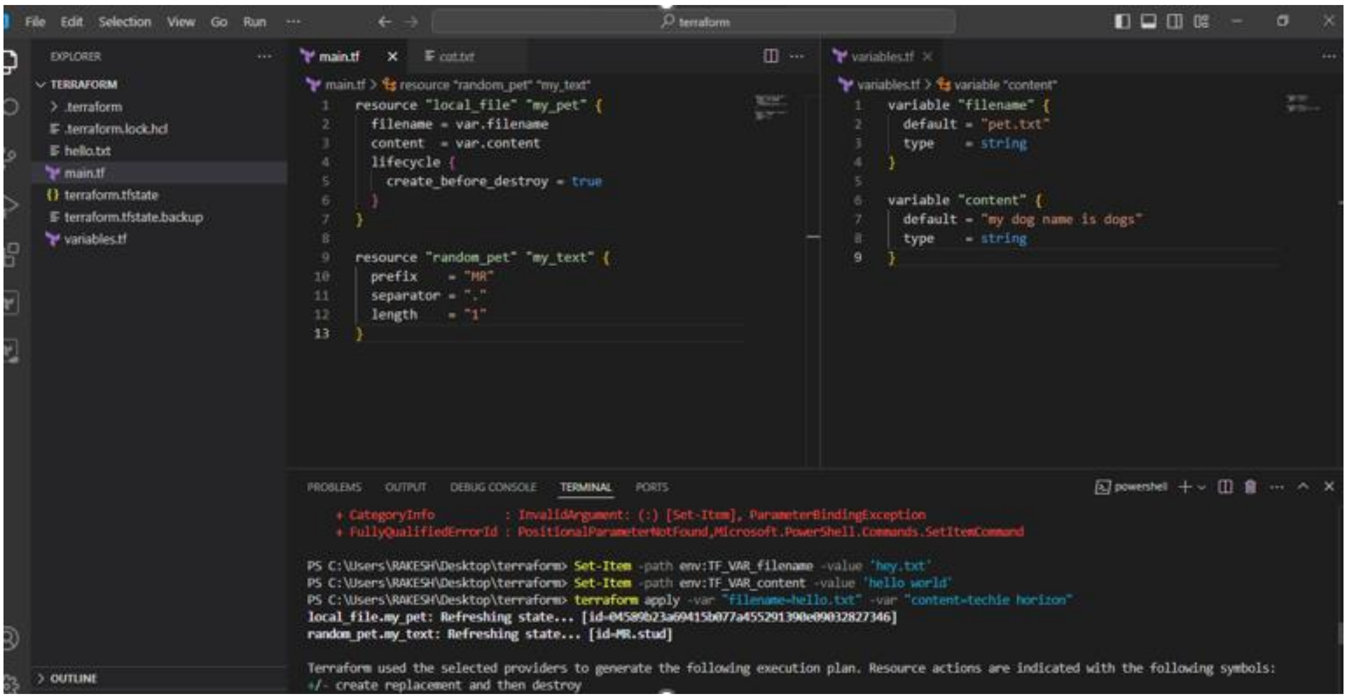
→ if we are saving the file with other name like varible.tfvars then we need to pass this in CLI

→ terraform apply -var-file varibale.tfvars

→ Precedence order:

If we use multiple ways to define varibles for the same file then terraform uses varible definition precedence.

Order	Option
1	Environment variables
2	Terraform.tfvars
3	*.auto.tfvars(alphabetical order)
4	-var or -var-file (Command line flags)



```
resource "local_file" "my_pet" {
  filename = var.filename
  content  = var.content
  lifecycle {
    create_before_destroy = true
  }
}

resource "random_pet" "my_text" {
  prefix    = "MR"
  separator = "."
  length    = "1"
}
```

```
variable "filename" {
  default = "pet.txt"
  type    = string
}

variable "content" {
  default = "my dog name is dogs"
  type    = string
}
```

```
    + CategoryInfo          : InvalidArgument: (:) [Set-Item], ParameterBindingException
    + FullyQualifiedErrorId : PositionalParameterNotFound,Microsoft.PowerShell.Commands.SetItemCommand

PS C:\Users\RAKESH\Desktop\terraform> Set-Item -path env:TF_VAR_filename -value 'hey.txt'
PS C:\Users\RAKESH\Desktop\terraform> Set-Item -path env:TF_VAR_content -value 'hello world'
PS C:\Users\RAKESH\Desktop\terraform> terraform apply -var "filename=hello.txt" -var "content=techie horizon"
local_file.my_pet: Refreshing state... [id=04589b23a69415b077a455291390e090328273460]
random_pet.my_text: Refreshing state... [id=MR.stud]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+/- create replacement and then destroy
```

→ Resource Attribute reference:

If i want to link two rerouces together by using resource attributes.
main.tf

```
resource "local_file" "pet" {
filename = "/root/pets.txt"
content = "My cat is MR.Cat"
}
resource "random_pet" "mypet" {
prefix = "MR"
separator = "."
length = "1"
}
```

When we execute terraform apply it will create random id with pet name,
now i want to add this pet name in my content file (using output of one resource as input for another resource).

main.tf

```
resource "local_file" "pet" {
filename = "/root/pets.txt"
content = "My cat is ${random_pet.mypet.id}"     (random_pet = resource type,mypet = resource name, id = attribute)
}
resource "random_pet" "mypet" {
prefix = "MR"
separator = "."
length = "1"
}
```

→ Resource Dependencies:

From the above main.tf we have local_file which is dependent on random_pet resource.
We are not mentioning that anywhere but still terraform will figure that out, we call them as implicit dependecy.
Still we can define the dependency explicit by ourself using "depends_on" module
Main.tf:

```
resource "local_file" "pet" {
filename = "/root/pets.txt"
content = "My cat is MR.CAT"
depends_on = [
random_pet.mypet
]
}
resource "random_pet" "mypet" {
prefix = "MR"
separator = "."
length = "1"
}
```

Output variables:

These are used to display the output of the resources.
Ex:

```
resource "random_pet" "mypet" {
prefix = "MR"
separator = "."
length = "1"
}

output my-pet {
value = random_pet.my-pet.id
description = optional name
}
```

when we use terraform apply we can see the id as output.
we can use terraform output command to see the output of the resource.

→ Terraform state:

- Terraform state file will have the complete record of the infra created by terraform.
- State file is considered as a blue print of all the resources terraform manages.
- terraform.tfstate will be the name of the file and this will created only after using terraform apply command.
- When we execute terraform apply then terraform will check for the state file config and main.tf configuration and make the changes.
- If both the files are in sync and we are again trying to execute terraform apply then terraform will not make the changes but show's
  "Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed."
- Each resource created by terraform will have the unique ID.
- State files also capture the Metadata of the configuration file.
- State file will help for better performance because of the cache of the data.
- state file benefits in collaborating with different team members.
- State files should be shared in the remote backend place so that team can access the state file.
- State files also store the sensitive data so not recommended to store in public repo's like github, gitlab.
- Terraform state is a json format file, never try to edit the state file manually.

→ Terraform Commands:

Terraform init --> to initiliaze terraform and download the dependecies for the resources.

Terraform plan --> To dry run or show how terraform will be executing the resource file.

Terraform apply --> To execute and make changes of the resource file.

Terraform output --> to print the output present in the resource file

Terraform validate --> to check the syntax is correct or not

Terraform fmt (format) --> to correct the format of the configuration file

Terraform show --> to display the current state of the infrastructure.

Terraform providers --> to see the list of providers used in configuration file

Terraform refresh --> used to sync terraform with real world infrastructure

Terraform graph --> to create visual representation of dependency

→ Terraform mutable vs immutable infrastructure:

- Terraform as a IAC tool uses immutable infrastructure stratgey.
- Immutable means deleting the older infra and creating a newer one with new update.
- Mutable means using the existing infra and updating the system with newer version.

→ Lifecycle rules:

- We can configure lifecycle rules to our resource file.
- Terraform will destroy the file before creating a new file by default.
- We can create a file before destroy by using the lifecycle rules.

→ create_before_destory:

```
resource "local_file" "pet" {
filename = "/root/pets.txt"
content = "My cat is MR.CAT"
lifecylce {
create_before_destory = true
}
}
```

→ prevent_destory:

If we don’t want to delete for any reason
This will be only helpful if we are using terraform apply command, if we use terraform destroy then it will destroy the resources.

```
resource "local_file" "pet" {
filename = "/root/pets.txt"
content = "My cat is MR.CAT"
lifecylce {
prevent_destroy = true
}
}
```

→ ignore_change:

If we want to ignore the changes which has been by third party tool or manually then we can use the ignore_change

```
resource "aws_ec2" "My-ec2" {
ami = "....."
instance_type = "t2.micro"
tags = {
Name = "ProjectA-webserver"
}
lifecycle {
ignore_changes = [
 tags,ami
]
}
}
```

→ Data sources:

Apart from terraform we have multiple other tools where the infra can be created.
Ex: ansible, salt, puppet, bash script, manual process.
Data sources are used to read the content of the infrastructure
for example: If we want terraform to read the content of the file which has been created by any other tool.
create a file called in dogs.txt in the same terrafrom working directory.

main.tf:

```
resource "local_file" "my-pet" {
filename = "pets.txt"
content = data.local_file.dog.content
}
data "local_file" "dog" {
filename = "dogs.txt"
}
```

Difference between resources and data ?

>Resources starts with keyword resource
resource is used to create, modify, delete the infra

>Data source start with keyword data.
data sources are used to read the infrastructure.

→ Meta-Arguments:

Meta arguments are used if we want to create multiple resources.
Meta arguments can be used within any resource block to change the behaviour of the resources.
Examples for meta-arguments:
1) Depends_on
2) lifecycle rules
3) Count
4) For_each

Example of count:
If we use count as 3 then it will create 3 files with pet[0], pet[1], pet[2]

```
resource "local_file" "my-pet" {
filename = "pets.txt"
content = "I love cats!"
count = 3
}
```

This is not the idea way to use because these are getting replaced.

```
resource "local_file" "my-pet" {
filename = var.filename[count.index]
content = "I love cats!"
count = 3
}
```

variables.tf

```
variable "filename" {
default = [
"pets.txt"
"cats.txt"
"dogs.txt"
]
}
```

Still we have problem in the above configuration, if in future the list of variables then we need to change the count value manually.

to avoid this, we can use the inbuilt length function in terraform.

```
resource "local_file" "my-pet" {
filename = var.filename[count.index]
content = "I love cats!"
count = length(var.filename)
}
```

variables.tf

```
variable "filename" {

default = [

"pets.txt"
"cats.txt"
"dogs.txt"
]
}
```

But when we want to update/destroy any one file then we will see un wanted results in count as count will store the output in list and works on index number. to overcome the issue we have for_each meta argument.

Example of for_each:

main.tf:

```
resource "local_file" "pet" {
filename = each.value
for_each = var.filename
}
```

variables.tf:

```
variable "filename" {
type=set(string)    --> list type will throw error for_each argument.
default = [
"pets.txt"
"cats.txt"
"dogs.txt"
]
}
```

or if you want to use list variable then we can change the main.tf with toset inbuilt function.

main.tf:

```
resource "local_file" "pet" {
filename = each.value
for_each = toset(var.filename)
}
```

variables.tf:

```
variable "filename" {
default = [
"pets.txt"
"cats.txt"
"dogs.txt"
]
}
```

Count will store the output as list and identified based on index number

foreach store the output as map and identified based on filename.

→ Version Constraints:

Changing in terraform providers version may get us into incompatibility issues.
By default terraform will always try to download the latest version of provider available from registry.
To make sure to use the specific version provider we can add the provider block in configuration.
Example:

```
terraform {
  required_providers {
    local = {
      source = "hashicorp/local"
      version = "2.3.0"
    }
  }
}
```

- version = "2.3.0" --> download the exact version
- version = "!=2.3.0" --> will not use the mentioned version
- version = "< 2.3.0" --> lesses than the mention version
- version = "> 2.3.0" --> greater than the given version
- version = "~> 2.3.0" --> specific version or higher version.

→ Terraform Remote state and state locking:

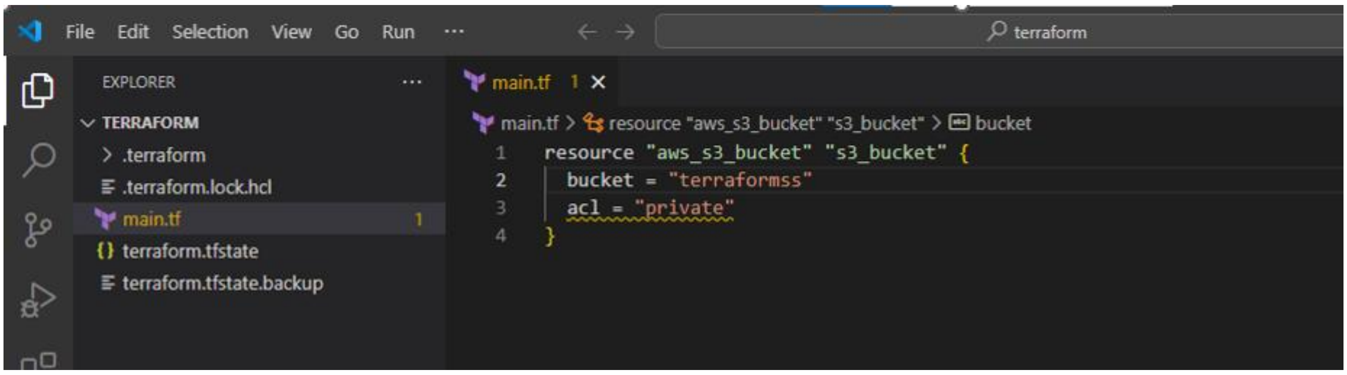
We can store terraform configuration files and state file in github or any other repository but it is not good practice.
We use s3, terraform storage, hashicorp console to store the state file.
State locking is used to lock the state file so that no two users can execute the state file at the same point of time.

Remote backend and state locking:

We can use s3 as remote backend and dynamo db for state locking.

Create s3 using terraform:

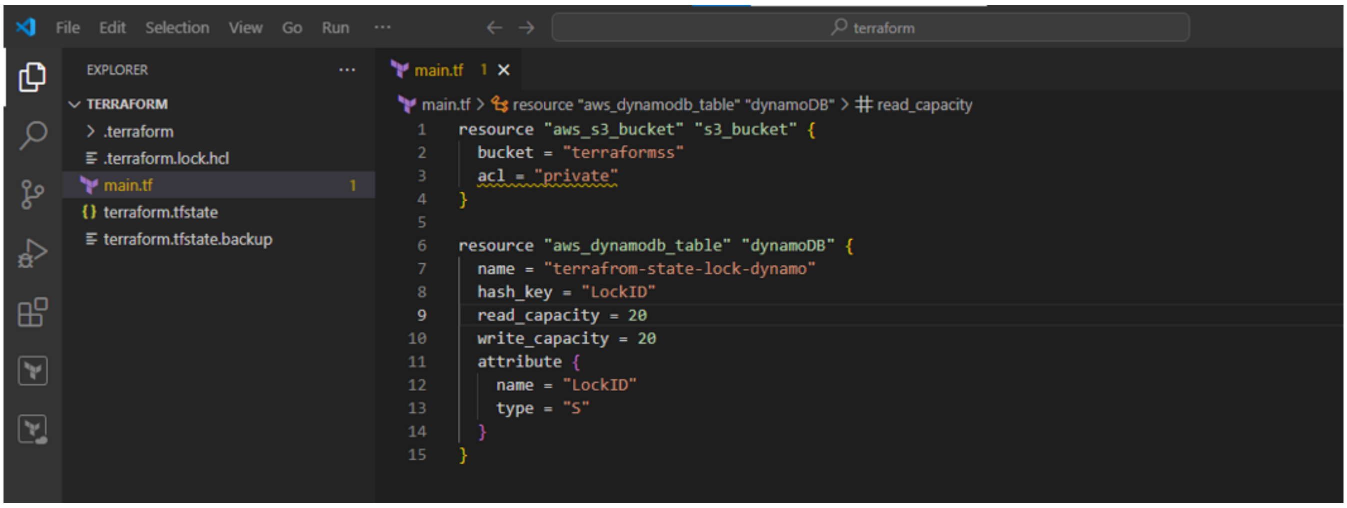
```
resource "aws_s3_bucket" "s3_bucket" {
   bucket = "s3backend"
   acl = "private"
}
```



```
resource "aws_s3_bucket" "s3_bucket" {
  bucket = "terraformss"
  acl = "private"
}
```

Create dynamo db using terraform:

```
resource "aws_dynamodb_table" "dynamodb-terraform-state-lock" {
  name = "terraform-state-lock-dynamo"
  hash_key = "LockID"
  read_capacity = 20
  write_capacity = 20
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

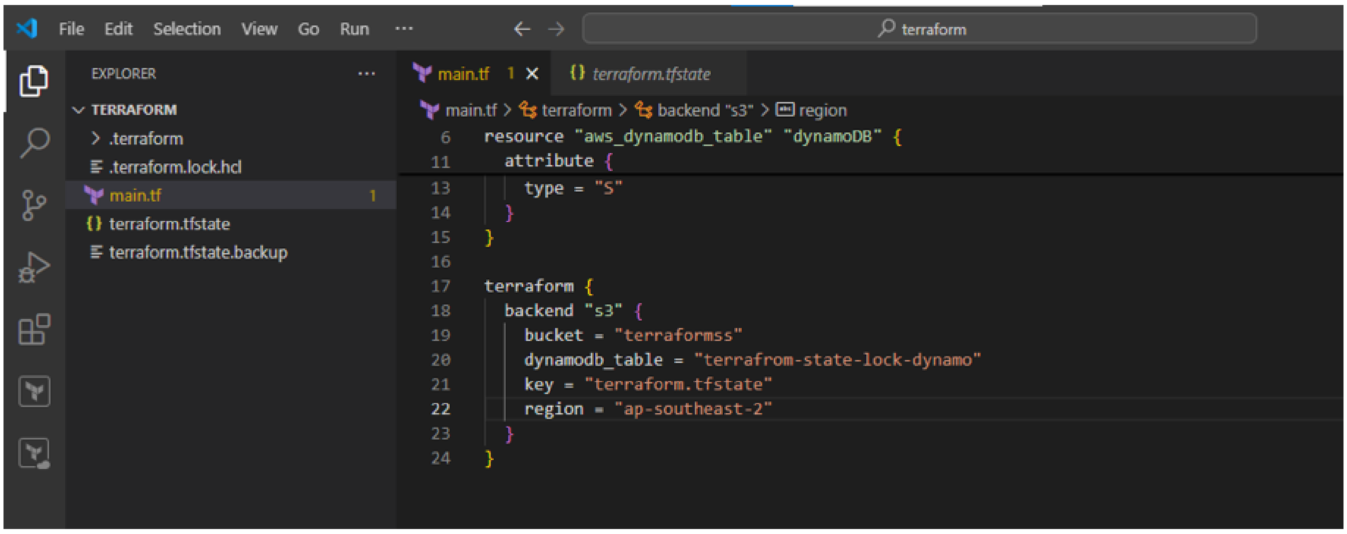


```
resource "aws_s3_bucket" "s3_bucket" {
  bucket = "terraformss"
  acl = "private"
}

resource "aws_dynamodb_table" "dynamoDB" {
  name = "terrafrom-state-lock-dynamo"
  hash_key = "LockID"
  read_capacity = 20
  write_capacity = 20
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

S3 as backend for terraform.tfstate file:

```
terraform {
  backend "s3" {
    bucket = "s3backend"
    dynamodb_table = "terraform-state-lock-dynamo"
    key    = "terraform.tfstate"
    region = "us-east-1"
  }
}
```



```
resource "aws_dynamodb_table" "dynamoDB" {
  attribute {
    type = "S"
  }
}

terraform {
  backend "s3" {
    bucket = "terraformss"
    dynamodb_table = "terrafrom-state-lock-dynamo"
    key = "terraform.tfstate"
    region = "ap-southeast-2"
  }
}
```

Terraform state commands:

<u>Terraform show</u> --> To show the resources of the state file

<u>Terraform state list</u> --> List resources in the state

<u>Terraform mv</u> --> Move an item in the state

<u>Terraform pull</u> --> Pull current state and output to stdout

<u>Terraform push</u> --> Update remote state from a local state file

<u>Terraform replace-provider</u> --> Replace provider in the state

Terraform rm --> Remove instances from the state

Terraform show --> Show a resource in the state

→ Terraform Provisioners:

Terraform provisioners allow us to execute command, scripts on remote machines or
local place were terraform is installed.
Provisioners will be written inside the resource blocks.
We have two types of provisioners
1) Remote provisioner
This is used to execute commands at the run time on remote machines.
2) Local provisioner
This is used to execute commands at the run time on local machine. (means where terraform is installed)

Example of Local provisioner:

```
resource "aws_instance" "test-server" {
 ami = "ami-005f9685cb30f234b"
 instance_type = "t2.micro"
 key_name = "linux-01"
 tags = {
  Name = "Test-server"
    }
   provisioner "local-exec" {
     command = "echo Instance ${aws_instance.test-server.public_ip} created! > instance_state.txt"
 }
}
```

We can use local provider after creating/destroying resource. After destroy we need to add when condition in provisioner.

```
resource "aws_instance" "test-server" {
 ami = "ami-005f9685cb30f234b"
 instance_type = "t2.micro"
 key_name = "linux-01"
 tags = {

  Name = "Test-server"
    }
   provisioner "local-exec" {
     when = destroy
     command = "echo Instance ${aws_instance.test-server.public_ip} created! > instance_state.txt"
 }
}
```

If the script failed then terraform apply command will also throw error.
if we want the resource to be completed if the script is failed then we can use on_failure module.

```
resource "aws_instance" "test-server" {
 ami = "ami-005f9685cb30f234b"
 instance_type = "t2.micro"
 key_name = "linux-01"
 tags = {
  Name = "Test-server"
    }
   provisioner "local-exec" {
     on_failure = fail
     command = "echo Instance ${aws_instance.test-server.public_ip} created! > instance_state.txt"
 }
}
```

Terraform provisioner behaviors:

1) Create at the time creating resource (Default)
2) create at the time of destroyin resource (when = destroy)
3) on_failure = fail --> to create the resource if the script gets failed.(But terraform will mark the reource as tainted)
4) On_fail = continue --> to create the resource and ignore the changes.

→ Terraform taint and untaint:

Thee would be cases when resource creation will get failed, if this happens then terraform will be marked as "Tainted".
we can see this when we execute terraform plan command and this will be replaced when we use terraform apply command.
If for any reason you have installed manually in ec2 and you want that to be replaced in next apply then we can manually mark that resource as taint in terraform so that it will be replaced next time when use apply.

```
  #terraform taint aws_instance.webserver
```

to undo the changes we can use untaint command.

```
  #terraform untaint aws_instance.webserver
```

this resource will not be created when using terraform apply.

→ Debugging:

terraform apply will provide us the logs/cause of the issue, but still if we want to dig deeper then we need to export a variable.

```
export TF_LOG=TRACE
Set-Item -Path env:TF_LOG -value "TRACE"
```

Terraform provides 5 levels of logs:

1) INFO
2) WARNING
3) ERROR
4) DEBUG
5) TRACE

To store the logs permanently then we can export a path

export TF_LOG_PATH=/tmp/terraform.log

Set-Item -Path env:TF_LOG_PATH -value "terraform.log"

to unset or disable logs

#unset TF_LOG_PATH

→ Terraform import:

Terraform import is used to import the existing infrastructure in terraform state file. Once import is done then we can be able to create/delete and manage the infrastructure.
In order to import any resource, we need to write the resource details in configuration file.

#terraform import aws_instance.<name> instance_id

→ Terraform Modules:

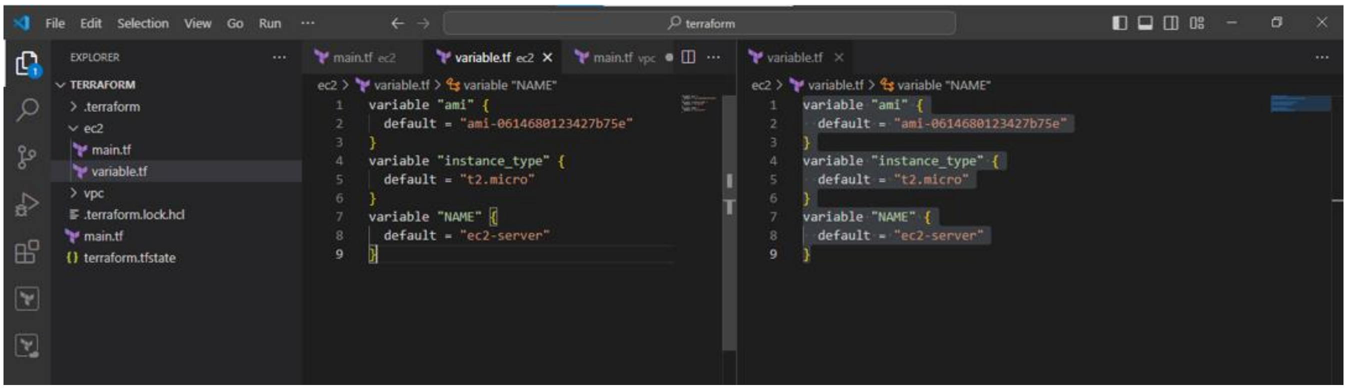
Modules is a collection of configuration files in a directory.
For example if we want to create two ec2 for different environments
then I can create a directory for aws-instance configuration.
Example:

```
resource "aws_instance" "webserver"
ami = "var.ami"
instance_type = "t2.micro"
}
```

now let me create a directory for dev under root directory.

example:

```
module "dev-webserver" {
   source = "../aws_instance"
   ami = "ami-02f3f602d23f1659d"
}
```



```
variable "ami" {
  default = "ami-0614680123427b75e"
}
variable "instance_type" {
  default = "t2.micro"
}
variable "NAME" {
  default = "ec2-server"
}
```

```
variable "ami" {
  default = "ami-0614680123427b75e"
}
variable "instance_type" {
  default = "t2.micro"
}
variable "NAME" {
  default = "ec2-server"
}
```

→ Terraform workspaces:

Workspace can be used if we have multiple project/environments using the same configuration.

Workspace commands:

#terraform workspace new projectA --> to create a workspace names projectA
#terraform workspace list --> to list the workspaces available
#terraform workspace select projectA --> to swtich to specific workspace

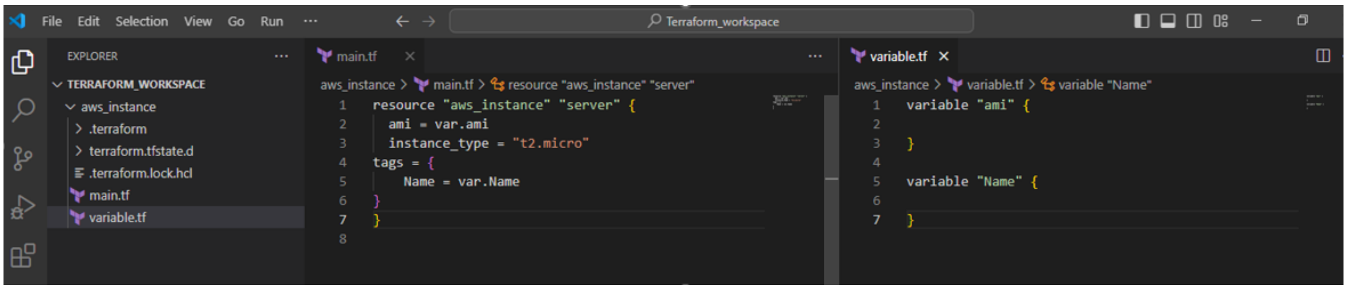
Example:

main.tf:

```
resource "aws_instance" "webserver"
ami = "var.ami"
instance_type = "t2.micro"
}
```

variable.tf:

```
variable "ami" {
type = map
default = {
  "ProjectA" = "ami-02f3f602d23f1659d"
  "ProjectB" = "ami-02f3f602d23f1659d"
}
}
```


File Edit Selection View Go Run
Terraform_workspace
EXPLORER
TERRAFORM_WORKSPACE
aws_instance
.terraform
terraform.tfstate.d
.terraform.lock.hcl
main.tf
variable.tf
aws_instance > main.tf > resource "aws_instance" "server"
aws_instance > variable.tf > variable "Name"

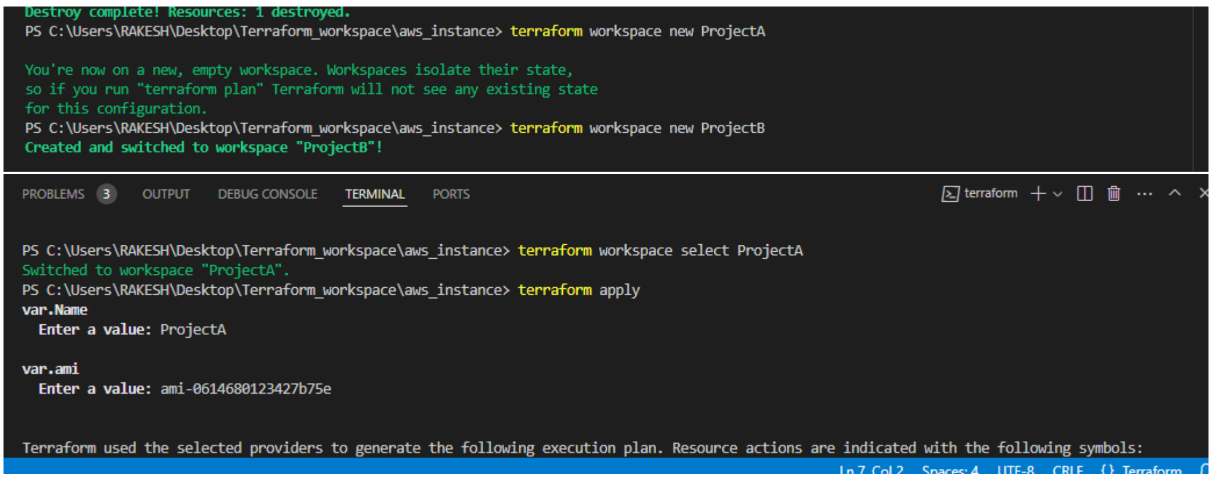
```
resource "aws_instance" "server" {
  ami = var.ami
  instance_type = "t2.micro"
tags = {
    Name = var.Name
}
}
```

```
variable "ami" {

}

variable "Name" {

}
```



```
Destroy complete! Resources: 1 destroyed.
PS C:\Users\RAKESH\Desktop\Terraform_workspace\aws_instance> terraform workspace new ProjectA

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.
PS C:\Users\RAKESH\Desktop\Terraform_workspace\aws_instance> terraform workspace new ProjectB
Created and switched to workspace "ProjectB"!
```

```
PS C:\Users\RAKESH\Desktop\Terraform_workspace\aws_instance> terraform workspace select ProjectA
Switched to workspace "ProjectA".
PS C:\Users\RAKESH\Desktop\Terraform_workspace\aws_instance> terraform apply
var.Name
  Enter a value: ProjectA

var.ami
  Enter a value: ami-0614680123427b75e


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

when using workspaces state files will be created in the main directory

under a folder "terraform.tfstate.d"