DVS Technologies

Dvs Technologies Aws Terraform

Compiled and Scrutinized by
Mr. Shaan Shaik
(Senior DevOps Lead)

Words To The Students

Though we have taken utmost efforts to present you this book error free, but still it may contain some errors or mistakes. Students are encouraged to bring, if there are any mistakes or errors in this document to our notice. So that it may be rectified in the next edition of this document.

"Suppressing your doubts is Hindering your growth".

We urge you to work hard and make use of the facilities we are providing to you, because there is no substitute for hard work. We wish you all the best for your future.

"The grass isn't greener on the other side; the grass is greener where you water it."

You and your suggestions are valuable to us; Help us to serve you better. In case of any suggestions, grievance, or complaints, please feel free to write us your suggestions, grievance and feedback on the following

Dvs.training@gmail.com

1. Infrastructure as a Code (IaC)

Terraform Introduction:

Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation

If you search the Internet for “infrastructure-as-code”, it’s pretty easy to come up with a list of the most popular tools:

Chef
Puppet
Ansible
SaltStack
CloudFormation
Terraform

All the above tools helps us to manage our infrastructure in the form of code.

But question is simple why ?

should we choose "terraform" than all these. If you observe all the above tools are open source and they
have their own communities and the contribution & one more thing is they all are enterprise tools.

Even we have "cloud formation" for automating the things with AWS than terraform. but question remains same why terraform ??

Configuration Management vs Orchestration :

The above mentioned tools except "CloudFormation & Terraform" all other tools are basically configuration
management tools.

Which means that they are used to manage and install the s/w or helps to maintain a state of the particular
machine.

But "Terraform" & "CloudFormation" are the Orchestration tools which means that they are designed to provison the machines & their infrastructure. Once the machine is builded you can use the configuration management tools for performing your task.

Mutable Infrastructure vs Immutable Infrastructure :

Mutable --> Configuration management tools

using configuration management tools, we can deploy the new software versions based up on the environment we are choosing. But if you observe each server will be having a separate version based up on the environment.

Immutable --> Orchestration tools

But if you are choosing the Orchestration tools you can simply maintain all the servers with a single version
of OS. It's simple create a simple OS image and start deploy the servers as per the requirement and all your
old machines will be replaced with the newly builded machines and all the machines will have same version of
the package installed !!

Procedural vs Declarative :

Procedural approach means if you want to achieve something you need to mention the things in an programmatic approach. "Chef & Ansible" works on the same.

But in Declarative approach you no need to worry about flow it will automatically gets the respective information based up the resource what we are choosing.

For example if you want to create 10 servers with app version v1 then the code for different tools will be like below.

using Ansible : (Procedural approach)

```
- ec2:
    count: 10
    ami: app-v1
    instance_type: t2.micro
```

Using terraform : (Declarative)

```
resource "aws_instance" "example" {
  count = 10
  ami = "ami-v1"
```

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

Till now its fine no much changes in both the configuration. But question is what will happen if the load is high and if you want to add 5 more servers.

Using Ansible you need to specify the code like below.

```
- ec2:
        count: 15
        ami: app-v1
        instance_type: t2.micro
```

Soon after executing this code, you will get a 15 more servers along with 10 machines so total will be 25 servers. But your desire state is to have only 5 machines without changing the code. Which means that you need to again re-write the entire code and find the previous machines and has to do all the other stuff.

Using Terraform you need to specify the code like below.

```
resource "aws_instance" "exampe" {
        count = 15
        ami = "ami-v1"
        instance_type = "t2.micro"
        }
```

Now what Terraform will do it, it won't create 15 more servers it will simply create 5 servers because it is well aware of the current state whatever it is having. Hence you no need to break you head to write new code.

Disadvantages:

Of course, there are downsides to declarative languages too. Without access to a full programming language, your expressive power is limited. For example, some types of infrastructure changes, such as a rolling, zero-downtime deployment, are hard to express in purely declarative terms. Similarly, without the ability to do “logic”
(e.g. if-statements, loops), creating generic, reusable code can be tricky (especially in CloudFormation).

Client/Server Architecture vs Client-Only Architecture :

Chef,puppet & salt stack are purely based on "Client/Server". Which indeed there are many hiccups when you are dealing with these tools, like

1. you need to install the client in all the machines in order to get the desired state as per the requirement
2. you should require a manageable server which should give the instructions to the client machines.
3. you will get all the issues with the network, client,management and etc.

Ansible, CloudFormation & Terraform are purely client-only Architecture, which in deed you no need to install any agents as part of your machines in order to do the management.

CloudFormation is also client/server, but AWS handles all the server details so transparently, that as an end user, you only have to think about the client code. The Ansible client works by connecting directly to your servers over SSH

Terraform uses cloud provider APIs to provision infrastructure, so there are no new authentication mechanisms beyond what you're using with the cloud provider already, and there is no need for direct access to your servers

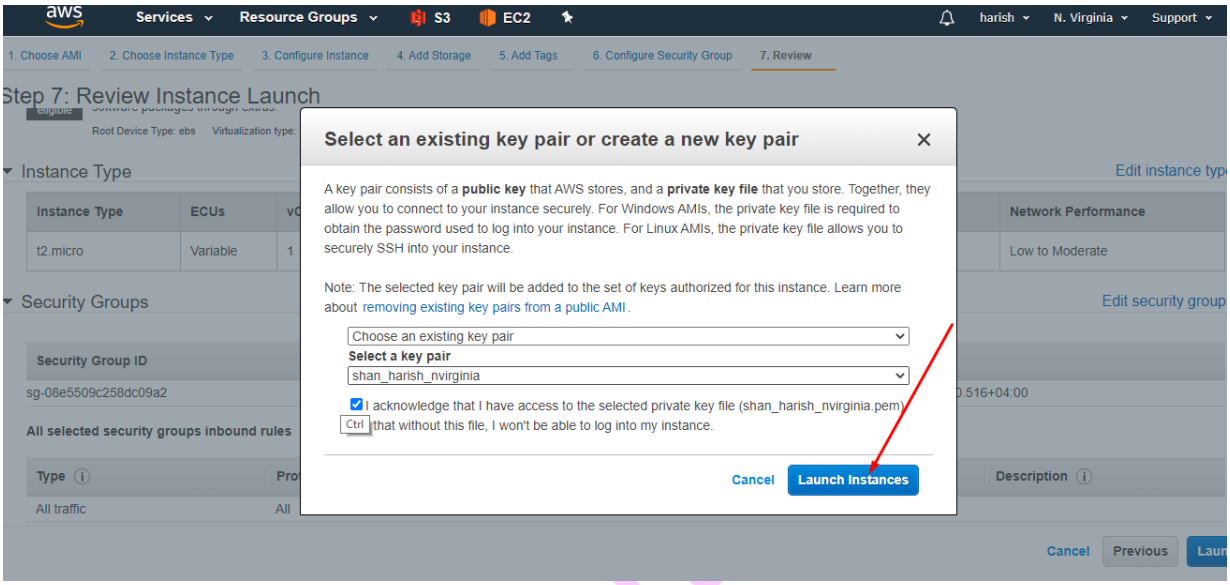
Final Conclusion:

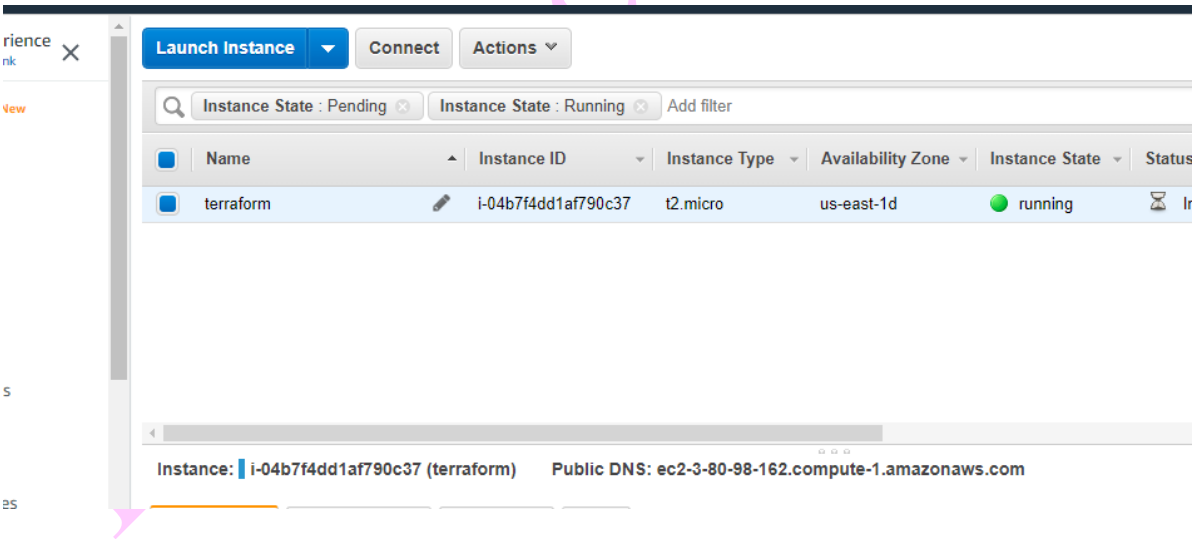
Of course, Terraform isn't perfect. It's younger and less mature than all the other tools on the list: whereas Puppet came out in 2005, Chef in 2009, SaltStack and CloudFormation in 2011, and Ansible in 2012,
Terraform came out just 4 years ago, in 2014.

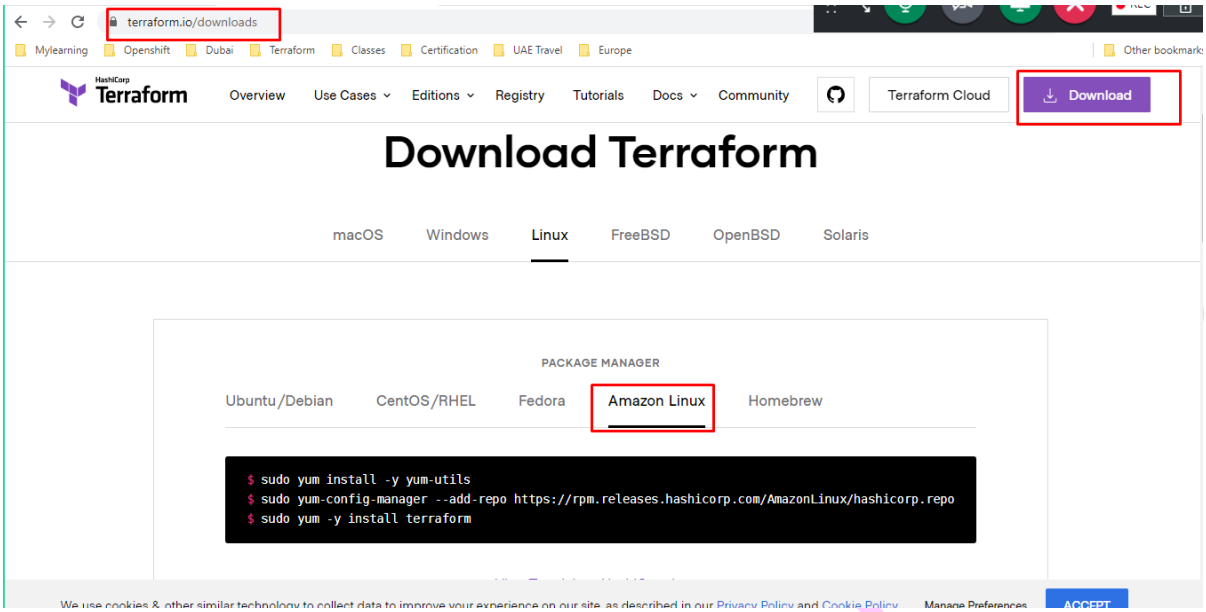
Bugs are relatively common (e.g. there are over 800 open issues with the label "bug"), although the vast majority are harmless eventual consistency issues that go away when you rerun Terraform

2. Installation and Configuration

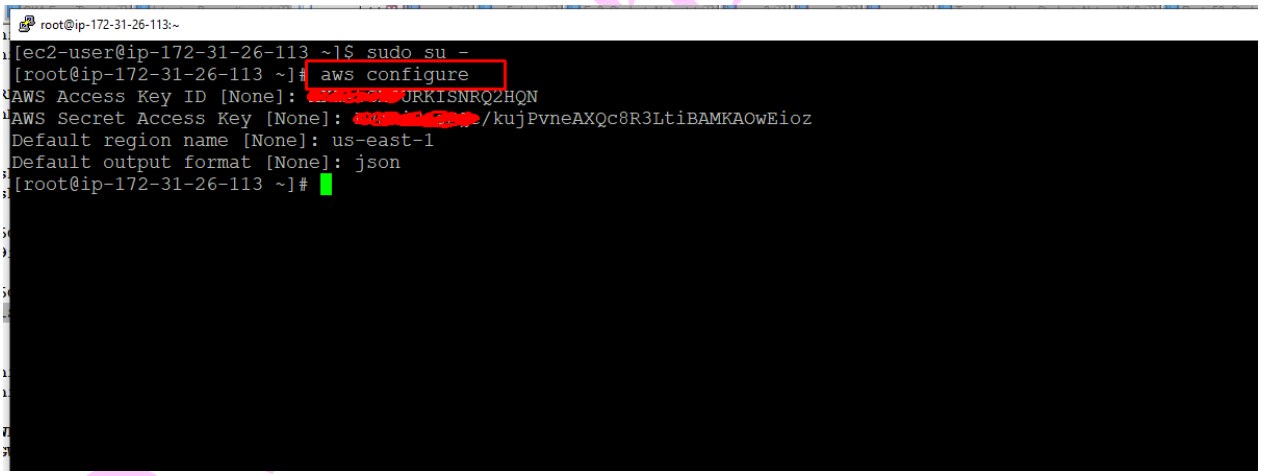
Create one Ec2







Configure your Aws Cli as specified below:



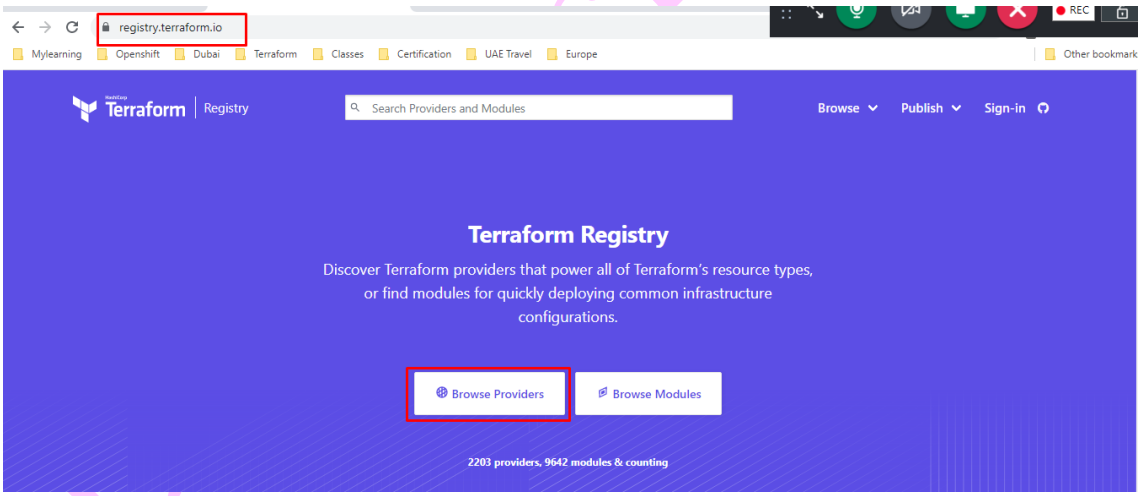
Let's Install the terraform:

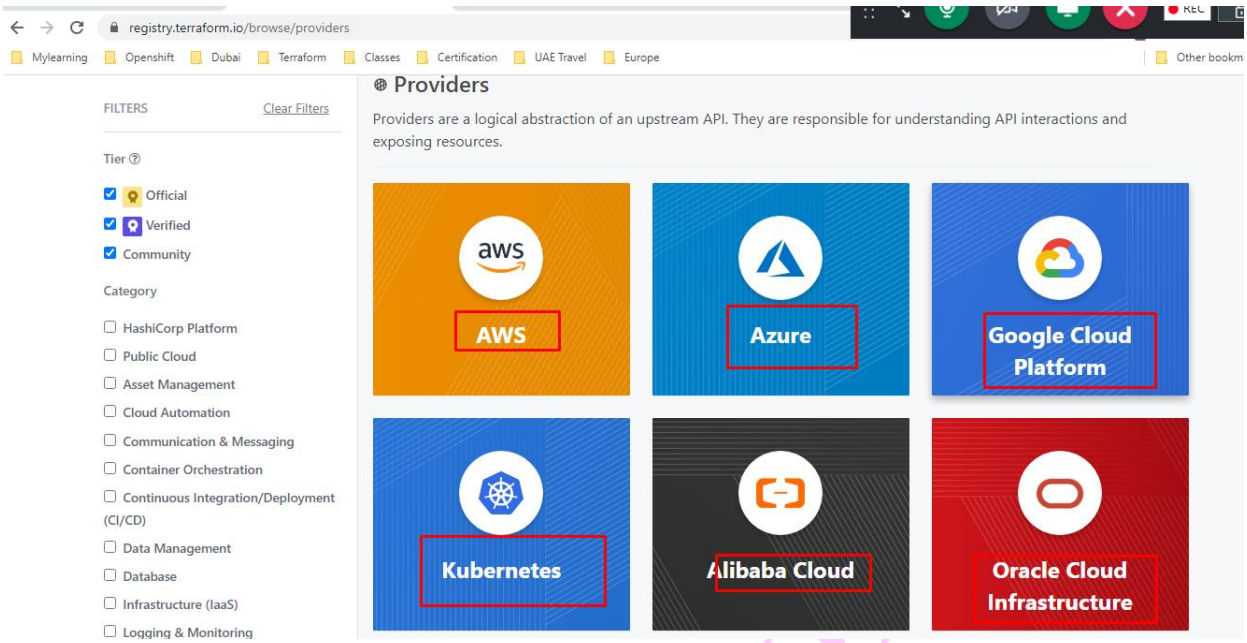
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
sudo yum -y install terraform

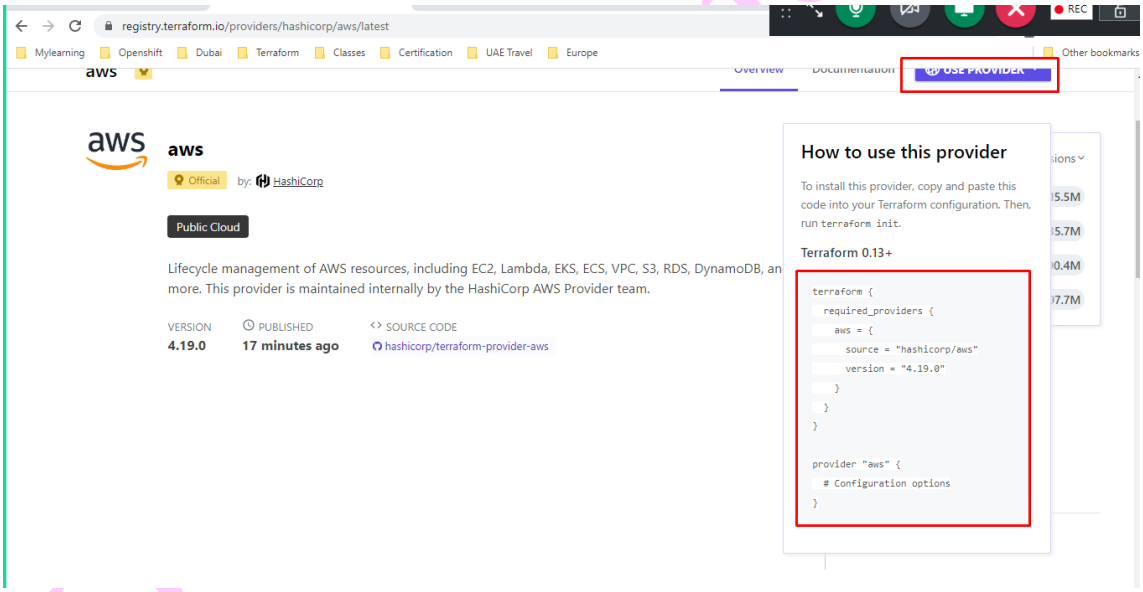
```
[root@ip-172-31-94-172 ~]# hostnamectl set-hostname terraform-workstation
[root@ip-172-31-94-172 ~]# bash
[root@terraform-workstation ~]# sudo yum install -y yum-utils
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
Package yum-utils-1.1.31-46.amzn2.0.1.noarch already installed and latest version
Nothing to do
[root@terraform-workstation ~]# sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/A
mazonLinux/hashicorp.repo
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
adding repo from: https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
grabbing file https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo to /etc/yum.repos.d/hashico
rp.repo
repo saved to /etc/yum.repos.d/hashicorp.repo
[root@terraform-workstation ~]# sudo yum -y install terraform
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
hashicorp                                                | 1.4 kB  00:00:00
hashicorp/x86_64/primary                                 |  91 kB  00:00:00
hashicorp                                                             650/650
```

```
Complete!
[root@terraform-workstation ~]# terraform version
Terraform v1.2.3
on linux_amd64
[root@terraform-workstation ~]#
```

3 Working with terraform







Providers:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
```

```
  }
}

provider "aws" {
  region = "us-east-1"
/* Optional not recommended to use
  access_key = "my-access-key"
  secret_key = "my-secret-key"
*/
}
```

Note: version you can use > or >= for supporting multiple versions
Ex: version = ">=4.17.0"

```
[root@terraform-workstation mycode]# vi providers.tf
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
/* Optional not recommended to use
  access_key = "my-access-key"
  secret_key = "my-secret-key"
*/
}
[root@terraform-workstation mycode]#
```

Diff stages of terraform code execution:

init:

```
-rw-r--r-- 1 root root 259 Jun 17 01:37 providers.tf
[root@terraform-workstation mycode]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

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.
[root@terraform-workstation mycode]# cat providers.tf
```

```
-rw-r--r-- 1 root root 1181 Jun 17 01:39 .terraform.lock.hcl
[root@terraform-workstation mycode]# ls -l .terraform/providers/registry.terraform.io/hashicorp/aws/4.19.0/linux_amd64/terraform-provider-aws_v4.19.0_x5
-rwxr-xr-x 1 root root 269729792 Jun 17 01:39 .terraform/providers/registry.terraform.io/hashicorp/aws/4.19.0/linux_amd64/terraform-provider-aws_v4.19.0_x5
[root@terraform-workstation mycode]#
```

Resource:

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
/* Optional not recommended to use
  access_key = "my-access-key"
  secret_key = "my-secret-key"
*/
}

resource "aws_key_pair" "anyrefname" {
  key_name   = "dvsbatch2-key"
  public_key = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2WjP76sGzM2kM7n
juP1yl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzRdoFpnYFSoXxLqd2MyV6rfReJZqHjdwX62VUcVZgAVFM
Alm4L55fXVkG7yfDSTw0tFG3WgieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cNn4CSDlIyS66
JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ6a4wdTDZIVAZT1yN3KKciLT imported-openss
h-key"
}
```

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}
```

```
provider "aws" {
  region = "us-east-1"
/* Optional not recommended to use
  access_key = "my-access-key"
  secret_key = "my-secret-key"
*/
}

resource "aws_key_pair" "mykey" {
  key_name   = "dvsbatch2-key"
  public_key = "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2Wj
P76sGzM2kM7njuP1yl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzR
doFpnYFSoXxLqd2MyV6rfReJZqHjdwX62VUcVZgAVFMAlm4L55fXVkG7yfDSTw0tFG3W
gieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cNn4
CSDlIyS66JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ
6a4wdTDZIVAZT1yN3KKciLT imported-openssh-key"
}
```

Validate:

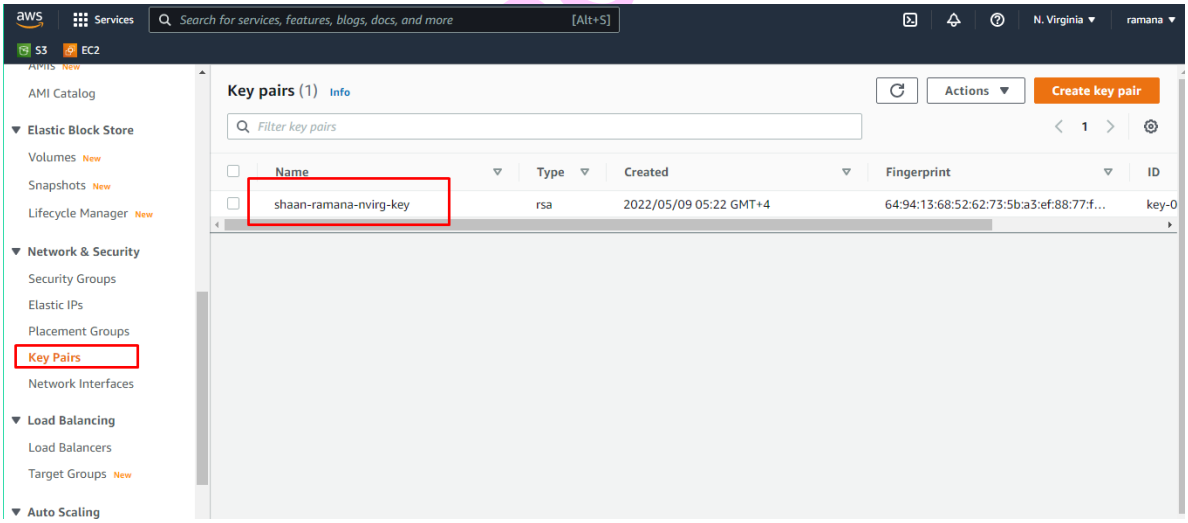
```
[root@terraform-workstation mycode]#
[root@terraform-workstation mycode]# terraform validate
Success! The configuration is valid.

[root@terraform-workstation mycode]#
```

Plan:

```
[root@terraform-workstation mycode]# 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:

  # aws_key_pair.mykey will be created
  + resource "aws_key_pair" "mykey" {
      + arn             = (known after apply)
      + fingerprint     = (known after apply)
      + id              = (known after apply)
      + key_name        = "dvsbatch2-key"
      + key_name_prefix = (known after apply)
      + key_pair_id     = (known after apply)
      + public_key      = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2WjP
76sGzM2kM7njuP1yl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzRdoFpnYFSoXxLqd2MyV6rfReJZqHjdwX6
2VUcVZgAVFMAlm4L55fXVkG7yfDSTw0tFG3WgieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cN
n4CSDlIyS66JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ6a4wdTDZIVAZT1yN3KKciLT impo
rted-openssh-key"
      + tags_all        = (known after apply)
    }

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.
[root@terraform-workstation mycode]#
```

Apply:

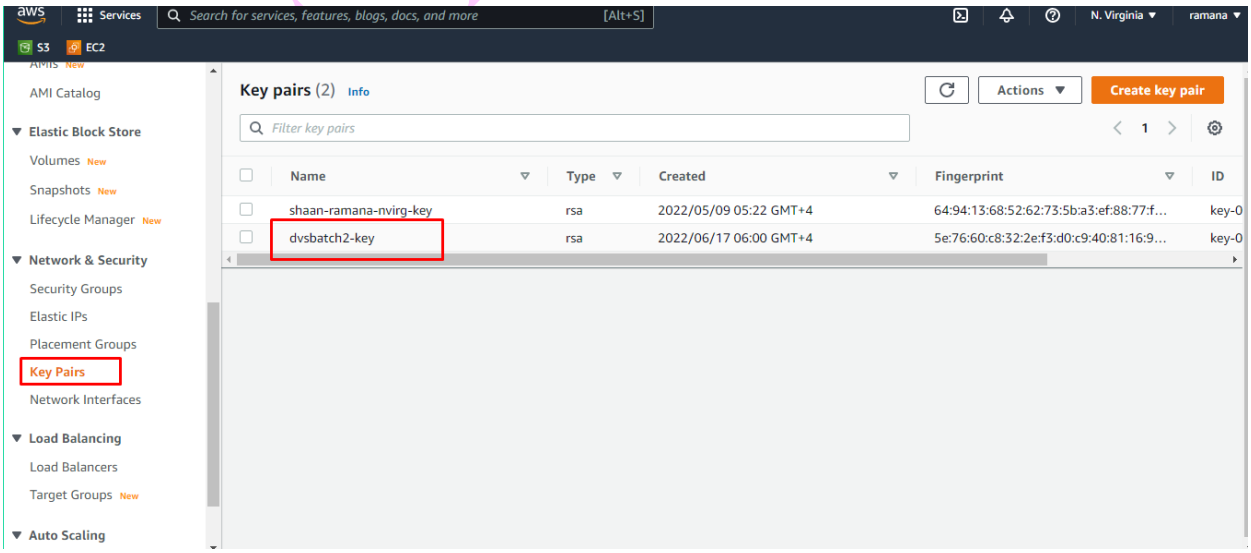
Before:



```
[root@terraform-workstation mycode]# 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:

  # aws_key_pair.mykey will be created
  + resource "aws_key_pair" "mykey" {
      + arn             = (known after apply)
      + fingerprint     = (known after apply)
      + id              = (known after apply)
      + key_name        = "dvsbatch2-key"
      + key_name_prefix = (known after apply)
      + key_pair_id     = (known after apply)
      + public_key      = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2WjP
76sGzM2kM7njuPIyl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzRdoFpnYFSoXxLqd2MyV6rfReJZqHjdwX6
2VUcVZgAVFMAlm4L55fXVkG7yfDSTw0tFG3WgieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cN
n4CSDlIyS66JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ6a4wdTDZIVAZT1yN3KKciLT impo
rted-openssh-key"
      + tags_all        = (known after apply)
    }

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
```

```
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

aws_key_pair.mykey: Creating...
aws_key_pair.mykey: Creation complete after 0s [id=dvsbatch2-key]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@terraform-workstation mycode]#
```

Post apply:

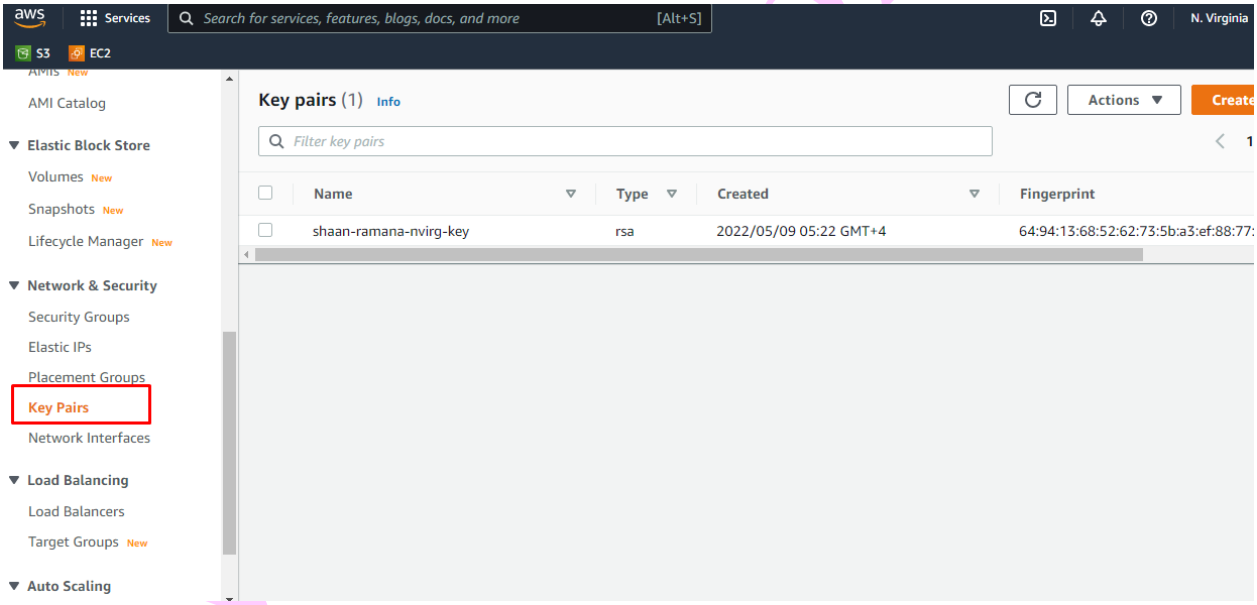


```
[root@terraform-workstation mycode]# ls -al
total 16
drwxr-xr-x 3 root root  128 Jun 17 02:11 .
dr-xr-x--- 6 root root  165 Jun 17 01:56 ..
-rw-r--r-- 1 root root  748 Jun 17 01:56 providers.tf
drwxr-xr-x 3 root root   23 Jun 17 01:39 .terraform
-rw-r--r-- 1 root root 1181 Jun 17 01:39 .terraform.lock.hcl
-rw-r--r-- 1 root root 1335 Jun 17 02:11 terraform.tfstate
-rw-r--r-- 1 root root 1337 Jun 17 02:11 terraform.tfstate.backup
[root@terraform-workstation mycode]# cat terraform.tfstate
{
  "version": 4,
  "terraform_version": "1.2.3",
  "serial": 2,
  "lineage": "40ce356f-6690-ccd5-04c7-1769ad5f4b5b",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_key_pair",
      "name": "mykey",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "arn": "arn:aws:ec2:us-east-1:156118558224:key-pair/dvsbatch2-key",
            "fingerprint": "5e:76:60:c8:32:2e:f3:d0:c9:40:81:16:93:f3:72:03",
            "id": "dvsbatch2-key",
            "key_name": "dvsbatch2-key",
```

```
[root@terraform-workstation mycode]#
[root@terraform-workstation mycode]# terraform show
# aws_key_pair.mykey:
resource "aws_key_pair" "mykey" {
    arn         = "arn:aws:ec2:us-east-1:156118558224:key-pair/dvsbatch2-key"
    fingerprint = "5e:76:60:c8:32:2e:f3:d0:c9:40:81:16:93:f3:72:03"
    id          = "dvsbatch2-key"
    key_name    = "dvsbatch2-key"
    key_pair_id = "key-0028ee04827d5a1f9"
    public_key  = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2WjP76s
M7njuP1yl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzRdoFpnYFSoXxLqd2MyV6rfReJZqHjdwX62VU
VFMAlm4L55fXVkG7yfDSTw0tFG3WgieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cNn4C
S66JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ6a4wdTDZIVAZT1yN3KKciLT importe
nssh-key"
    tags        = {}
    tags_all    = {}
}
[root@terraform-workstation mycode]#
```

Destroy:

```
[root@terraform-workstation mycode]# terraform destroy -auto-approve
aws_key_pair.mykey: Refreshing state... [id=dvsbatch2-key]

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:

  # aws_key_pair.mykey will be destroyed
  - resource "aws_key_pair" "mykey" {
      - arn         = "arn:aws:ec2:us-east-1:156118558224:key-pair/dvsbatch2-key" -> null
      - fingerprint = "5e:76:60:c8:32:2e:f3:d0:c9:40:81:16:93:f3:72:03" -> null
      - id          = "dvsbatch2-key" -> null
      - key_name    = "dvsbatch2-key" -> null
      - key_pair_id = "key-0028ee04827d5a1f9" -> null
      - public_key  = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCejHfC0soRlaHPvTFISxESTi2jnq0EXZUc2WjP7
zM2kM7njuPlyl/hC7ZTkWki9HfA2LZ+wLFIKVe9p7epHfoRm/ivFGO0ASNzjI6MizzRdoFpnYFSoXxLqd2MyV6rfReJZqHjdwX62
VZgAVFMAlm4L55fXVkG7yfDSTw0tFG3WgieTO22wfp4YHr5GEUOhB3VLFxFFCq8ESvl0swp6K8C445WfE6GxLXlS9KeJG4bQ6cNn
DlIyS66JAqSYCBWGKzQj4h3ojWEdkBodJnNbhDuPQvudcTjlimXZzfOTSFs819v0tJsrgpJ6a4wdTDZIVAZT1yN3KKciLT impor
-openssh-key" -> null
      - tags        = {} -> null
      - tags_all    = {} -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.
aws_key_pair.mykey: Destroying... [id=dvsbatch2-key]
aws_key_pair.mykey: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
[root@terraform-workstation mycode]#
```




```
Destroy complete! Resources: 1 destroyed.
[root@terraform-workstation mycode]# terraform show

[root@terraform-workstation mycode]#
```

4. Variables and datatypes

In case if we want to define the variables in terraform there are many ways let's explore them one by one.

variable section:

```
variable "your-var-name" {
        default = "any default values"
        type = (optional) string

}
```

It's not mandatory to give the type of variable but you can define it based on your requirement.

Different data types for variables in terraform:

https://www.terraform.io/language/values/variables

Type:

string
list(<TYPE>)
set(<TYPE>)
map(<TYPE>)
object({<ATTR NAME> = <TYPE>, ... })
tuple([<TYPE>, ...])

Output section:

Whenever you have a requirement of printing the output on the screen then you can use the output section.

```
Output "name-on-screen" {
        value  = "${var.varname}"
        OR
        value = "${resourcename.resource-local-name.idname}"

}
```

Data types:

When we are delcaring the variables it’s a good practise to provide the variable type as mentioned above we have diff data types like string,number,map,list,object. Let's check them one by one

1 String data type:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "batchno" {
        type = string
        default = "dvsbatch2"
}


/*output*/
output "myoutput" {
        value = "${var.batchno}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "batchno" {
        type = string
        default = "dvsbatch2"
}


/*output*/
output "myoutput" {
        value = "${var.batchno}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  + myoutput = "dvsbatch2"

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:

myoutput = "dvsbatch2"
[root@terraform-workstation mycode]#
```

2 Number data type:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
```

```
variable "batchno" {
        type = number
        default = 2
}


/*output*/
output "myoutput" {
        value = "${var.batchno}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "batchno" {
        type = number
        default = 2
}


/*output*/
output "myoutput" {
        value = "${var.batchno}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = "dvsbatch2" -> 2

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:

myoutput = 2
[root@terraform-workstation mycode]#
```

3 List data type:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_courses" {
        type = list
        default = ["aws","devops","azure","azuredevops"]
}


/*output*/
output "myoutput" {
        value = "${var.dvs_courses[2]}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_courses" {
        type = list
        default = ["aws","devops","azure","azuredevops"]
}


/*output*/
output "myoutput" {
        value = "${var.dvs_courses[2]}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = 2 -> "azure"

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:

myoutput = "azure"
[root@terraform-workstation mycode]#
```

4 Map data type:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
   type = map
   default = {
      "azure" = "pipelines,cicd"
      "azcloud" = "vm,lb,fw,vnet"
    }
}

/*output*/
output "myoutput" {
        value = "${var.dvs_course.azure}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
   type = map
   default = {
       "azure" = "pipelines,cicd"
       "azcloud" = "vm,lb,fw,vnet"
     }
}



/*output*/
output "myoutput" {
        value = "${var.dvs_course.azure}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = "azure" -> "pipelines,cicd"

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

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

Outputs:

myoutput = "pipelines,cicd"
[root@terraform-workstation mycode]#
```

5 Object data type:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

```
/* Variables */
variable "dvs_course" {
   type = object({course_name=string,duration=number})
   default = {
      course_name = "az&azdevops"
      duration = 80
     }
}

/*output*/
output "myoutput" {
       value = "${var.dvs_course.course_name}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
   type = object({course_name=string,duration=number})
   default = {
       course_name = "az&azdevops"
       duration = 80
     }
}

/*output*/
output "myoutput" {
       value = "${var.dvs_course.course_name}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = "pipelines,cicd" -> "az&azdevops"

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:

myoutput = "az&azdevops"
[root@terraform-workstation mycode]# vi providers.tf
```

5 Working with variable

We can declare variable values in multiple ways they are as follows

1 Variable with default value:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
        type = string
        default = "az&azdevops"
}

/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
        type = string
        default = "az&azdevops"
}

/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```

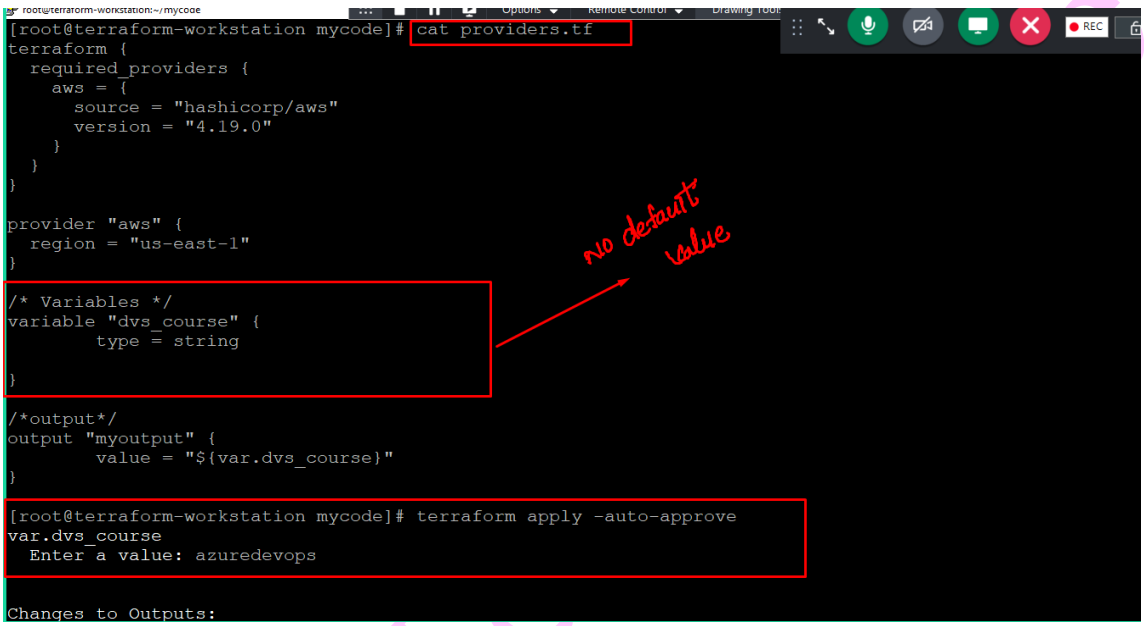
Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = "8000" -> "az&azdevops"

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

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

Outputs:

myoutput = "az&azdevops"
[root@terraform-workstation mycode]#
```

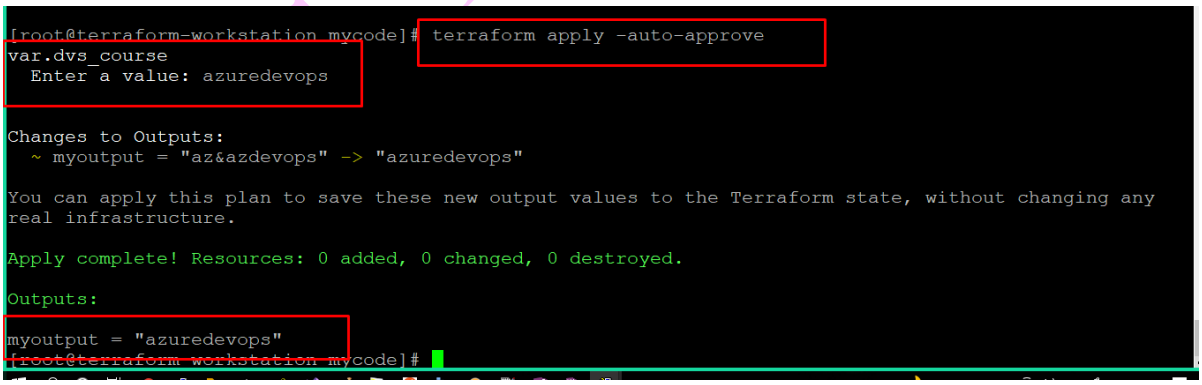
2 Variable value during runtime:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
```

```
        type = string
        /*Just don't give the default value*/
}

/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```



Execution:



3 Passing values from command line: (-var)

Syntax: terraform apply -auto-approve **-var "dvs_course=azurecloud"**

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
        type = string
        default = "az&azdevops"

}

/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```

```
[root@terraform-workstation mycode]# cat providers.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
        type = string
        default = "az&azdevops"

}

/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform apply -auto-approve

Changes to Outputs:
  ~ myoutput = "azuredevops" -> "az&azdevops"

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:

myoutput = "az&azdevops"
```

We are going to overwrite the value of our variable "dvs_course" to "azurecloud"

```
[root@terraform-workstation mycode]# terraform apply -auto-approve -var "dvs_course=azurecloud"

Changes to Outputs:
  ~ myoutput = "az&azdevops" -> "azurecloud"

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:

myoutput = "azurecloud"
```

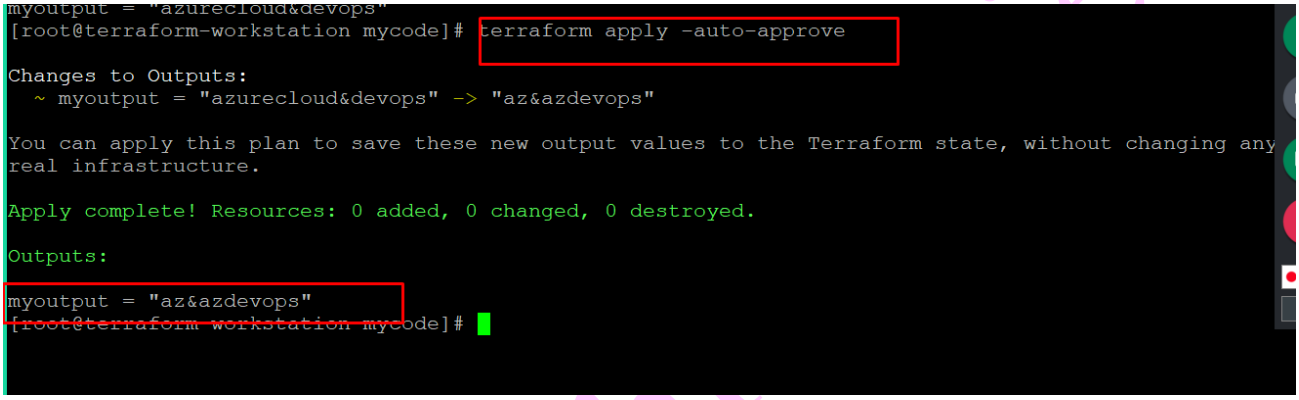
4 Overwriting multiple values from a file: (-var-file)

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* Variables */
variable "dvs_course" {
        type = string
        default = "az&azdevops"

}
```

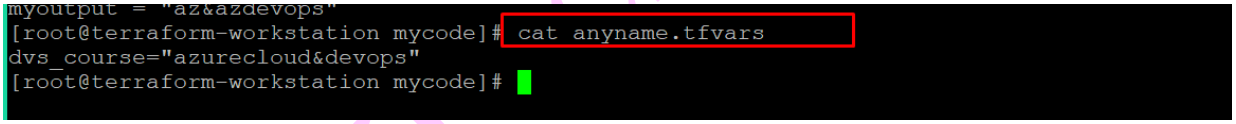
```
/*output*/
output "myoutput" {
        value = "${var.dvs_course}"
}
```

Execution:

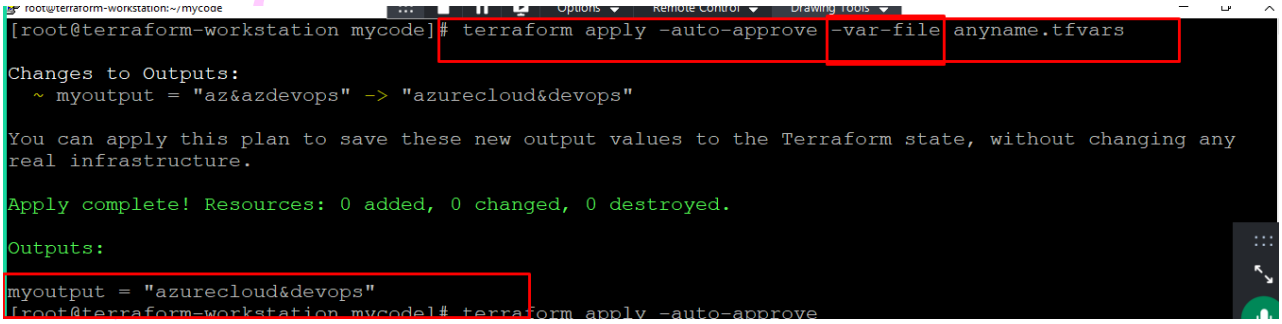
without overwriting we get default value



Overwriting values from a file

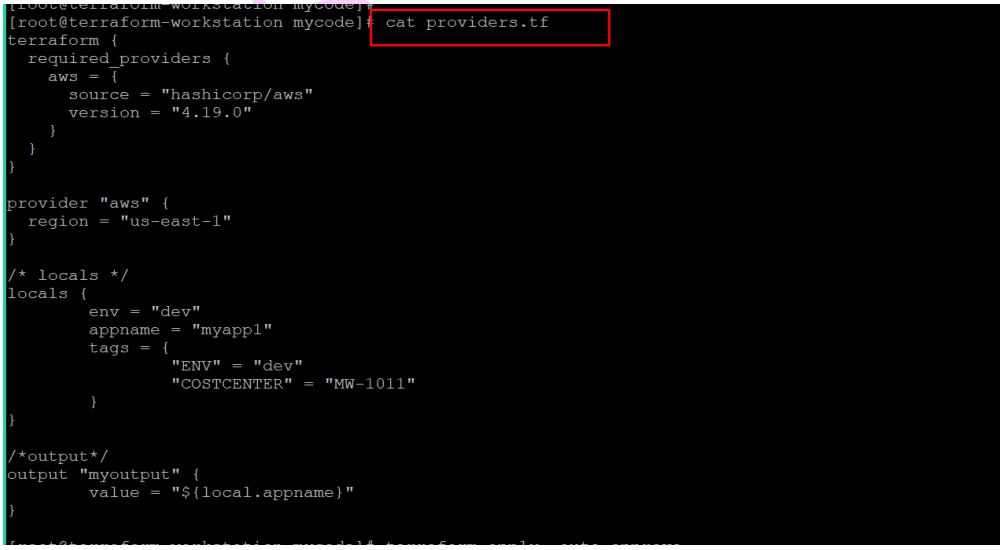


terraform apply -auto-approve -var-file anyname.tfvars

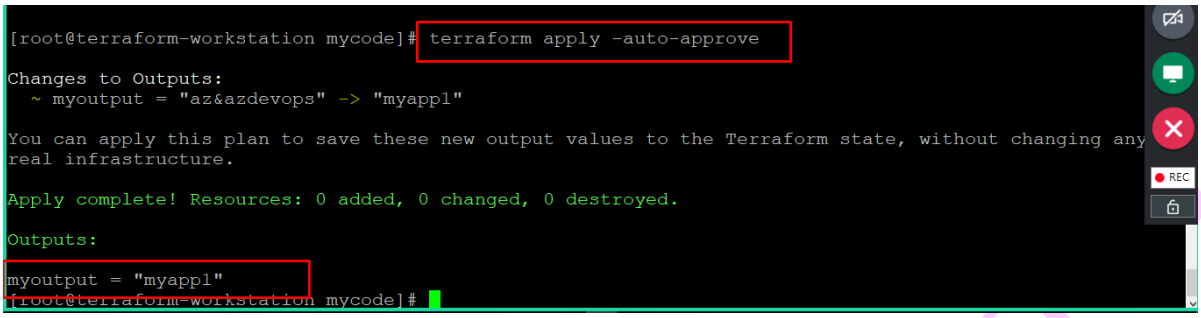


5 Locals:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

/* locals */
locals {
        env = "dev"
        appname = "myapp1"
        tags = {
                "ENV" = "dev"
                "COSTCENTER" = "MW-1011"
        }
}

/*output*/
output "myoutput" {
        value = "${local.appname}"
}
```



Execution:



6 Segregating code

It's not mandatory to dump all the code in the single file called main.tf you can keep it in different files.

You can keep diff sections in diff files like variables,locals,main,tfvars,etc ..

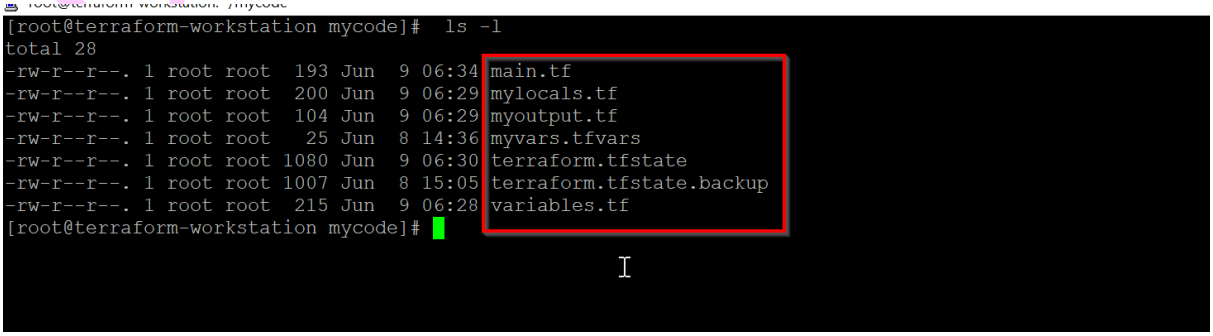
Note: But entire code should be in same folder.

Our total code:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}


resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name            = "${var.ec2_key}"
```

```
  security_groups = ["${var.ec2_secgroup}"]
  tags            = "${local.tags}"

}


/*local*/
locals {
        tags = {
       "ENV" = "DEV"
       "NAME" = "dvsserver1"
  }
}


/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}


/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"

/*outputs*/
output "myprivip" {
       value = "${aws_instance.myec2.private_ip}"
}
```

Segregated code:



providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags          = "${local.tags}"

}
```

locals.tf

```
/*local*/
locals {
        tags = {
       "ENV" = "DEV"
       "NAME" = "dvsserver1"
  }
}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
```

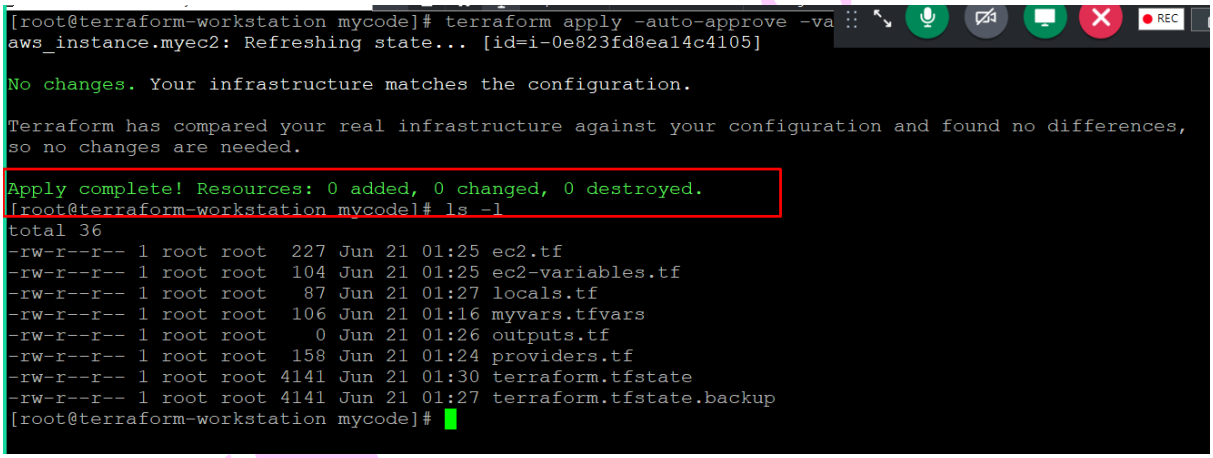
variable "ec2_secgroup" {}

myvars.tfvars

/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"

outputs.tf

output "myprivip" {
value = "${aws_instance.myec2.private_ip}"
}



7 Meta arguments

Terraform supports multiple meta arguments which help us to develop the code in precise way.

depends on
Count with element
Count with index
for_each

1 depends_on:

Imagine a situation where you have a requirement that you need to create multiple resources where one of the resource creation depends on other one. In simple if resource1 got created **then only** other resource 2 should get created otherwise it should fail/skip/stop the execution.

Let's say that I need to create the file with server information and it should happen iff s3 bucket creation is completed.

Total Code:

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
              "ENV" = "DEV"
```

```
            "Name" = "dvsserver"
  }

}

resource "null_resource" "printresult" {
  provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} got created successfully >
/tmp/myoutput.txt"
  }
  depends_on = [aws_instance.myec2]
}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}
```

myvars.tfvars

```
/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"
```

outputs.tf

```
output "myprivip" {
      value = "${aws_instance.myec2.private_ip}"
}
```

Execution:

```
[root@terraform-workstation mycode]# terraform plan -var-file myvars.tfvars

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:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0cff7528ff583bf9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
```

```
[root@terraform-workstation mycode]# ls -l /tmp/myoutput.txt
ls: cannot access /tmp/myoutput.txt: No such file or directory
[root@terraform-workstation mycode]# terraform apply -var-file myvars.tfvars

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:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0cff7528ff583bf9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
```

Before we apply it says no such file or directory, we will check once done with apply.

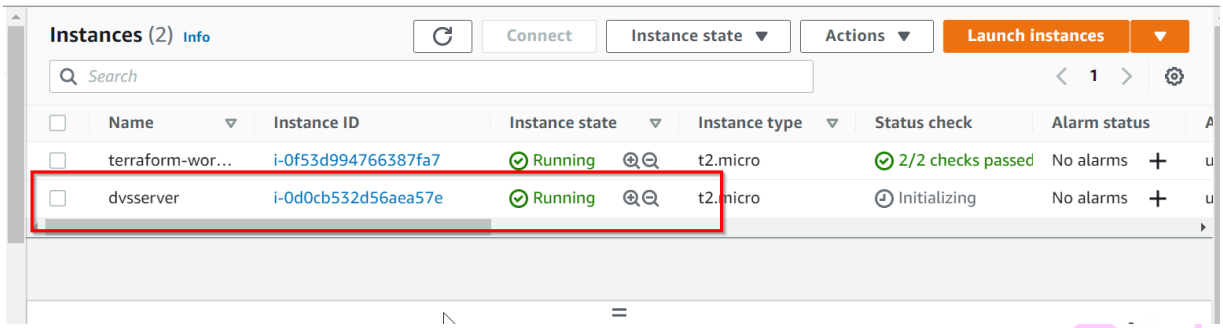
```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

myprivip = "172.31.94.227"
[root@terraform-workstation mycode]# ls -l /tmp/myoutput.txt
-rw-r--r-- 1 root root 39 Jun 22 12:36 /tmp/myoutput.txt
[root@terraform-workstation mycode]# cat /tmp/myoutput.txt
172.31.94.227 got created successfully
[root@terraform-workstation mycode]#
```

2 Count:

Each and every resource we can specify an option call "count", at any point of time if you want to create more number of same type of resources then you can create them by specifying the count value.

Let's say in the above example we are creating only one server let's say we want to create two servers then ??

From the console we can say that we have only one server



Let's say that I want to have 3 servers then ? We should get the servers as "dvsserver1","dvsserver2","dvsserver3"

Let's see what will happen if we give count=3 to our resource.

root@terraform-workstation:/tmp/mycode

```
[root@terraform-workstation mycode]# cat ec2.tf
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
                "ENV" = "DEV"
                "Name" = "dvsserver"
  }
  count = 3

}
[root@terraform-workstation mycode]#
```

Total Code:

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
              "ENV" = "DEV"
              "Name" = "dvsserver"
  }
  count = 3

}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}
```

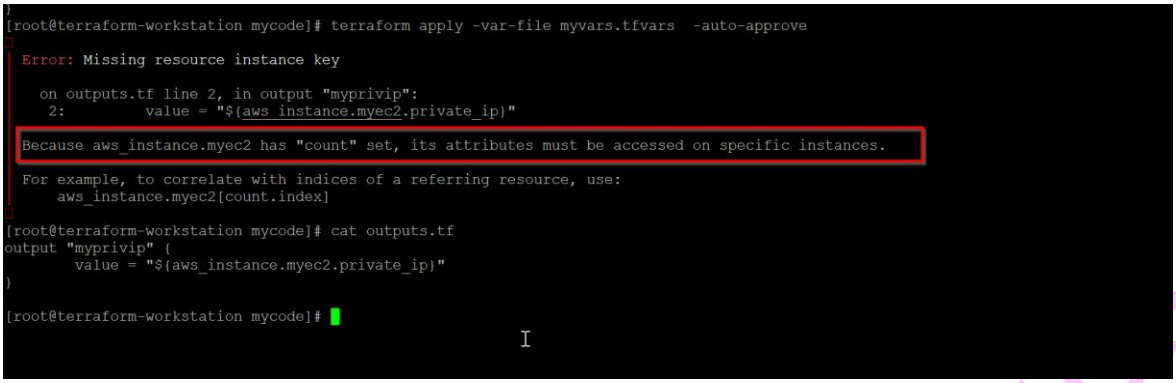
myvars.tfvars

```
/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"
```

outputs.tf

```
output "myprivip" {
    value = "${aws_instance.myec2.private_ip}"
}
```

Execution:

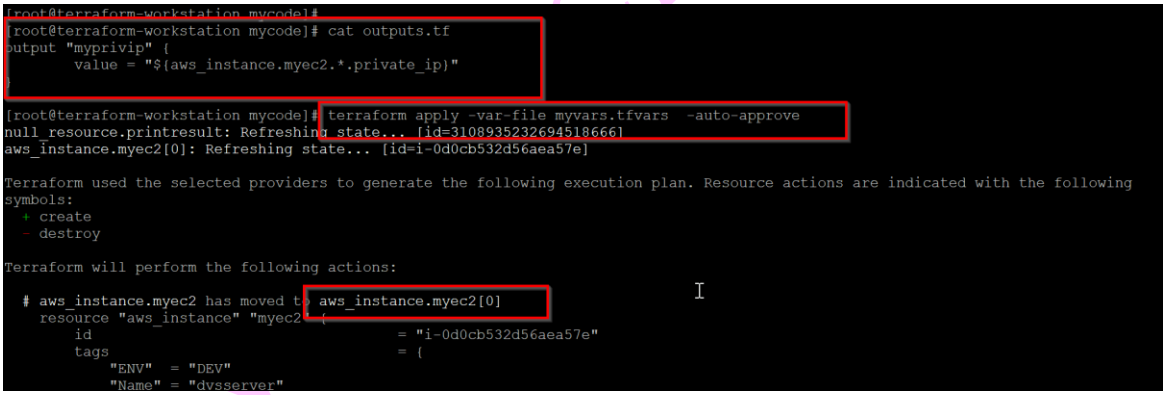


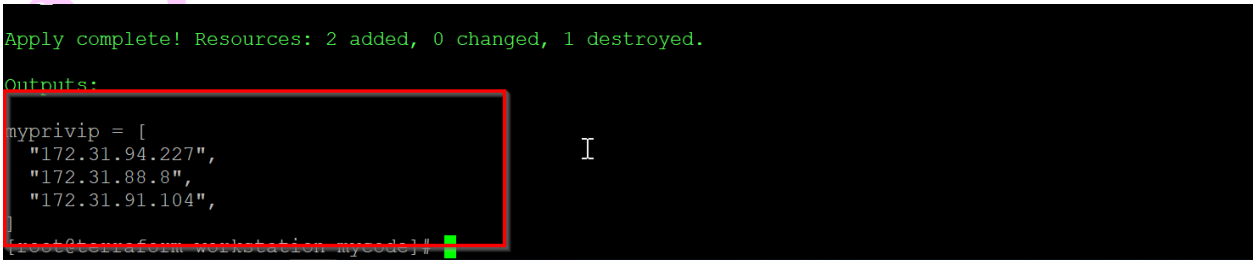
Here we start getting error messages in the outputs.tf file why ?? Because we are going to have 3 servers hence the syntax of printing the content should be different. It should be looking like below.

Outputs.tf:

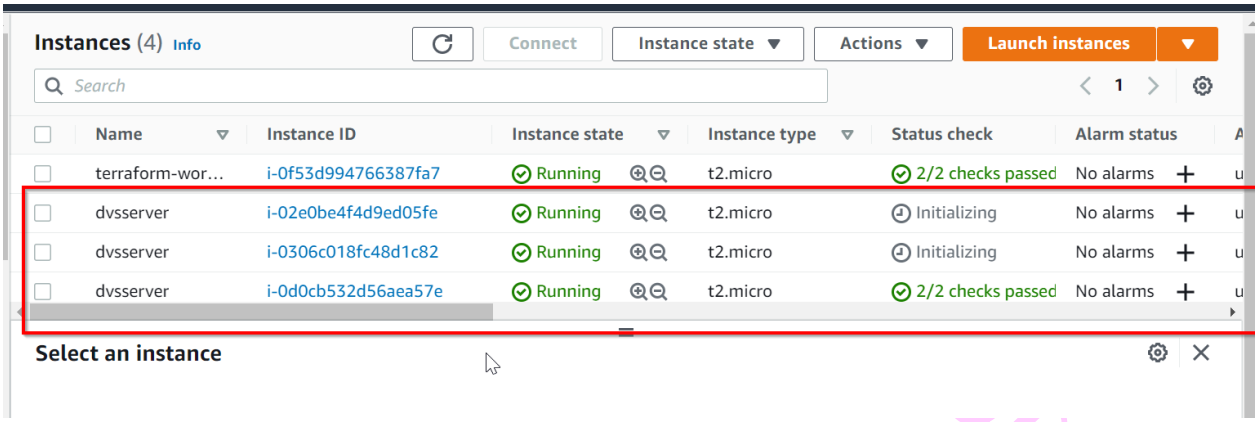
```
output "myprivip" {
      value = "${aws_instance.myec2.*.private_ip}"
}
```

Final execution post output changes:





Here we got the output as well, but did we get server names as expected in the console??



We got 3servers but all have same names. Then how we can make sure that we are getting the right values for our servers. Then you can use count.index

3 count with index:

Here our requirement is to have our servers with proper names ie., dvsserver1,dvsserver2,dvsserver3

Syntax:
Just add "${count.index}" to get the values as per the count. In our case we gave count as 3 hence we are going to get index as 0,1,2

But we need to have server as 1,2,3 hence we can increment count.index with 1 so it will be like

Syntax:

"${count.index+1}"

Total Code:

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}
```

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

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
              "ENV" = "DEV"
              "Name" = "dvsserver${count.index+1}"
  }
  count = 3

}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}
```

myvars.tfvars

```
/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"
```

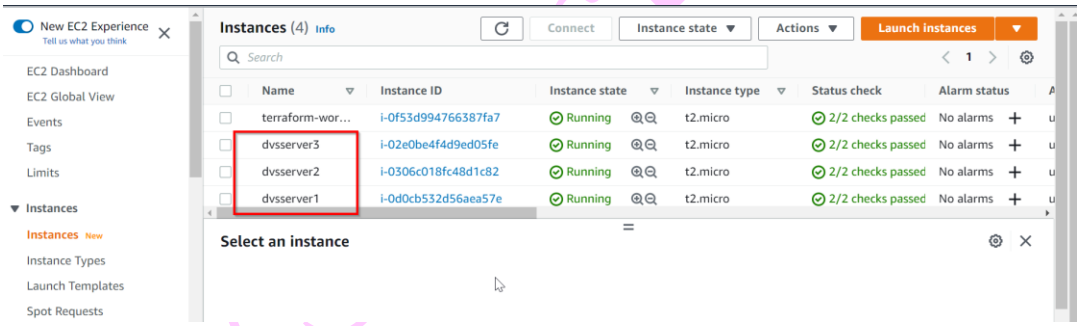
outputs.tf

```
output "myprivip" {
      value = "${aws_instance.myec2.*.private_ip}"
}
```

Execution:

```
[root@terraform-workstation mycode]# cat ec2.tf
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
            "ENV" = "DEV"
            "Name" = "dvsserver${count.index+1}"
  }
  count = 3

}
[root@terraform-workstation mycode]# terraform apply -var-file myvars.tfvars  -auto-approve
aws_instance.myec2[2]: Refreshing state... [id=i-02e0be4f4d9ed05fe]
aws_instance.myec2[0]: Refreshing state... [id=i-0d0cb532d56aea57e]
aws_instance.myec2[1]: Refreshing state... [id=i-0306c018fc48d1c82]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_instance.myec2[0] will be updated in-place
  ~ resource "aws_instance" "myec2" {
      id                                   = "i-0d0cb532d56aea57e"
    ~ tags                                 = {
        ~ "Name" = "dvsserver" -> "dvsserver1"
          # (1 unchanged element hidden)
      }
    ~ tags_all                             = {
        ~ "Name" = "dvsserver" -> "dvsserver1"
```

Now we see that the server names as below



4 count with element:

Just imagine a situation that I don't want the server names as "dvsserver1,dvsserver2,dvsserver3" instead of that I want to have names as "appserver,dbserver,backendserver" then ??? How are we going to achieve it ?? In such cases we can make use of our element with count.

So our syntax looks like below:

```
variable "server_names" {
        type = list
        Default = ["appserver","dbserver","backendserver"]
}
```

Accessing values:

"${element(var.server_names,count.index)}"

Total code:

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
            "ENV" = "DEV"
            "Name" = "${element(var.server_names,count.index)}"
  }
  count = 3

}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}
variable "server_names" {
        type = list
        Default = ["appserver","dbserver","backendserver"]

}
```

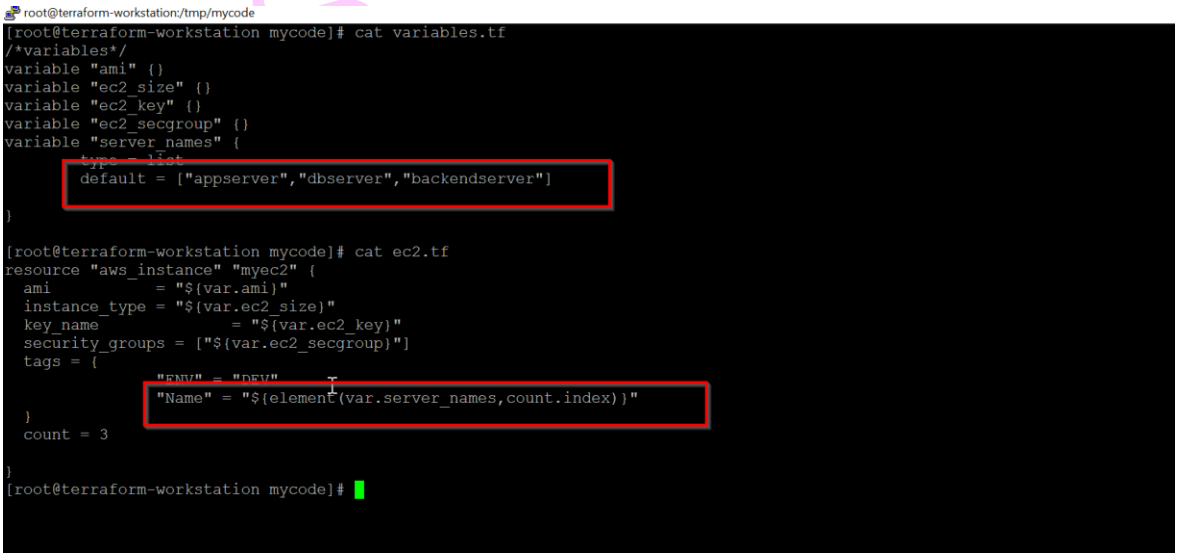
myvars.tfvars

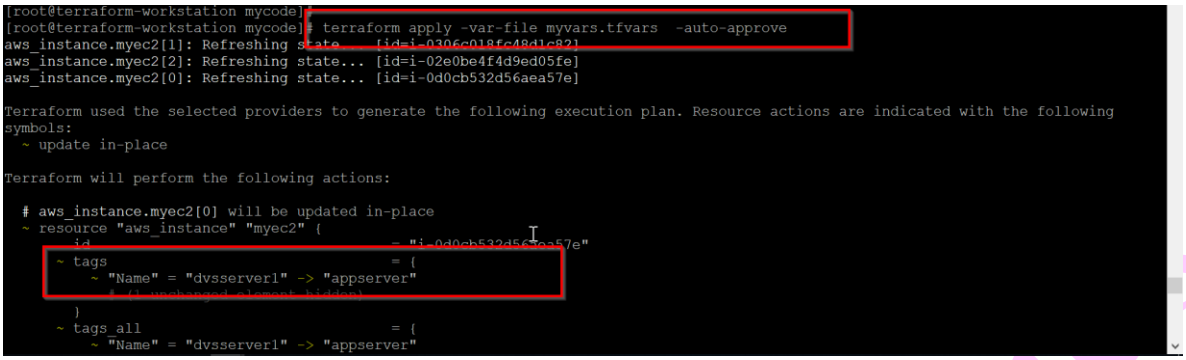
```
/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"
```

outputs.tf

```
output "myprivip" {
      value = "${aws_instance.myec2.*.private_ip}"
}
```

Execution:





5 for_each with map:

Till now we are building the infrastructure in same availability zone, just image that we need to have the ec2 servers from different subnets/availibility zones then ??

We can make use of our maps with for_each combination like below.

Syntax:

```
variable "server_details" {
        type = map
        default = {
                "appserver" = "us-east-1a"
                "dbserver" = "us-east-1b"
                "backendserver" = "us-east-1c"

        }

}


resource "aws_instance" "myec2" {
  for_each              = "${var.server_details}"
  ami           = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  availability_zone = "${each.value}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
              "ENV" = "DEV"
             "Name" = "${each.key}"
  }
}
```

Total code:

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

ec2.tf

```
resource "aws_instance" "myec2" {
  for_each          = "${var.server_details}"
  ami               = "${var.ami}"
  instance_type = "${var.ec2_size}"
  key_name          = "${var.ec2_key}"
  availability_zone = "${each.value}"
  security_groups = ["${var.ec2_secgroup}"]
  tags = {
            "ENV" = "DEV"
           "Name" = "${each.key}"
  }
}
```

variables.tf

```
/*variables*/
variable "ami" {}
variable "ec2_size" {}
variable "ec2_key" {}
variable "ec2_secgroup" {}
variable "server_details" {
        type = map
        default = {
                "appserver" = "us-east-1a"
```

```
        "dbserver" = "us-east-1b"
        "backendserver" = "us-east-1c"

    }

}
```

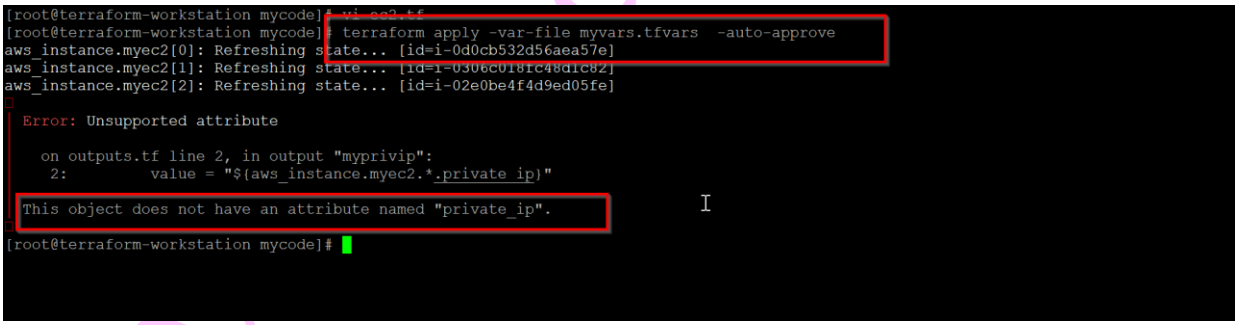
myvars.tfvars

```
/*tfvars*/
ami="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_key="shaan-ramana-nvirg-key"
ec2_secgroup="batch2-sg"
```

outputs.tf

```
output "myprivip" {
    value = "${aws_instance.myec2.*.private_ip}"
}
```

Execution:



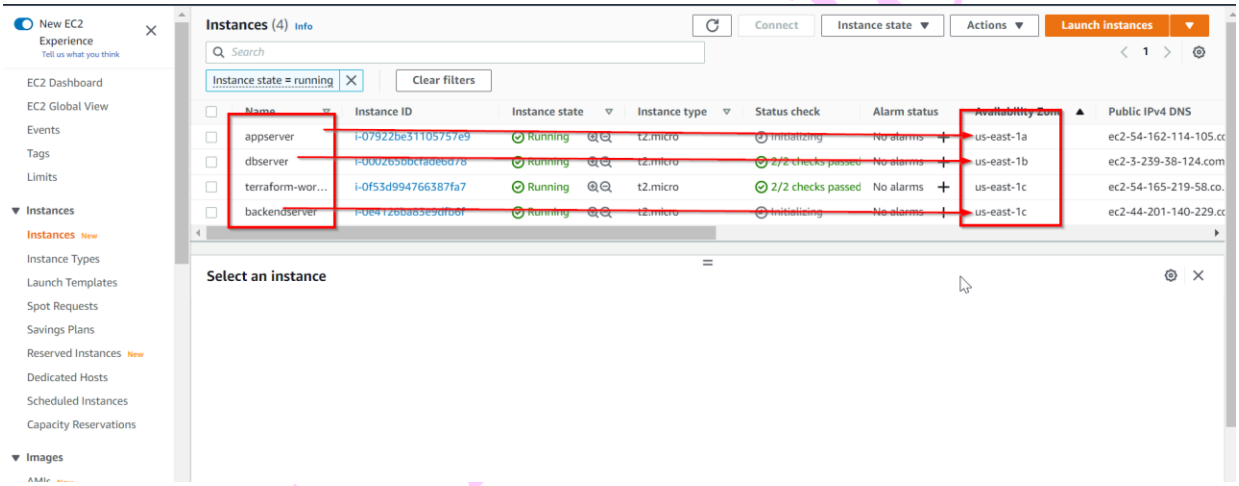
In the above we see that outputs.tf is throwing the error in simple since we are refering to the map we have to change the outputs.tf file as below.

Outputs.tf:

```
output "myprivip" {
    value = "${values(aws_instance.myec2)[*].private_ip}"
}
```

```
[root@terraform-workstation mycode]# vi outputs.tf
[root@terraform-workstation mycode]# cat outputs.tf
output "myprivip" {
        value = "${values(aws_instance.myec2)[*].private_ip}"
}

[root@terraform-workstation mycode]# ^C
[root@terraform-workstation mycode]# terraform apply -var-file myvars.tfvars  -auto-approve
aws_instance.myec2[1]: Refreshing state... [id=i-0306c018fc48d1c82]
aws_instance.myec2[2]: Refreshing state... [id=i-02e0be4f4d9ed05fe]
aws_instance.myec2[0]: Refreshing state... [id=i-0d0cb532d56aea57e]

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

Terraform will perform the following actions:

  # aws_instance.myec2[0] will be destroyed
  # (because resource does not use count)
  - resource "aws_instance" "myec2" {
      - ami                                  = "ami-0cff7528ff583bf9a" -> null
      - arn                                  = "arn:aws:ec2:us-east-1:156118558224:instance/i-0d0cb532d56aea57e" -> null
      - associate_public_ip_address          = true -> null
```

From the console:



Now our servers got created in the respective availability zones.

8 Working with infrastructure

In this example we are going to build infrastructure for s3,vpc,ec2. Here we are going make sure that our ec2 is coming from the same subnet of our vpc.

Final code:

Providers.tf:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

s3.tf:

```
resource "aws_s3_bucket" "mys3" {
  bucket = "${var.bucketname}"
  tags = "${var.tags}"
}
```

ec2.tf

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami_id}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  vpc_security_group_ids = ["${aws_security_group.mysecgrp.id}"]
  subnet_id             = "${aws_subnet.mysub1.id}"

  tags = {
    Name = "${var.ec2_name}"
  }
}
```

vpc.tf

```
resource "aws_vpc" "myvpc" {
  cidr_block       = "${var.vpc_cidr}"
  tags = {
       Name = "${var.vpc_name}"
  }
}

resource "aws_internet_gateway" "myigw" {
  vpc_id = "${aws_vpc.myvpc.id}"
  tags = {
       Name = "${var.igw_name}"
  }
}

resource "aws_route_table" "myroute" {
  vpc_id = "${aws_vpc.myvpc.id}"

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = "${aws_internet_gateway.myigw.id}"
  }
  tags = {
    Name = "${var.route_name}"
  }
}

resource "aws_subnet" "mysub1" {
  vpc_id     = "${aws_vpc.myvpc.id}"
  cidr_block = "${var.vpc_sub_cidr}"
  tags = {
    Name = "${var.sub_name}"
  }
}

resource "aws_route_table_association" "subassociation" {
  subnet_id      = "${aws_subnet.mysub1.id}"
  route_table_id = "${aws_route_table.myroute.id}"
}

resource "aws_security_group" "mysecgrp" {
  name        = "${var.sec_grp_name}"
  description = "Dvsbatch2 sec group"
  vpc_id      = "${aws_vpc.myvpc.id}"
```

```
 ingress {
   description      = "Allow everyone"
   from_port        = 0
   to_port          = 0
   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"]
 }
 tags = {
   Name = "${var.sec_grp_name}"
 }
}
```

variables.tf:

```
/*vpc variables*/
variable "vpc_cidr" {}
variable "vpc_name" {}
variable "igw_name" {}
variable "route_name" {}
variable "vpc_sub_cidr" {}
variable "sub_name" {}
variable "sec_grp_name" {}

/*ec2 variables*/
variable "ami_id" {}
variable "ec2_size" {}
variable "ec2_name" {}
variable "ec2_key" {}

/*s3 variables*/
variable "bucketname" {}
variable "tags" {
      type = map
}
```

outputs.tf:

```
output "myvpcid" {
      value = "${aws_vpc.myvpc.id}"
}
output "myigwid" {
      value = "${aws_internet_gateway.myigw.id}"
}
output "mysubid" {
      value = "${aws_subnet.mysub1.id}"
}

output "mysecgrpid" {
      value = "${aws_security_group.mysecgrp.id}"
}
```

myvars.tfvars

```
/*vpc values*/
vpc_cidr = "10.10.0.0/16"
vpc_name = "dvsvpc1"
igw_name = "dvsvpc1-igw"
route_name = "dvsvpc1-route1"
vpc_sub_cidr = "10.10.15.0/24"
sub_name = "dvsvpc1-sub1"
sec_grp_name = "dvsvpc1-sg1"

/*ec2 values*/
ami_id="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_name="dvs-server1"
ec2_key="shaan-ramana-nvirg-key"


/*s3 values*/
bucketname="dvsbatch2-2022"
tags = {
   Environment = "Dev"
      Department      = "DevOps"
  }
```

Execution:

```
root@terraform-workstation:~/mydata
[root@terraform-workstation mydata]# terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v4.19.0

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.
[root@terraform-workstation mydata]# terraform plan -var-file myvars.tfvars

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:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0cff7528ff583bf9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
```

```
[root@terraform-workstation mydata]# terraform apply -var-file myvars.tfvars -auto-approve

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:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0cff7528ff583bf9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "shaan-ramana-nvirg-key"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
```

```
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Still creating... [30s elapsed]
aws_instance.myec2: Creation complete after 33s [id=i-089cb4f5fb775d680]

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

myigwid = "igw-03758d0d68fa8b040"
mysecgrpid = "sg-097e96e65c3659637"
mysubid = "subnet-0a0cae03d4f9b7a00"
myvpcid = "vpc-0d8ff30a269d1d4d8"
[root@terraform-workstation mydata]#
```

Entire code is available under
https://github.com/shan5a6/aws-terraform.git

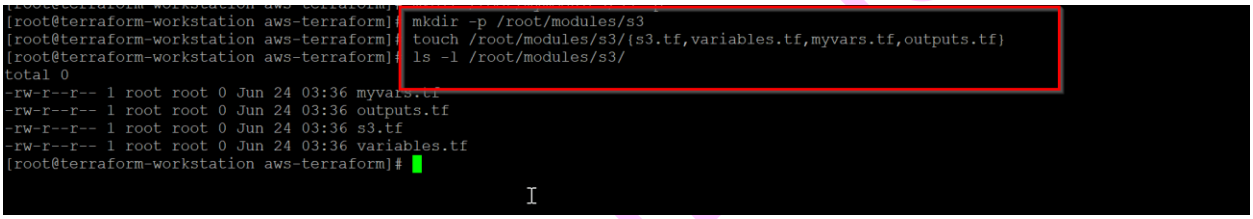
9 Modules

Modules help a lot in code resuability, in simple! define once and consume "n" of times. We just need to pass the parameters to the modules.

We are going to create modules for s3,vpc,ec2. We should make sure that our ec2 is coming from the same subnet as of our vpc.

Creating S3 Module:

Let's create the folder structure for the modules.



S3 Module Code:

/root/modules/s3/s3.tf

```
resource "aws_s3_bucket" "mys3" {
  bucket = "${var.bucketname}"
  tags = "${var.tags}"
}
```

/root/modules/s3/variables.tf

```
variable "bucketname" {}
variable "tags" {
        type = map
}
```

/root/modules/s3/myvars.tfvars

```
bucketname="dvsbatch2-2022"
tags = {
    Environment = "Dev"
        Department      = "DevOps"
```

```
}
```

/root/modules/s3/outputs.tf

```
output "mys3id" {
      value = "${aws_s3_bucket.mys3.id}"
}
```

Verify & execute to see if s3 code is working or not:

Here for testing purpose we are going to give the providers.tf file otherwise it's not required. Once we are done with testing will remove providers.tf file.

/root/modules/s3/providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

```
[root@terraform-workstation s3]# pwd
/root/modules/s3
[root@terraform-workstation s3]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

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.
[root@terraform-workstation s3]# terraform plan -var-file myvars.tfvars

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:

  # aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
```

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

Changes to Outputs:
  + mys3id = (known after apply)

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.
[root@terraform-workstation s3]#
```

From the above output we can confirm that code is working as expected.

Note: Don't forget to delete the providers.tf file

VPC Module Code:

```
drwxr-xr-x 3 root root 143 Jun 24 03:42 s3
[root@terraform-workstation s3]# mkdir -p /root/modules/vpc
[root@terraform-workstation s3]# touch /root/modules/vpc/{vpc.tf,variables.tf,outputs.tf,myvars.tfvars}
[root@terraform-workstation s3]# ls -l /root/modules/vpc
total 0
-rw-r--r-- 1 root root 0 Jun 24 03:46 myvars.tfvars
-rw-r--r-- 1 root root 0 Jun 24 03:46 outputs.tf
-rw-r--r-- 1 root root 0 Jun 24 03:46 variables.tf
-rw-r--r-- 1 root root 0 Jun 24 03:46 vpc.tf
[root@terraform-workstation s3]# ls -l /root/modules/
total 0
drwxr-xr-x 3 root root 143 Jun 24 03:42 s3
drwxr-xr-x 2 root root  79 Jun 24 03:46 vpc
```

File structure:

```
[root@terraform-workstation s3]# tree /root/modules/
/root/modules/
├── s3
│   ├── myvars.tfvars
│   ├── outputs.tf
│   ├── s3.tf
│   └── variables.tf
└── vpc
    ├── myvars.tfvars
    ├── outputs.tf
    ├── variables.tf
    └── vpc.tf

2 directories, 8 files
[root@terraform-workstation s3]#
```

/root/modules/vpc/vpc.tf

```
resource "aws_vpc" "myvpc" {
  cidr_block       = "${var.vpc_cidr}"
  tags = {
       Name = "${var.vpc_name}"
  }
}

resource "aws_internet_gateway" "myigw" {
  vpc_id = "${aws_vpc.myvpc.id}"
```

```
  tags = {
       Name = "${var.igw_name}"
  }
}

resource "aws_route_table" "myroute" {
  vpc_id = "${aws_vpc.myvpc.id}"

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = "${aws_internet_gateway.myigw.id}"
  }
  tags = {
    Name = "${var.route_name}"
  }
}

resource "aws_subnet" "mysub1" {
  vpc_id     = "${aws_vpc.myvpc.id}"
  cidr_block = "${var.vpc_sub_cidr}"
  tags = {
    Name = "${var.sub_name}"
  }
}

resource "aws_route_table_association" "subassociation" {
  subnet_id      = "${aws_subnet.mysub1.id}"
  route_table_id = "${aws_route_table.myroute.id}"
}

resource "aws_security_group" "mysecgrp" {
  name        = "${var.sec_grp_name}"
  description = "Dvsbatch2 sec group"
  vpc_id      = "${aws_vpc.myvpc.id}"

  ingress {
    description      = "Allow everyone"
    from_port        = 0
    to_port          = 0
    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"]
  }
  tags = {
    Name = "${var.sec_grp_name}"
  }
}
```

/root/modules/vpc/variables.tf

```
variable "vpc_cidr" {}
variable "vpc_name" {}
variable "igw_name" {}
variable "route_name" {}
variable "vpc_sub_cidr" {}
variable "sub_name" {}
variable "sec_grp_name" {}
```

/root/modules/vpc/outputs.tf

```
output "myvpcid" {
        value = "${aws_vpc.myvpc.id}"
}
output "myigwid" {
        value = "${aws_internet_gateway.myigw.id}"
}
output "mysubid" {
        value = "${aws_subnet.mysub1.id}"
}
output "mysecgrpid" {
        value = "${aws_security_group.mysecgrp.id}"
}
```

/root/modules/vpc/myvars.tfvars

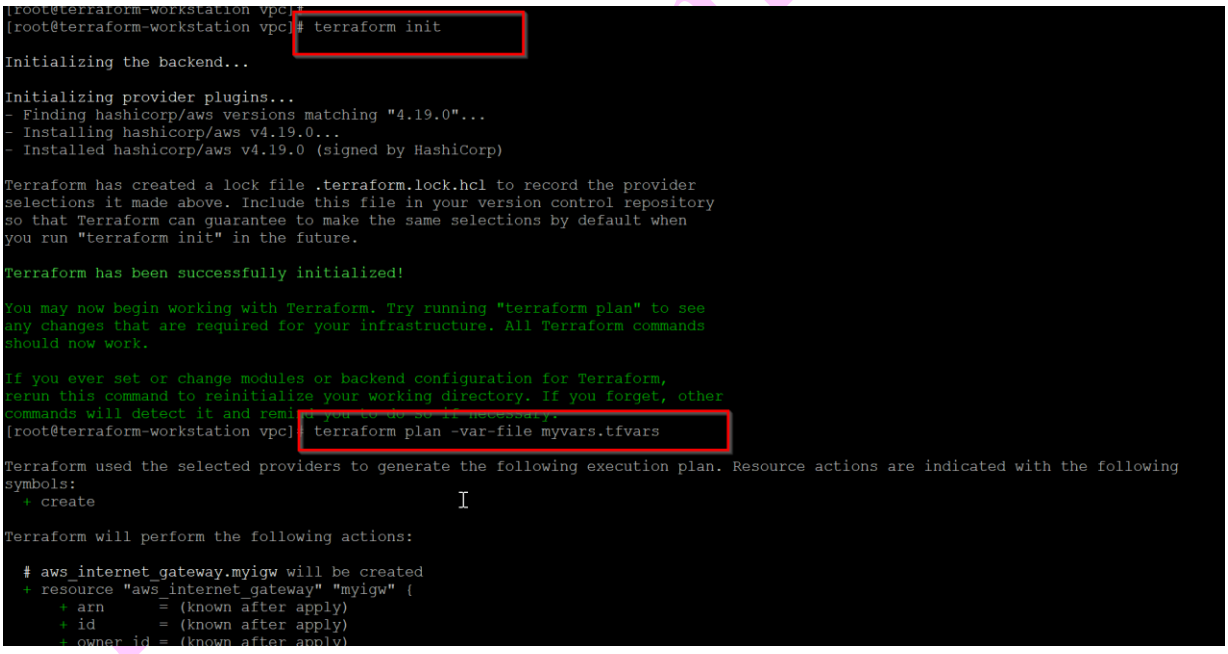
```
vpc_cidr = "10.10.0.0/16"
vpc_name = "dvsvpc1"
igw_name = "dvsvpc1-igw"
route_name = "dvsvpc1-route1"
vpc_sub_cidr = "10.10.15.0/24"
sub_name = "dvsvpc1-sub1"
sec_grp_name = "dvsvpc1-sg1"
```

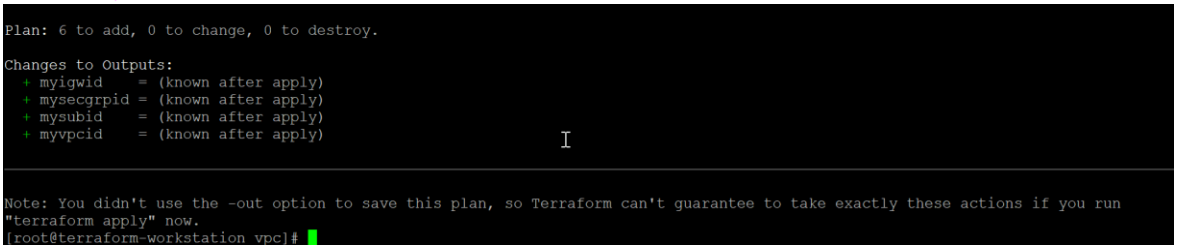
Verify & execute to see if vpc code is working or not:

Here for testing purpose we are going to give the providers.tf file otherwise it's not required. Once we are done with testing will remove providers.tf file.

/root/modules/vpc/providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

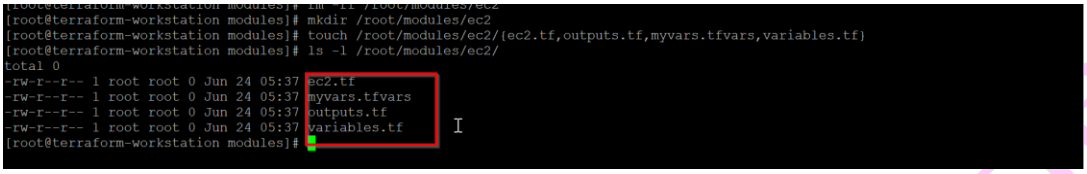




From the above output we can confirm that code is working as expected.

Note: Don't forget to delete the providers.tf file

EC2 Module Code:



For reference you can take ec2 Code from topic8 "working with infrastructure"

/root/modules/ec2/variables.tf

```
/*ec2 variables*/
variable "ami_id" {}
variable "ec2_size" {}
variable "ec2_name" {}
variable "ec2_key" {}
variable "sec_grp_id" {}
variable "sub_id" {}
```

/root/modules/ec2/myvars.tfvars

```
/*ec2 values*/
ami_id="ami-0cff7528ff583bf9a"
ec2_size="t2.micro"
ec2_name="dvs-server1"
ec2_key="shaan-ramana-nvirg-key"
sec_grp_id= "XXXXXXXXX"
sub_id="XXXXXXXXXXX"
```

/root/modules/ec2/ec2.tf:

```
resource "aws_instance" "myec2" {
  ami           = "${var.ami_id}"
  instance_type = "${var.ec2_size}"
  key_name              = "${var.ec2_key}"
  vpc_security_group_ids = ["${var.sec_grp_id}"]
  subnet_id             = "${var.sub_id}"
```

```
  tags = {
    Name = "${var.ec2_name}"
  }
}
```

/root/modules/ec2/outputs.tf

```
output "myprivaip" {
       value = "${aws_instance.myec2.private_ip}"
}
```

Verify & execute to see if ec2 code is working or not:

Here for testing purpose we are going to give the providers.tf file otherwise it's not required. Once we are done with testing will remove providers.tf file.

/root/modules/ec2/providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

Note: here sec_grp_id & sub_id are missing for testing purpose we can take it from our default vpc.

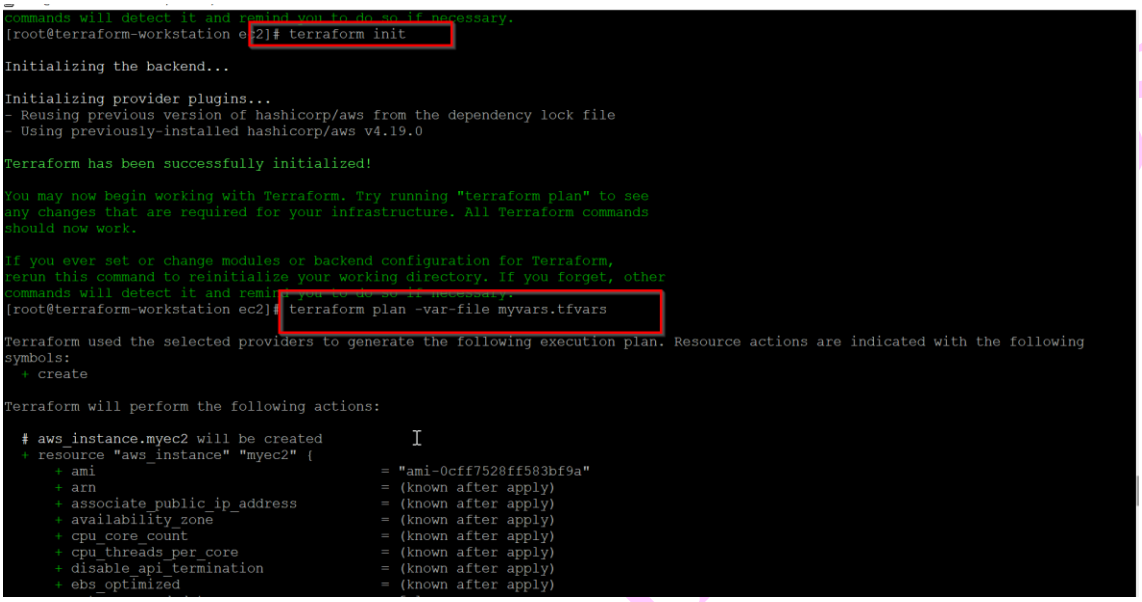
```
sec_grp_id="sg-0821bb050a1d8a9c5"  /*security group id*/
sub_id="subnet-08b7ef5a1839d735d" /*subnet id*/
```

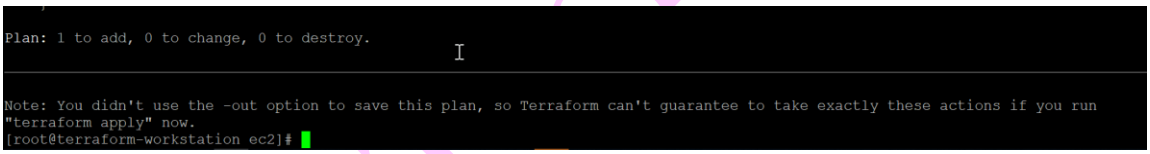
So our final myvars.tfvars for ec2 look like below.

/root/modules/ec2/myvars.tfvars

```
/*ec2 values*/
ami_id="ami-0cff7528ff583bf9a"
```

ec2_size="t2.micro"
ec2_name="dvs-server1"
ec2_key="shaan-ramana-nvirg-key"
secgrp_id="sg-0821bb050a1d8a9c5"
sub_id="subnet-08b7ef5a1839d735d"





From the above output we can confirm that code is working as expected.

Note: Don't forget to delete the providers.tf file

Consuming Moduels:

Now let's finally consume our modules for our application. Imagine that one of the application called "application1" wants to consume the modules then how are we going to define it ??

Please follow the below steps:

Make sure that you are collecting all variables & output of all the modules like below.

S3 Variables:

[root@terraform-workstation ec2]# cat /root/modules/s3/variables.tf

```
variable "bucketname" {}
variable "tags" {
      type = map
}
```

Vpc variables:

```
[root@terraform-workstation ec2]# cat /root/modules/vpc/variables.tf
variable "vpc_cidr" {}
variable "vpc_name" {}
variable "igw_name" {}
variable "route_name" {}
variable "vpc_sub_cidr" {}
variable "sub_name" {}
variable "sec_grp_name" {}
```

Ec2 variables:

```
[root@terraform-workstation ec2]# cat /root/modules/ec2/variables.tf
variable "ami_id" {}
variable "ec2_size" {}
variable "ec2_name" {}
variable "ec2_key" {}
variable "sec_grp_id" {}
variable "sub_id" {}
```

S3 output variables:

```
[root@terraform-workstation ec2]# cat /root/modules/s3/outputs.tf
output "mys3id" {
      value = "${aws_s3_bucket.mys3.id}"
}
```

Vpc output variables:

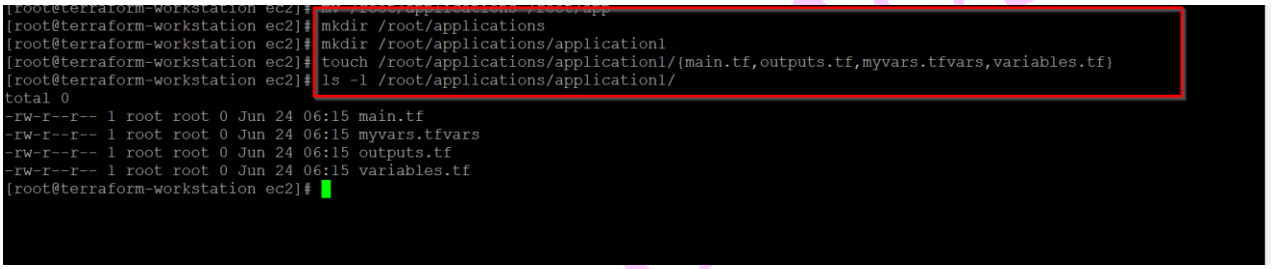
```
[root@terraform-workstation ec2]# cat /root/modules/vpc/outputs.tf
output "myvpcid" {
      value = "${aws_vpc.myvpc.id}"
}
output "myigwid" {
      value = "${aws_internet_gateway.myigw.id}"
}
output "mysubid" {
      value = "${aws_subnet.mysub1.id}"
```

```
}
output "mysecgrpid" {
      value = "${aws_security_group.mysecgrp.id}"
}
```

Ec2 output variables:

```
[root@terraform-workstation ec2]# cat /root/modules/ec2/outputs.tf
output "myprivaip" {
      value = "${aws_instance.myec2.private_ip}"
}
```

Let's build the application folder structure:



Here we have our application1 which wants to consume our modules s3,vpc,ec2. Our final code is going to look like below.

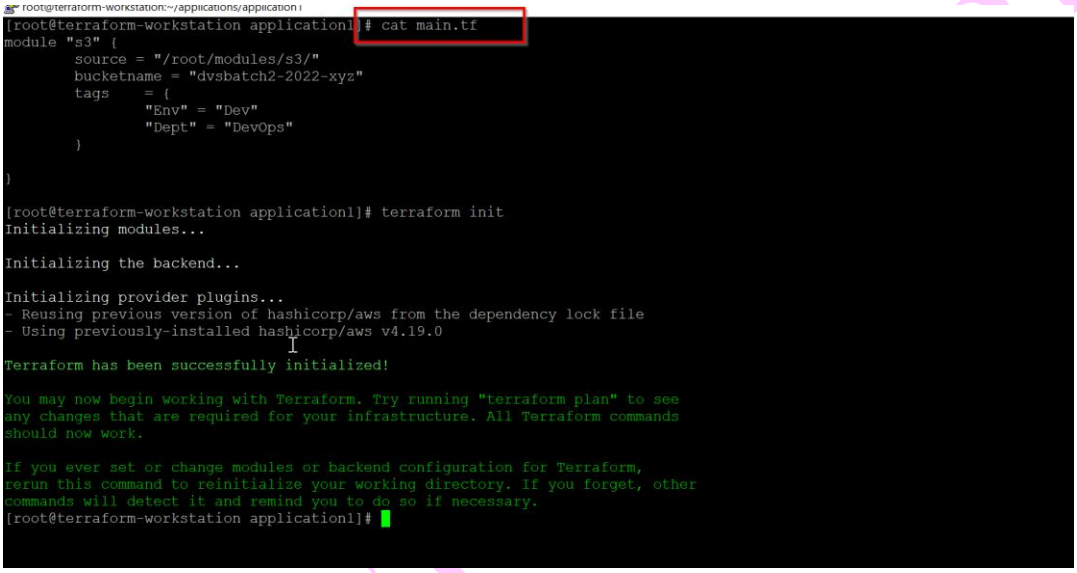
Make sure that you are having providers.tf file.

providers.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

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

Our application main.tf file for S3 looks like below.

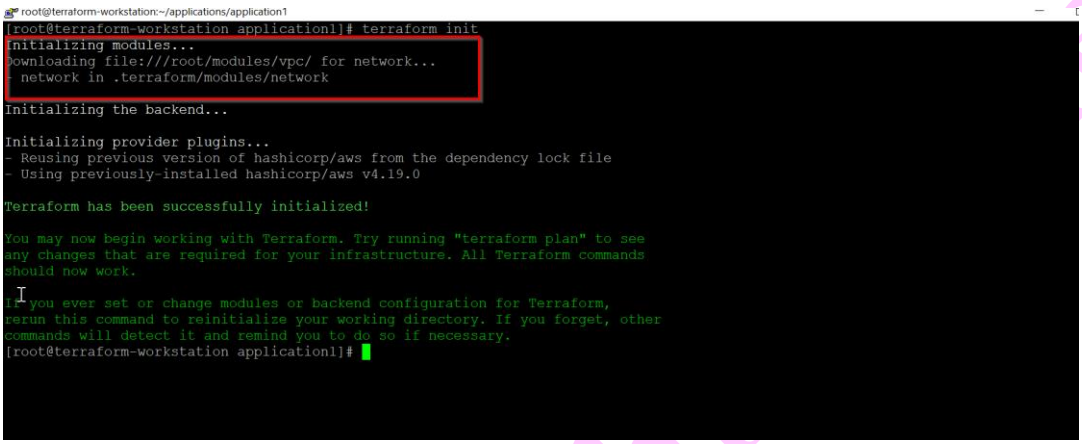
```
module "s3" {
      source = "/root/modules/s3/"
      bucketname = "dvsbatch2-2022-xyz"
      tags    = {
            "Env" = "Dev"
            "Dept" = "DevOps"
      }

}
```



Similarly main.tf with s3 & vpc looks like below.

```
module "s3" {
      source = "/root/modules/s3/"
      bucketname = "dvsbatch2-2022-xyz"
      tags    = {
                  "Env" = "Dev"
                  "Dept" = "DevOps"
      }

}

module "network" {
      source = "/root/modules/vpc/"
      vpc_cidr = "10.10.0.0/16"
```

```
    vpc_name = "dvsvpc1"
    igw_name = "dvsvpc1-igw"
    route_name = "dvsvpc1-route1"
    vpc_sub_cidr = "10.10.15.0/24"
    sub_name = "dvsvpc1-sub1"
    sec_grp_name = "dvsvpc1-sg1"
}
```



Let's finally add our ec2 module code as well.

Total application1 code

```
module "s3" {
    source = "/root/modules/s3/"
    bucketname = "dvsbatch2-2022-xyz"
    tags    = {
                "Env" = "Dev"
                "Dept" = "DevOps"
    }

}

module "network" {
    source = "/root/modules/vpc/"
    vpc_cidr = "10.10.0.0/16"
    vpc_name = "dvsvpc1"
    igw_name = "dvsvpc1-igw"
    route_name = "dvsvpc1-route1"
    vpc_sub_cidr = "10.10.15.0/24"
    sub_name = "dvsvpc1-sub1"
    sec_grp_name = "dvsvpc1-sg1"
```

```
}

module "compute" {
        source = "/root/modules/ec2/"
        ami_id="ami-0cff7528ff583bf9a"
        ec2_size="t2.micro"
        ec2_name="dvs-server1"
        ec2_key="shaan-ramana-nvirg-key"
        sec_grp_id="$$$$$$$$$$$$$$"
        sub_id="$$$$$$$$$$$$$$"
}
```

Here how are we going to give the values of "sec_grp_id" & "sub_id" ????? Because these values we are going to get from VPC module.

So always remember that output of previous module we can use it as input. Hence in our VPC module we defined the outputs

```
output "mysubid" {
      value = "${aws_subnet.mysub1.id}"
}
output "mysecgrpid" {
      value = "${aws_security_group.mysecgrp.id}"
}
```

Hence we are going to them as inputs to our EC2 module. So its goanna look like below.

```
sec_grp_id="${module.vpc.mysecgrpid}"
sub_id="${module.vpc.mysubid}"
```

So final code looks like below:

```
module "s3" {
        source = "/root/modules/s3/"
        bucketname = "dvsbatch2-2022-xyz"
        tags    = {
                        "Env" = "Dev"
                        "Dept" = "DevOps"
        }

}
```

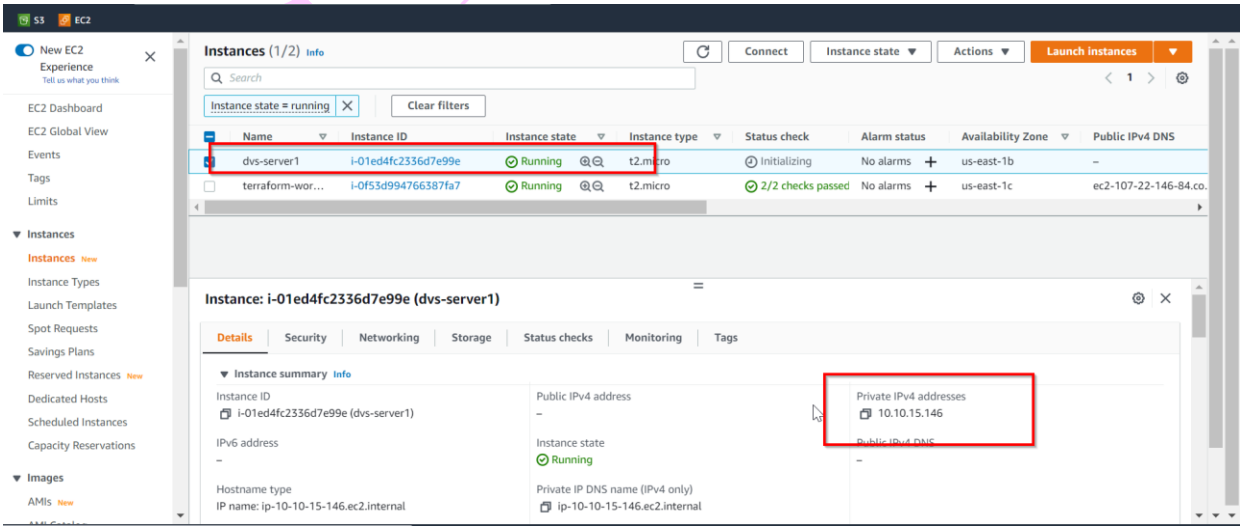
```
module "network" {
        source = "/root/modules/vpc/"
        vpc_cidr = "10.10.0.0/16"
        vpc_name = "dvsvpc1"
        igw_name = "dvsvpc1-igw"
        route_name = "dvsvpc1-route1"
        vpc_sub_cidr = "10.10.15.0/24"
        sub_name = "dvsvpc1-sub1"
        sec_grp_name = "dvsvpc1-sg1"
}

module "compute" {
        source = "/root/modules/ec2/"
        ami_id="ami-0cff7528ff583bf9a"
        ec2_size="t2.micro"
        ec2_name="dvs-server1"
        ec2_key="shaan-ramana-nvirg-key"
        sec_grp_id="${module.network.mysecgrpid}"
        sub_id="${module.network.mysubid}"
}
```

Execution:

```
/root/applications/application1
[root@terraform-workstation application1]# terraform init
Initializing modules...

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v4.19.0

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.
[root@terraform-workstation application1]# 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:

  # module.compute.aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0cff7528ff583bf9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_termination              = (known after apply)
```

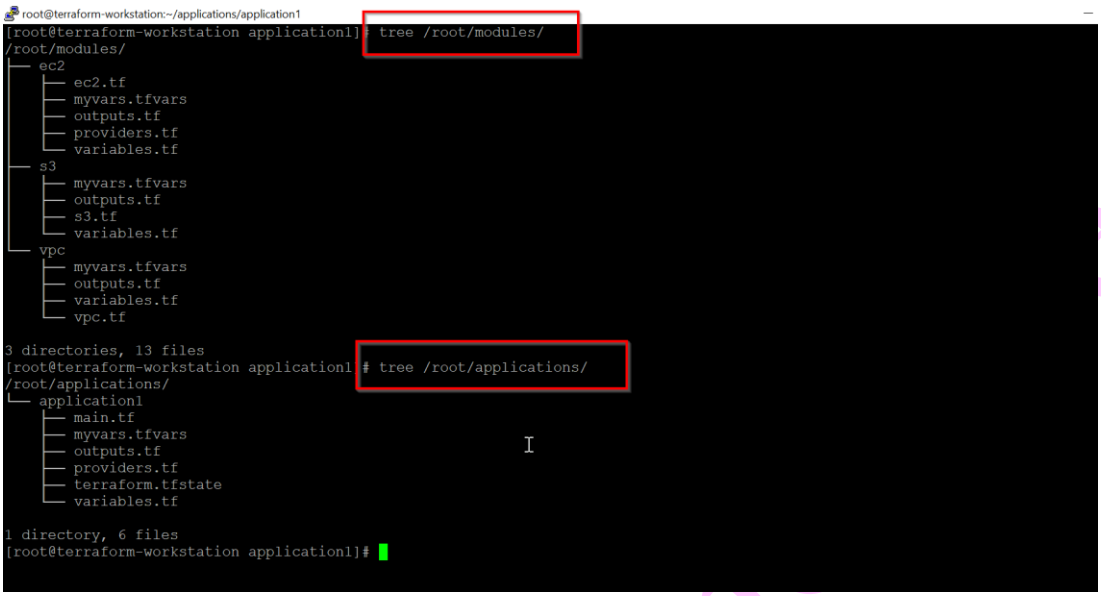
```
module.network.aws_vpc.myvpc: Creating...
module.s3.aws_s3_bucket.mys3: Creating...
module.s3.aws_s3_bucket.mys3: Creation complete after 0s [id=dvsbatch2-2022-xyz]
module.network.aws_vpc.myvpc: Creation complete after 1s [id=vpc-093d7051784226d95]
module.network.aws_security_group.mysecgrp: Creating...
module.network.aws_subnet.mysubl: Creating...
module.network.aws_internet_gateway.myigw: Creating...
module.network.aws_internet_gateway.myigw: Creation complete after 0s [id=igw-03eeced6aa700f368]
module.network.aws_route_table.myroute: Creating...
module.network.aws_subnet.mysubl: Creation complete after 0s [id=subnet-0bdd119d954f47d6a]
module.network.aws_route_table.myroute: Creation complete after 1s [id=rtb-006717babfc95bc45]
module.network.aws_route_table_association.subassociation: Creating...
module.network.aws_route_table_association.subassociation: Creation complete after 0s [id=rtbassoc-02654525dd2d38d48]
module.network.aws_security_group.mysecgrp: Creation complete after 3s [id=sg-03c0e25d2f0d007e9]
module.compute.aws_instance.myec2: Creating...
module.compute.aws_instance.myec2: Still creating... [10s elapsed]
module.compute.aws_instance.myec2: Still creating... [20s elapsed]
module.compute.aws_instance.myec2: Still creating... [30s elapsed]
module.compute.aws_instance.myec2: Creation complete after 32s [id=i-01ed4fc2336d7e99e]

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.
[root@terraform-workstation application1]#
```

Now finally verify the private ipaddress of our server



So finally we are good to go with modules !!!

Our File structure looks like below.



Entire code is available under
https://github.com/shan5a6/aws-terraform.git

10 Working with workspaces

Let's imagine a situation that you want to scale your infrastructure like dev,sit,uat,pt, and so on. Here I don't want to write the same code again & again. Hence we have an option called "workspace", we can reuse the same code as per our requirement. But only one thing we define is the environment and goanna define all the values under that environment.

As per the previous code we have modules and create our infrastructure using the same. Now let's use the same code but we will modify it as per our requirement.

From the previous code let's take only example of "s3" creation

Our old main.tf looks like below:

```
module "s3" {
      source = "/root/modules/s3/"
      bucketname = "dvsbatch2-2022-xyz"
      tags    = {
                  "Env" = "Dev"
```

```
                "Dept" = "DevOps"
        }

}
```

Let's modify the code to support multiple environments.

```
variable "env" {
        type = string
}

locals {
        env = {
                dev = {
                        bucketname = "dvsbatch2-2022-dev"
                        tags = {
                                        "Env" = "Dev"
                                        "Dept" = "DevOps"
                                }
                }
                prod = {
                        bucketname = "dvsbatch2-2022-prod"
                        tags = {
                                        "Env" = "Prod"
                                        "Dept" = "DevOps"
                                }
                }
        }

}
module "s3" {
        source = "/root/modules/s3/"
        bucketname = "${local.env[var.env].bucketname}"
        tags    = "${local.env[var.env].tags}"

}
```

Let's plan the changes for dev & prod and see what happens.

Executing for dev environment:

```
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[root@terraform-workstation application1]# terraform plan -var "env=dev"

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:

  # module.s3.aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "dvsbatch2-2022-dev"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
      + policy                      = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + tags                        = {
          + "Dept" = "DevOps"
          + "Env"  = "Dev"
        }
      + tags_all                    = {
          + "Dept" = "DevOps"
          + "Env"  = "Dev"
        }
      + website_domain              = (known after apply)
      + website_endpoint            = (known after apply)
```

Executing for prod environment:

```
root@terraform-workstation:~/applications/application1
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# terraform plan -var "env=prod"

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:

  # module.s3.aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "dvsbatch2-2022-prod"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
      + policy                      = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + tags                        = {
          + "Dept" = "DevOps"
          + "Env"  = "Prod"
        }
      + tags_all                    = {
          + "Dept" = "DevOps"
          + "Env"  = "Prod"
        }
      + website_domain              = (known after apply)
      + website_endpoint            = (known after apply)

      + cors_rule {
          + allowed_headers = (known after apply)
```

So for dev & prod same code is working. Let's see if this really going to happen by applying the changes.

Execute terraform apply:

```
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# terraform apply -var "env=dev" -auto-approve

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:

  # module.s3.aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "dvsbatch2-2022-dev"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
      + policy                      = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + tags                        = {
          + "Dept" = "DevOps"
          + "Env"  = "Dev"
        }
      + tags_all                    = {
          + "Dept" = "DevOps"
          + "Env"  = "Dev"
        }
      + website_domain              = (known after apply)
```

```
    }

Plan: 1 to add, 0 to change, 0 to destroy.
module.s3.aws_s3_bucket.mys3: Creating...
module.s3.aws_s3_bucket.mys3: Creation complete after 1s [id=dvsbatch2-2022-dev]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@terraform-workstation application1]#
```

It clearly says that 1 to add 0 to change.

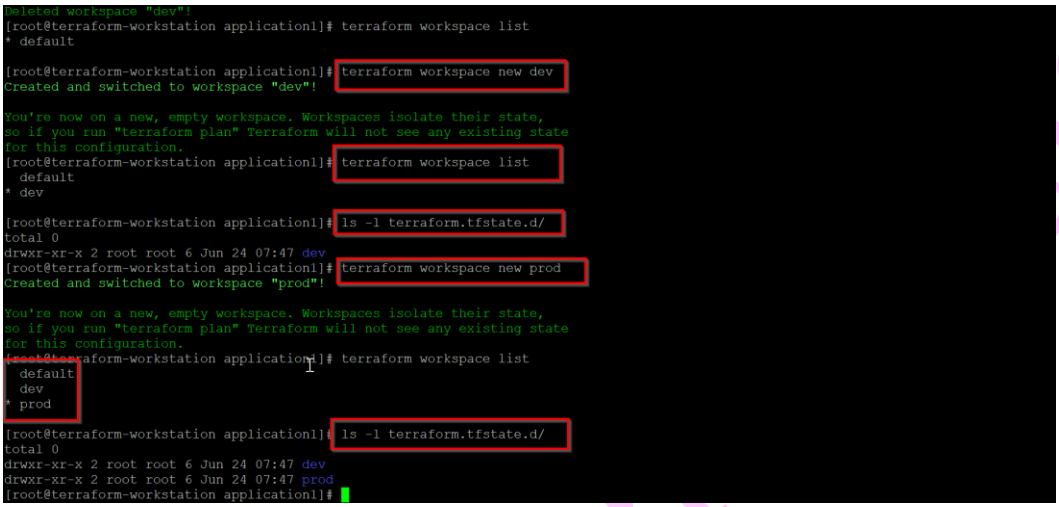
Now let's run the same code for prod.

```
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# terraform apply -var "env=prod"
module.s3.aws_s3_bucket.mys3: Refreshing state... [id=dvsbatch2-2022-dev]

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

Terraform will perform the following actions:

  # module.s3.aws_s3_bucket.mys3 must be replaced
-/+ resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      ~ arn                         = "arn:aws:s3:::dvsbatch2-2022-dev" -> (known after apply)
      ~ bucket                      = "dvsbatch2-2022-dev" -> "dvsbatch2-2022-prod" # forces replacement
      ~ bucket_domain_name          = "dvsbatch2-2022-dev.s3.amazonaws.com" -> (known after apply)
      ~ bucket_regional_domain_name = "dvsbatch2-2022-dev.s3.amazonaws.com" -> (known after apply)
      ~ hosted_zone_id              = "Z3AQBSTGFYJSTF" -> (known after apply)
      ~ id                          = "dvsbatch2-2022-dev" -> (known after apply)
      ~ object_lock_enabled         = false -> (known after apply)
      + policy                      = (known after apply)
      ~ region                      = "us-east-1" -> (known after apply)
      ~ request_payer               = "BucketOwner" -> (known after apply)
      ~ tags                        = {
          ~ "Env"  = "Dev" -> "Prod"
            # (1 unchanged element hidden)
        }
      ~ tags_all                    = {
          ~ "Env"  = "Dev" -> "Prod"
            # (1 unchanged element hidden)
        }
      + website_domain              = (known after apply)
      + website_endpoint            = (known after apply)
      # (1 unchanged attribute hidden)
```

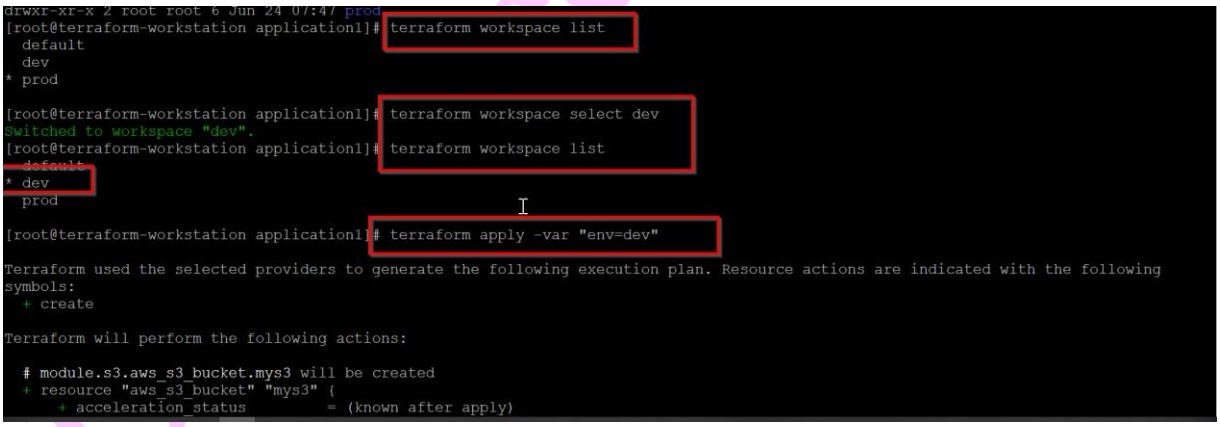
Here it clearly says that it's going to recreate the environment from dev to prod ??? Why this behaviour ???

This is happening because we have only one tfstate file in the local so in order to achieve this we are going to use **workspace**.

Creating workspace for dev & prod:



Apply the changes to dev workspace:



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

Do you want to perform these actions in workspace "dev"?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

module.s3.aws_s3_bucket.mys3: Creating...
module.s3.aws_s3_bucket.mys3: Creation complete after 1s [id=dvsbatch2-2022-dev]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@terraform-workstation application1]# ls -l terraform.tfstate/dev
ls: cannot access terraform.tfstate/dev: Not a directory
[root@terraform-workstation application1]# ls -l terraform.tfstate.d/dev/terraform.tfstate
-rw-r--r-- 1 root root 2149 Jun 24 07:50 terraform.tfstate.d/dev/terraform.tfstate
[root@terraform-workstation application1]#
```

Apply the changes to prod workspace:

```
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# terraform workspace list
  default
* dev
  prod

[root@terraform-workstation application1]# terraform workspace select prod
Switched to workspace "prod".
[root@terraform-workstation application1]# terraform workspace list
  default
  dev
* prod

[root@terraform-workstation application1]# terraform apply -var "env=prod"

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:

  # module.s3.aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "dvsbatch2-2022-prod"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
```

```
[root@terraform-workstation application1]# cat main.tf
variable "env" {
        type = string
}

locals {
        env = {
                dev = {
                        bucketname = "dvsbatch2-2022-dev"
                        tags = {
                                        "Env" = "Dev"
                                        "Dept" = "DevOps"
                                }
                }
                prod = {
                        bucketname = "dvsbatch2-2022-prod"
                        tags = {
                                        "Env" = "Prod"
                                        "Dept" = "DevOps"
                                }
                }
        }

}
module "s3" {
        source = "/root/modules/s3/"
        bucketname = "${local.env[var.env].bucketname}"
        tags    = "${local.env[var.env].tags}"

}
[root@terraform-workstation application1]# tree terraform.tfstate.d/
terraform.tfstate.d/
├── dev
│   └── terraform.tfstate
└── prod
    └── terraform.tfstate
```

So finally its same code but with two diff state files per environment. Hence we are getting two diff buckets with different names

```
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# aws s3 ls|grep -e "dev\|prod"
2022-06-24 07:40:20 dvsbatch2-2022-dev
2022-06-24 02:24:23 dvsbatch2-2022-prod
2022-05-25 02:11:15 s3-shaandevops-live
[root@terraform-workstation application1]#
```

11 Protect state file

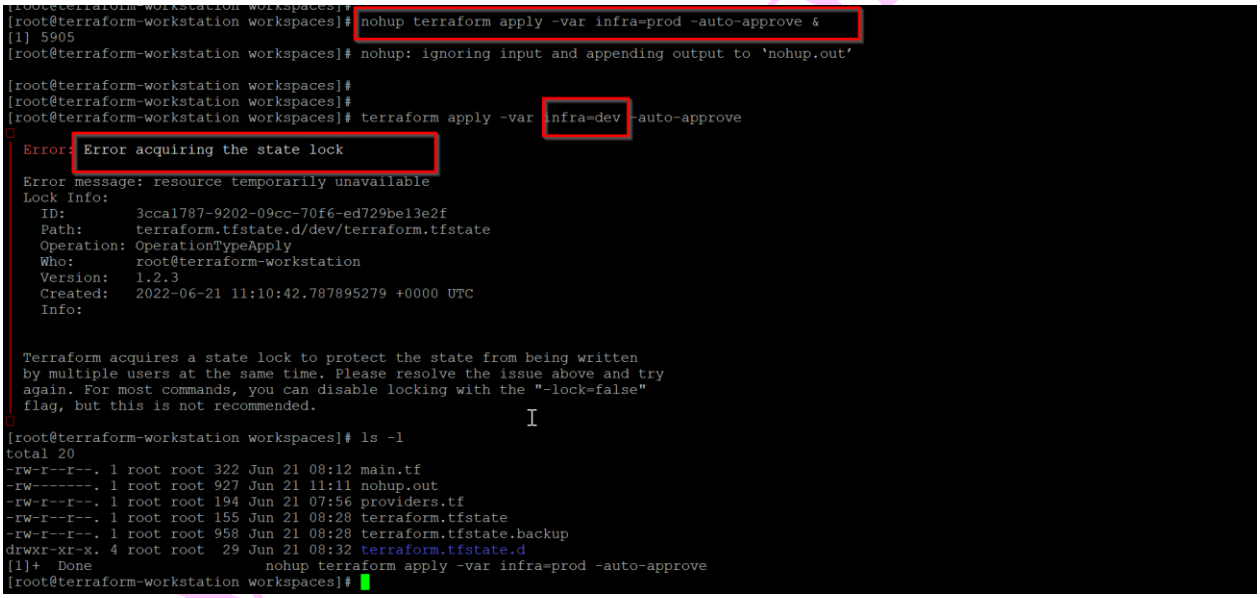
Since we are completely depending on our state file for our infrastructure, just imagine what happens if we lose that file ???

Hence it's our primary responsibility to safe gurad the file hence we have to keep it safe and in common location.

Other common problem is imagine a situation that you and your college is applying the changes to the infrastructure at the same time then ??

In order to avoid such kind of issues we have to make sure that we are keeping state file in common and safe location along with some locking mechanism. So that only one guy can do the changes, not everyone.

Issue:

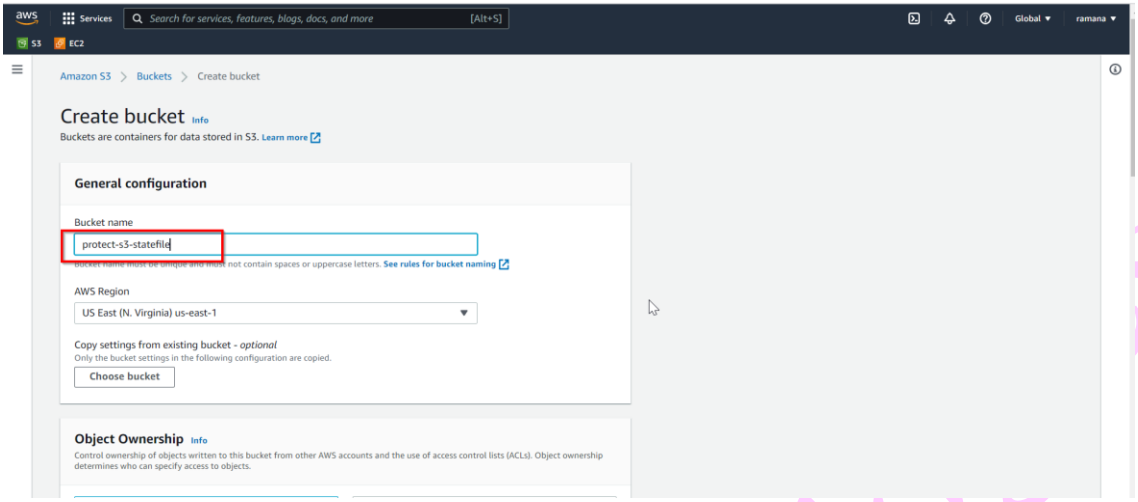


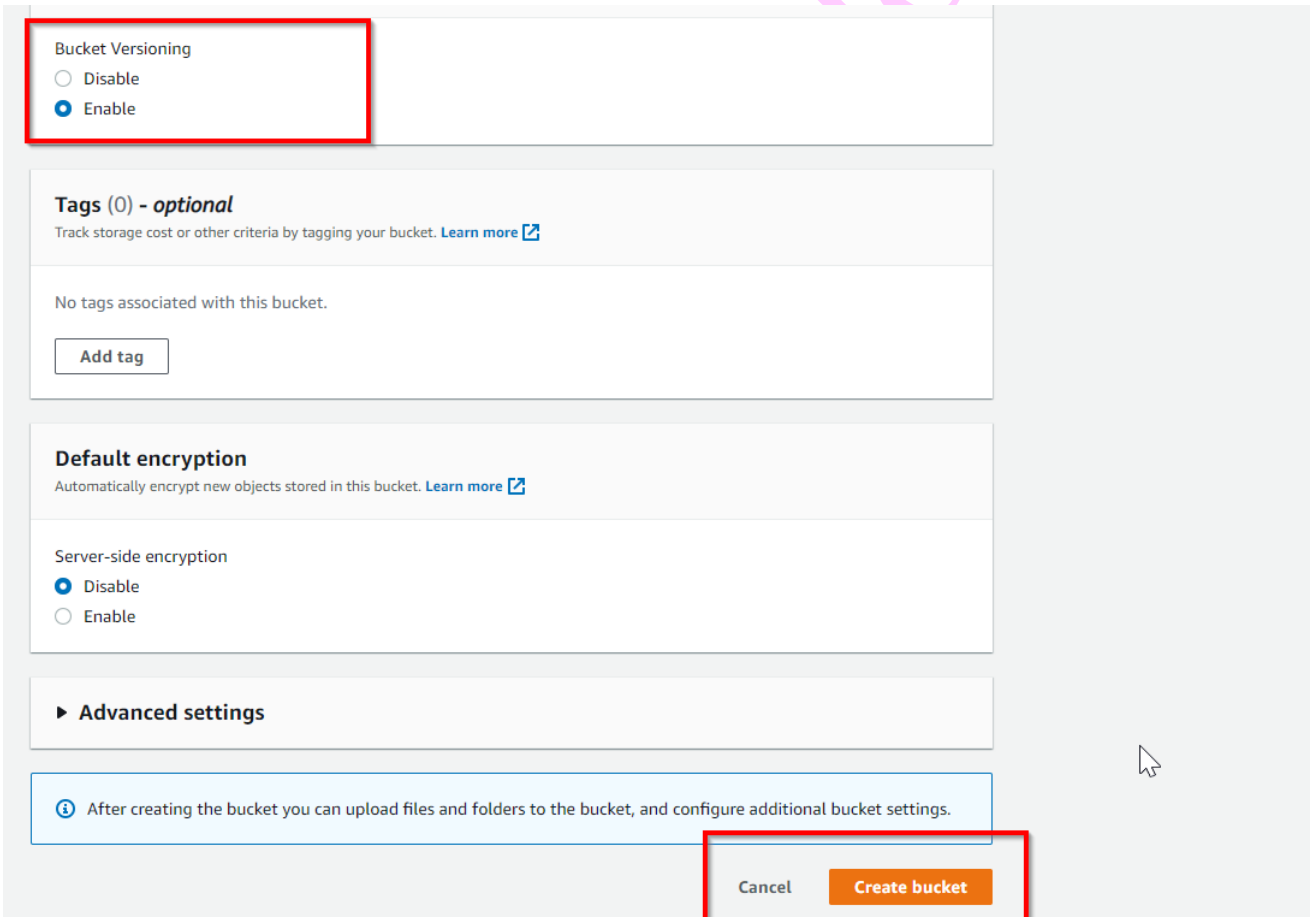
Here we try to build the infra in different environments hence our state file is locked sometimes we may corrupt the entire infrastructure.

Solution:

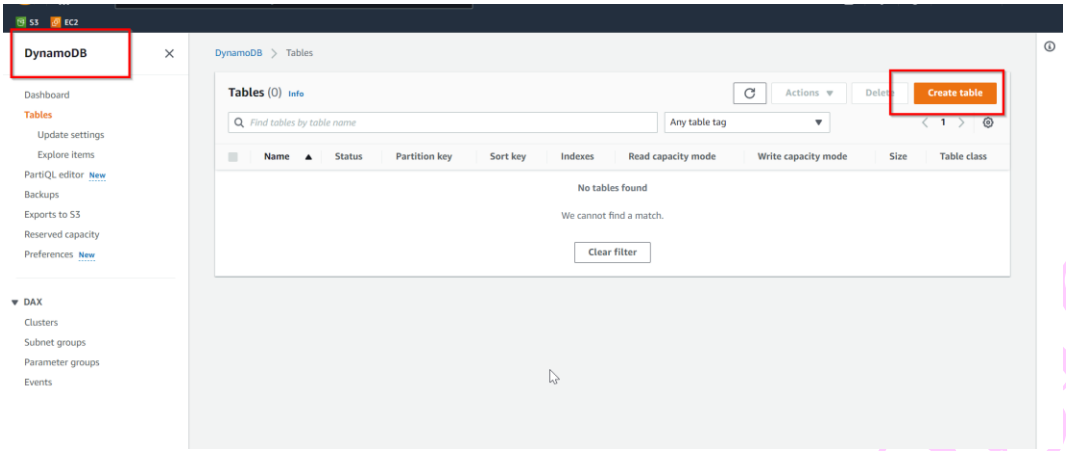
We have to make sure that we are providing the backend for the state file as "s3" & for lock we are going to use "dynamodb" along with versioning enabled for S3 bucket.

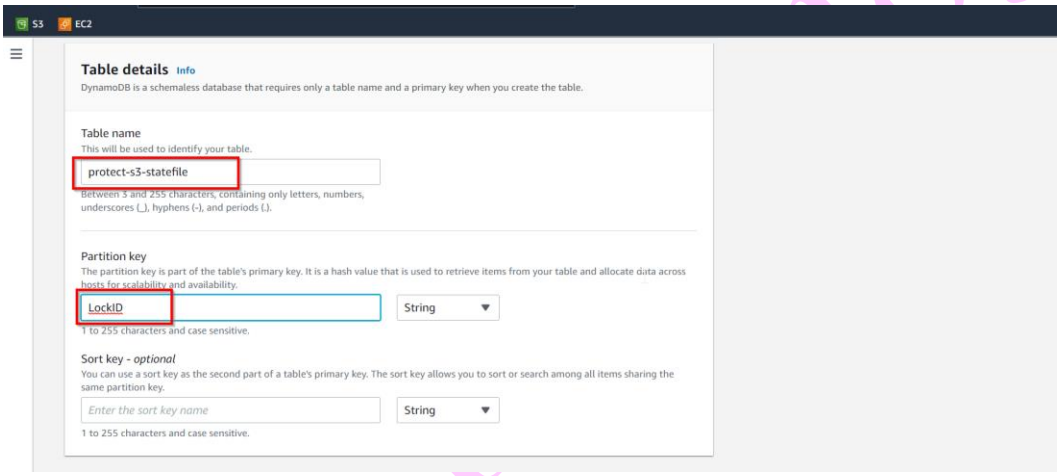
Create one S3 bucket:


Amazon S3 > Buckets > Create bucket
Create bucket Info
Buckets are containers for data stored in S3. Learn more
General configuration
Bucket name
protect-s3-statefile
AWS Region
US East (N. Virginia) us-east-1
Copy settings from existing bucket - optional
Only the bucket settings in the following configuration are copied.
Choose bucket
Object Ownership Info
Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

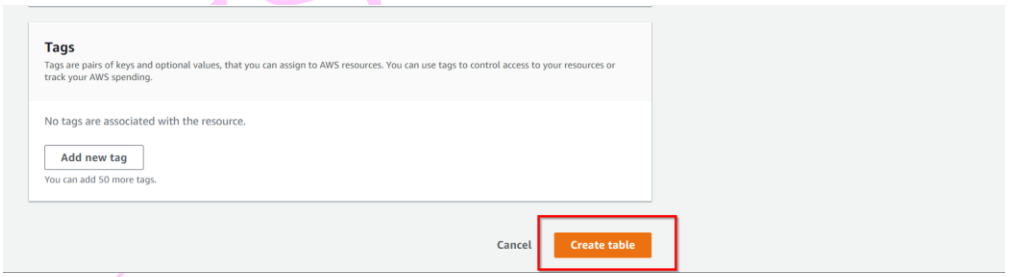

Bucket Versioning
Disable
Enable
Tags (0) - optional
Track storage cost or other criteria by tagging your bucket. Learn more
No tags associated with this bucket.
Add tag
Default encryption
Automatically encrypt new objects stored in this bucket. Learn more
Server-side encryption
Disable
Enable
Advanced settings
After creating the bucket you can upload files and folders to the bucket, and configure additional bucket settings.
Cancel
Create bucket

Create one dynamodb table:





Note: It should be "LockID" mandatory



Now we have to mention backend as "s3" along with "dynamodb'. Let's create the file with below content.

S3statefilebackup.tf

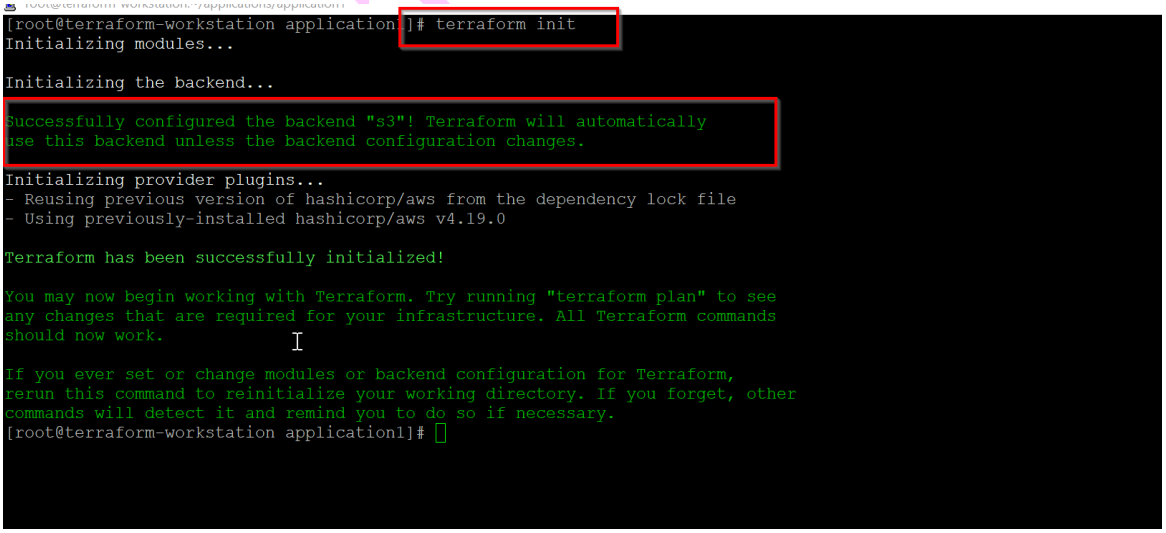
Example:

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

s3statebackup.tf

```
terraform {
  backend "s3" {
    bucket = "protect-s3-statefile"
    key    = "terraform.tfstate"
    region = "us-east-1"
    dynamodb_table = "protect-s3-statefile"
  }
}
```

Once you add this file we need to initialize the terraform



Now it says S3 backend initialized. If in case if you already have initialized earlier then make sure that you are issuing "-reconfigure"

terraform init -reconfigure

Now let's apply the changes & see what happens.

```
[root@terraform-workstation application1]# cat s3statebackup.tf
terraform {
  backend "s3" {
    bucket = "protect-s3-statefile"
    key    = "terraform.tfstate"
    region = "us-east-1"
    dynamodb_table = "protect-s3-statefile"
  }
}
[root@terraform-workstation application1]# terraform apply -var "env=dev" -auto-approve

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:

  # module.s3.aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "dvsbatch2-2022-dev"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
      + policy                      = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + routing_rules               = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
module.s3.aws_s3_bucket.mys3: Creating...
module.s3.aws_s3_bucket.mys3: Creation complete after 1s [id=dvsbatch2-2022-dev]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@terraform-workstation application1]#
[root@terraform-workstation application1]# cat terraform.tfstate
{
  "version": 4,
  "terraform_version": "1.2.3",
  "serial": 21,
  "lineage": "60a3fdef-0d86-4cd4-7ba4-bf3925b1652d",
  "outputs": {},
  "resources": []
}
[root@terraform-workstation application1]#
```

Since we have configured our terraform to use backend as S3 its placing statefile in the S3 bucket hence we don't see the state file content in local system.

But if we observe the same in S3 bucket we should see the content.