TERRAFORM

History:

Terraform is an open source IaC (Infrastructure as Code) tool created by HashiCorp. It was developed by Mitchell Hashimoto with Go language in the year 2014.

What is Terraform?

Terraform is an open source "Infrastructure as a Code (IaaC)" tool created by HashiCorp. This tool is used to automate the creation of an entire infrastructure setup for an application like servers, networks, databases, VPC etc. All the configuration files that are used to write a code are written using a declarative configuration language known as a HashiCorp Configuration Language (HCL) or optionally JSON. It uses a simple syntax, it's a Cloud agnostic which doesn't depends on single provider and can provision infrastructure for multiple clouds and On premises. This automation is based on a file (should end with **'.tf'**) that we write and execute for setting up the infrastructure in an automated way instead of a manual process.

Why Terraform?

Let's say we're working on a project and if a new project comes in and instead of writing all the code for the infrastructure setup, we can just reuse the code by changing the requirements as per the project. Here there's a reusability with Terraform.

Even if we've to create or delete a server at a particular as per the requirement without doing it manually we can just write a code and the job will be done. Also, for example if we've to change the instance type from t2.micro to t2.medium/large we need to stop the instance manually and then change the type. However, with the help of terraform we can just make changes in the code and the changes will be made.

What is IAAC?

IAAC refers to Infrastructure as a Code. It's a practice in DevOps that involves managing and provisioning infrastructure resources using code and automation.

- By using IaaC we can automate the creation of infrastructure instead of a manual process.
- Server automation (ec2) and configuration management tools can be often used to achieve IaaC.
- IaaC brings the principles of software development to infrastructure management allowing for more streamlined and agile operations (with just writing the code, the entire setup can be created which is as similar to a developer writes a code to get a vision into reality)
- IaaC tools such as Terraform or Ansible are used to write scripts or playbooks that automatically create, configure, and manage your infrastructure in a consistent and repeatable manner.

Alternatives of IaaC tools:

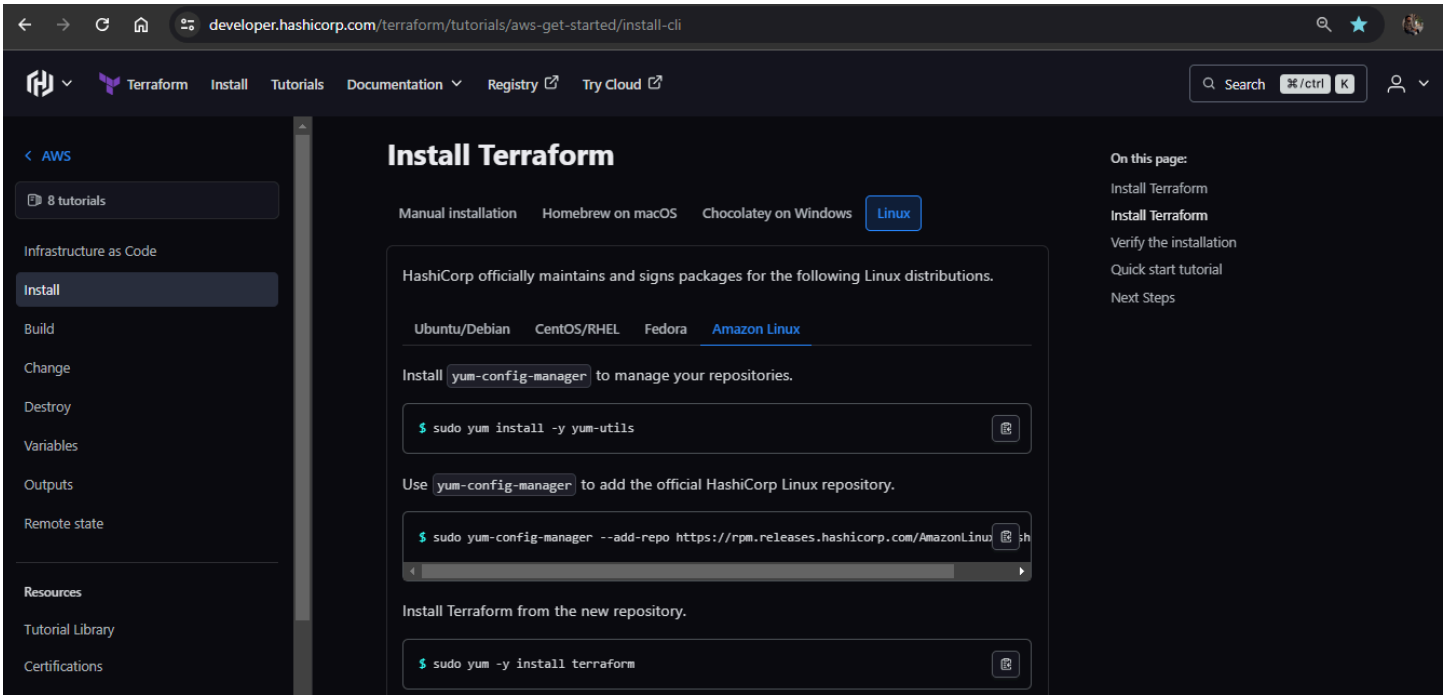
1) AWS - - CFT (Cloud formation template) (JSON/YAML)
2) Azure - - ARM templates (JSON)

3) GCP - - Cloud Deployment Manager (YAML/PYTHON)
4) Pulumi - - (Python, JS, C#, GO and Type script)
5) Ansible - - (YAML)
6) Puppet
7) Chef
8) Vagrant
9) Crossplane

Installation of Terraform:

1) Go to official documentation of terraform and select the operating system that matches the instance to install the correct version of terraform.

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli



Syntax of a Terraform code: The file name of the code should end with **'.tf'** (means terraform files) only.

There are three important sections in the code. Provider, Resource and Variables.

1) Provider: This section is used to define in which cloud provider, which account and in which region you want to create the infrastructure setup.

2) Resource: This is used to define what you want to create like EC2, S3 etc.

3) Variables: This is used to define the variables.

TYPE: 1

1) The below image represents a basic terraform code to create an instance in an automated way.

Correction: tags = {

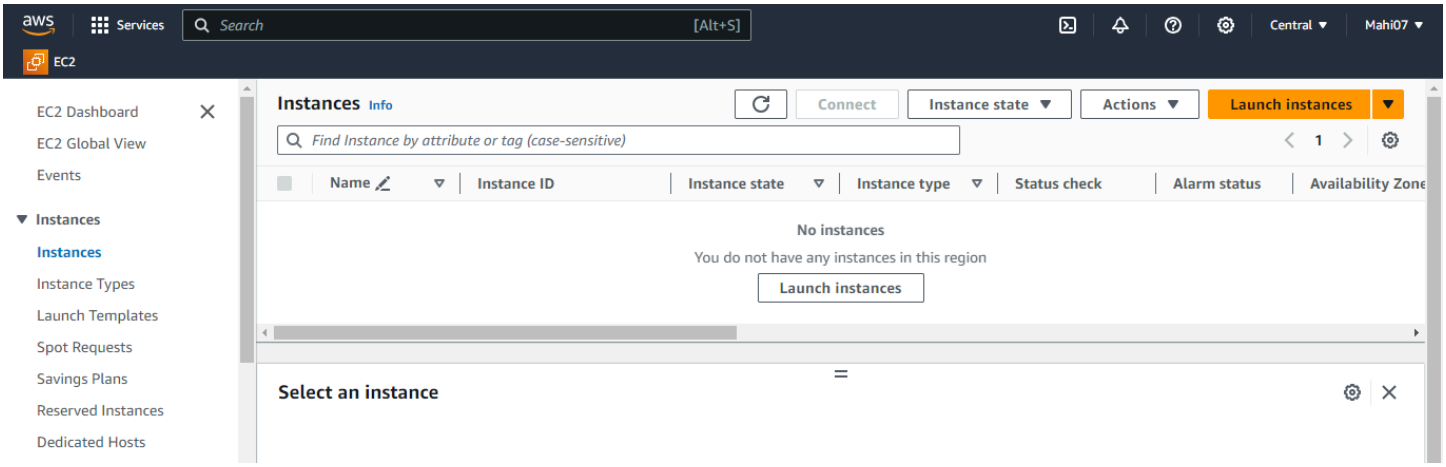
```
provider "aws" {
region = "ca-central-1"
access_key = "AKIA365MTWIRUSIHHIAZ"
secret_key = "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"
}

resource "aws_instance" "rawan" {
tags {
Name = "terra-inst"
Environment = "Prod"
Project = "ONDC"
}
ami = "ami-095819c19b51bc983"
instance_type = "t2.micro"
key_name = "master-slave"
availability_zone = "ca-central-1b"
}
```

The provider section from the above code demonstrates that, '**aws**' is the cloud provider, region defines the region where we are going to create our resource. In the above code we're creating an ec2 instance. The instance is getting created under an account created through IAM with AmazonEC2Full Access with the help of access and secret key that we got from the aws console.

The resource section defines that we're creating an instance (**aws_instance**) with a tag to differentiate as '**rawan**'. The instance name is given as '**terra-inst**', the environment where the instance has to be created to filter out, project name for which the instance is creating. Moreover, **'ami'** defines the ami-id of the region where we're creating the instance and these ami-id's are unique from region to region. The instance type is '**t2.micro**', the keypair name is '**master-slave**' and the availability zone in ca-central-1 is '**ca-central-1b**'.

If you do not mention the security group and keypair value in the code then it will automatically takes the default security group and without keypair instance will be created.

Here we do not have any instance in ca-central-1 (Canada) region. After the code is written, we need to execute it and then refresh the aws console then you can be able to see that an instance is created.



2) To execute the code, first we need to **'init the terraform', 'terraform plan'** and then **'terraform apply'**.

The core Terraform workflow consists of three main steps after you have written your Terraform configuration:

Initialize prepares your workspace so Terraform can apply your configuration.

Plan allows you to preview the changes Terraform will make before you apply them.

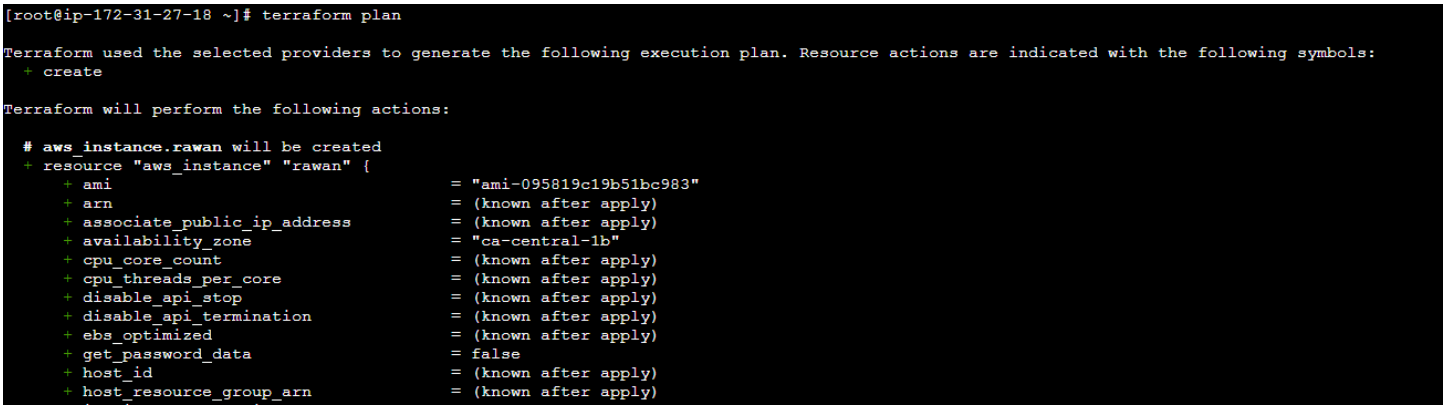
Apply makes the changes defined by your plan to create, update, or destroy resources.

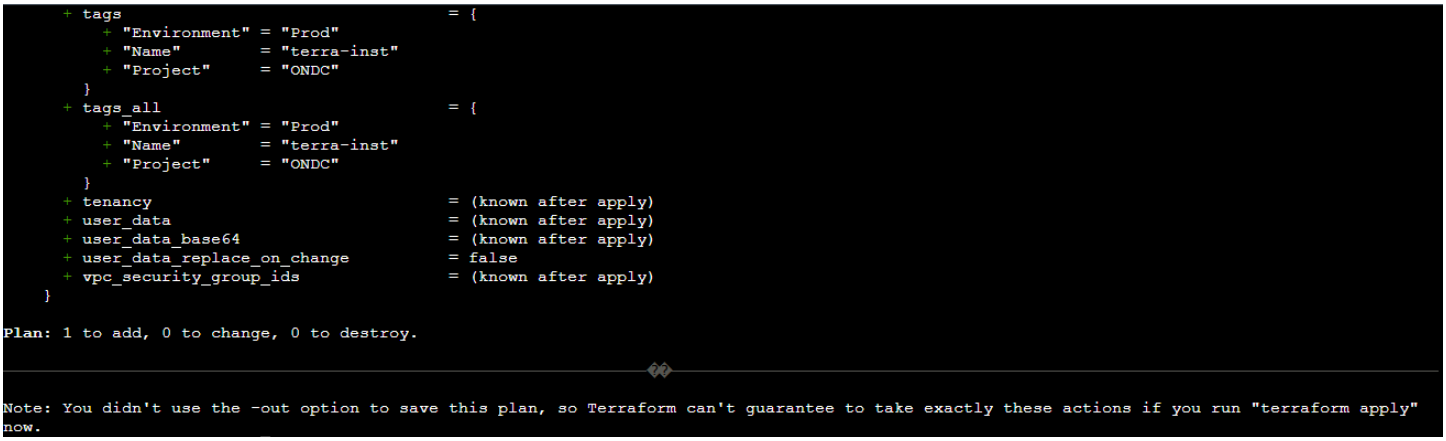
2 (a) Terraform init

```
[root@ip-172-31-27-18 ~]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.30.0...
- Installed hashicorp/aws v5.30.0 (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.
```

After you give terraform plan, you'll get all the information that you mentioned in the code to create and the overall changes that are getting made like how many resources are getting added, changes made, and number of things getting destroyed will be given at the end.

2 (b) Terraform plan shows the preview of action that will be performed or task that's get executed.





So as per the above image, one resource would be added as per the terraform plan, 0 to change and 0 to destroy.

Plan: 1 to add, 0 to change, 0 to destroy, these things would be applied after it is executed.

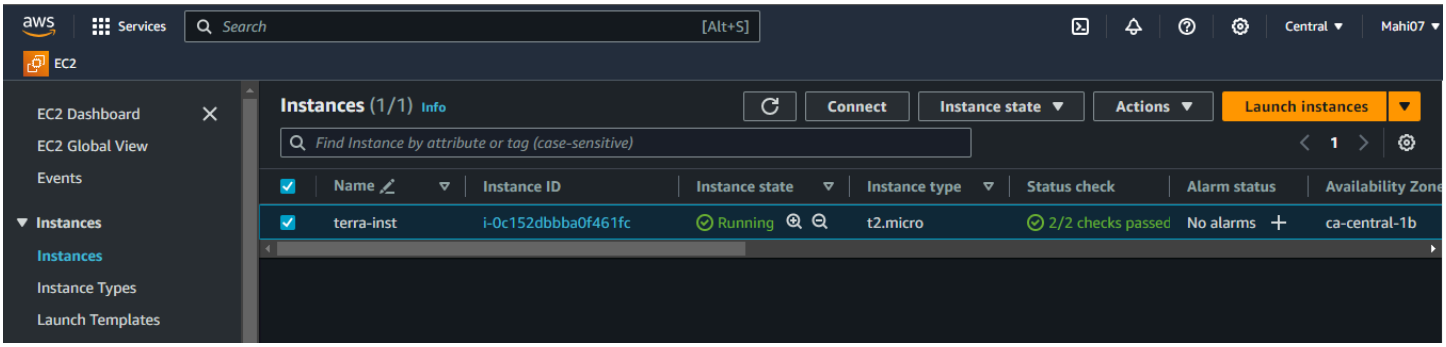
2 (c) After giving the command as "t**erraform apply**", it'll execute the preview that shown after the terraform plan. Here we need to enter a value as 'yes' for the plan to approve and execute.

```
      + tags                                 = {
          + "Environment" = "Prod"
          + "Name"        = "terra-inst"
          + "Project"     = "ONDC"
        }
      + tags_all                             = {
          + "Environment" = "Prod"
          + "Name"        = "terra-inst"
          + "Project"     = "ONDC"
        }
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (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
```

After entering the value as yes we'll get the logs about the status of creation of an ec2 instance and once the instance is created we get an instance id as shown below. Resources : 1 added means one instance is created, 0 changed and 0 destroyed. Now, you can go to the aws console and refresh the page to check if an instance is created as per the code.

```
  Enter a value: yes

aws_instance.rawan: Creating...
aws_instance.rawan: Still creating... [10s elapsed]
aws_instance.rawan: Still creating... [20s elapsed]
aws_instance.rawan: Creation complete after 21s [id=i-0c152dbbba0f461fc]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@ip-172-31-27-18 ~]#
```

3) Here is the final result of the terraform code.



The above image represents that, an instance is created through automation with the help of a terraform code. This is how Terraform is used as a 'IaaC' tool.

TYPE - 2:

If you want to add a few more instances, you can just add it's '**count**' by mentioning in the same code.

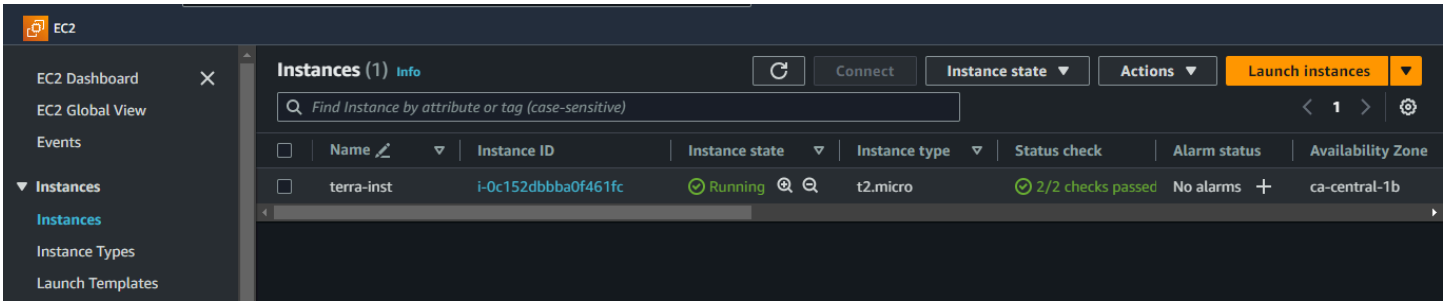
1) Terraform code

```
provider "aws" {
region = "ca-central-1"
access_key = "AKIA365MTWIRUSIHHIAZ"
secret_key = "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"
}

resource "aws_instance" "rawan" {
tags = {
Name = "terra-inst"
Environment = "Prod"
Project = "ONDC"
}
ami = "ami-095819c19b51bc983"
instance_type = "t2.micro"
count = 3
availability_zone = "ca-central-1b"
```

As we already have one instance running above with the code, so another two instances will be created in the same region and same availability zone.

2) Here we don't have any instances as of now. Once we give terraform plan and terraform apply we get to see two more instances.

2 (a) Before the execution of the terraform code we only have one instance.



2 (b) Terraform plan

```
Plan: 2 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@ip-172-31-27-18 ~]#
```

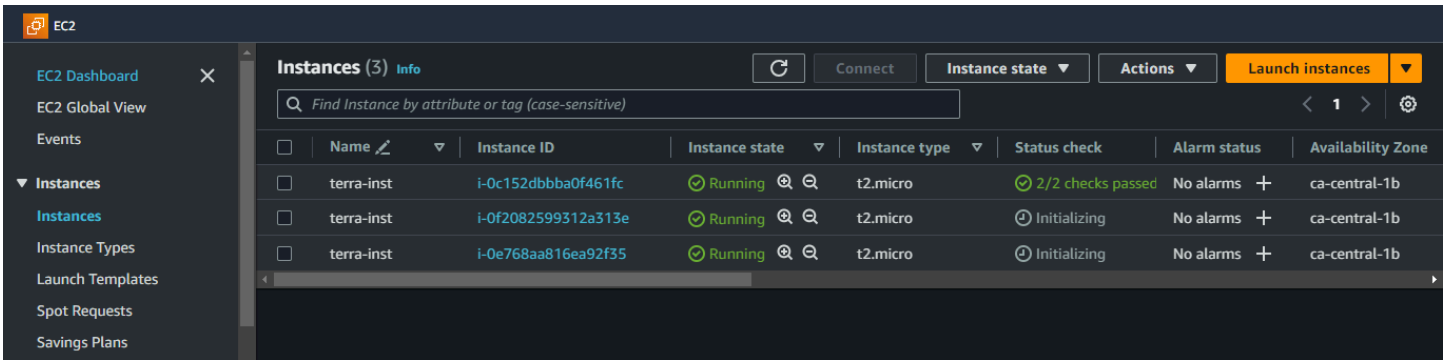
Plan: 2 to add, 0 to change and 0 to destroy, which means, 2 resources are getting added (instances), 0 to change and 0 to destroy.

2 (c) Terraform apply

```
  Enter a value: yes

aws_instance.rawan[2]: Creating...
aws_instance.rawan[1]: Creating...
aws_instance.rawan[2]: Still creating... [10s elapsed]
aws_instance.rawan[1]: Still creating... [10s elapsed]
aws_instance.rawan[2]: Still creating... [20s elapsed]
aws_instance.rawan[1]: Still creating... [20s elapsed]
aws_instance.rawan[2]: Still creating... [30s elapsed]
aws_instance.rawan[1]: Still creating... [30s elapsed]
aws_instance.rawan[1]: Creation complete after 32s [id=i-0f2082599312a313e]
aws_instance.rawan[2]: Creation complete after 32s [id=i-0e768aa816ea92f35]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
[root@ip-172-31-27-18 ~]#
```

Resources: 2 added (instances are created), 0 changed and 0 destroyed.

2 (d) We got two more instances launched as per the changes made in the terraform code.



TYPE - 3:

If you want to change the name, you can make changes in the code and again, plan it and apply it and the changes will be made. Refer to images for a better understanding.

a) New name '**salaar-inst**' is mentioned in the code in place of **'terra-inst'**.

```
provider "aws" {
region = "ca-central-1"
access_key = "AKIA365MTWIRUSIHHIAZ"
secret_key = "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"
}

resource "aws_instance" "rawan" {
tags = {
Name = "salaar-inst"
Environment = "Prod"
Project = "ONDC"
}
ami = "ami-095819c19b51bc983"
instance_type = "t2.micro"
count = 3
availability_zone = "ca-central-1b"
}
```

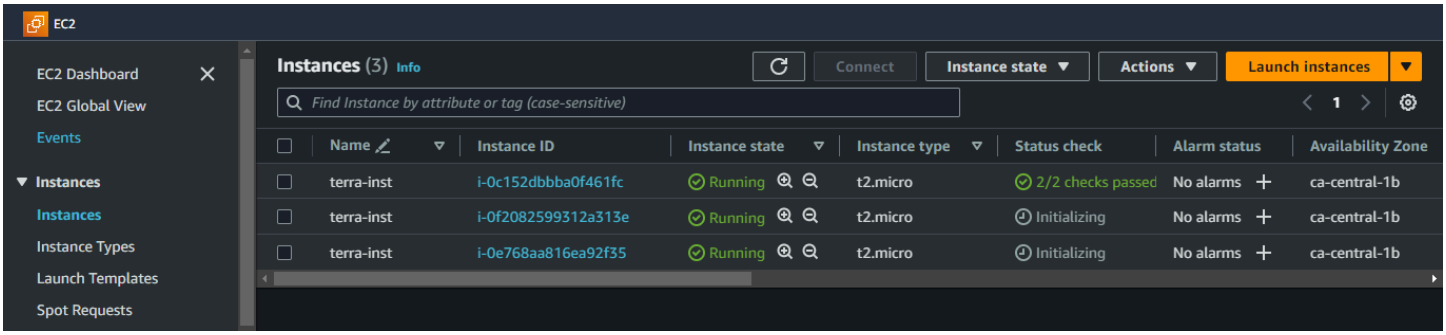
b) The name is getting changed from **'terra-inst' to 'salaar-inst'**.

```
  # aws_instance.rawan[2] will be updated in-place
  ~ resource "aws_instance" "rawan" {
        id                                   = "i-0e768aa816ea92f35"
      ~ tags                                 = {
            "Environment" = "Prod"
          ~ "Name"        = "terra-inst" -> "salaar-inst"
            "Project"     = "ONDC"
        }
      ~ tags_all                             = {
          ~ "Name"        = "terra-inst" -> "salaar-inst"
            # (2 unchanged elements hidden)
        }
        # (30 unchanged attributes hidden)

        # (8 unchanged blocks hidden)
    }

Plan: 0 to add, 3 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.
```

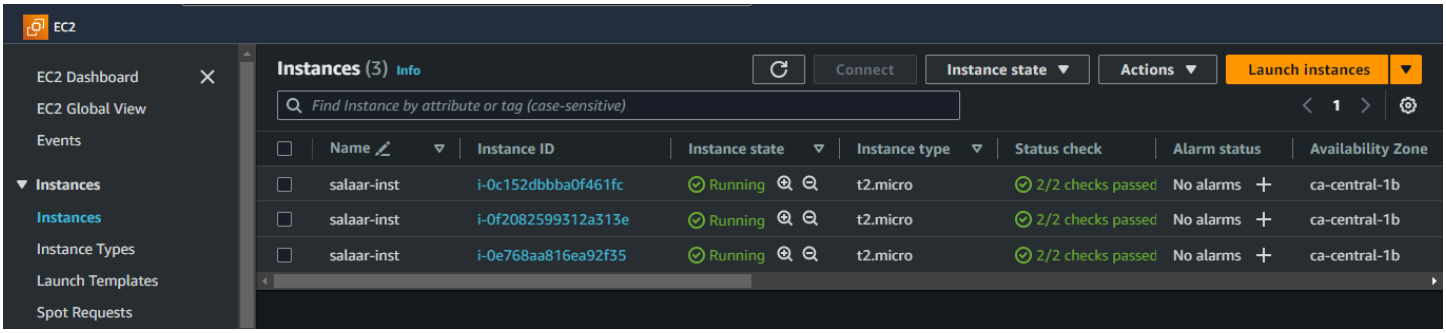
c) The changes are mentioned. As there are 2 instances, so 3 name changes have been made.

```
  Enter a value: yes

aws_instance.rawan[1]: Modifying... [id=i-0f2082599312a313e]
aws_instance.rawan[2]: Modifying... [id=i-0e768aa816ea92f35]
aws_instance.rawan[0]: Modifying... [id=i-0c152dbbba0f461fc]
aws_instance.rawan[0]: Modifications complete after 1s [id=i-0c152dbbba0f461fc]
aws_instance.rawan[1]: Modifications complete after 1s [id=i-0f2082599312a313e]
aws_instance.rawan[2]: Modifications complete after 2s [id=i-0e768aa816ea92f35]

Apply complete! Resources: 0 added, 3 changed, 0 destroyed.
[root@ip-172-31-27-18 ~]#
```

d) Before refreshing the aws console and after applying the actions with terraform apply.



e) After refreshing the aws console.



Changing the instance type:

If you want to change the instance type, like for example from t2.micro to t2.medium, you can mention the same in the terraform code and then execute it. First the instance state changes to stopping - - > stopped - - > instance type gets changed and then the instance state changes from pending - - > running.

Deleting a specific instance from a group of instances:

If you want to delete a specific instance, you can't do it with the help of code because we can only just mention the count and the terraform doesn't know which specific one to delete. Hence, we go with the CLI

so that a particular instance can be destroyed or deleted.

Since we mentioned the label as ‘rawan’ and launched multiple instances so the instances will be named as shown below.

1) aws_instance.rawan[0]
2) aws_instance.rawan[1]
3) aws_instance.rawan[2]

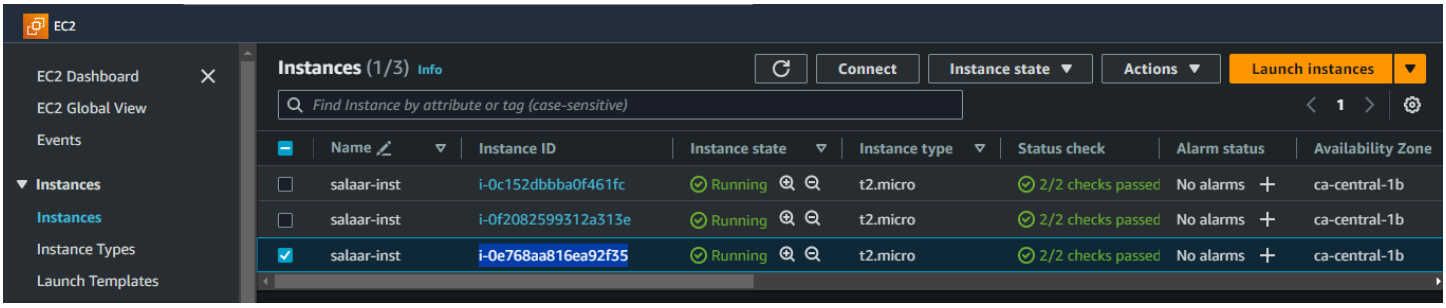
If you want to delete the 3rd instance, we need to select the index value of the third instance.

Command Syntax: terraform destroy --target=aws_instance.labelname[index value of the particular instance]

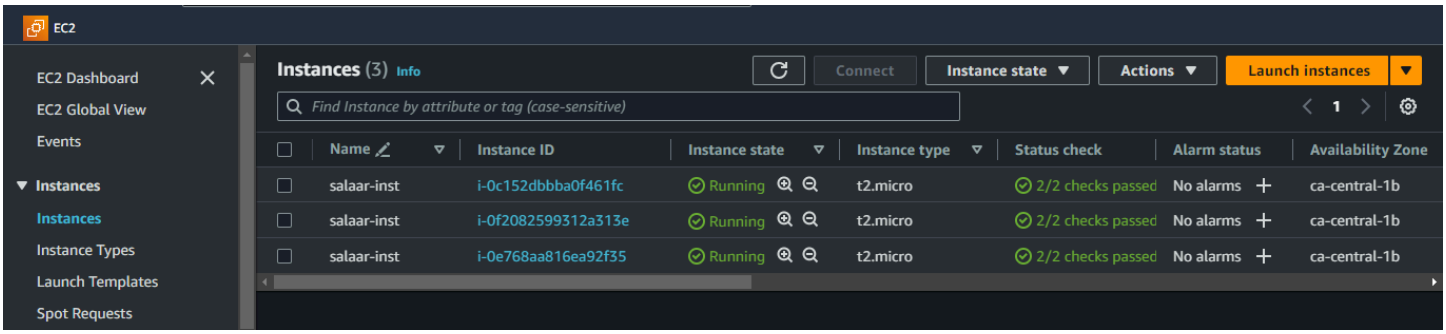
terraform destroy --target=aws_instance.rawan[2]

a) The instance that’s going to be deleted.

```
# aws_instance.rawan[2] will be updated in-place
~ resource "aws_instance" "rawan" {
      id                                   = "i-0e768aa816ea92f35"
    ~ tags                                 = {
        "Environment" = "Prod"
      ~ "Name"        = "terra-inst" -> "salaar-inst"
        "Project"     = "ONDC"
    }
    ~ tags_all                             = {
      ~ "Name"        = "terra-inst" -> "salaar-inst"
        # (2 unchanged elements hidden)
    }
      # (30 unchanged attributes hidden)
```



b) Before deletion, we’ve all three instances.



c) Once we give the command terraform destroy --target=aws_instance.rawan[2] then the below process undergoes before it is deleted.

```
aws_instance.rawan[2]: Refreshing state... [id=i-0e768aa816ea92f35]

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_instance.rawan[2] will be destroyed
  - resource "aws_instance" "rawan" {
      - ami                                  = "ami-095819c19b51bc983" -> null
      - arn                                  = "arn:aws:ec2:ca-central-1:822311367203:instance/i-0e768aa816ea92f35" -> null
      - associate_public_ip_address          = true -> null
      - availability_zone                    = "ca-central-1b" -> null
      - cpu_core_count                       = 1 -> null
      - cpu_threads_per_core                 = 1 -> null
      - disable_api_stop                     = false -> null
      - disable_api_termination              = false -> null
      - ebs_optimized                        = false -> null
```

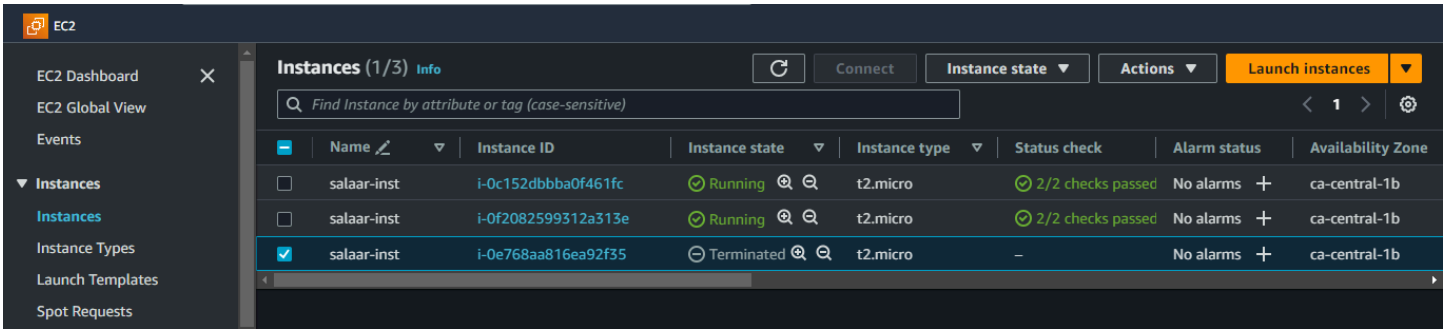
The below image shows the plan of action before getting destroyed that is, Plan: 0 to add, 0 to change, 1 to destroy.

```
          - throughput           = 125 -> null
          - volume_id            = "vol-0b3cd6736cd83ddf3" -> null
          - volume_size          = 8 -> null
          - volume_type          = "gp3" -> null
        }
    }

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

  Warning: Resource targeting is in effect

  You are creating a plan with the -target option, which means that the result of this plan may not represent all of the changes requested by
  the current configuration.

  The -target option is not for routine use, and is provided only for exceptional situations such as recovering from errors or mistakes, or when
  Terraform specifically suggests to use it as part of an error message.


Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

The below image shows that 1 resource is destroyed (one instance is deleted) after the value is entered as yes.

```
  Enter a value: yes

aws_instance.rawan[2]: Destroying... [id=i-0e768aa816ea92f35]
aws_instance.rawan[2]: Still destroying... [id=i-0e768aa816ea92f35, 10s elapsed]
aws_instance.rawan[2]: Still destroying... [id=i-0e768aa816ea92f35, 20s elapsed]
aws_instance.rawan[2]: Destruction complete after 30s

  Warning: Applied changes may be incomplete

  The plan was created with the -target option in effect, so some changes requested in the configuration may have been ignored and the output
  values may not be fully updated. Run the following command to verify that no other changes are pending:
      terraform plan

  Note that the -target option is not suitable for routine use, and is provided only for exceptional situations such as recovering from errors
  or mistakes, or when Terraform specifically suggests to use it as part of an error message.


Destroy complete! Resources: 1 destroyed.
[root@ip-172-31-27-18 ~]#
```

d) After deletion, we've only two except the last one with instance id: i-0e768aa816ea92f35.



The instance index value with label we mentioned in the CLI is terminated and the state changes from stopping - - > stopped - - > terminated.

Deleting a specific instance from a group of instances by not entering the value as yes:

If you want to delete a specific instance without approving the action all the time by entering yes after plan, we need to mention auto approve in the command.

Command Syntax: terraform destroy --auto-approve --target=aws_instance.labelname[index value of the particular instance]

terraform destroy --auto-approve --target=aws_instance.rawan[1]

a) First page of destruction of resource (instance).

```
[root@ip-172-31-27-18 ~]# terraform destroy --auto-approve --target=aws_instance.rawan[1]
aws_instance.rawan[1]: Refreshing state... [id=i-0f2082599312a313e]

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_instance.rawan[1] will be destroyed
  - resource "aws_instance" "rawan" {
      - ami                                  = "ami-095819c19b51bc983" -> null
      - arn                                  = "arn:aws:ec2:ca-central-1:822311367203:instance/i-0f2082599312a313e" -> null
      - associate_public_ip_address          = true -> null
      - availability_zone                    = "ca-central-1b" -> null
      - cpu_core_count                       = 1 -> null
      - cpu_threads_per_core                 = 1 -> null
      - disable_api_stop                     = false -> null
      - disable_api_termination              = false -> null
      - ebs_optimized                        = false -> null
      - get_password_data                    = false -> null
      - hibernation                          = false -> null
```

b) No value is asked to enter to approve the action to delete.

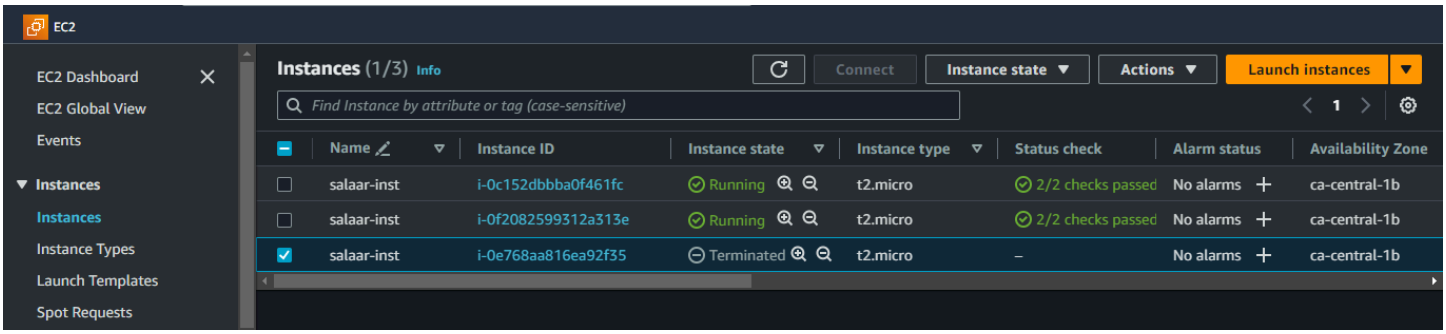
```
EC2
          - throughput           = 125 -> null
          - volume_id            = "vol-06db9ba1ba86477e8" -> null
          - volume_size          = 8 -> null
          - volume_type          = "gp3" -> null
        }
    }

Plan: 0 to add, 0 to change, 1 to destroy.
aws_instance.rawan[1]: Destroying... [id=i-0f2082599312a313e]
aws_instance.rawan[1]: Still destroying... [id=i-0f2082599312a313e, 10s elapsed]
aws_instance.rawan[1]: Still destroying... [id=i-0f2082599312a313e, 20s elapsed]
aws_instance.rawan[1]: Still destroying... [id=i-0f2082599312a313e, 30s elapsed]
aws_instance.rawan[1]: Still destroying... [id=i-0f2082599312a313e, 40s elapsed]
aws_instance.rawan[1]: Destruction complete after 40s

Warning: Resource targeting is in effect

You are creating a plan with the -target option, which means that the result of this plan may not represent all of the changes requested by
the current configuration.

The -target option is not for routine use, and is provided only for exceptional situations such as recovering from errors or mistakes, or when
Terraform specifically suggests to use it as part of an error message.
```

c) Specific instance is destroyed automatically without entering any value yes (without our permission).

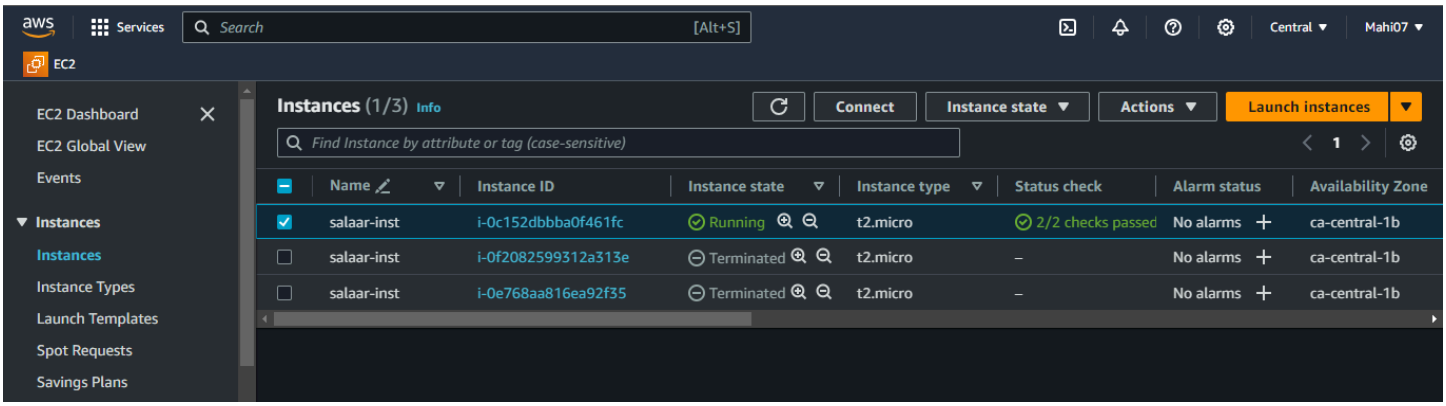
```
aws_instance.rawan[1]: Still destroying... [id=i-0f2082599312a313e, 40s elapsed]
aws_instance.rawan[1]: Destruction complete after 40s

Warning: Resource targeting is in effect

You are creating a plan with the -target option, which means that the result of this plan may not represent all of the changes requested by
the current configuration.

The -target option is not for routine use, and is provided only for exceptional situations such as recovering from errors or mistakes, or when
Terraform specifically suggests to use it as part of an error message.


Warning: Applied changes may be incomplete

The plan was created with the -target option in effect, so some changes requested in the configuration may have been ignored and the output
values may not be fully updated. Run the following command to verify that no other changes are pending:
    terraform plan

Note that the -target option is not suitable for routine use, and is provided only for exceptional situations such as recovering from errors
or mistakes, or when Terraform specifically suggests to use it as part of an error message.


Destroy complete! Resources: 1 destroyed.
[root@ip-172-31-27-18 ~]#
```

```
i-0637cb63905eb097d (TFORM)
PublicIPs: 54.162.157.65  PrivateIPs: 172.31.27.18
```

d) Earlier to the deletion, we had two instances running.



e) After the deletion, we only have one instance.



The instance index value with label we mentioned in the CLI is terminated and the state changes from stopping - - > stopped - - > terminated.

Deletion of all resources mentioned in the .tf file:

If you want to delete all the resources mentioned in the .tf file then use the below command.

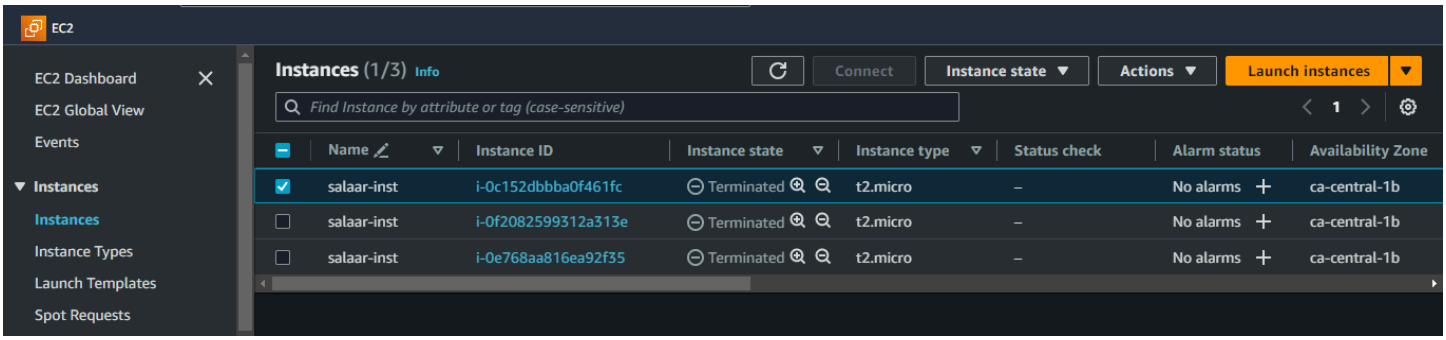
terraform destroy --auto-approve

As we already deleted two instances above, so the remaining one last instance is deleted as shown below.

```
[root@ip-172-31-27-18 ~]# terraform destroy --auto-approve
aws_instance.rawan[0]: Refreshing state... [id=i-0c152dbbba0f461fc]

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_instance.rawan[0] will be destroyed
  - resource "aws_instance" "rawan" {
      - ami                          = "ami-095819c19b51bc983" -> null
      - arn                          = "arn:aws:ec2:ca-central-1:822311367203:instance/i-0c152dbbba0f461fc" -> null
      - associate_public_ip_address  = true -> null
      - availability_zone            = "ca-central-1b" -> null
      - cpu_core_count               = 1 -> null
      - cpu_threads_per_core         = 1 -> null
      - disable_api_stop             = false -> null
      - disable_api_termination      = false -> null
      - ebs_optimized                = false -> null
      - get_password_data            = false -> null
```

```
      - root_block_device {
          - delete_on_termination = true -> null
          - device_name           = "/dev/xvda" -> null
          - encrypted             = false -> null
          - iops                  = 3000 -> null
          - tags                  = {} -> null
          - throughput            = 125 -> null
          - volume_id             = "vol-084a9c63851cab878" -> null
          - volume_size           = 8 -> null
          - volume_type           = "gp3" -> null
        }
    }

Plan: 0 to add, 0 to change, 1 to destroy.
aws_instance.rawan[0]: Destroying... [id=i-0c152dbbba0f461fc]
aws_instance.rawan[0]: Still destroying... [id=i-0c152dbbba0f461fc, 10s elapsed]
aws_instance.rawan[0]: Still destroying... [id=i-0c152dbbba0f461fc, 20s elapsed]
aws_instance.rawan[0]: Still destroying... [id=i-0c152dbbba0f461fc, 30s elapsed]
aws_instance.rawan[0]: Still destroying... [id=i-0c152dbbba0f461fc, 40s elapsed]
aws_instance.rawan[0]: Destruction complete after 40s

Destroy complete! Resources: 1 destroyed.
[root@ip-172-31-27-18 ~]#
```

Now the last instance from the group of three instances, which is the first one in the below image has also deleted.



Steps to launch instances in multiple regions:

The terraform code needs to be written twice for provider, resource and most importantly alias needs to be mentioned in the code in order to create two instances in two different regions as show in the image below.

1) Write a terraform code as per the requirement.

NOTE: As the code is lengthy and unable to take a screenshot as a single image so took two. Also, _ is missed for access_key in the first image.

```
provider "aws" {
region = "ap-south-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

provider "aws" {
region = "us-east-2"
alias = "ohio"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}
```

```
resource "aws_instance" "jawaan" {
tags {
Name = "mbi-instance"
}
ami = "ami-074f77adfeee318d3"
instance_type = "t2.micro"
availability_zone = "ap-south-1a"
}

resource "aws_instance" "pathan" {
tags {
Name = "ohio-instance"
}
provider = "aws.ohio"
ami = "ami-0f599bbc07afc299a"
instance_type = "t2.micro"
availability_zone = "us-east-2a"
}
```

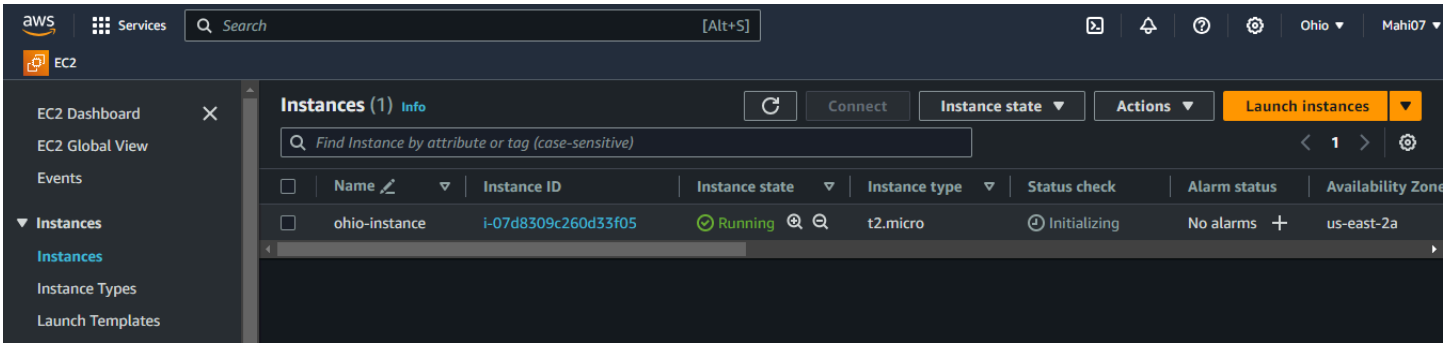
2) You can preview the changes that Terraform will make before you apply them.

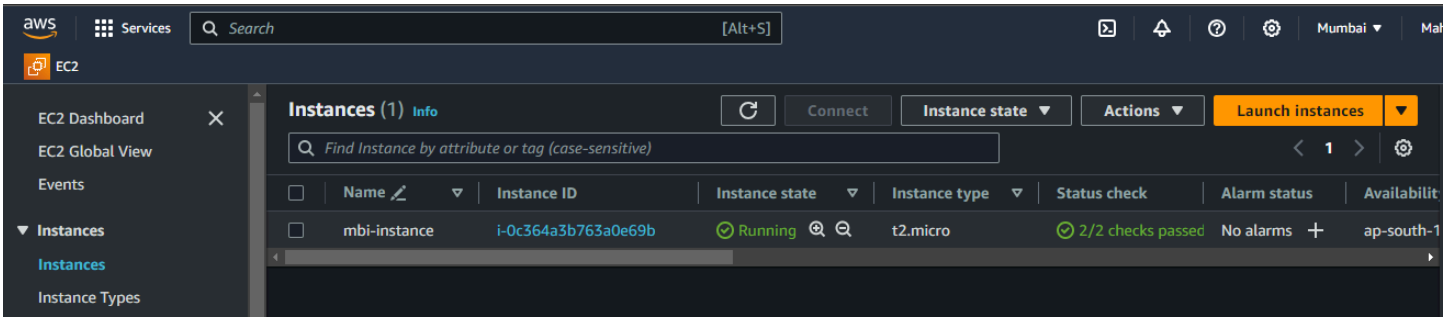
```
Plan: 2 to add, 0 to change, 0 to destroy.

  Warning: Quoted references are deprecated

    on mahi.tf line 27, in resource "aws_instance" "pathan":
    27: provider = "aws.ohio"

  In this context, references are expected literally rather than in quotes. Terraform 0.11 and earlier required quotes, but quoted references
  are now deprecated and will be removed in a future version of Terraform. Remove the quotes surrounding this reference to silence this warning.


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@ip-172-31-27-18 ~]#
```

3) The changes that are made after applying.

```
Plan: 2 to add, 0 to change, 0 to destroy.
aws_instance.pathan: Creating...
aws_instance.jawaan: Creating...
aws_instance.pathan: Still creating... [10s elapsed]
aws_instance.jawaan: Still creating... [10s elapsed]
aws_instance.pathan: Still creating... [20s elapsed]
aws_instance.jawaan: Still creating... [20s elapsed]
aws_instance.pathan: Still creating... [30s elapsed]
aws_instance.jawaan: Still creating... [30s elapsed]
aws_instance.pathan: Creation complete after 32s [id=i-07d8309c260d33f05]
aws_instance.jawaan: Creation complete after 34s [id=i-0c364a3b763a0e69b]

  Warning: Quoted references are deprecated

    on mahi.tf line 27, in resource "aws_instance" "pathan":
    27: provider = "aws.ohio"

  In this context, references are expected literally rather than in quotes. Terraform 0.11 and earlier required quotes, but quoted references
  are now deprecated and will be removed in a future version of Terraform. Remove the quotes surrounding this reference to silence this warning.


Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
[root@ip-172-31-27-18 ~]#
```

4) One instance is created in the region Ohio and another one in the Mumbai region as shown below.





To delete these two instances you can either use the CLI command or you can also just mention the count as “0” in the code under instance type.

Structuring the code with correct indentation:

If you want to structure the code with correct indentation then you can just give the command as **terraform fmt** (terraform format). It arranges the YAML code into a rearranged one.

Importance of Terraform backup files:

If no changes have been made to the terraform code and if you give the terraform plan command it shows as no changes have been made. How does it define that there are no changes is because of the **'terraform backup'** files.

```
[root@ip-172-31-27-18 ~]# ll
total 16
-rw-r--r-- 1 root root  345 Dec 13 08:06 mahi.tf
-rw-r--r-- 1 root root  181 Dec 13 09:28 terraform.tfstate
-rw-r--r-- 1 root root 4978 Dec 13 09:28 terraform.tfstate.backup
[root@ip-172-31-27-18 ~]#
```

Usually, when you write a code and apply (terraform apply) it for the first time, a backup file will be created. Later, if any changes are made to the code like the instance name has been changed and if you apply it again, then terraform compares the **'.tf'** file with the **'backup file'** that already has all the configurations in it. If there are any changes found then it shows in the format of 0 to add, 1 to change (instance name changed) and 0 to destroy and if there aren't any changes it shows 0 to add, 0 to change, 0 to destroy.

Summary: With the help of terraform backup files, we can compare the data of main.tf file with the backup file and then it displays whether the changes are made when we give terraform plan.

Importance of deletion of code and inclusion of new code in the .tf file:

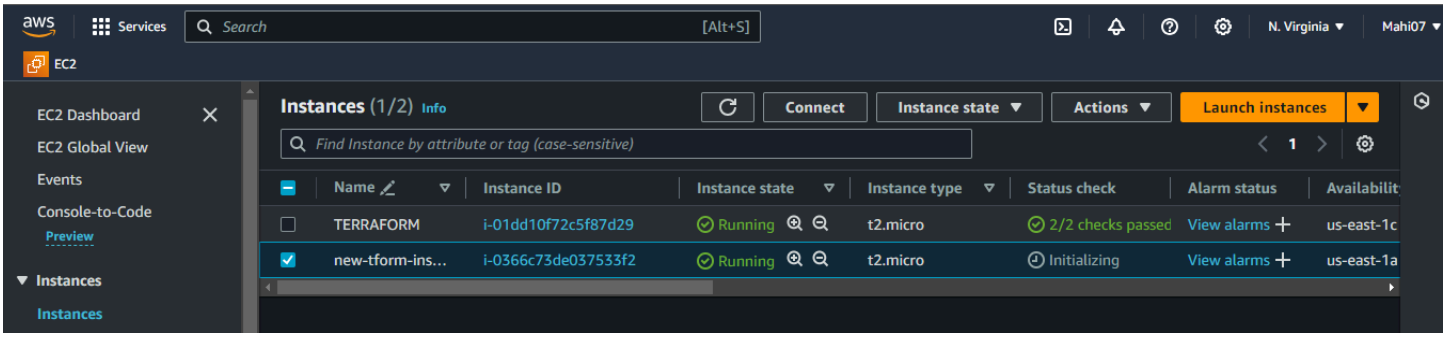
If the terraform code in the .tf file is deleted and if a new code is written in the same .tf file then the resource which is live it gets deleted and the new code that we've in the .tf file will go live and we can access them and not the old one.

So the conclusion here is, if the old code is deleted it deletes the entire infra of the old code and whatever the code is currently running only those resources can be accessed live.

1) The old code for instance created in Northern Virginia.

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_instance" "LEO" {
ami = "ami-018ba43095ff50d08"
instance_type = "t2.micro"
tags = {
Name = "new-tform-instance"
}
}
```

a) The instance named **'new-tform-instance'** is created.



2) The old terraform code is removed and a new code is written to create an instance in the same Northern Virginia region again with name **'VIKRAM'**.

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_instance" "VIKRAM" {
ami = "ami-0230bd60aa48260c6"
instance_type = "t2.micro"
tags = {
Name = "north-instance"
}
}
```

3) a) After giving terraform plan, we got the result that **'aws_instance.LEO'** is getting destroyed as it is

the old instance of the old code.

```
[root@ip-172-31-95-242 ~]# terraform plan
aws_instance.LEO: Refreshing state... [id=i-0366c73de037533f2]

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.LEO will be destroyed
  # (because aws_instance.LEO is not in configuration)
  - resource "aws_instance" "LEO" {
      - ami                                  = "ami-018ba43095ff50d08" -> null
      - arn                                  = "arn:aws:ec2:us-east-1:822311367203:instance/i-0366c73de037533f2" -> null
      - associate_public_ip_address          = true -> null
      - availability_zone                    = "us-east-1a" -> null
      - cpu_core_count                       = 1 -> null
      - cpu_threads_per_core                 = 1 -> null
      - disable_api_stop                     = false -> null
```

b) The new instance is getting created with name **'aws_instance.VIKRAM'**.

```
# aws_instance.VIKRAM will be created
+ resource "aws_instance" "VIKRAM" {
    + ami                                  = "ami-0230bd60aa48260c6"
    + 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_stop                     = (known after apply)
    + disable_api_termination              = (known after apply)
    + ebs_optimized                        = (known after apply)
    + get_password_data                    = false
    + host_id                              = (known after apply)
    + host_resource_group_arn              = (known after apply)
    + iam_instance_profile                 = (known after apply)
    + id                                   = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_lifecycle                   = (known after apply)
    + instance_state                       = (known after apply)
    + instance_type                        = "t2.micro"
```

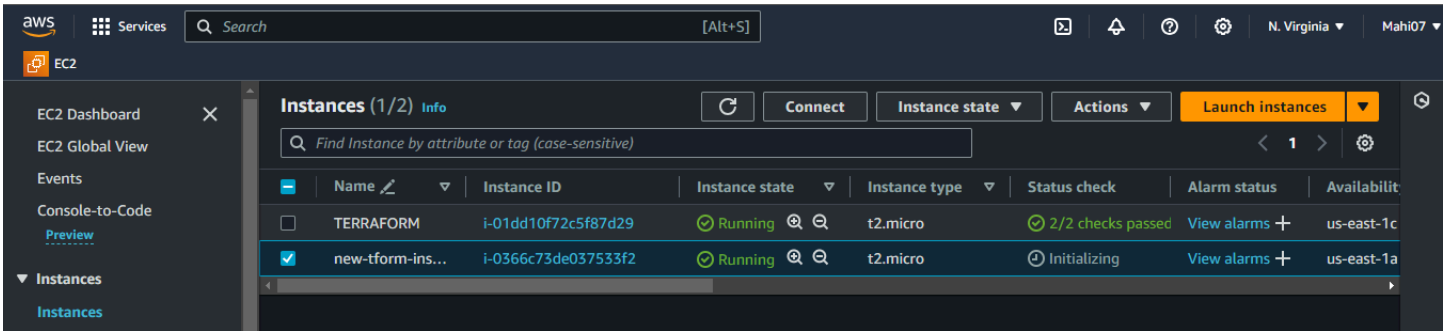
c) The preview of changes that Terraform will make it happen before applying them.

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

```
    + tags                                 = {
        + "Name" = "north-instance"
      }
    + tags_all                             = {
        + "Name" = "north-instance"
      }
    + tenancy                              = (known after apply)
    + user_data                            = (known after apply)
    + user_data_base64                     = (known after apply)
    + user_data_replace_on_change          = false
    + vpc_security_group_ids               = (known after apply)
  }

Plan: 1 to add, 0 to change, 1 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@ip-172-31-95-242 ~]#
```

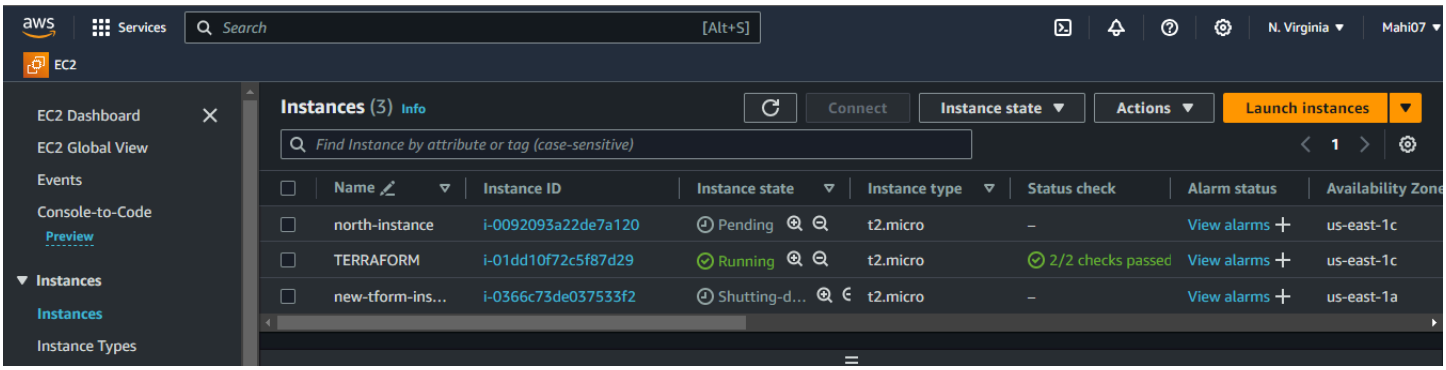
3) First we've only one instance which is **'new-tform-instance'** but, a new one will be added with name **'north-instance'** in the same region.

a) Old instance

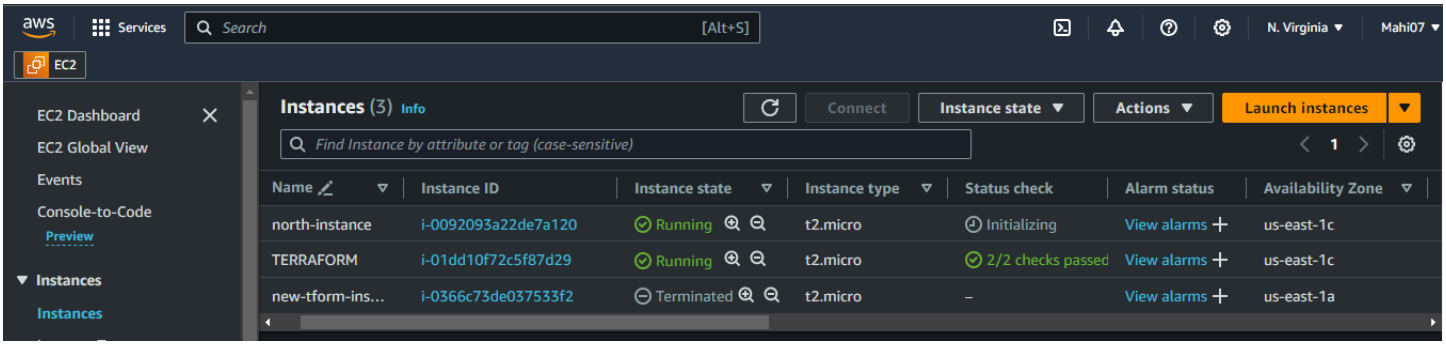


b) Creation of new instance and deletion of old instance.

i) The state of new-tform-instance is changing from running to shutting-down and at the same time the instance north-instance is getting ready to launch which is currently in pending as shown below.



ii) The old instance is deleted and the new one has come into running.



CONCLUSION/SUMMARY OF ABOVE PICTORIAL REPRESENTATION:

If the old code is deleted, it deletes the entire infra of the old code and whatever the code that's currently running, only those resources can be accessed live.

Terraform code with Variables:

Variables is the last syntax of the terraform code. In Terraform, we use variables to parameterize the infrastructure code, allowing to reuse and customize configurations.

If we declare a variable and define a value to it for any component (volume, instance type) to execute the code and if you want to make any changes to the components/parameters in the code, we can just make changes to the variables instead of altering the source code.

Every variable in the terraform code consists of description, type and default.

Creation of EC2 instance with terraform code along with the inclusion of Keypair, EBS, Security groups.

1) Terraform code for variables.

a) First half of the code with declaration of variables for all the parameters.

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIA365MTWIRWXXCWCHL"
  secret_key = "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"
}

resource "aws_instance" "salaar" {
  tags = {
    Name = var.name
  }
  ami                    = var.ami_id
  instance_type          = var.inst_type
  key_name               = var.kpair
  availability_zone      = var.azone
  vpc_security_group_ids = [aws_security_group.trial-sg.id]
  root_block_device {
  volume_size = var.vsize
  }
  count = var.icount
}
```

b) The variable values are defined in this second half of the code.

You can either mention the description for every variable or not, that's as per requirement.

```
variable "name" {
  description = "this is instance name"
  type        = string
  default     = "deva"
}

variable "ami_id" {
  type    = string
  default = "ami-0230bd60aa48260c6"
}

variable "inst_type" {
  type    = string
  default = "t2.micro"
}

variable "kpair" {
  type    = string
  default = "master-keypair"
}
```

c) The rest of the code with variable values.

```
variable "azone" {
  type    = string
  default = "us-east-1b"
}

variable "vsize" {
  type    = number
  default = 10
}

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

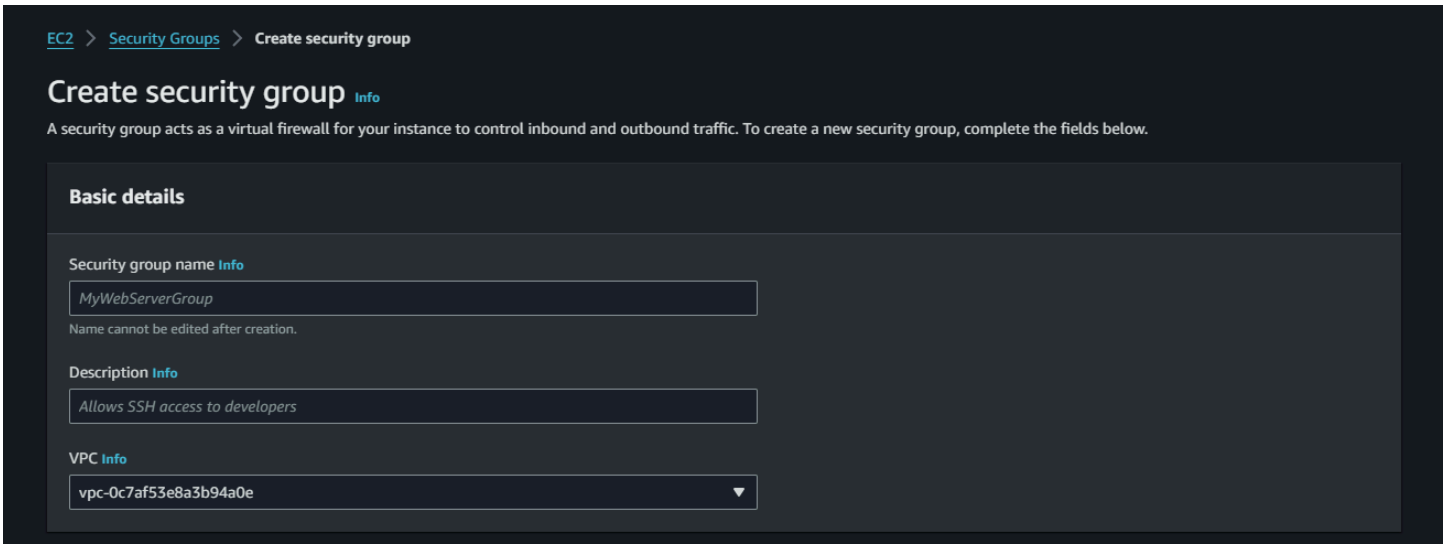
2) Terraform code for the security groups.

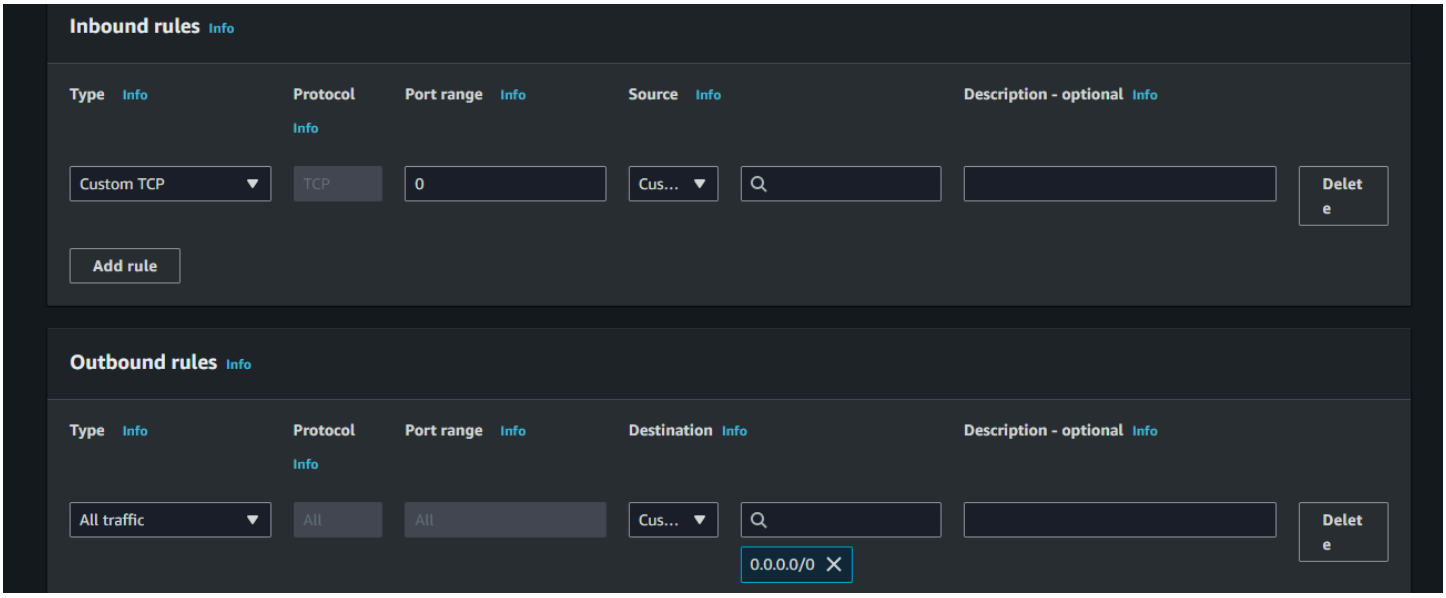
If you've multiple VPC id's as suggestions under VPC option then you can select and mention it as '**vpc_id**' after the description in the terraform code.

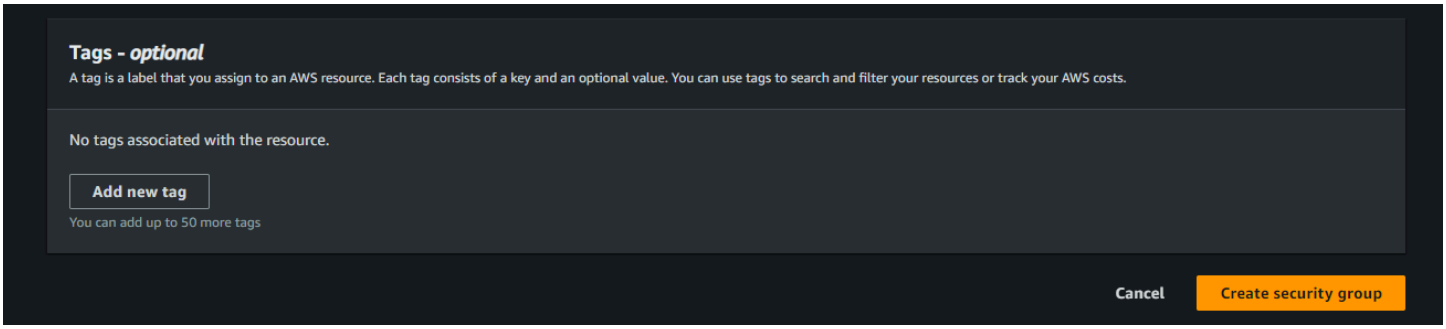
Inbound rules in the terraform code is called as **‘ingress’** and the outbound rules called as **‘egress’**.

The source in inbound rules is called as cidr block (Classless Inter-Domain Routing) in the terraform code.

a) Basic representation of the options available on the security groups page on aws console while creating manually.







You can refer to the above images that has details like name, description, vpc, inbound rules, outbound rules needs to be mentioned in the terraform code.

b) Terraform code for the Security group. Create a new file with tf and write the code.

```
resource "aws_security_group" "trial-sg" {
  name        = "NEW-sg"
  description = "this is the security group created by TCODE"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
ingress {
from_port = 8080
to_port = 8080
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}

ingress {
from_port = 0
to_port = 0
protocol = "-1"
cidr_blocks = ["0.0.0.0/0"]
}

egress {
from_port = 0
to_port = 0
protocol = "-1"
cidr_blocks = ["0.0.0.0/0"]
}
}
```

3) Result after Terraform plan.

a) Preview of the 1st instance creation prior to the execution (terraform apply).

```
[root@ip-172-31-95-242 ~]# terraform plan
aws_instance.VIKRAM: Refreshing state... [id=i-0092093a22de7a120]

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.salaar[0] will be created
  + resource "aws_instance" "salaar" {
      + ami                          = "ami-0230bd60aa48260c6"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = "us-east-1b"
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + disable_api_stop             = (known after apply)
      + disable_api_termination      = (known after apply)
      + ebs_optimized                = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + host_resource_group_arn      = (known after apply)
      + iam_instance_profile         = (known after apply)
```

b) Preview of the 2nd instance creation prior to the execution (terraform apply).

```
# aws_instance.salaar[1] will be created
+ resource "aws_instance" "salaar" {
    + ami                                  = "ami-0230bd60aa48260c6"
    + arn                                  = (known after apply)
    + associate_public_ip_address          = (known after apply)
    + availability_zone                    = "us-east-1b"
    + cpu_core_count                       = (known after apply)
    + cpu_threads_per_core                 = (known after apply)
    + disable_api_stop                     = (known after apply)
    + disable_api_termination              = (known after apply)
    + ebs_optimized                        = (known after apply)
    + get_password_data                    = false
    + host_id                              = (known after apply)
    + host_resource_group_arn              = (known after apply)
    + iam_instance_profile                 = (known after apply)
    + id                                   = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_lifecycle                   = (known after apply)
    + instance_state                       = (known after apply)
    + instance_type                        = "t2.micro"
```

c) Preview of Security group creation.

```
  # aws_security_group.trial-sg will be created
  + resource "aws_security_group" "trial-sg" {
      + arn                    = (known after apply)
      + description            = "this is the security group created by TCODE"
      + egress                 = [
          + {
              + cidr_blocks      = [
                  + "0.0.0.0/0",
                ]
              + description      = ""
              + from_port        = 0
              + ipv6_cidr_blocks = []
              + prefix_list_ids  = []
              + protocol         = "-1"
              + security_groups  = []
              + self             = false
              + to_port          = 0
```

d) Preview of two instances and one security group which are yet to be created. Hence the plan is 3 to add.

```
Plan: 3 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@ip-172-31-95-242 ~]#
```

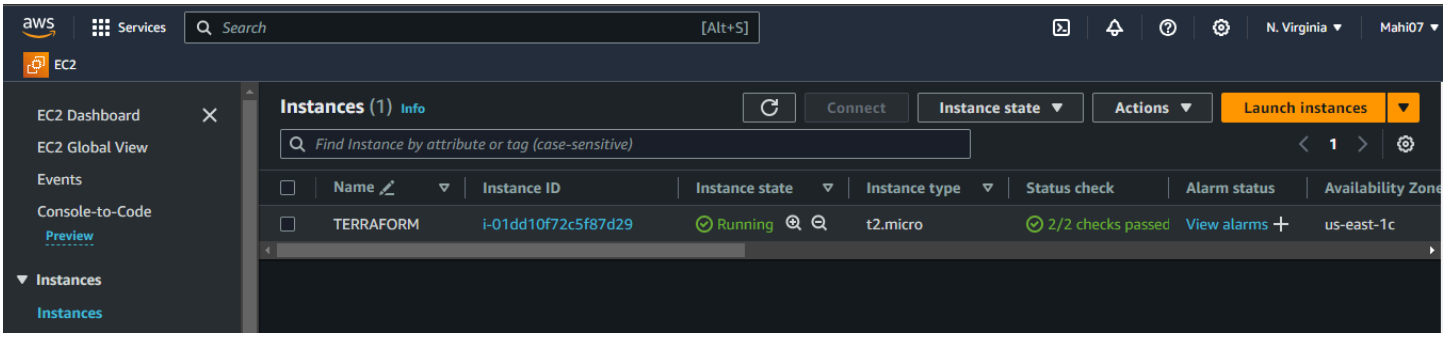
i-01dd10f72c5f87d29 (TERRAFORM)

PublicIPs: 54.204.75.247 PrivateIPs: 172.31.95.242

4) Result after Terraform apply.

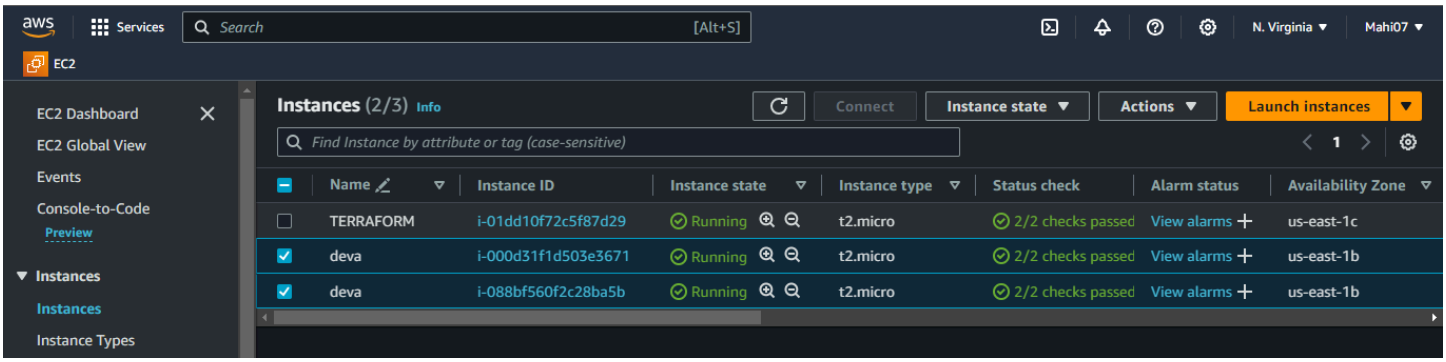
```
Plan: 3 to add, 0 to change, 0 to destroy.
aws_security_group.trial-sg: Creating...
aws_security_group.trial-sg: Creation complete after 2s [id=sg-0094d5e3f60844317]
aws_instance.salaar[1]: Creating...
aws_instance.salaar[0]: Creating...
aws_instance.salaar[1]: Still creating... [10s elapsed]
aws_instance.salaar[0]: Still creating... [10s elapsed]
aws_instance.salaar[1]: Still creating... [20s elapsed]
aws_instance.salaar[0]: Still creating... [20s elapsed]
aws_instance.salaar[1]: Still creating... [30s elapsed]
aws_instance.salaar[0]: Still creating... [30s elapsed]
aws_instance.salaar[1]: Creation complete after 31s [id=i-088bf560f2c28ba5b]
aws_instance.salaar[0]: Creation complete after 32s [id=i-000d31f1d503e3671]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
[root@ip-172-31-95-242 ~]#
```

4) Here we only have 1 instance before applying the changes after the terraform plan.

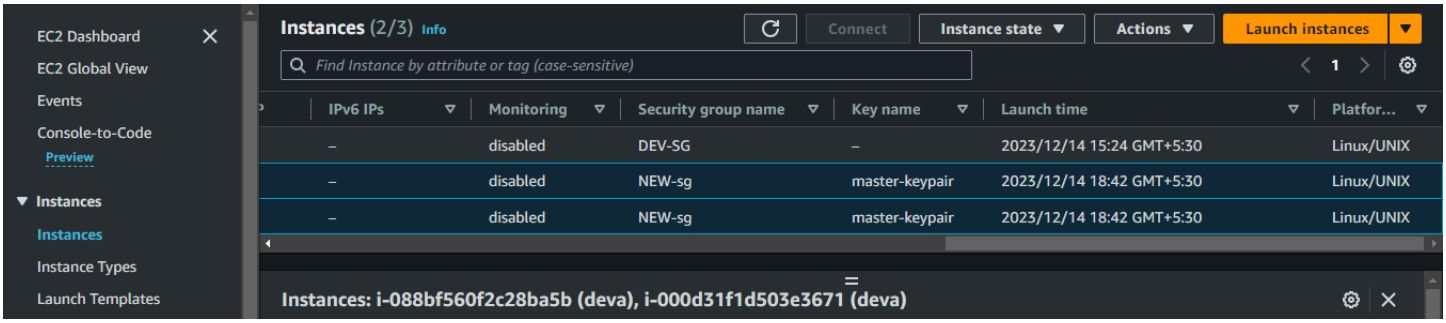


5) Now we've two instances and a security group as shown below.

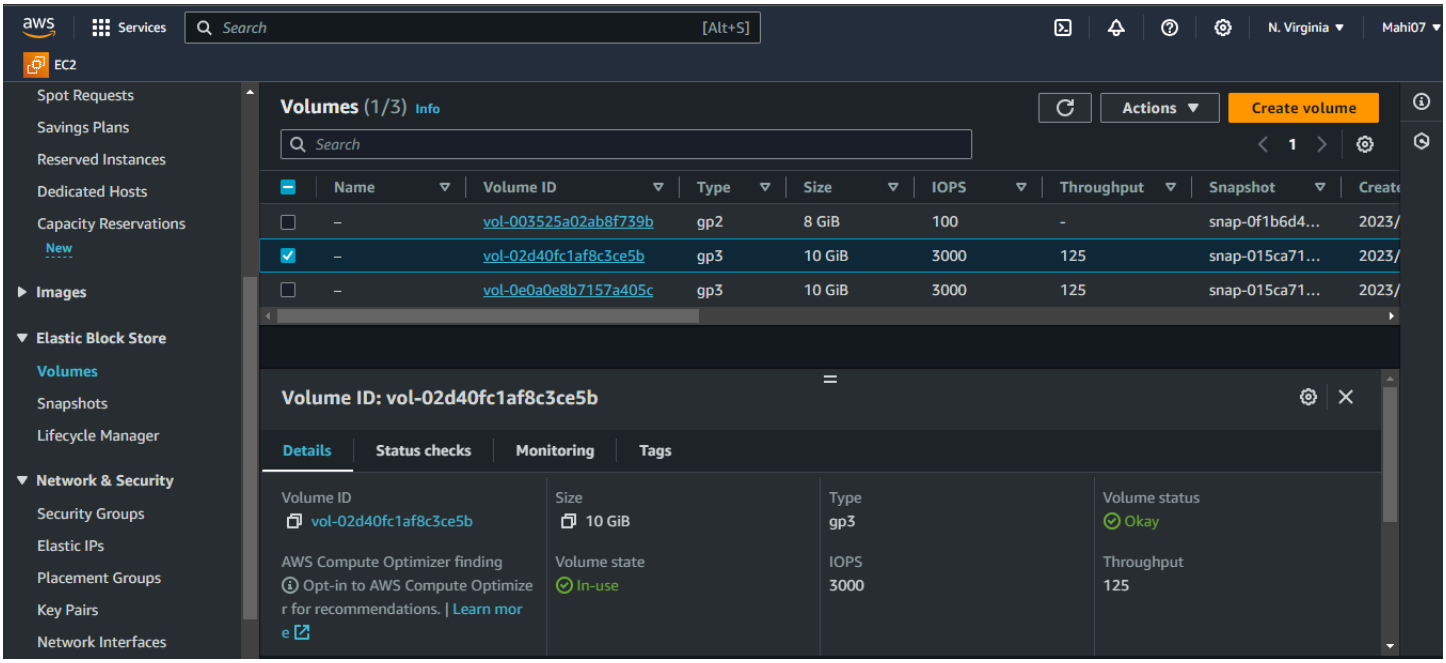
a) The below image represents that two instances are created with name "**deva**", in the availability zone "**us-east-1b**" and with instance type "**t2.micro**".



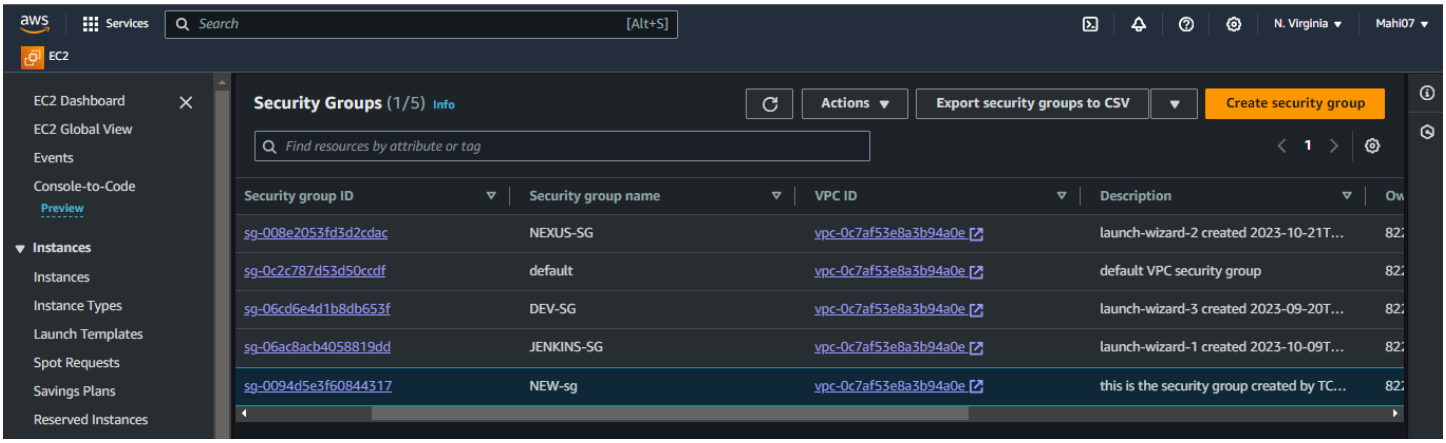
b) The below image represents that the instances are created with the security group "**NEW-sg**" and with keypair "**master-keypair**".



c) The below image represents that two instances are created with a "**10 GiB**" of volume.



d) The below image represents that the security group is created with name "**NEW-sg**" the description "this is the security group created by TCODE" mentioned in the terraform code.



CONCLUSION/SUMMARY OF ABOVE PICTORIAL REPRESENTATION:

As per the written terraform code, all the requirements mentioned such as, a count of '**two**' instances named **'deva'** have been launched with instance type as **'t2.micro'**, under the availability zone **'us-east-1b'**, in **'Northern Virginia'** region with given **'ami-id - ami-0230bd60aa48260c6'** along with a keypair named **'master-keypair'**, security group named **'NEW-sg'** with a volume of **'10 GiB'**.

Creation of users and user group in IAM using Terraform code:

A) Creation of a single user in IAM:

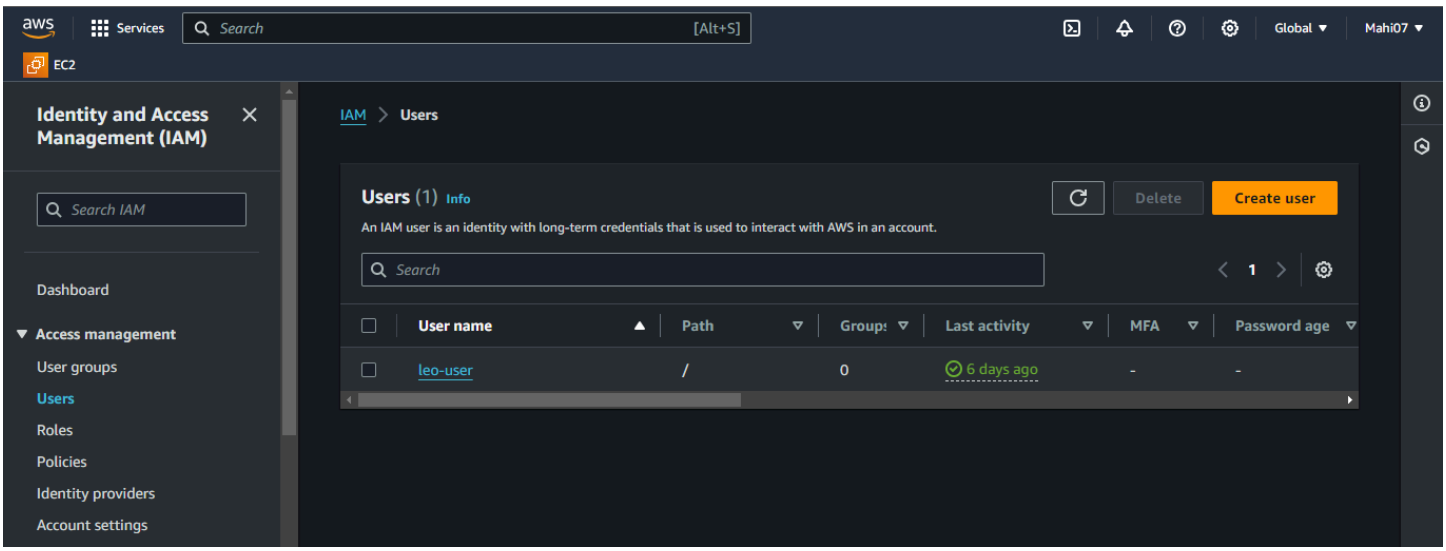
1) Creating a user with the help of a terraform code.

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_iam_user" "deva" {
name = "varadha-user"
}
```

2) Preview of user getting added.

```
[root@ip-172-31-18-198 ~]# terraform plan

Terraform used the selected providers to generate the following execution plan.
  + create

Terraform will perform the following actions:

  # aws_iam_user.deva will be created
  + resource "aws_iam_user" "deva" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "varadha-user"
      + path          = "/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

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

3) Before giving terraform apply, the list of users on aws console.



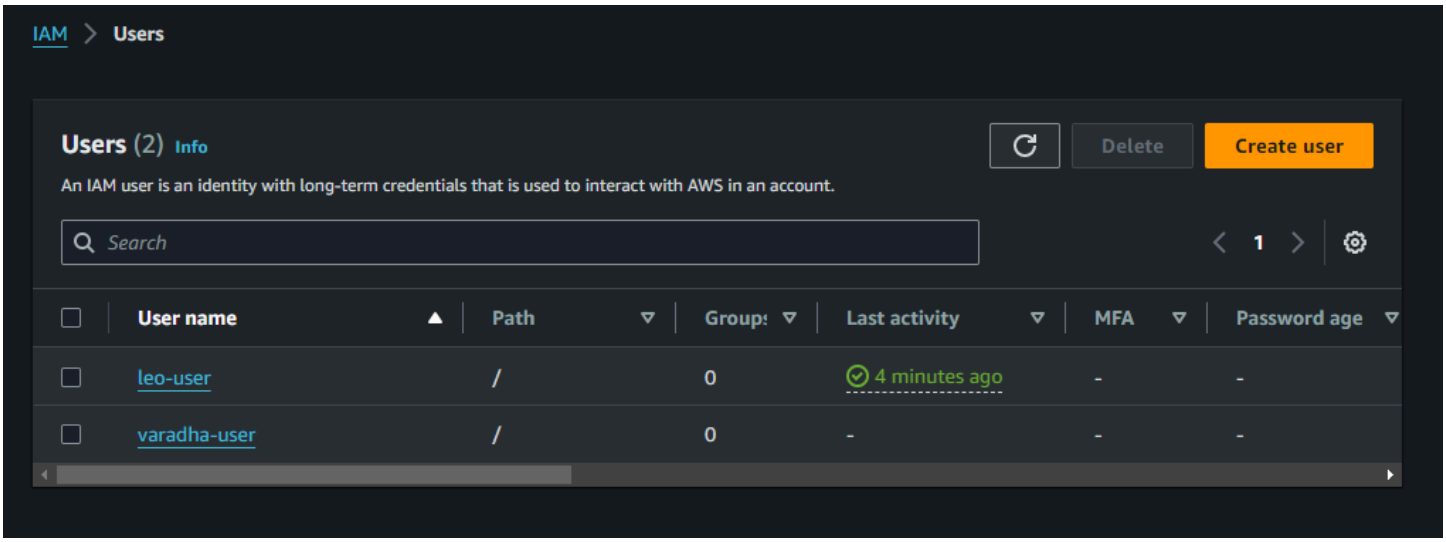
4) Result after terraform apply (ensure that the user you create should've the full permissions of IAM which is IAMFullAccess in order get this user create in IAM service, else you'll get an error as below).

```
[root@ip-172-31-18-198 ~]# terraform apply --auto-approve
aws_iam_user.deva: Refreshing state... [id=varadha-user]

Error: reading IAM User (varadha-user): AccessDenied: User: arn:aws:iam::822311367203:user/leo-user is not authorized to perform: iam:GetUser
on resource: user varadha-user because no identity-based policy allows the iam:GetUser action
        status code: 403, request id: 3826f8b5-9dba-4393-8e79-f2a5613459a0

   with aws_iam_user.deva,
   on mahi.tf line 7, in resource "aws_iam_user" "deva":
    7: resource "aws_iam_user" "deva" {
```

i) After IAMFullAccess permission is added.

```
  # aws_iam_user.deva will be created
  + resource "aws_iam_user" "deva" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "varadha-user"
      + path          = "/"
      + tags_all      = (known after apply)
      + unique_id     = (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

aws_iam_user.deva: Creating...
aws_iam_user.deva: Creation complete after 1s [id=varadha-user]

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

5) Now another user that wrote in the terraform code (**'varadha-user'**) has been added.



B) Creation of multiple users in IAM.

1) Terraform code to add/create multiple users in IAM. The description isn't mandatory in the below code.

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_iam_user" "uzi" {
count = length(var.abc)
name = var.abc[count.index]
}

variable "abc" {
description = "this is the variable to create multiple users"
type = list(string)
default = ["mahi", "ravi", "tiru"]
}
```

2) Three users are getting created as mentioned in the code.

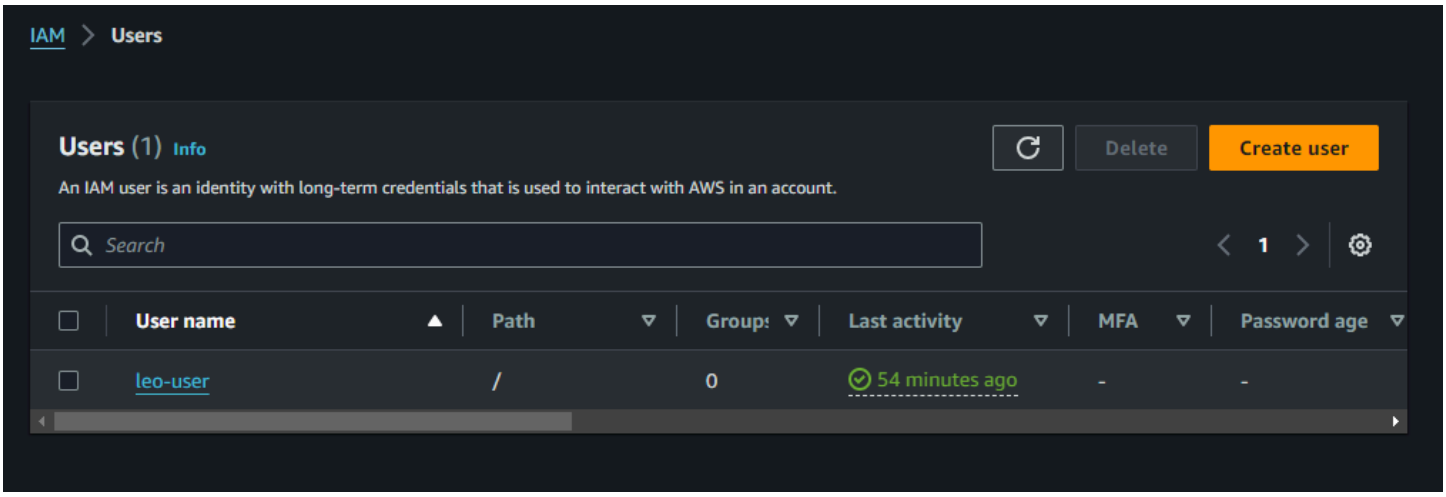
```
[root@ip-172-31-18-198 ~]# terraform plan

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_iam_user.uzi[0] will be created
  + resource "aws_iam_user" "uzi" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "mahi"
      + path          = "/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }
```

```
  # aws_iam_user.uzi[1] will be created
  + resource "aws_iam_user" "uzi" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "ravi"
      + path          = "/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

  # aws_iam_user.uzi[2] will be created
  + resource "aws_iam_user" "uzi" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "tiru"
      + path          = "/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

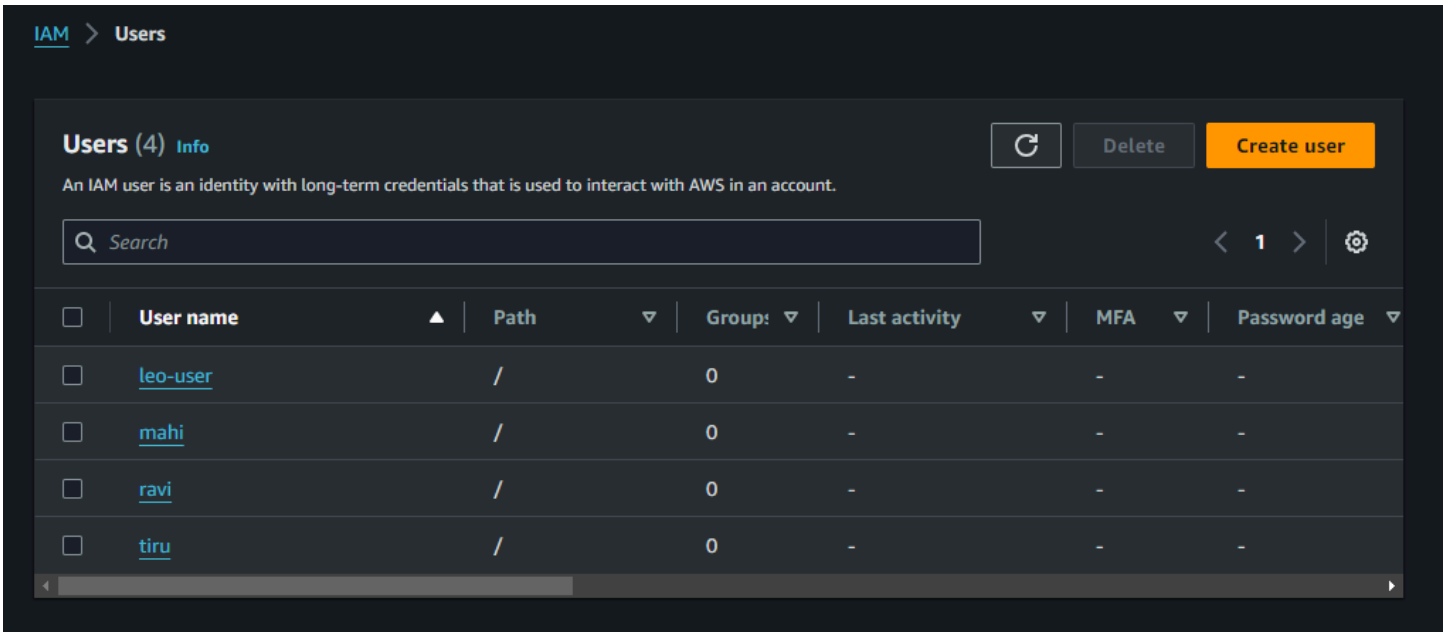
3) Execution result after terraform apply.

```
Plan: 3 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_iam_user.uzi[1]: Creating...
aws_iam_user.uzi[2]: Creating...
aws_iam_user.uzi[0]: Creating...
aws_iam_user.uzi[1]: Creation complete after 0s [id=ravi]
aws_iam_user.uzi[2]: Creation complete after 0s [id=tiru]
aws_iam_user.uzi[0]: Creation complete after 0s [id=mahi]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

4) Earlier to the terraform apply, we've only one user as shown below.



5) Now we've got all the three users created.



Simple Storage Service (S3)

S3 is abbreviated as Simple Storage Service. It's one of the services in the aws cloud which is widely used by the DevOps engineers.

CRR concept in S3 is a Cross region replication, if data is uploaded in one bucket, it reflects on another bucket as well. If we delete the data in any of the buckets, it doesn't replicate. It's only for upload.

There are a few different types of storages like object based storage (S3), block level storage (EBS) and file level storage (EFS).

1) S3 - Simple Storage Service. To store app logs, infra logs, config logs etc.

2) EBS - Elastic Block Storage. The actions like creation of files, folders, tools installation on aws cloud is stored on EBS which was selected while creating EC2 instance.

3) EFS - Elastic File Storage is used for file replication from one server to another.

About Simple Storage Service (S3):

1. S3 is highly scalable, durable, securable to store the data easily and easy to retrieve the data.
2. It's an object based storage and data is spread in multiple regions because it is not a region based service (works in all regions).
3. Customers of all sizes and industries can use to store the data and protect any amount of data range of use cases, such as data lakes, websites, mobile applications, archive, enterprise applications, IoT devices and big data analytics.

POINTS:

1) While creating a bucket you need to select the region, because first the bucket has to be created in any one of the regions to be accessed globally.

2) We can't upload or make any changes in the data in the file on S3 service, we need to make changes in the file in local and then upload it.

Bucket in S3 and it's importance:

A bucket is a root level folder created in S3 which stores all the objects (files).

1. A bucket ownership isn't transferable to another account and you can't change it's name or region after a bucket is created.
2. You can't delete the bucket directly, it needs to be emptied first.
3. The name of bucket must be unique. Once a bucket is deleted, the name becomes available for reuse.
4. The files stored in S3 must be from 0 Bytes (0B) to 5 Terra Bytes (5TB).
5. By default, you can create up to 100 buckets in each AWS account.
6. If you need additional buckets, then you can increase the limit to 1,000 buckets by submitting a service limit increase.

Important rules to create a bucket:

1. Name should be unique.
2. Only lower case is valid to be in the name.
3. The name should be in between 3 to 63 characters.
4. It must begin and end with a number or a letter.
5. The name shouldn't be in a IP format.

NOTE: Before March 1 2018, buckets created in US East (N. Virginia) region could've names that were up to 255 characters long and included upper case letters and underscores. However, beginning March 1 2018, new buckets in US East (N. Virginia) must conform to the same rules as all other regions.

S3 versioning:

1. It'll enable you to maintain the multiple versions of the same file.
2. It helps you to retrieve the data due to any accidental termination (It'll have the deleted file along with a new file as a copy with delete marker if S3 versioning is enabled for that bucket and also to the object).
3. It helps to retrieve the previous versions of the objects.
4. By default the versioning isn't enabled, we need to enable it manually.
5. We can't disable the versioning, we can only suspend it.

Object and it's importance:

A file that may be a text or a folder or an image or a video (mp4) or an audio (mp3) stored inside a S3 bucket is called as an Object.

1. A bucket is a container for objects.
2. An object is a file and any metadata that describes that file.
3. With amazon S3 you pay only for your use.

Major Components of Object: The major components are Key, Value, Version Id and Metadata.

i) Key: It's the name of the object.

ii) Value: It's the data denoted in Bytes.

iii) Version ID: It shows the versioning Id for the uniqueness of the object.

iv) Metadata: It's the data about the data we're sharing.

Depending on the size of data you're uploading Amazon S3 offers the following options:

i) Upload an object using in a single operation using the AWS SDK's RESET API, or AWS CLI: With a PUT operation you can upload a single object up to 5GB in size.

ii) Upload an object using in a single operation using the Amazon S3 console: You can upload a single object up to 160 GB in size.

iii) Upload an object in parts using the AWS SDK's RESET API, or AWS CLI: Using the multipart upload API, you can upload a single large object up to 5 TB in size.

The AWS CLI SYNTAX: **cloud_provider service command**

Ex: aws s3 ls (list of buckets)

Creation of S3 bucket using AWS CLI:

Command Syntax: cloud_provider service command

Command: aws s3 mb s3://deva.bucket

Here: **aws** - cloud provider, **s3** - service, **mb** - make bucket, **s3://mahi.mudhiraj** - bucket name.

CLI COMMANDS:

1) To create a bucket: **aws s3 mb s3://bucket_name**

```
[root@ip-172-31-18-198 ~]# aws s3 mb s3://deva.bucket
make_bucket: deva.bucket
[root@ip-172-31-18-198 ~]#
```

2) To check the list of buckets: **aws s3 ls**

```
[root@ip-172-31-18-198 ~]# aws s3 ls
2023-12-21 13:35:03 deva.bucket
[root@ip-172-31-18-198 ~]#
```

3) To check the list of objects in a specific bucket: **aws s3 ls s3://bucket_name**

```
[root@ip-172-31-18-198 ~]# touch mahi.pdf
[root@ip-172-31-18-198 ~]# aws s3 cp mahi.pdf s3://deva.bucket
upload: ./mahi.pdf to s3://deva.bucket/mahi.pdf
[root@ip-172-31-18-198 ~]#
```

```
[root@ip-172-31-18-198 ~]# aws s3 ls s3://deva.bucket
2023-12-21 13:38:57          0 mahi.pdf
[root@ip-172-31-18-198 ~]#
```

3) To copy a file from server to the bucket: **aws s3 cp filename s3://bucket_name**

```
[root@ip-172-31-18-198 ~]# touch mahi.pdf
[root@ip-172-31-18-198 ~]# aws s3 cp mahi.pdf s3://deva.bucket
upload: ./mahi.pdf to s3://deva.bucket/mahi.pdf
[root@ip-172-31-18-198 ~]#
```

4) If you want to download the objects from s3 bucket to the local machine/local server:

Copy the S3 URI from the console and then use the below command

aws s3 cp S3 URI - - - > aws s3 cp s3://deva.bucket/mahi.pdf .

mahi.pdf object in the deva.bucket is downloading to the present directory (.)

```
[root@ip-172-31-18-198 ~]# aws s3 cp s3://deva.bucket/mahi.pdf .
download: s3://deva.bucket/mahi.pdf to ./mahi.pdf
[root@ip-172-31-18-198 ~]#
```

5) To delete a bucket: **aws s3 rb s3://bucket_name**

```
[root@ip-172-31-18-198 ~]# aws s3 rb s3://deva.bucket
remove_bucket: deva.bucket
[root@ip-172-31-18-198 ~]#
```

If the bucket has objects then it needs to be emptied first, then it can be deleted. Else it gives error.

```
[root@ip-172-31-18-198 ~]# aws s3 rb s3://deva.bucket
remove_bucket failed: s3://deva.bucket An error occurred (BucketNotEmpty) when calling the DeleteBucket operation: The bucket you tried to dele
te is not empty
[root@ip-172-31-18-198 ~]# aws s3 rm s3://deva.bucket --recursive
delete: s3://deva.bucket/ravi-tarak/IMG_20231218_134709.jpg
delete: s3://deva.bucket/ravi-tarak/
```

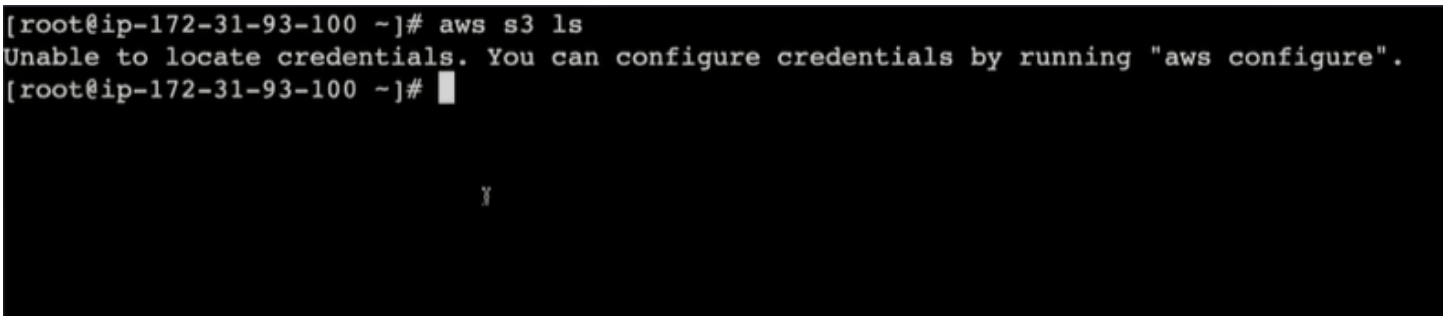
To empty or to remove the objects in the bucket: **aws s3 rm s3://bucket_name --recursive**

```
[root@ip-172-31-18-198 ~]# aws s3 rm s3://deva.bucket --recursive
delete: s3://deva.bucket/ravi-tarak/IMG_20231218_134709.jpg
delete: s3://deva.bucket/ravi-tarak/
delete: s3://deva.bucket/mahi.pdf
delete: s3://deva.bucket/ravi-tarak/Resume.docx
[root@ip-172-31-18-198 ~]#
```

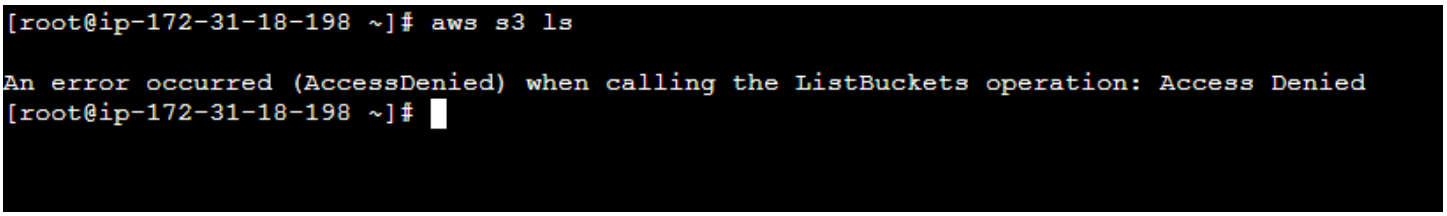
POINTS:

1) Ensure that a user is created in IAM, because the commands that you want to execute or actions that you want to perform, there must be an account in which you can perform all the actions. So the IAM user details can be configured on ec2 instance with the help of that user's security credentials (access_key and secret_key).

Error:1 - You'll get this error if there's no user in IAM.

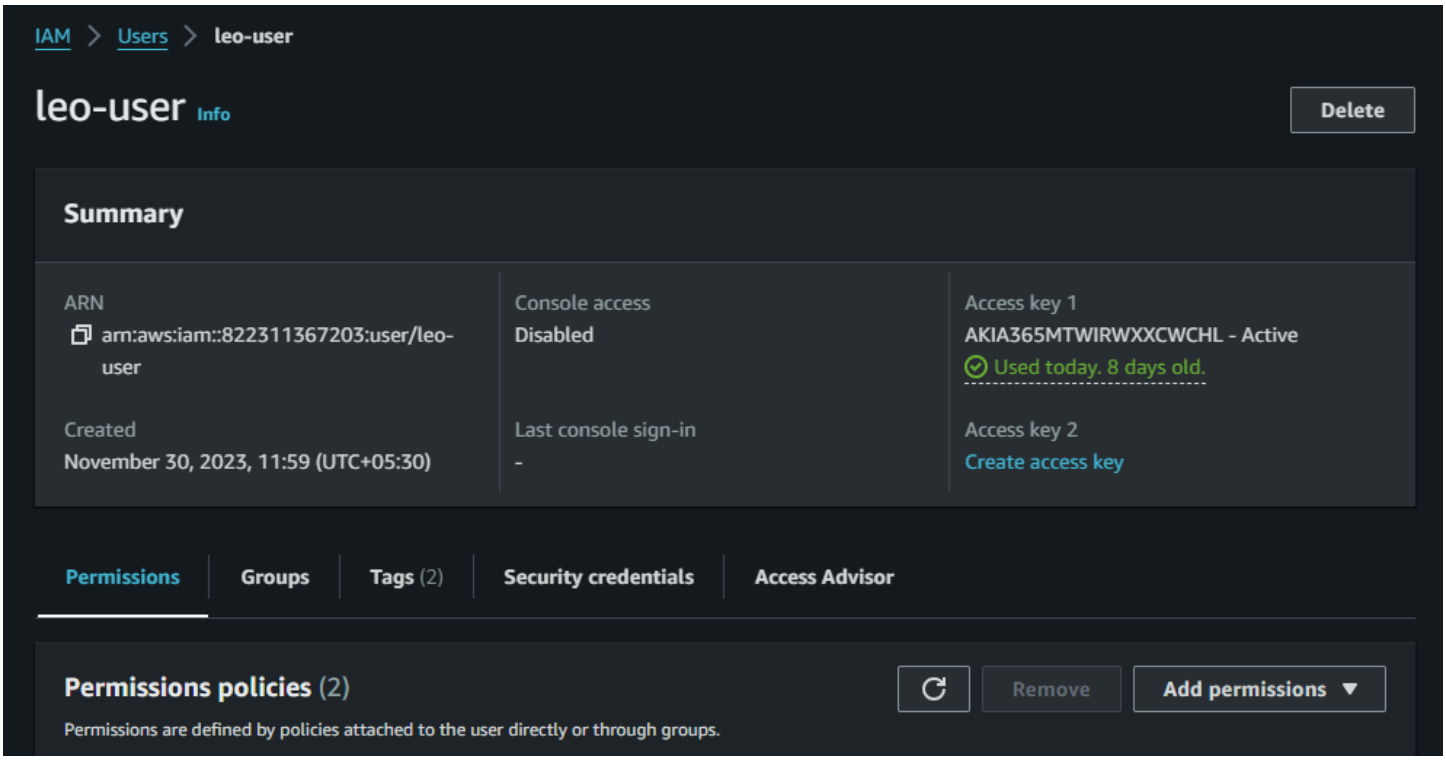


2) AmazonS3FullAccess permissions needs to be added to the user, so that the S3 actions can be performed in that account.

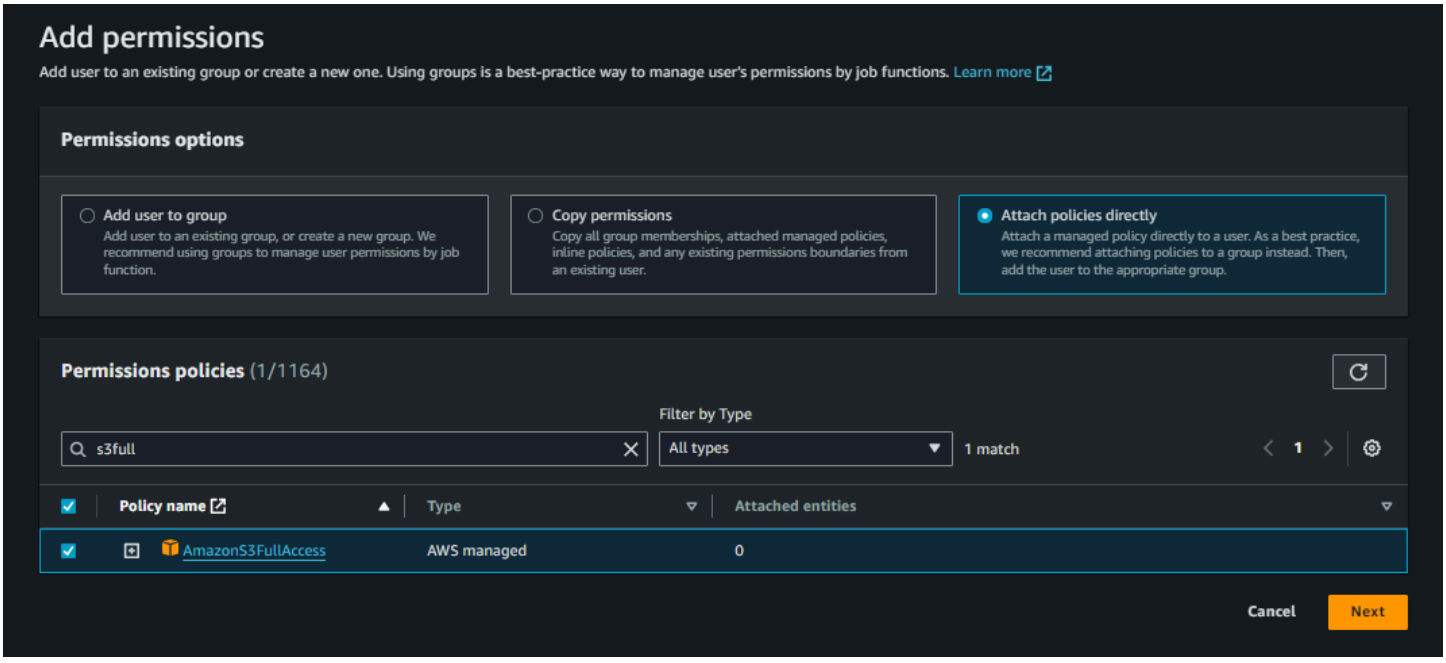


Refer to the below screenshots to add the permissions.

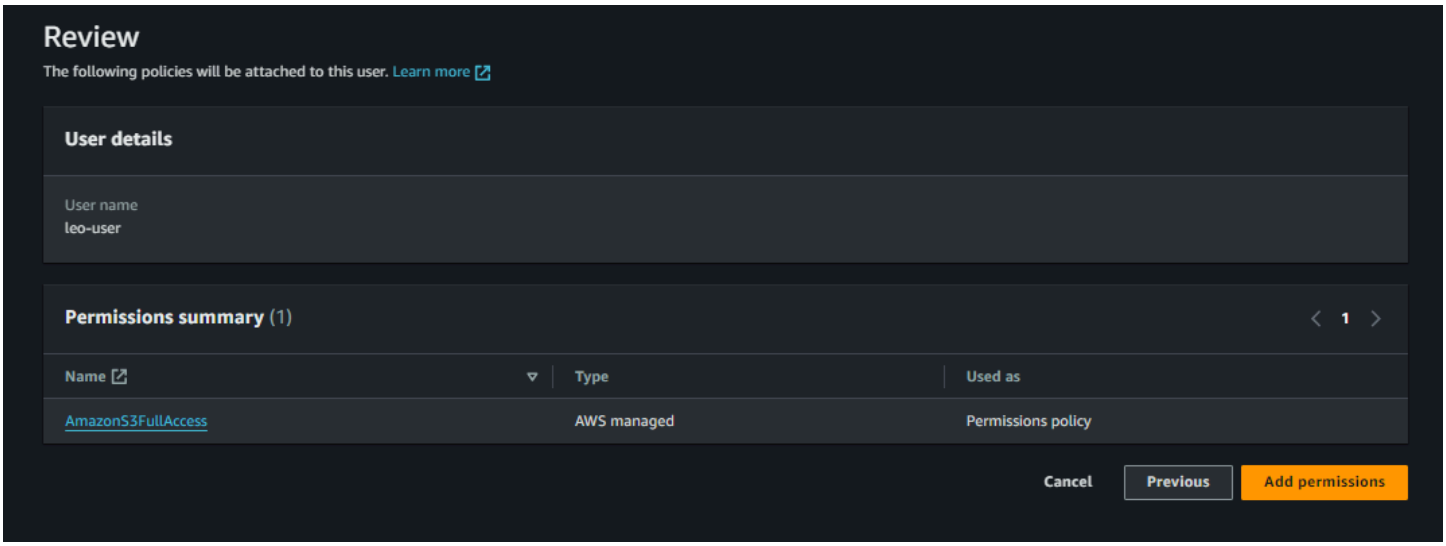
i) Search for IAM in the services, select the user and click on Add permissions on the bottom right corner of the screen as show below.



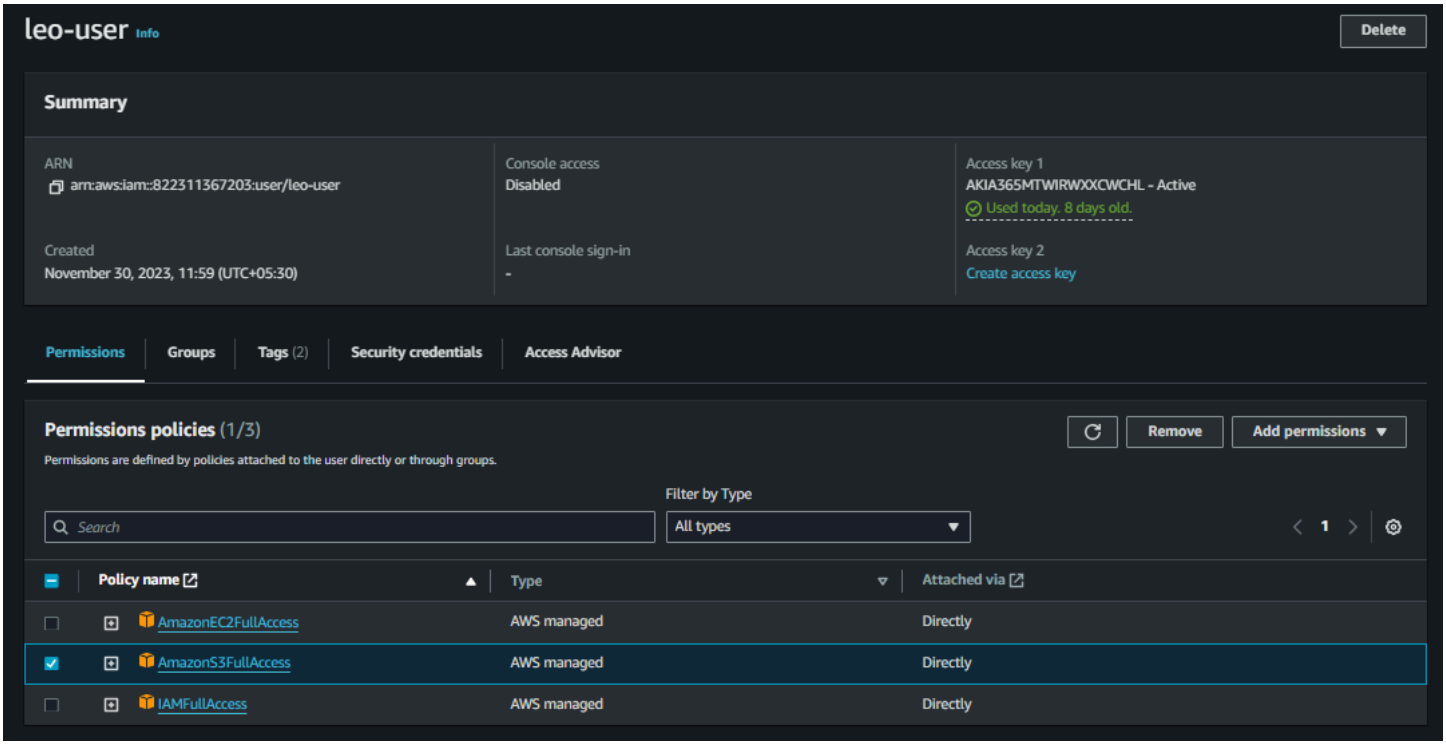
ii) Click on 'Attach policies directly' and choose the AmazonS3FullAccess and click on next.



iii) Click on Add permissions.



iv) Here you can view the permission is added to the user.



3) Now if you give the command aws s3 ls, it doesn't show anything since we don't have any buckets.

```
Last login: Thu Dec 21 13:01:04 2023 from ec2-18-206-107-28.compute-1.amazonaws.com
   ,     #_
   ~\_  ####_        Amazon Linux 2
  ~~  \_#####\
  ~~     \###|       AL2 End of Life is 2025-06-30.
  ~~       \#/ ___
   ~~       V~' '->
    ~~~         /    A newer version of Amazon Linux is available!
      ~~._.   _/
         _/ _/       Amazon Linux 2023, GA and supported until 2028-03-15.
       _/m/'           https://aws.amazon.com/linux/amazon-linux-2023/

[ec2-user@ip-172-31-18-198 ~]$ sudo -i
[root@ip-172-31-18-198 ~]# aws s3 ls
[root@ip-172-31-18-198 ~]#
```

Creation of S3 bucket using terraform code:

1) Terraform code

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_s3_bucket" "salaar" {
bucket = "deva.terraform.bucket"
}
```

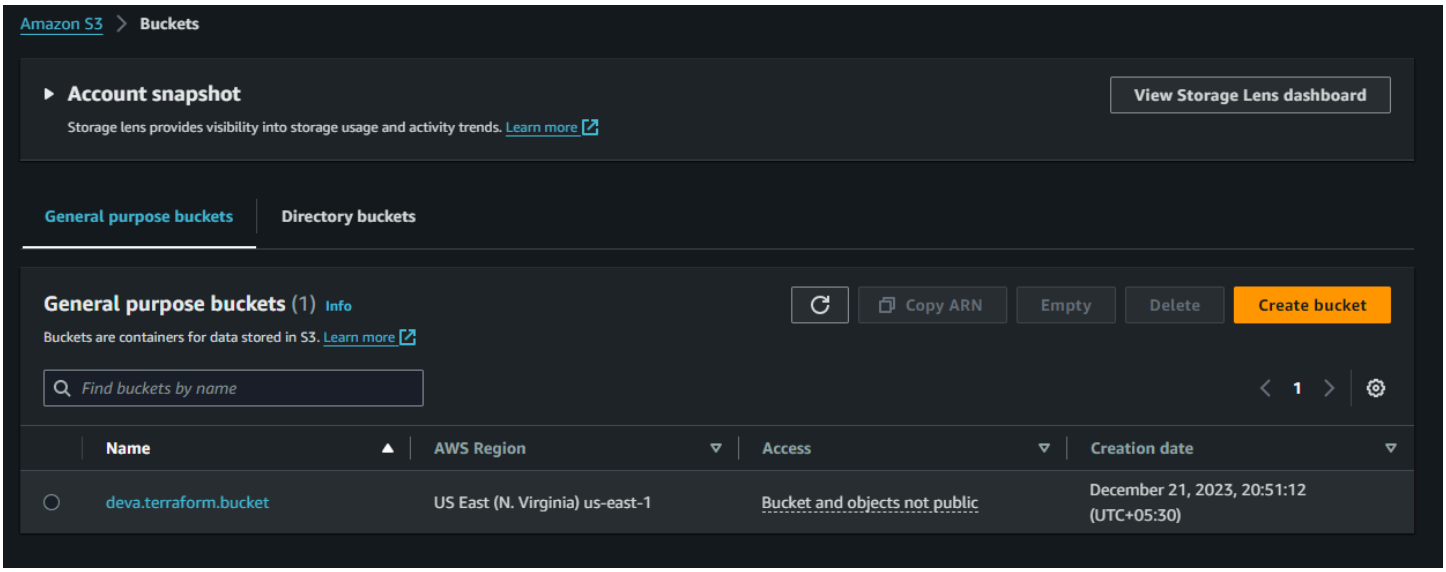
2) Terraform plan

```
Plan: 1 to add, 0 to change, 3 to destroy.


Note: You didn't use the -out option to save this plan,
apply" now.
[root@ip-172-31-18-198 ~]#
```

3) Execution result of above terraform plan. We had a code in the .tf file earlier for users, hence after the code is deleted and a code is written to create a bucket we got the result as destroying and creating as shown below.

```
Plan: 1 to add, 0 to change, 3 to destroy.
aws_iam_user.uzi[2]: Destroying... [id=tiru]
aws_iam_user.uzi[0]: Destroying... [id=mahi]
aws_iam_user.uzi[1]: Destroying... [id=ravi]
aws_s3_bucket.salaar: Creating...
aws_iam_user.uzi[2]: Destruction complete after 0s
aws_iam_user.uzi[1]: Destruction complete after 0s
aws_iam_user.uzi[0]: Destruction complete after 0s
aws_s3_bucket.salaar: Still creating... [10s elapsed]
aws_s3_bucket.salaar: Creation complete after 16s [id=deva.terraform.bucket]

Apply complete! Resources: 1 added, 0 changed, 3 destroyed.
[root@ip-172-31-18-198 ~]#
```

4) The bucket with **'deva.terraform.bucket'** has been created as mentioned in the code.



Enabling the ACL (access control list) and versioning for the bucket.

1) Terraform code

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRWXXCWCHL"
secret_key = "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"
}

resource "aws_s3_bucket" "one" {
bucket = "salaar.terraform.bucket"
}

resource "aws_s3_bucket_ownership_controls" "two" {
bucket = aws_s3_bucket.one.id
rule {
object_ownership = "BucketOwnerPreferred"
}
}

resource "aws_s3_bucket_acl" "three" {
depends_on = [aws_s3_bucket_ownership_controls.two]
bucket = aws_s3_bucket.one.id
acl = "private"
}
```

```
resource "aws_s3_bucket_versioning" "four" {
bucket = aws_s3_bucket.one.id
versioning_configuration {
status = "Enabled"
}
}
```

2) Terraform plan

```
[root@ip-172-31-94-245 ~]# terraform plan

Terraform used the selected providers to generate the following execution plan.
  + create

Terraform will perform the following actions:

  # aws_s3_bucket.one will be created
  + resource "aws_s3_bucket" "one" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "salaar.terraform.bucket"
      + bucket_domain_name          = (known after apply)
      + bucket_prefix               = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
```

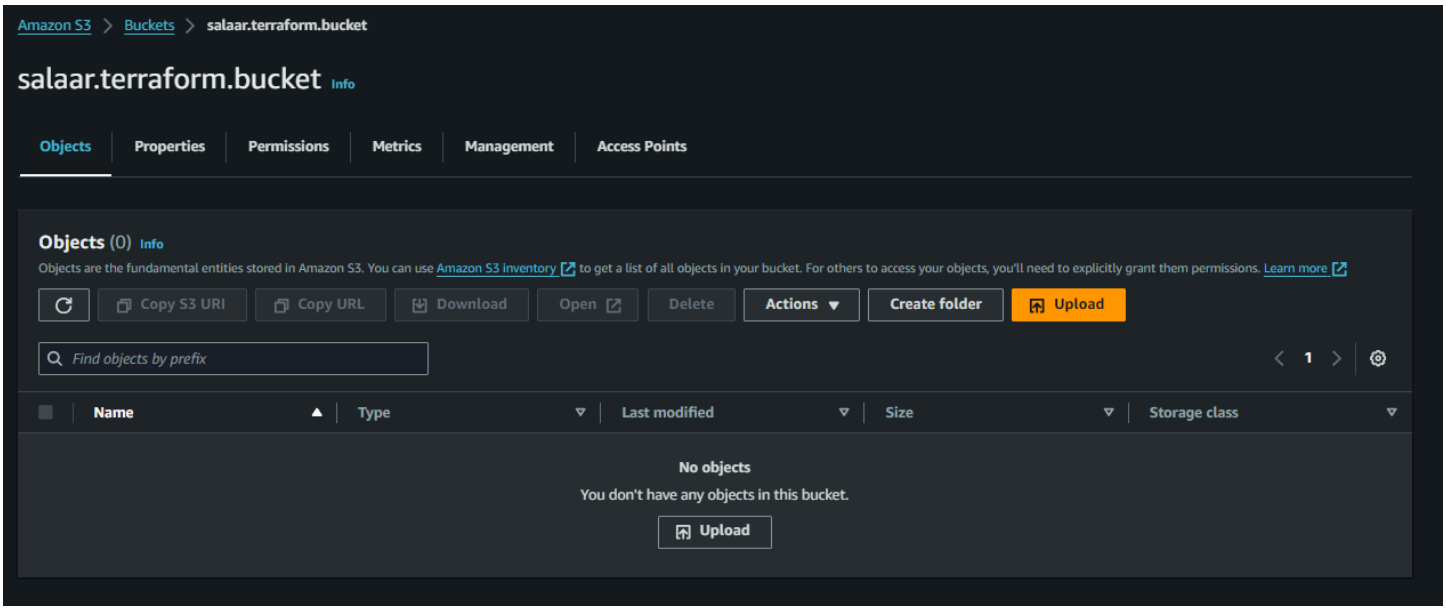
```
  # aws_s3_bucket_acl.three will be created
  + resource "aws_s3_bucket_acl" "three" {
      + acl    = "private"
      + bucket = "aws_s3_bucket.one.id"
      + id     = (known after apply)
    }

  # aws_s3_bucket_ownership_controls.two will be created
  + resource "aws_s3_bucket_ownership_controls" "two" {
      + bucket = "aws_s3_bucket.one.id"
      + id     = (known after apply)

      + rule {
          + object_ownership = "BucketOwnerPreferred"
        }
    }

  # aws_s3_bucket_versioning.four will be created
  + resource "aws_s3_bucket_versioning" "four" {
      + bucket = "aws_s3_bucket.one.id"
      + id     = (known after apply)

      + versioning_configuration {
```

```
Plan: 4 to add, 0 to change, 0 to destroy.

Warning: Quoted references are deprecated

  on main.tf line 19, in resource "aws_s3_bucket_acl" "three":
  19: depends_on = ["aws_s3_bucket_ownership_controls.two"]

In this context, references are expected literally rather than in quotes. Terraform 0.11 and earlier required quotes, but quoted references
are now deprecated and will be removed in a future version of Terraform. Remove the quotes surrounding this reference to silence this warning.


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.
```

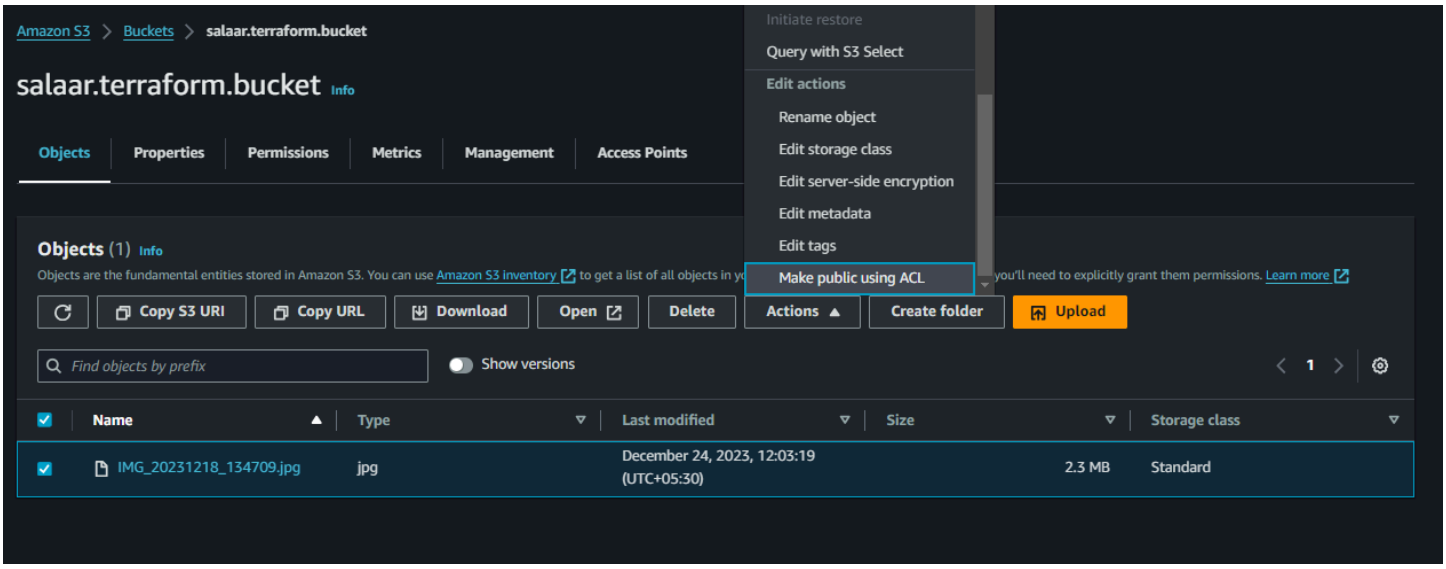
3) Execution result after terraform apply.

```
Plan: 4 to add, 0 to change, 0 to destroy.
aws_s3_bucket.one: Creating...
aws_s3_bucket.one: Still creating... [10s elapsed]
aws_s3_bucket.one: Creation complete after 16s [id=salaar.terraform.bucket]
aws_s3_bucket_versioning.four: Creating...
aws_s3_bucket_ownership_controls.two: Creating...
aws_s3_bucket_ownership_controls.two: Creation complete after 0s [id=salaar.terraform.bucket]
aws_s3_bucket_acl.three: Creating...
aws_s3_bucket_acl.three: Creation complete after 0s [id=salaar.terraform.bucket,private]
aws_s3_bucket_versioning.four: Creation complete after 3s [id=salaar.terraform.bucket]

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

4) The below image shows that the versioning hasn't enabled "**show versions**" isn't available.



5) Here you can see that the versioning and ACL options have been enabled, "**show versions**" and "**Make public using ACL**" is available.



This is how the versioning and ACL can be enabled for a bucket.

How the application log files, artifacts after the build (war/jar files) are stored in S3 bucket through Jenkins?

Earlier we used Nexus in order to store the data of an application like app log files, war files etc, after the build is success in Jenkins for future reference. In addition to that we can also use the S3 buckets because we've a lot of advantages with S3 and it differs from organization.

AIM: Here we're building a job in Jenkins (getting source code from GitHub to build with Maven) and ensuring whether the artifacts after the build are getting stored in S3 buckets or not.

Steps to integrate S3 with Jenkins:

1) Launch an EC2 instance with t2.micro, with/without keypair, 8GiB volume, security group with SSH and 8080 to access Jenkins.

2) Installation of Jenkins Git and Maven.

a) **Jenkins**:

sudo wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
amazon-linux-extras install java-openjdk11 -y
yum install jenkins -y
systemctl start jenkins
systemctl status jenkins

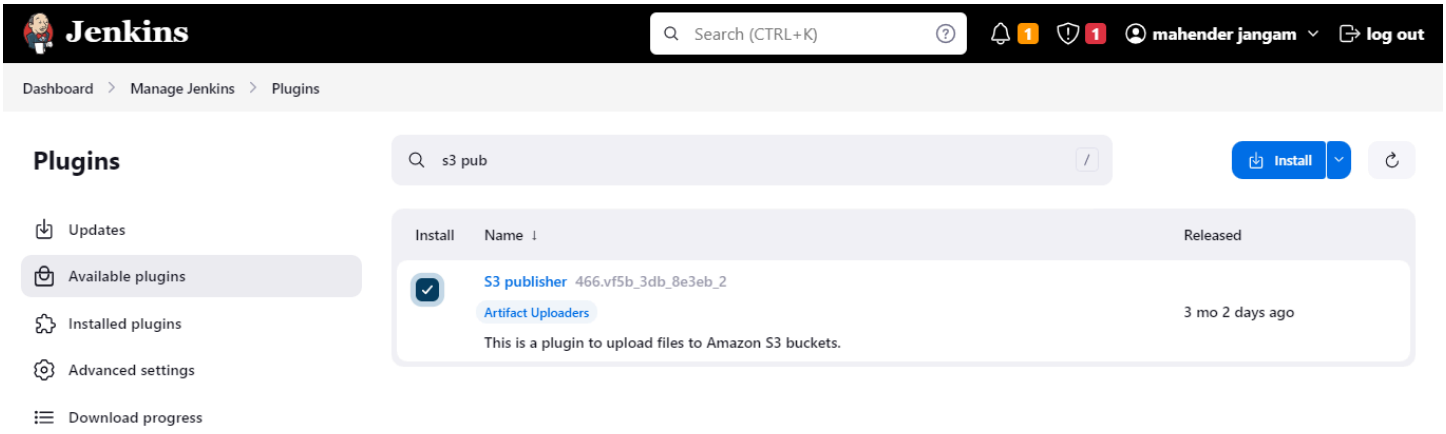
b) **Git**: yum install git -y

c) **Maven**: yum install java-1.8.0-openjdk -y and yum install maven -y

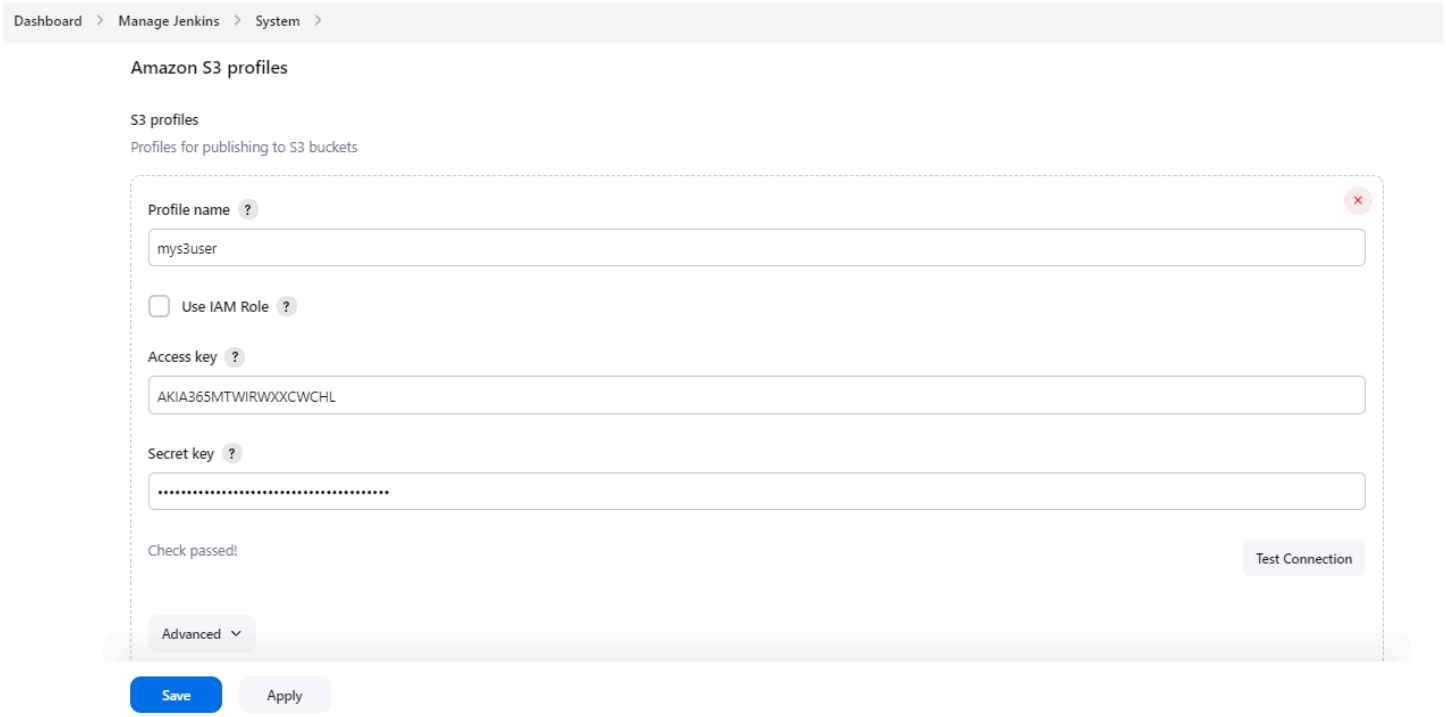
You can also install git, java1.8.0 and maven at the same time with the below command.

yum install git java-1.8.0-openjdk maven -y

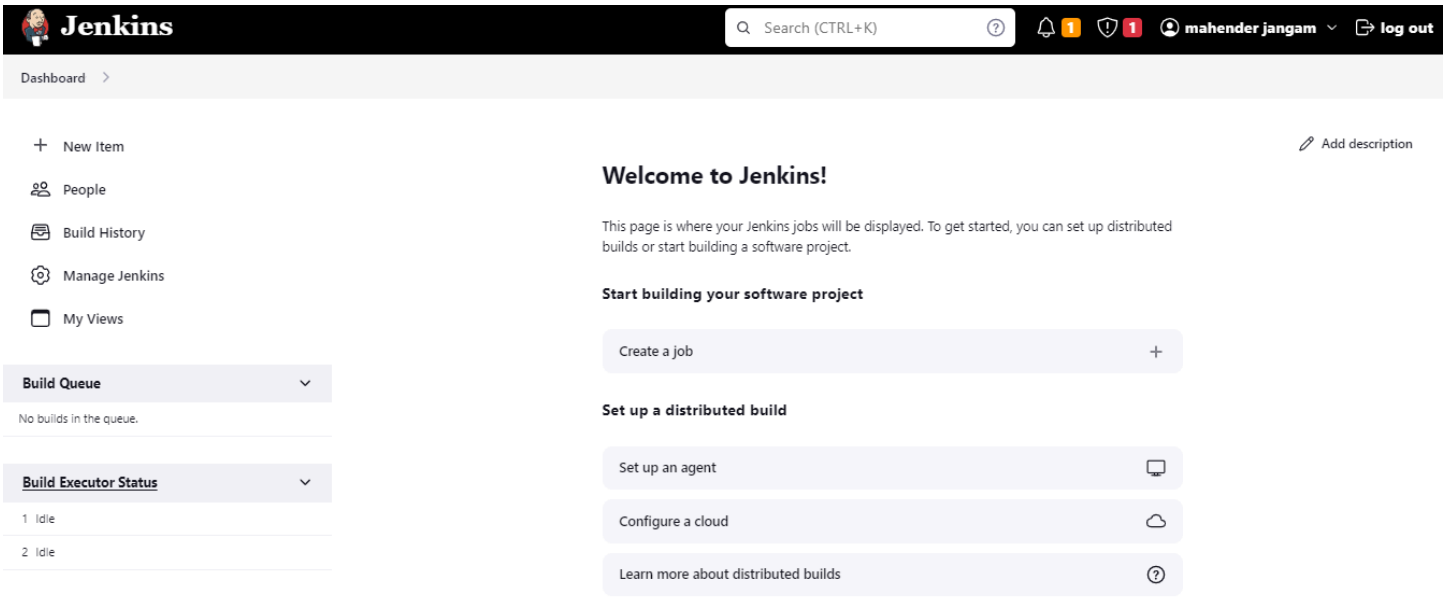
3) Install a plugin in Jenkins (**Jenkins Dashboard → Manage Jenkins → Plugins → Available Jenkins → S3-publisher**).



4) Configure S3 with Jenkins (**Jenkins dashboard → Manage jenkins → System → Amazon S3 profiles → Any profile name → Access Key → Secret Key**)

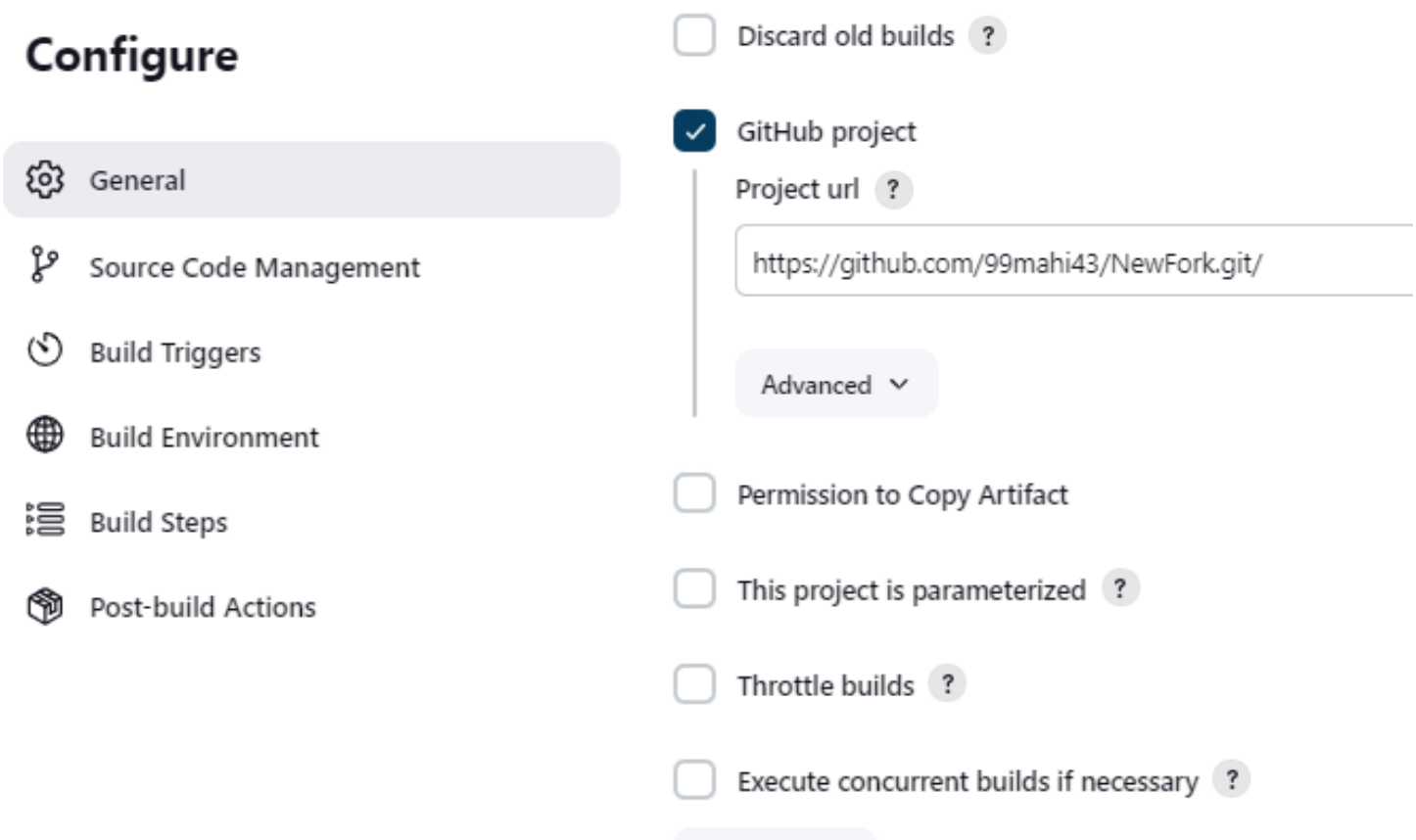


5) Go to dashboard and click on '**New item**' on the top left or click on '**Create a job**' in the mid of the screen and select any job type on the next step.

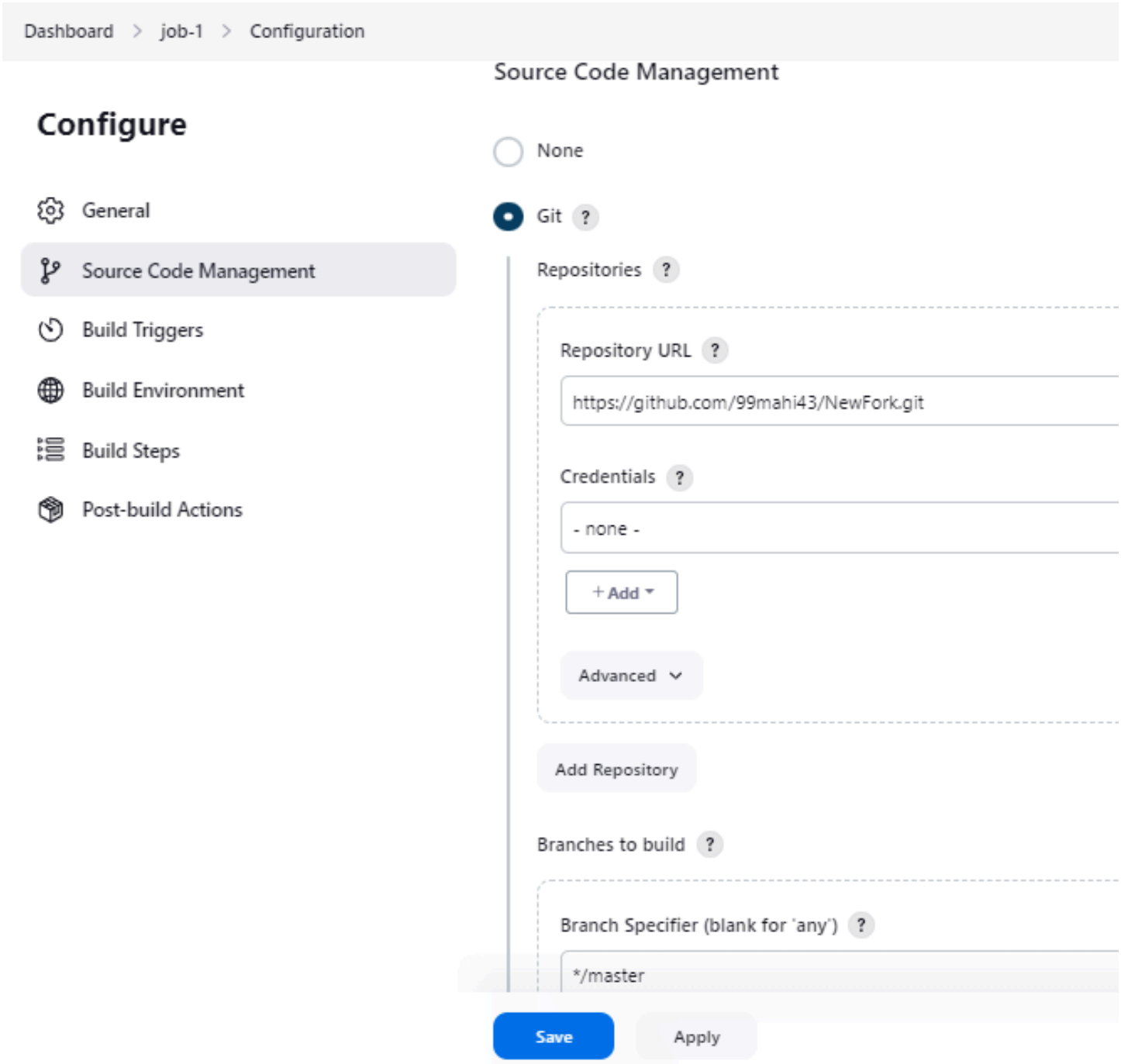


6) The below set of images represent the procedure of building a job in Jenkins.

a) Getting the source code from the GitHub.



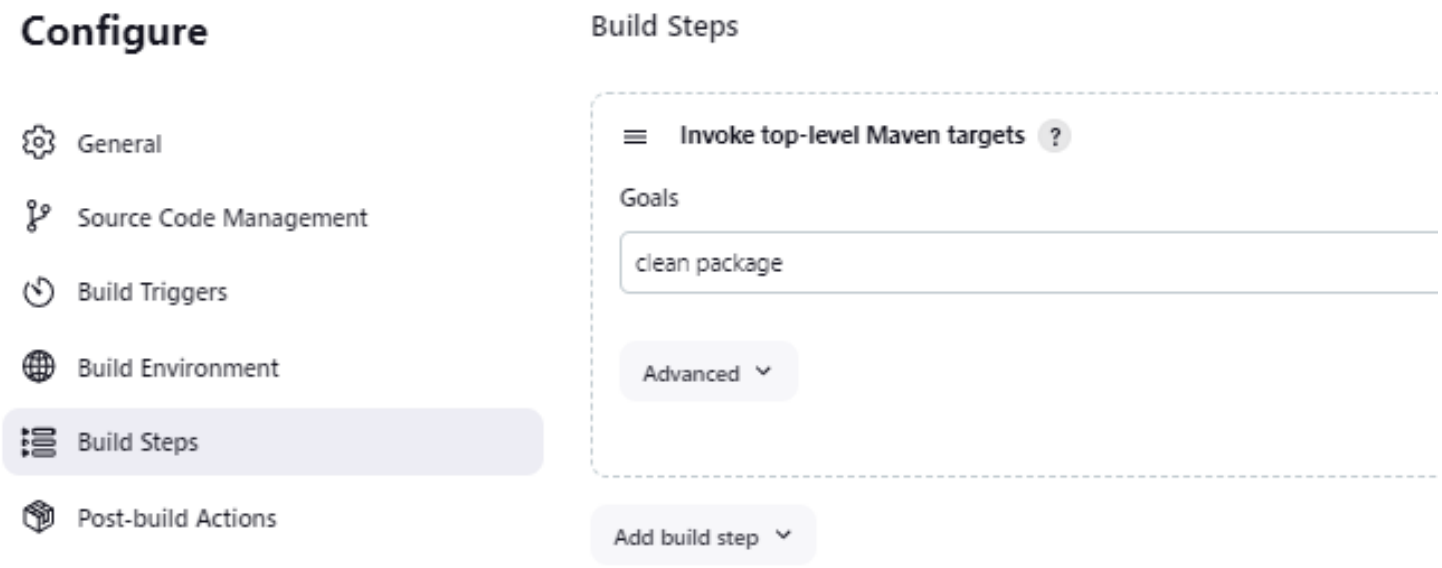
b) Mentioning the URL of the source code from GitHub.



c) The reason why we mention the ‘clean package’ goal.

To ensure a consistent and reliable build process, starting with a clean workspace for each build and producing the packaged application as the final output.

The detailed explanation is mentioned below.



Clean Goal (clean): This goal is used to clean up the project by deleting the target directory. The target directory is where Maven stores the compiled bytecode, JARs, and other build artifacts. Running the clean goal ensures that you start with a clean slate for each build, preventing any artifacts from previous builds from interfering with the current one.

Package Goal (package): This goal is used to compile the source code, run the unit tests, and package the application into its distributable format, such as a JAR file. The package goal is often used in conjunction with other goals like compile and test. When you select "Invoke top-level Maven targets" in Jenkins and specify "clean package" as the Maven goals, you are instructing Jenkins to execute these two goals in sequence. This ensures that the project is cleaned first, and then the source code is compiled, tests are run, and the application is packaged into its final distributable format.

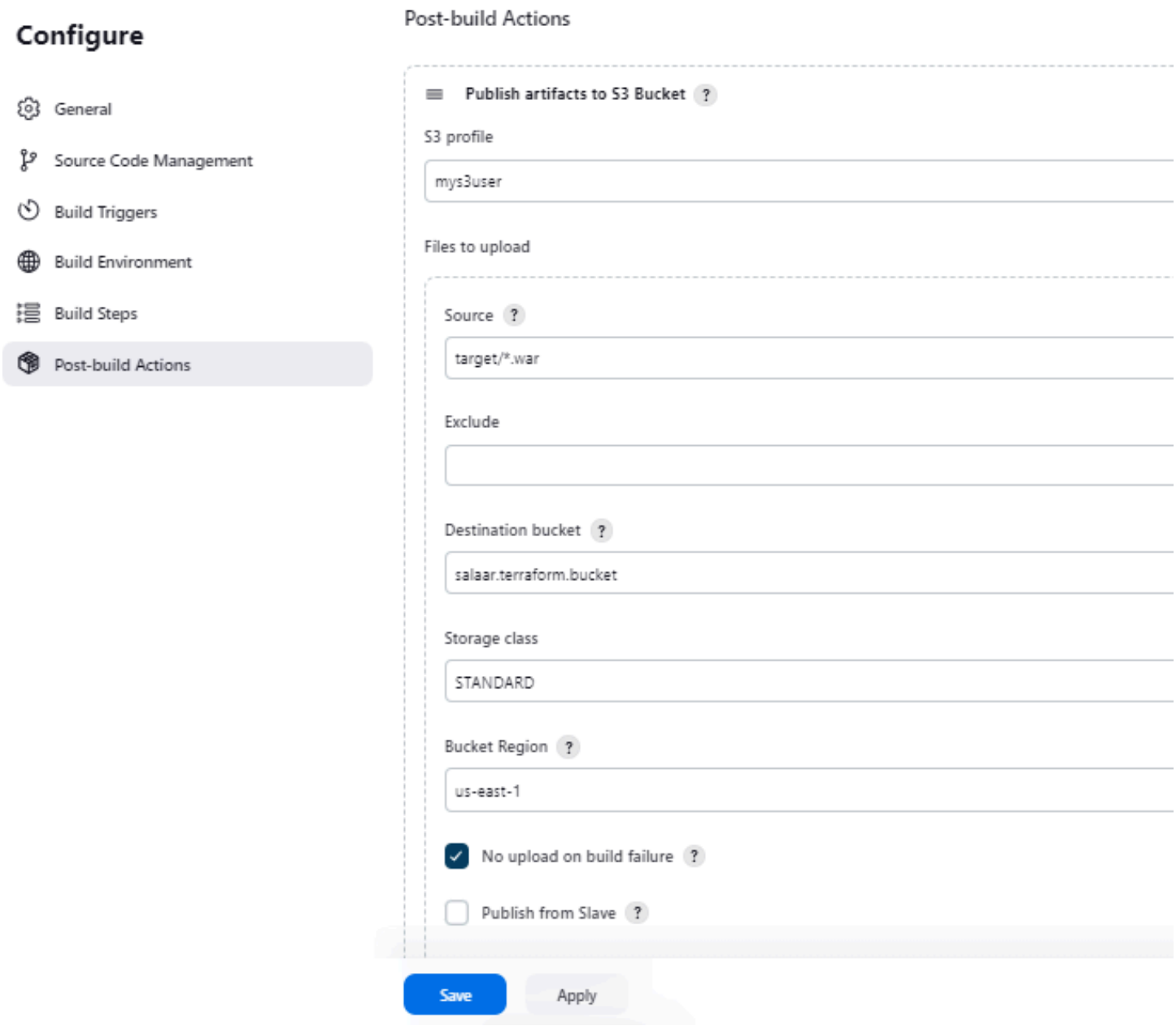
Here's a breakdown of what happens:

Clean (clean): Deletes the target directory.

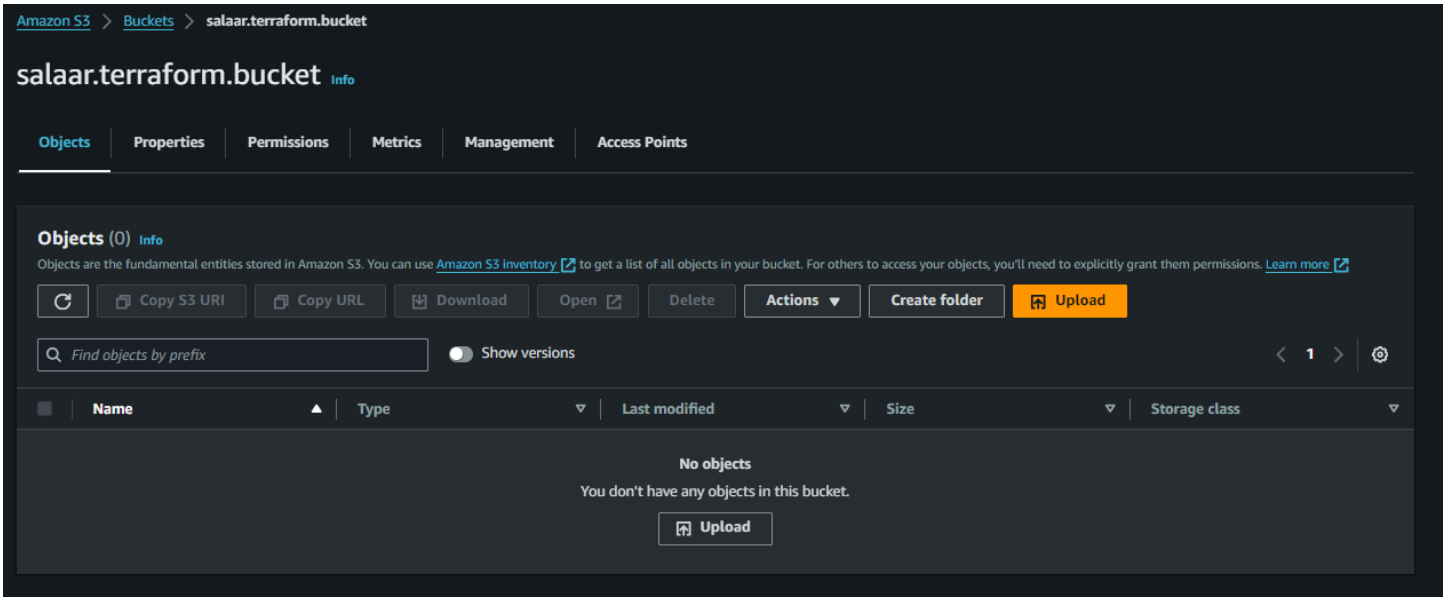
Package (package): Compiles the source code, runs tests, and packages the application.

By using "clean package" in Jenkins, you ensure a consistent and reliable build process, starting with a clean workspace for each build and producing the packaged application as the final output. This is a common practice to avoid issues that might arise from leftover artifacts in the build directory and to ensure that the build is reproducible.

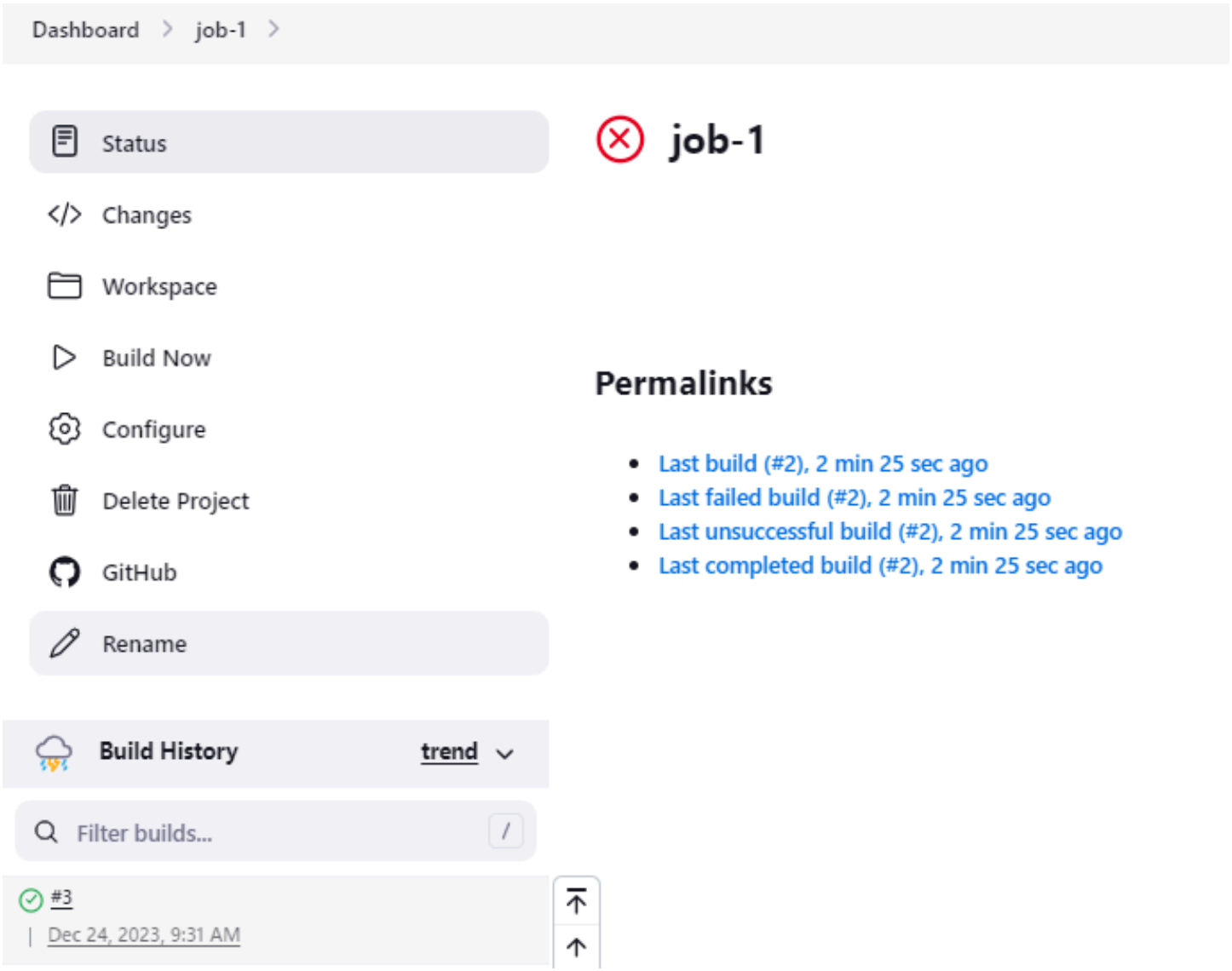
d) Configuring the S3 bucket in the job to get the files stored.



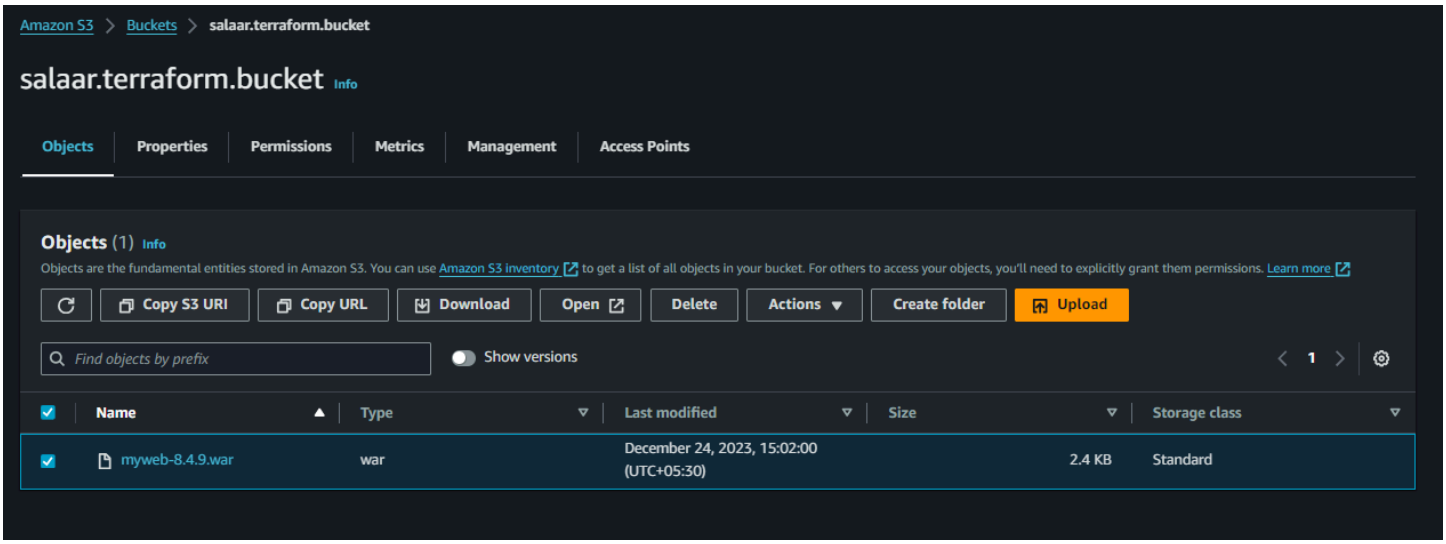
7) Here we don't see any war files before the job is built in jenkins.



8) Now we've got the artifacts stored in the S3 bucket after the job is build in Jenkins.

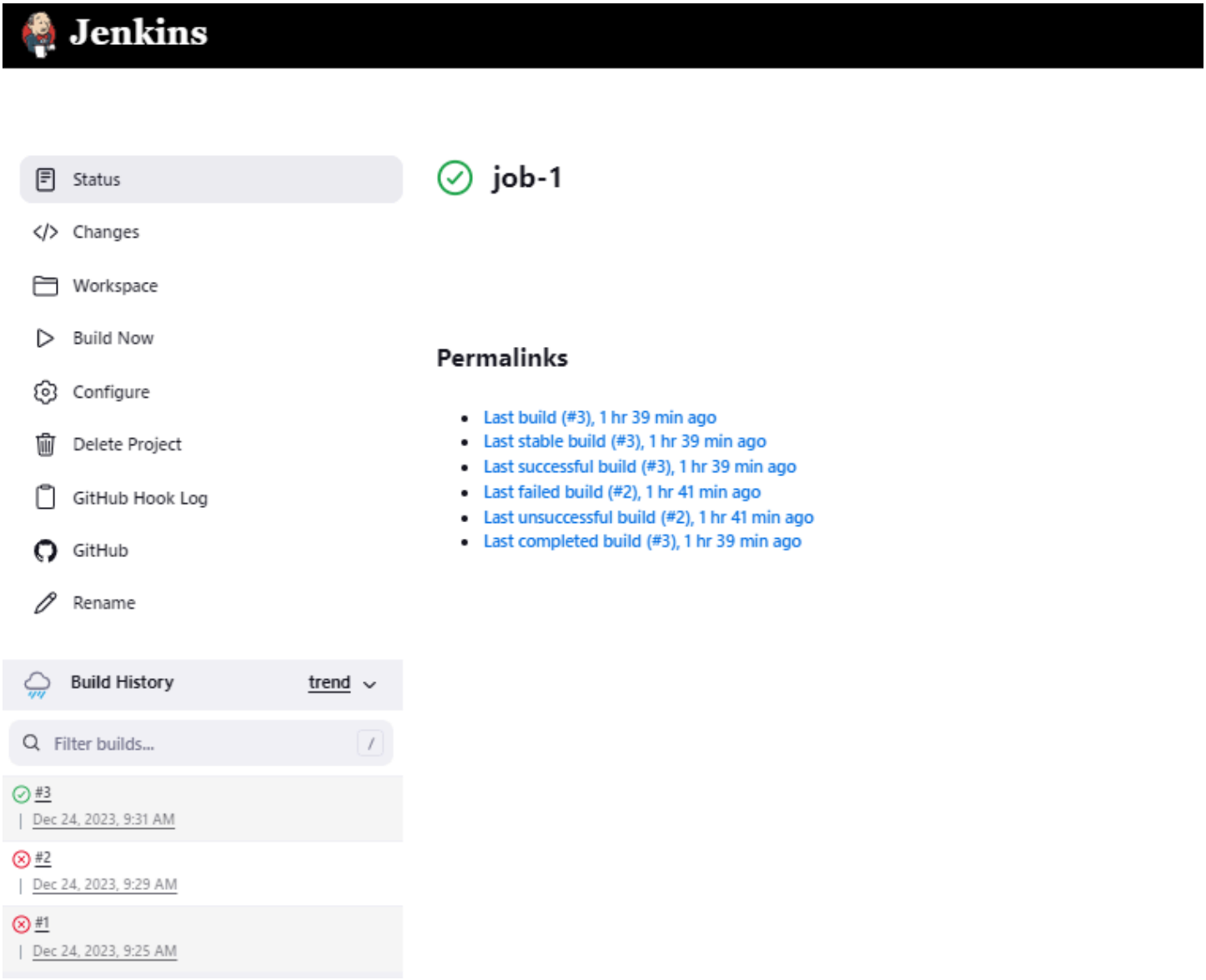


9) The artifact (.war type) is stored in the bucket of which the user we mentioned above.

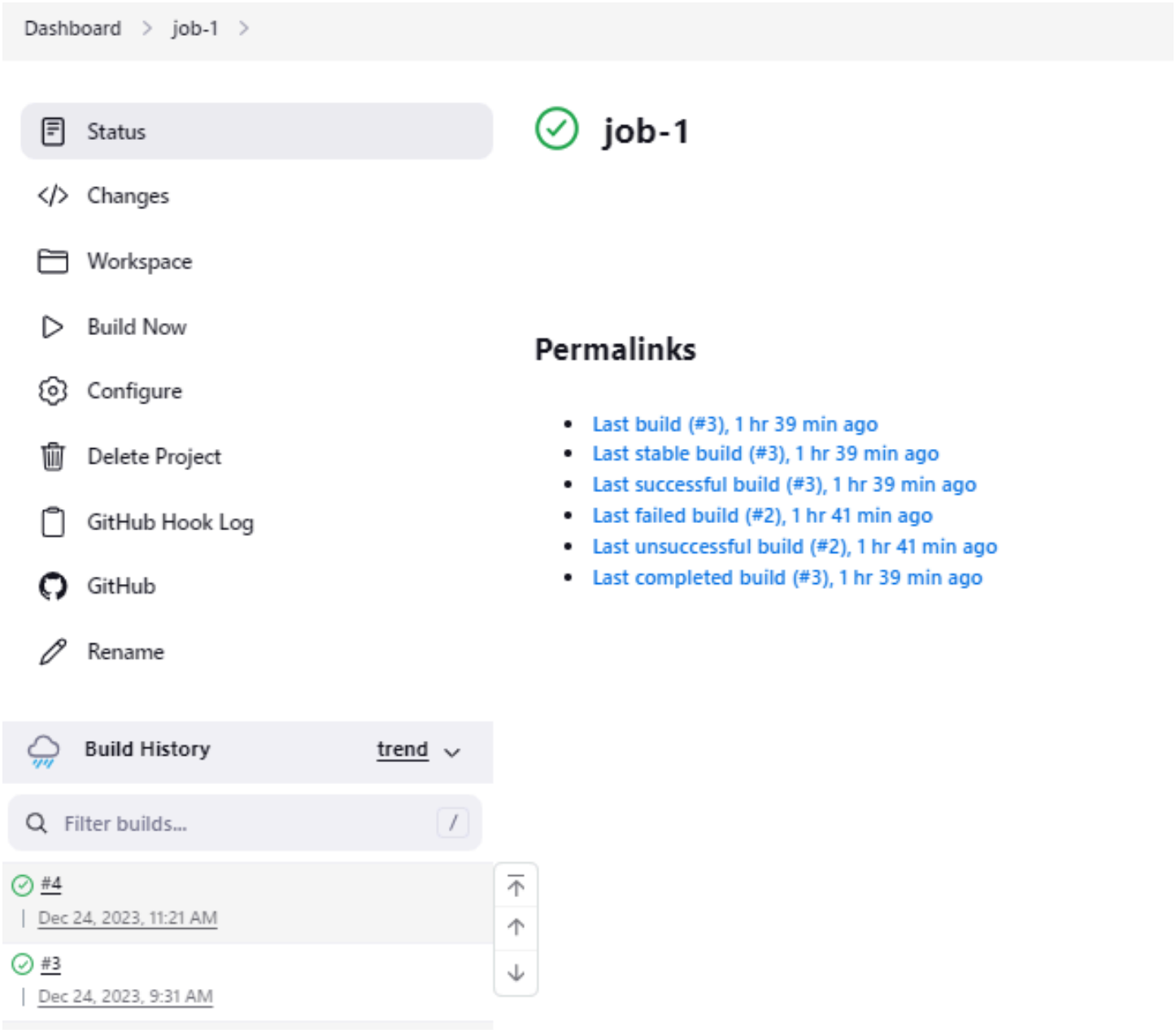


In case if the developers make any changes to the source and commit, we get the new source and the version changes from 8.4.9 to 8.5.0. If you wish you can use the Webhooks to integrate it with GitHub and Jenkins so that whenever the changes are made to the source code, it triggers the pipeline and the build will run automatically.

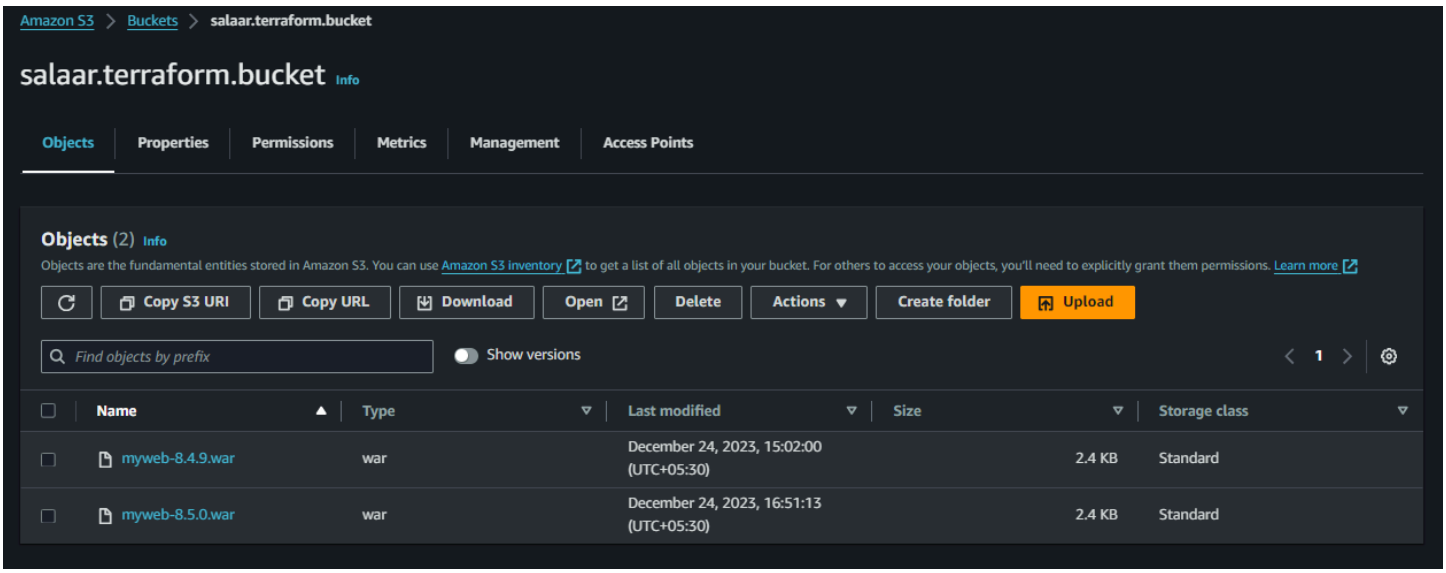
1) Here we've only 3 builds that were run manually from Jenkins.



2) Now, we've got 4 builds just by adding the Webhook to the Jenkins job and GitHub repository. I've made changes to the source code (pom.xml) and as soon I committed the changes with the help of Webhook that I've created, it trigged the free-style job and it ran automatically.



3) We've also got the war file generated as shown below.



VPC (Virtual Private Cloud)

VPC is one of the aws cloud services. It's abbreviated as Virtual Private Cloud. Usually, for example if we create instances, apart from ours we do also have many as well so here the security is less and aren't isolated. Here comes our VPC to resolve this drawback, which means we create a virtual cloud which is private and only a set of people can access.

By default, we've a vpc created by aws and we can create 4 vpc's in a single region. In a single VPC we can create 200 subnets.

IP address and it's types:

There are two types of IP addresses. The first one is IPV4 (Internet Protocol Version 4) which is a 32-bit address and the other is IPV6 (Internet Protocol Version 6) which is a 64-bit address. AWS generates IPV4 addresses and not IPV6 address whenever an instance is created and hence we take 32 in the formula.

How many IP addresses can VPC provide?

AWS provides up to a maximum of 65,536 IP addresses if a VPC is created (with a formula of 2 power 32-n). From those overall IP's, you can create subnets. With the help of CIDR block, you can mention it like 10.0.1.0/8, so based on the n value (here 8) in the cidr block range, IP's (here 256) will be directly allocated by aws from the overall limit of 65,536 IP addresses.

From 65,536 IP addresses, 5 IP's have been reserved by aws (**10.0.0.0, 10.0.0.1, 10.0.0.2, 10.0.0.3 and 10.256.256.256**) and we get 65,531 IP addresses within a range of IP's from (**10.0.0.4 to 10.256.256.255**).

CIDR block: CIDR is abbreviated as Classless Inter-Domain-Routing. It's the IP address allocation method that improves data routing efficiency on the internet. It's the place where you give the notation in order to create IP addresses in that particular subnet.

How the IP addresses are allocated by aws if a VPC is created?

AWS allocates both public and private IP addresses, the allocation mechanism for public IP addresses involves Elastic IP addresses, and for private IP addresses, it is based on the CIDR block range specified for the VPC and subnets.

The formula which is used to know how the IP addresses are allocated:

There's a formula that is used to calculate the number of IP addresses that can be provided with the n value you take. The formula is 2 power (32 - n). If the number 'n' is 17, then 2 power (32 - 17) = 2 power (15) = 32,768, these many can be created.

So the n value must be greater than or equal to 16. If the n value gets decreased then the IP addresses you get will be more than the aws limit (more than 65,536) which can't be provided. If n is 15, 2 power (32 - 15) = 2 power (17) = 131,072 which exceeds the aws limit of IP addresses that can be provided.

Ex: Let's say you've 2 subnets in a VPC. If n is 24, then the IP address would be as 10.0.1.0/24 and we'll get 256 IP addresses for subnet-1. If n is 18, then the IP address would be as 10.0.2.0/18 and we'll get 16,384 IP addresses for subnet-2.

About VPC:

1. Virtual Private Cloud is used to create a private cloud inside a public cloud.
2. This private cloud provides isolation and security for our resources and services.
3. Every VPC has it's own IP address, Subnets, Internet Gateways and Route Tables.
4. We can control access to our resources with the help of security groups and network access control lists.

Components in VPC:

1) Subnet: A subnet is a availability zone in a region. Subnets are used to divide a large networks into smaller networks. There are 2 types of subnets, one is public subnet and another is private subnet.

a) **Public Subnet:** The resources can be accessed through internet. Like if an application is running in the public subnet, then it can be accessed through internet directly.

b) **Private Subnet:** The resources running in private subnet can't be accessed directly through Internet, it needs to be accessed through load balancers or NAT Gateway which are present in public subnet.

Functionality of public and private subnets:

Let's consider an example of 3 tier architecture with front-end, backend and database.

So if a user is trying to login to facebook.com using internet then the request goes to the frontend of an application (login page) which is to the instance/server located in the public subnet. Since it's accessed through the internet directly it doesn't need any NAT gateway.

Whereas in private subnet, people can't access the backend of the application directly through the internet, if the username and password are correct in the frontend side of the application, then they'll be routed to the inside page to access the application. So in order allow the user to go to the backend of the application, the NAT gateway will allow the users and the NAT gateway is located in the public subnet.

2) Internet Gateway: An Internet gateway is a component that acts as a communicator between the VPC and internet.

If a request is sent to the VPC by the user over the internet, then the request first goes to the internet gateway and later it gets forwarded to the route table where it checks whether the user is having permission or not in order to access the page or not. The rules for incoming and outgoing traffic is mentioned in the route tables.

3) NAT Gateway: Network Address Translation gateway is used to access the resources in private subnet.

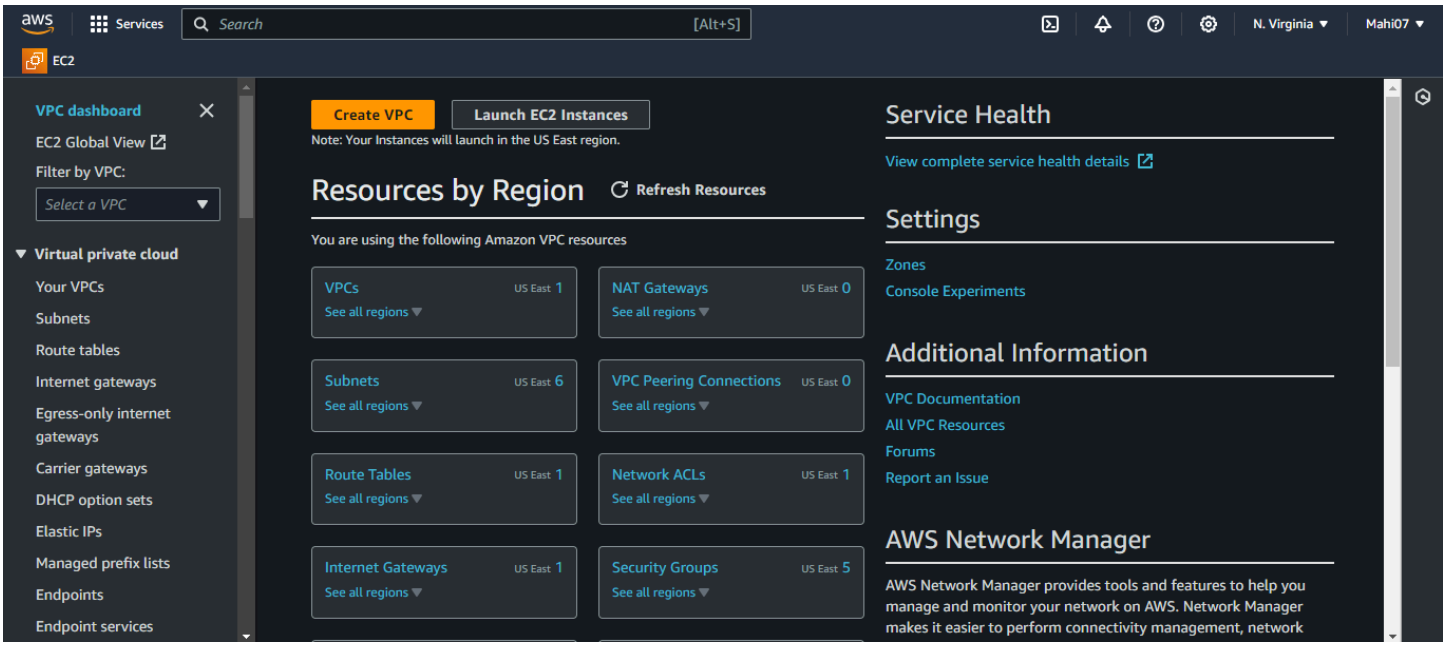
4) Route Tables: It's a set of rules, that defines how the network traffic is directed within the VPC.

5) Network Access Control Lists: A network access control list (NACL) is an optional layer of security for your VPC that acts as a firewall for controlling traffic in and out of one or more subnets.

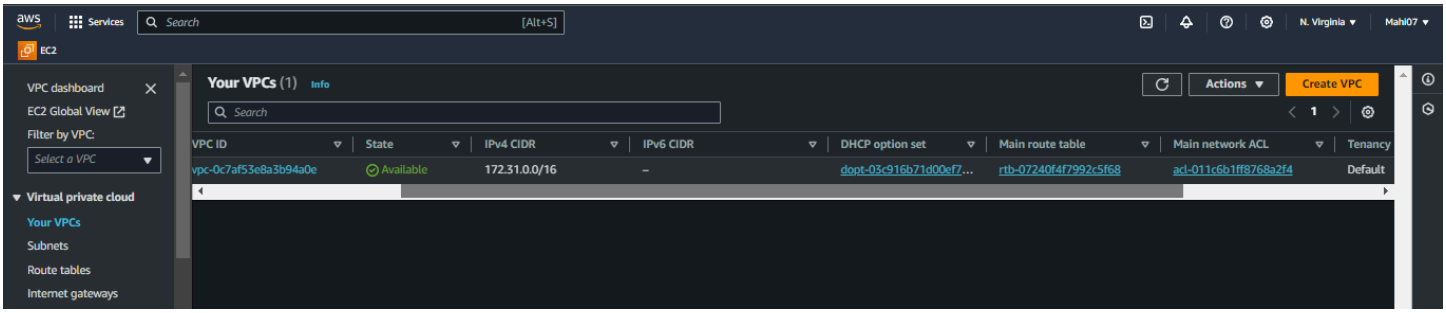
Pictorial representation of VPC creation using aws console (Automated):

This is an automated way of creation of VPC which alongside creates subnets, internet gateways, route tables etc if the option is selected as VPC and more. In case if the VPC only is selected while creation then you need to manually create subnets, internet gateways, route tables etc. If you delete the VPC alone, it deletes all the resources attached to it while creation like subnets, route tables, internet gateways.

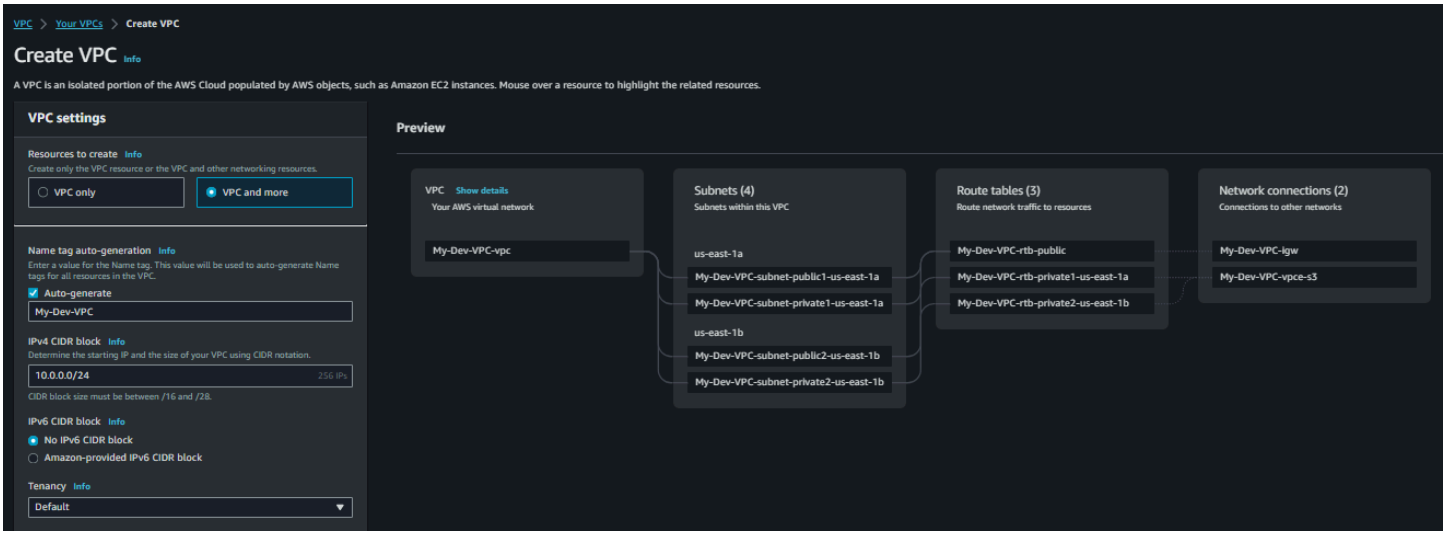
1) Go to aws console and search for VPC. You'll be landed on this page.



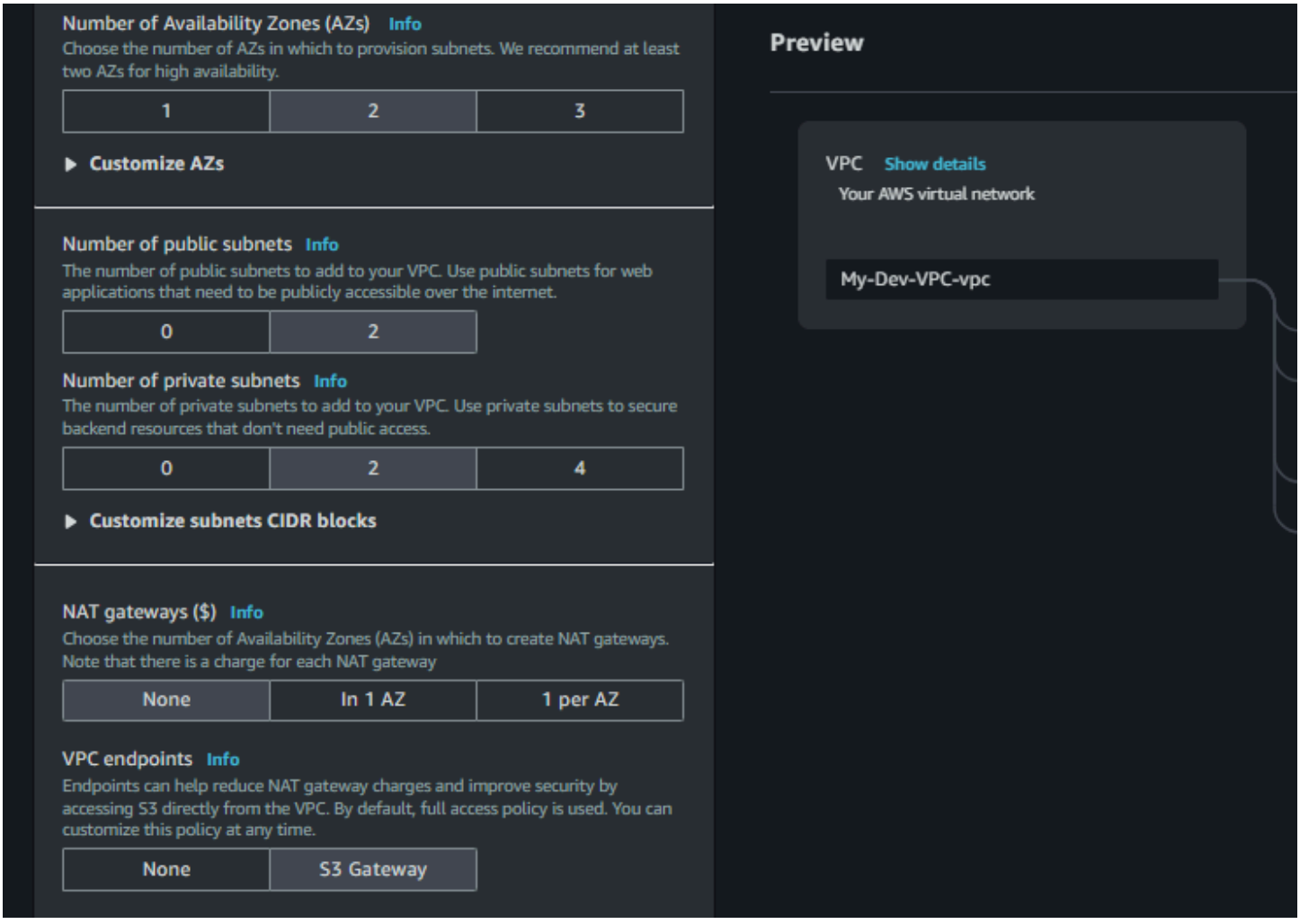
If you click on your vpc's you'll have a default one which is provided by aws.



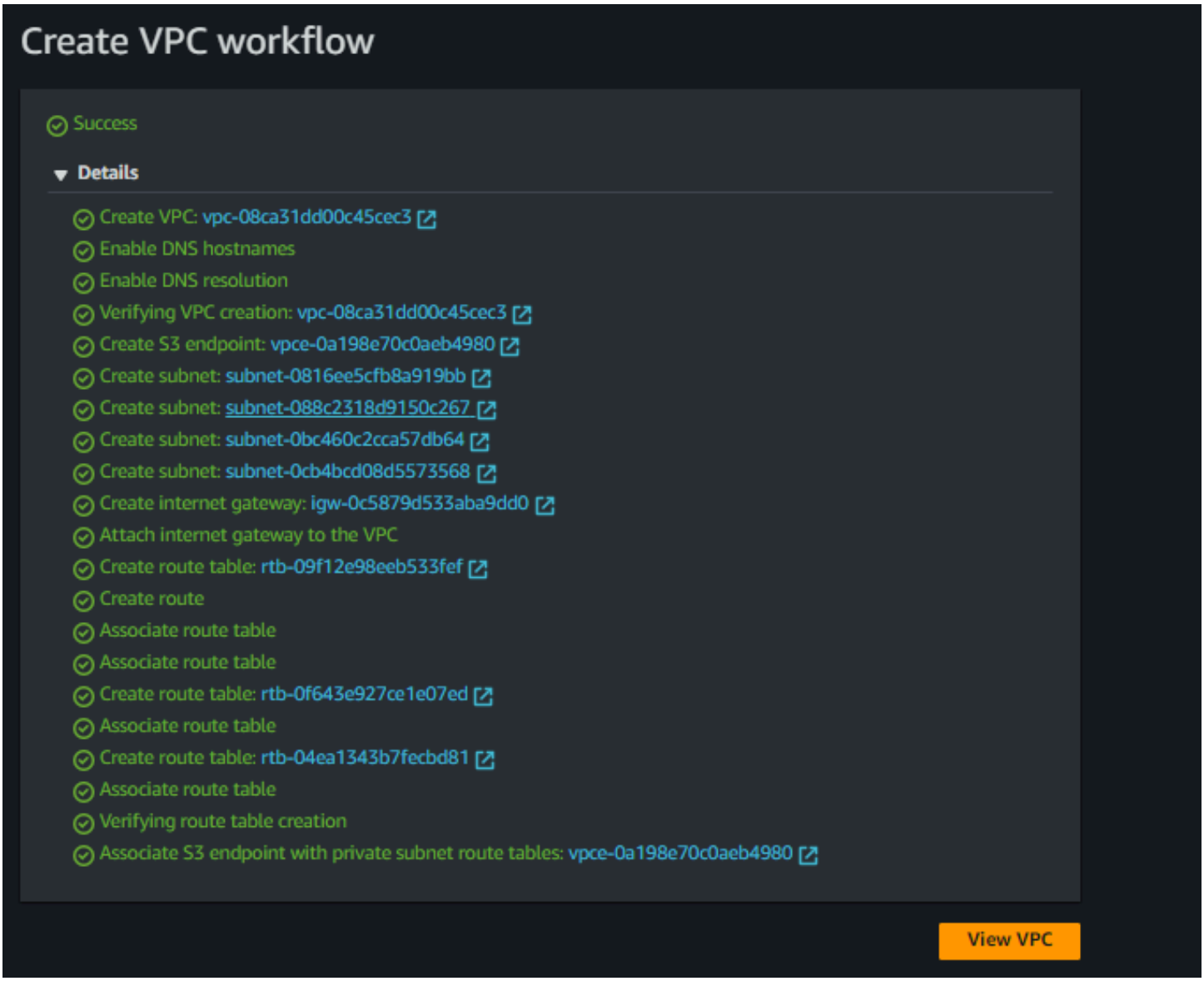
2) Click on Create VPC as show above and the whole page looks like shown below.



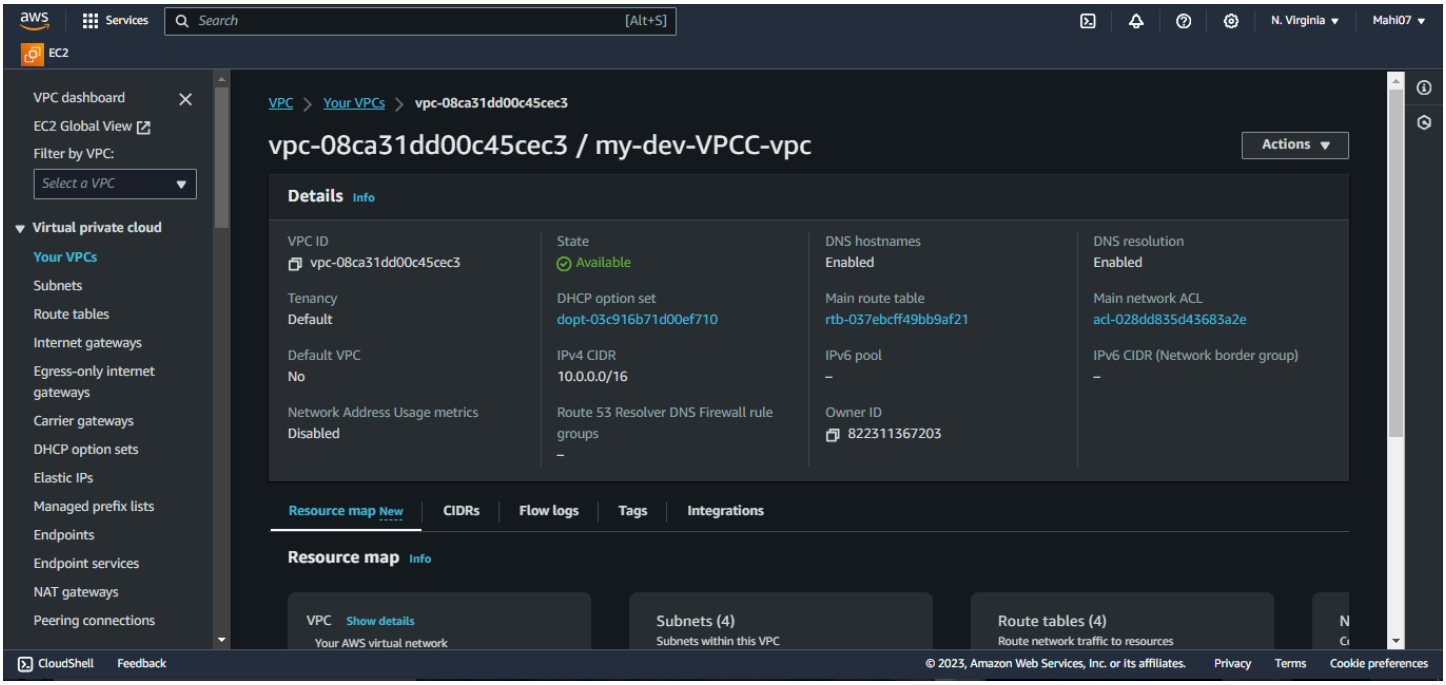
3) Give a name to the VPC under name tag auto-generation, select the cidr block range and tenancy should be default, if not there'll be billing if it's selected as dedicated, select the number of availability zones (based on that public and private subnets gets created, 1 Az = 2 subnets, 1 public and 1 private) and click on create VPC on the left bottom of the screen.



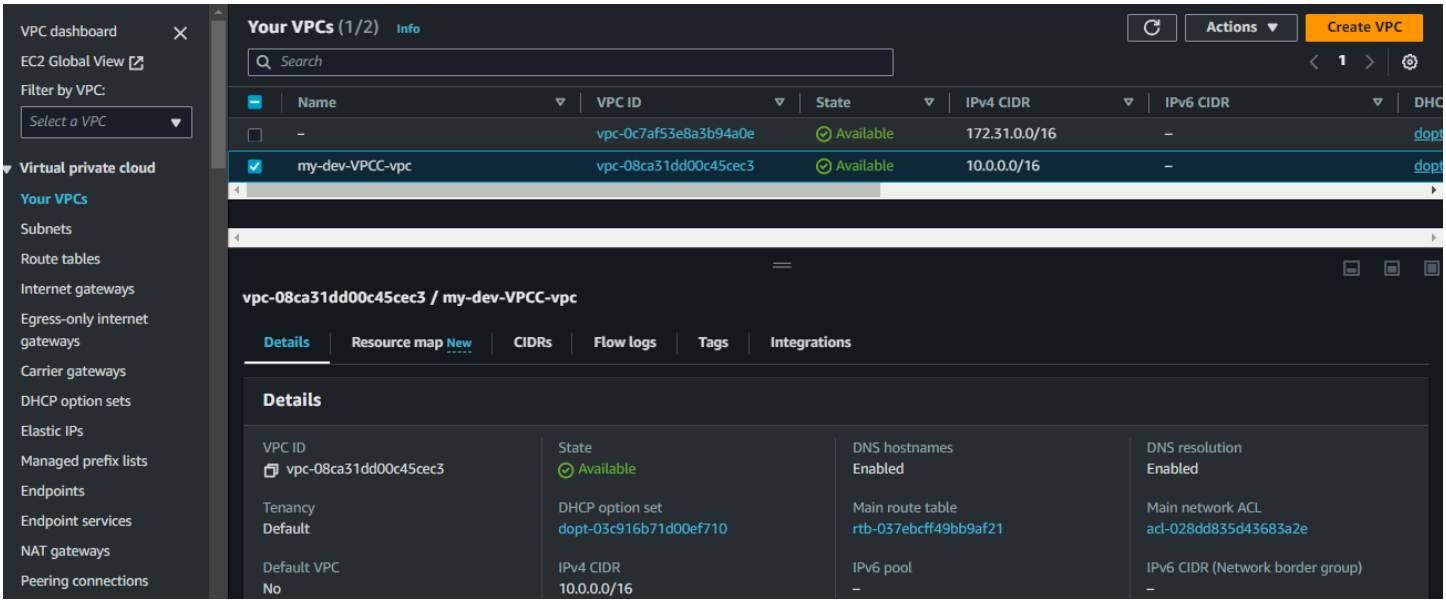
4) You'll get a page that shows the VPC is created along with all other resources that are necessary.



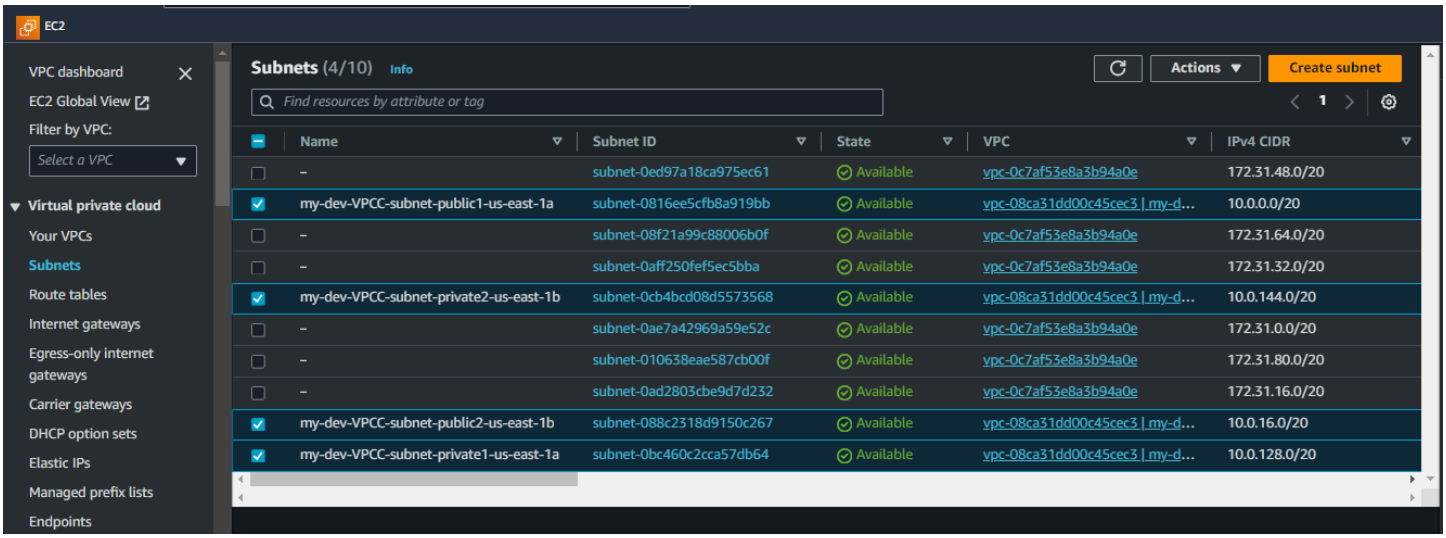
5) Click on view VPC and you'll see the below interface. You can click on subnets, route tables, internet gateways, all of them would be created.



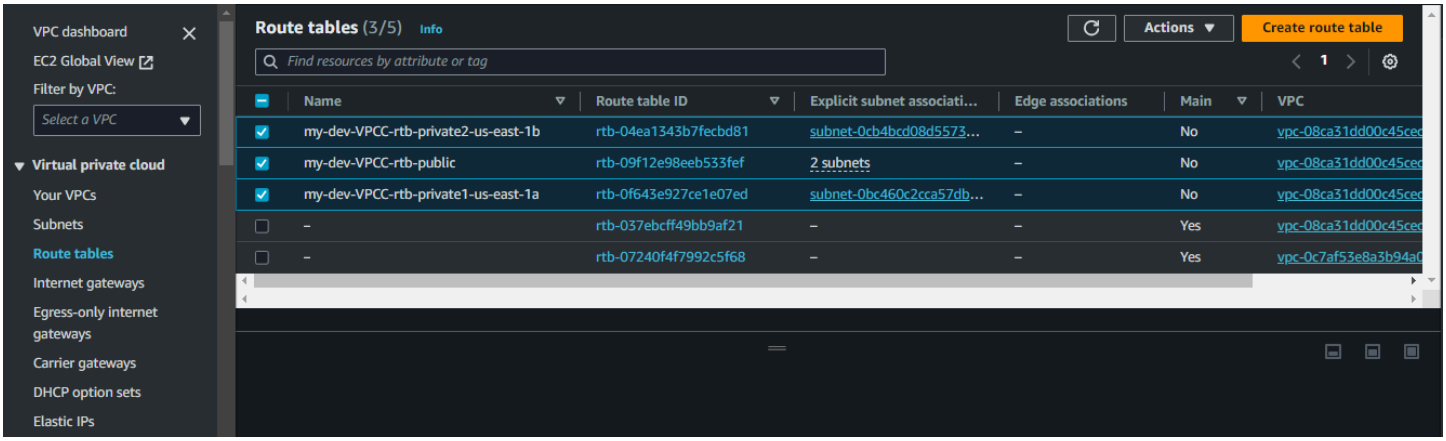
a) Your VPCs



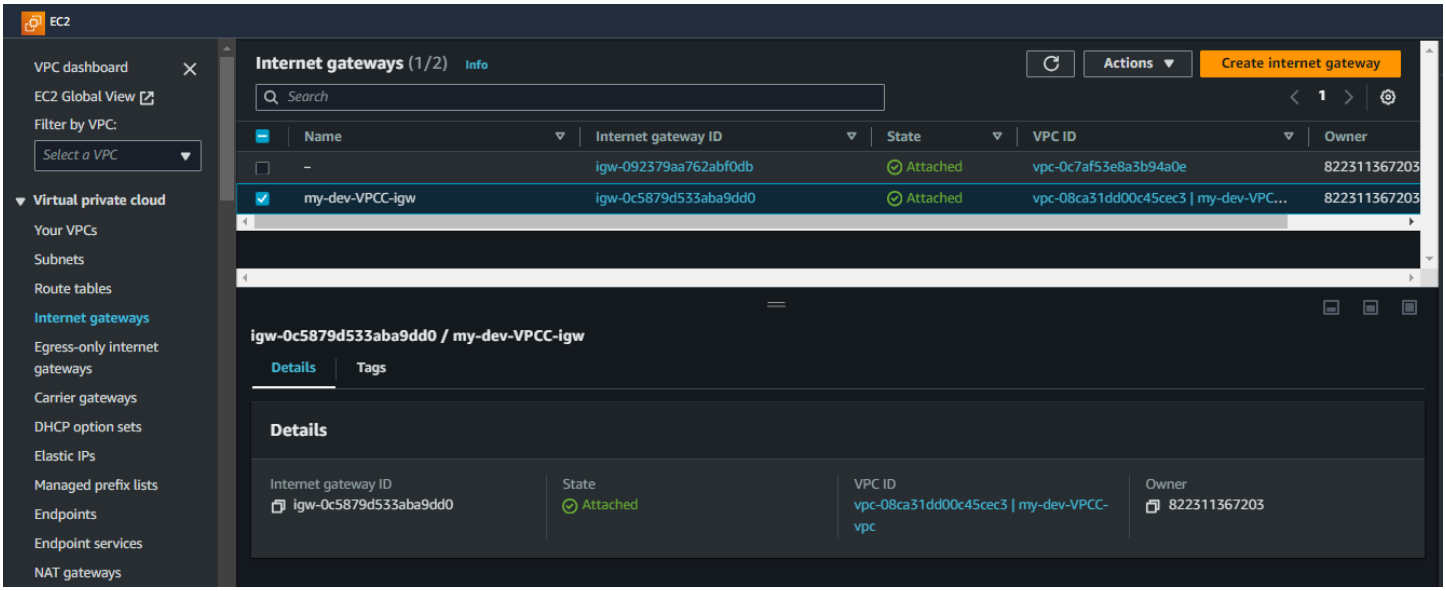
b) Subnets



c) Route tables



d) Internet gateways



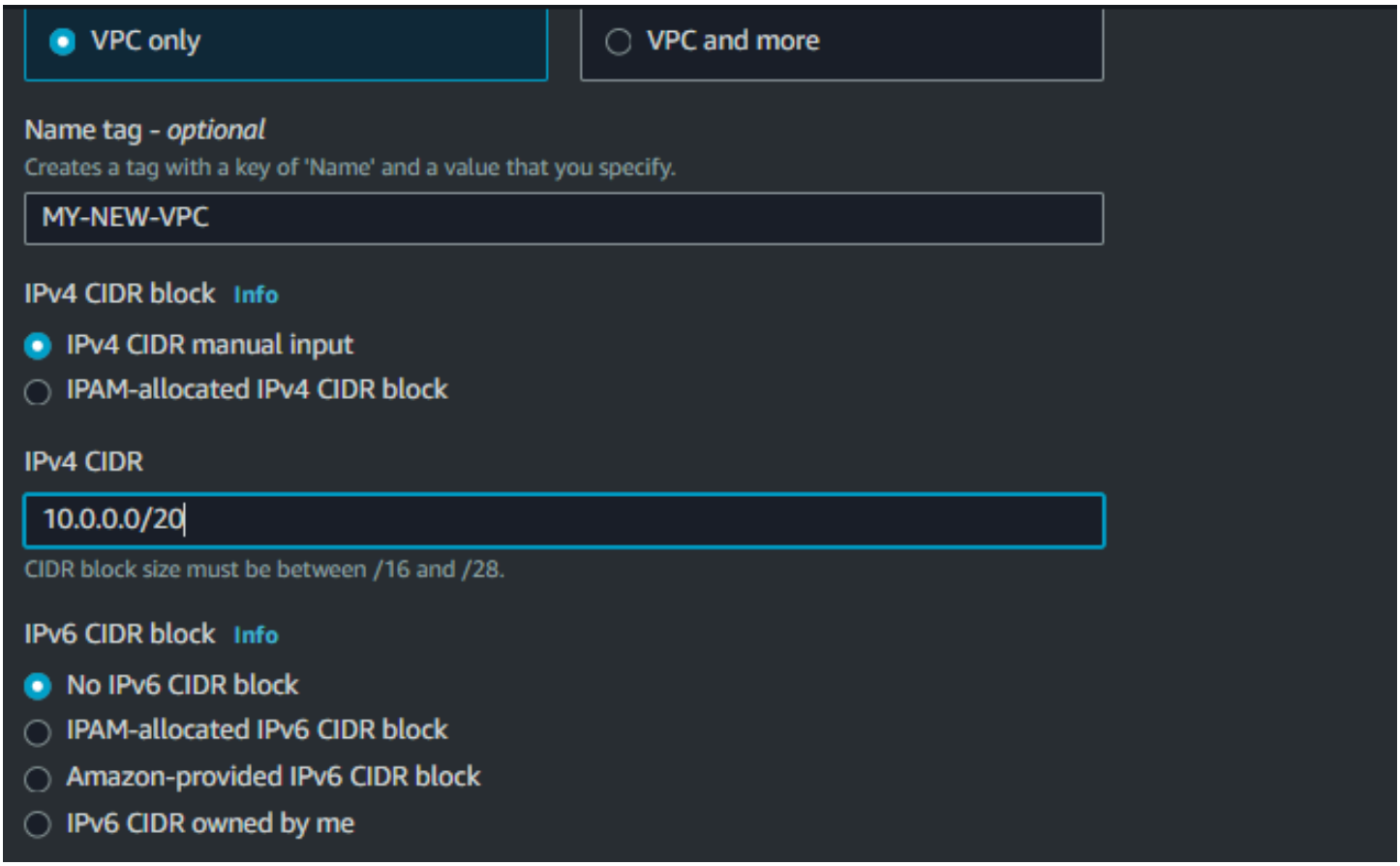
Pictorial representation of VPC creation using aws console (Manual):

This is a manual way of creation of VPC where VPC will only be created and we need to create subnets, internet gateways, route tables manually. VPC only option needs to be selected after you click on create VPC.

Virtual Private Cloud (VPC):

1) Click on create VPC and here mention the CIDR block range manually, based on that the IP's will be assigned to that VPC and with those IP's (4096) we can create public (4096) and private (4096) subnets.

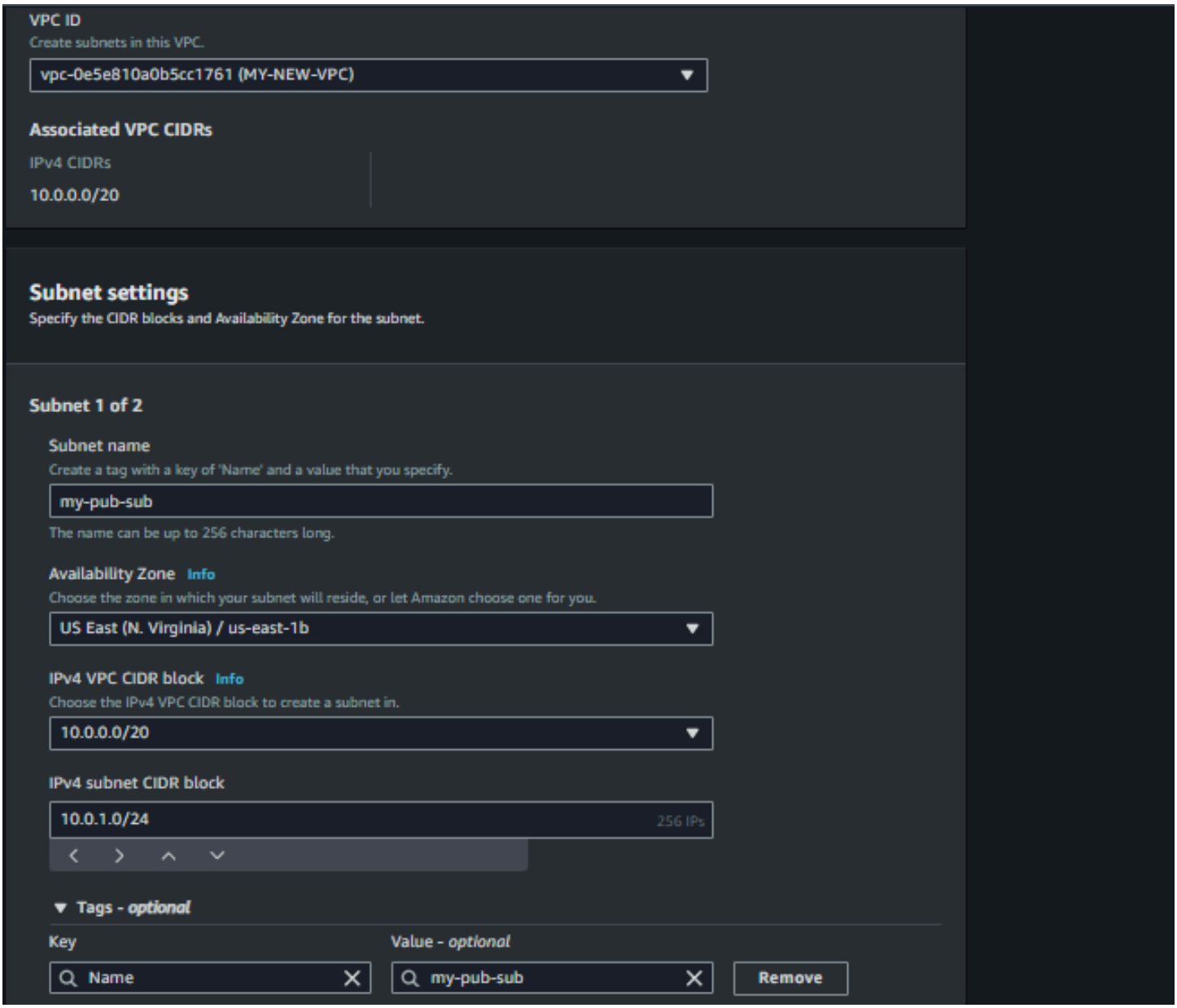
CIDR block range = 2 power (32 - n) = 2 power (32 -20) = 2 power (12) = 4096 IP's.



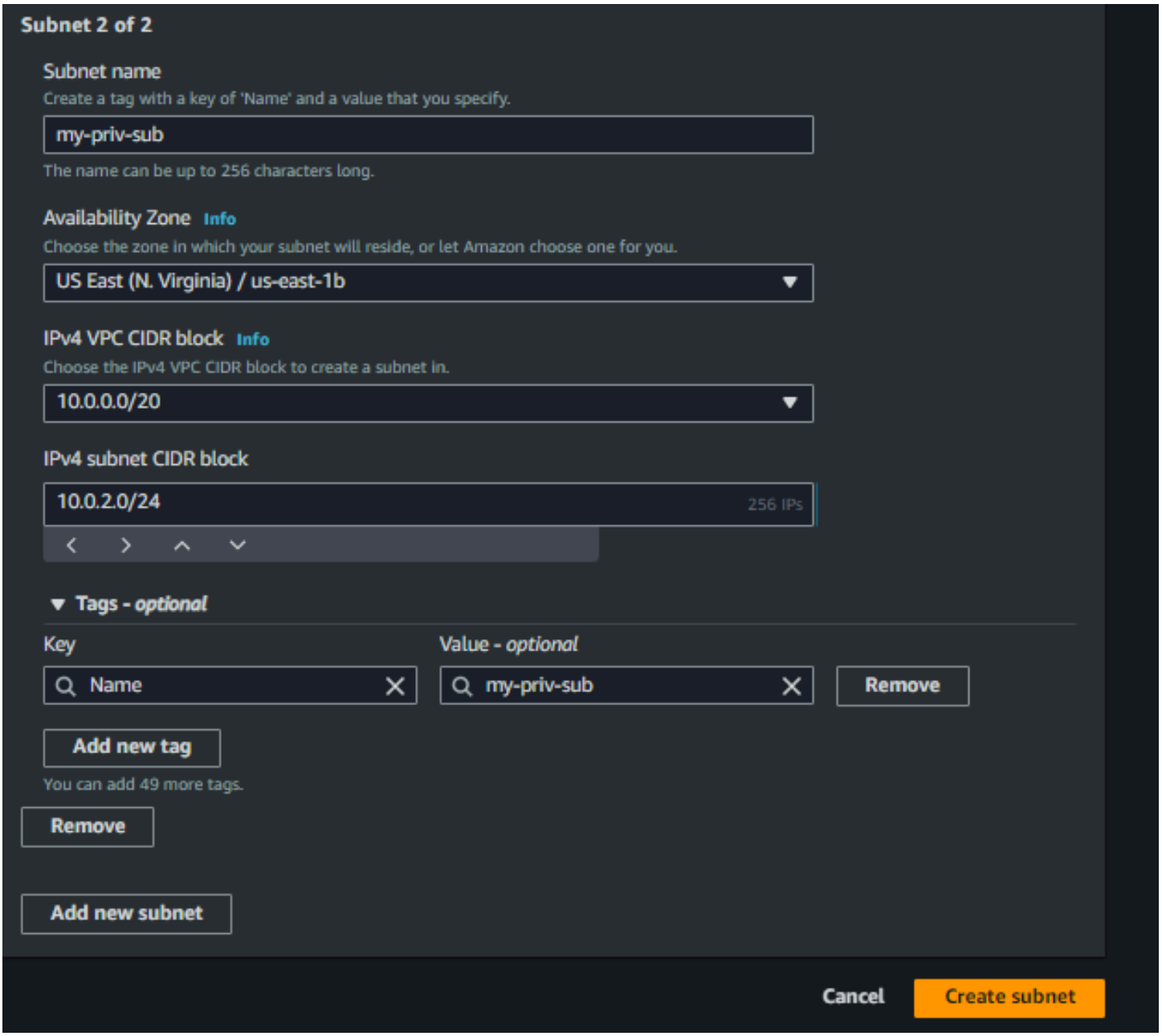
SUBNETS:

1) Go to Subnets option on the left hand side and click on create subnet. You'll be on this page.

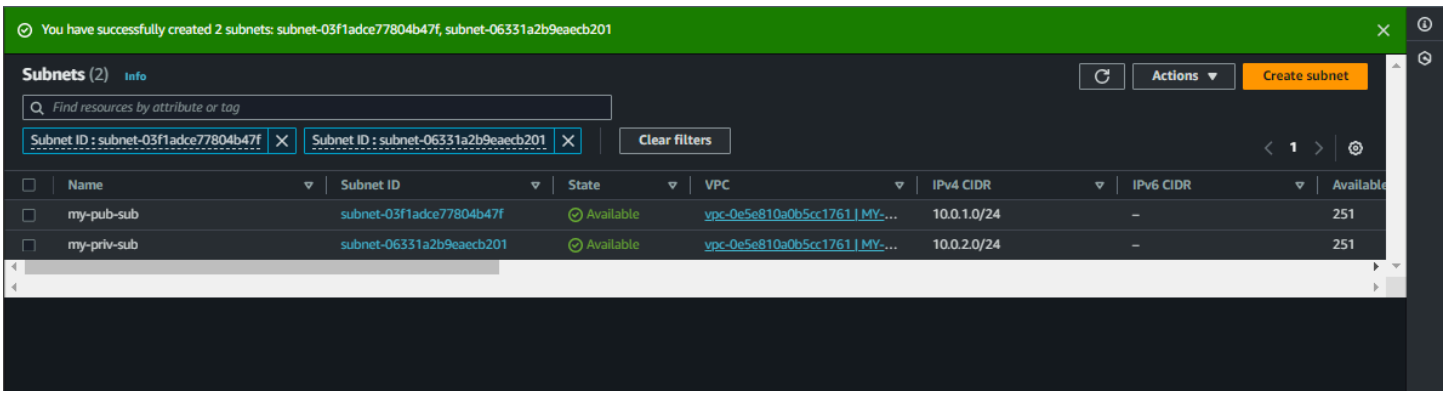
Select the VPC that you want to have your public and privates subnets created, give a name, select the availability zone, mention the n value in the cidr block.



2) Subnet - 2 (private) and then click on create subnet.

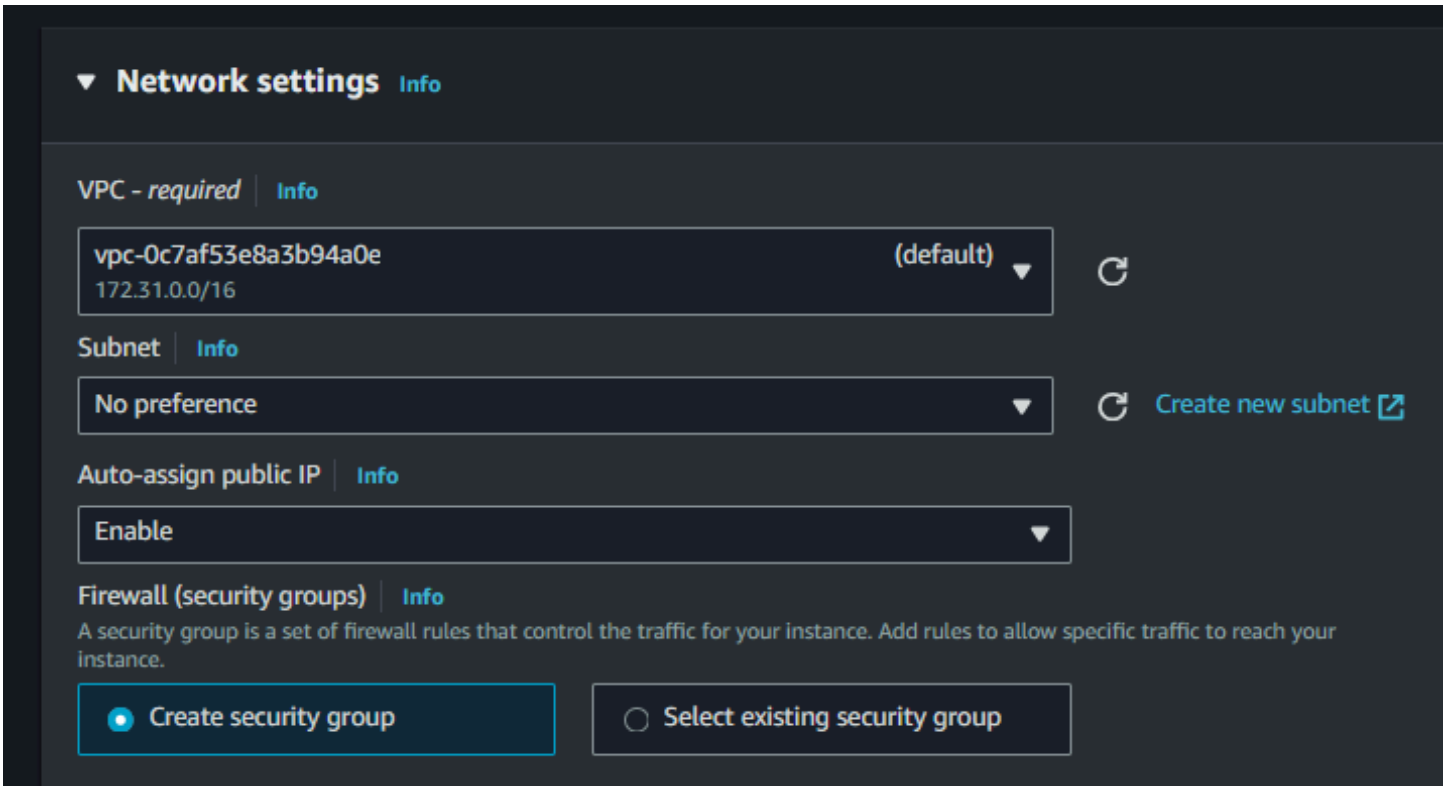


3) Public and private subnet have been created.



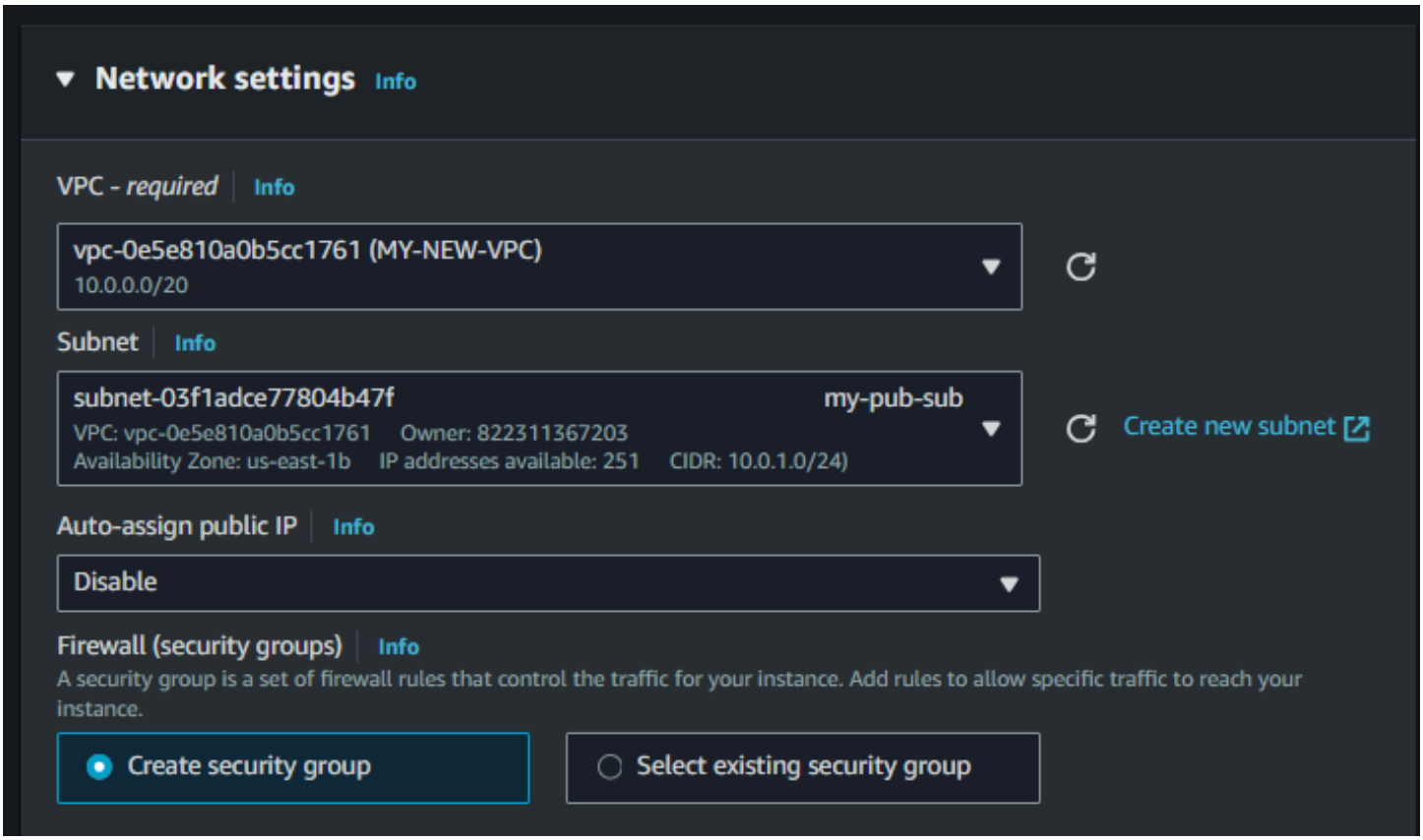
4) If you want to get the IPV4 address auto enable for the instance that you want to launch then you need to enable this option or else, by default it will be as disable as shown below.

a) By default, the auto assign public IP is enable because it's the default VPC that aws is providing while creating an instance.

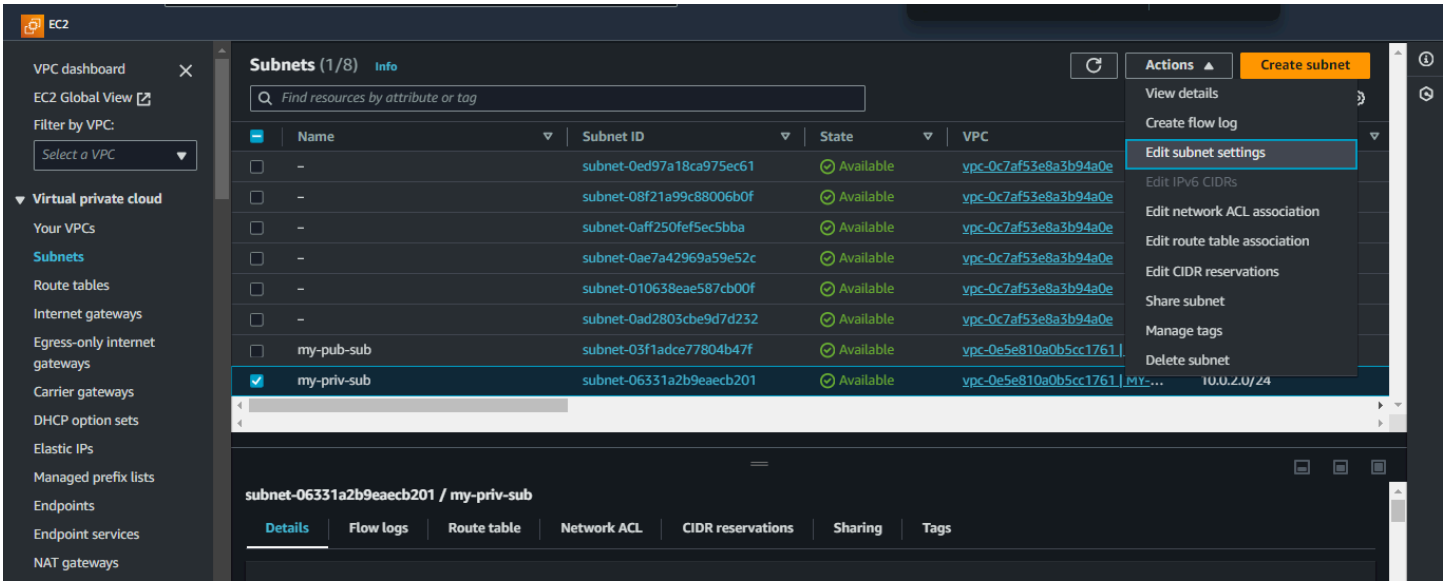


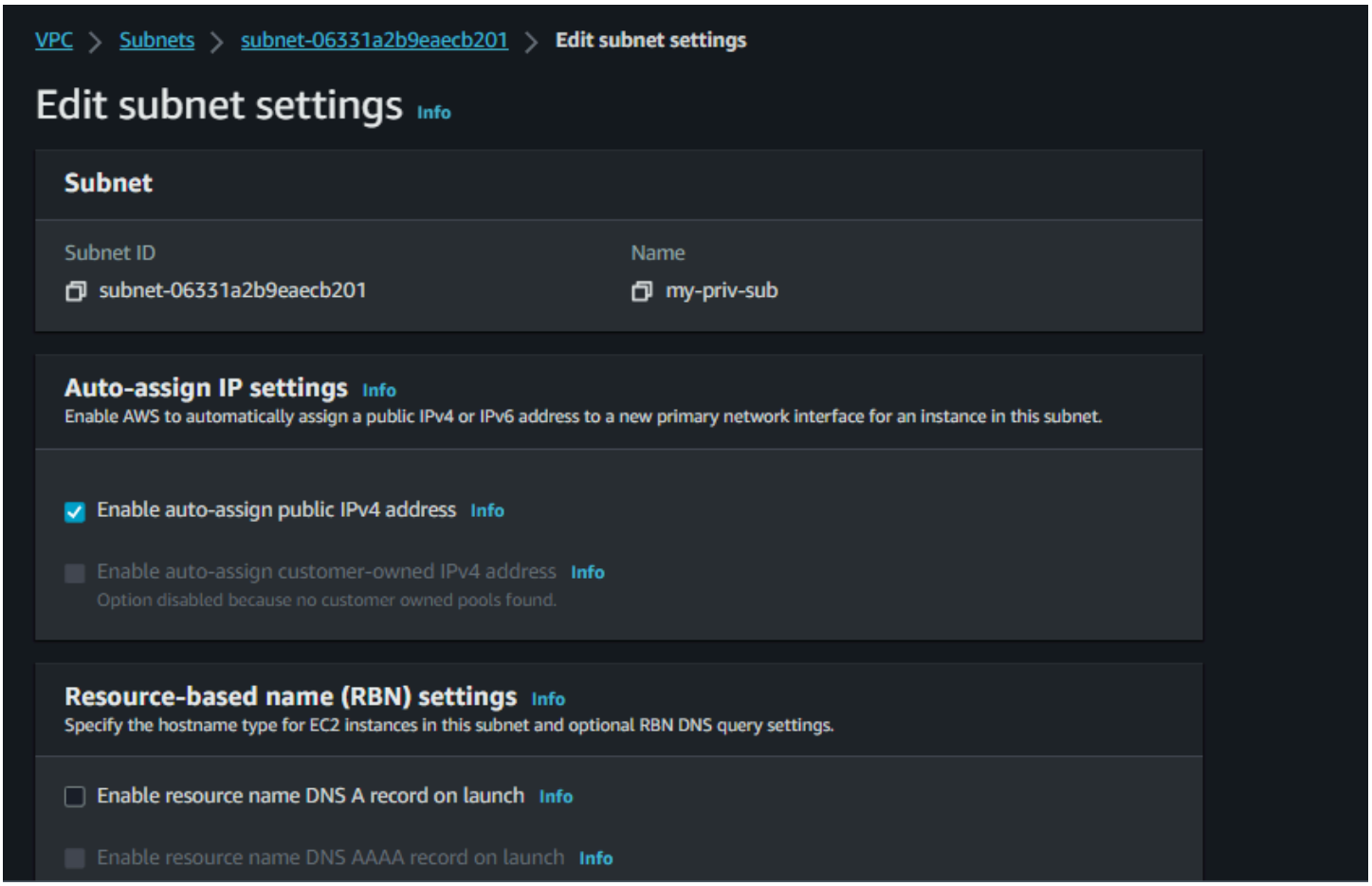
b) If you select the VPC that you've created and now it shows the auto assign public IP as disable only, because in order to enable it, you need to go to the subnet where you want to create an instance, then you need to enable the option as shown below.

Before enabling after selecting the VPC.

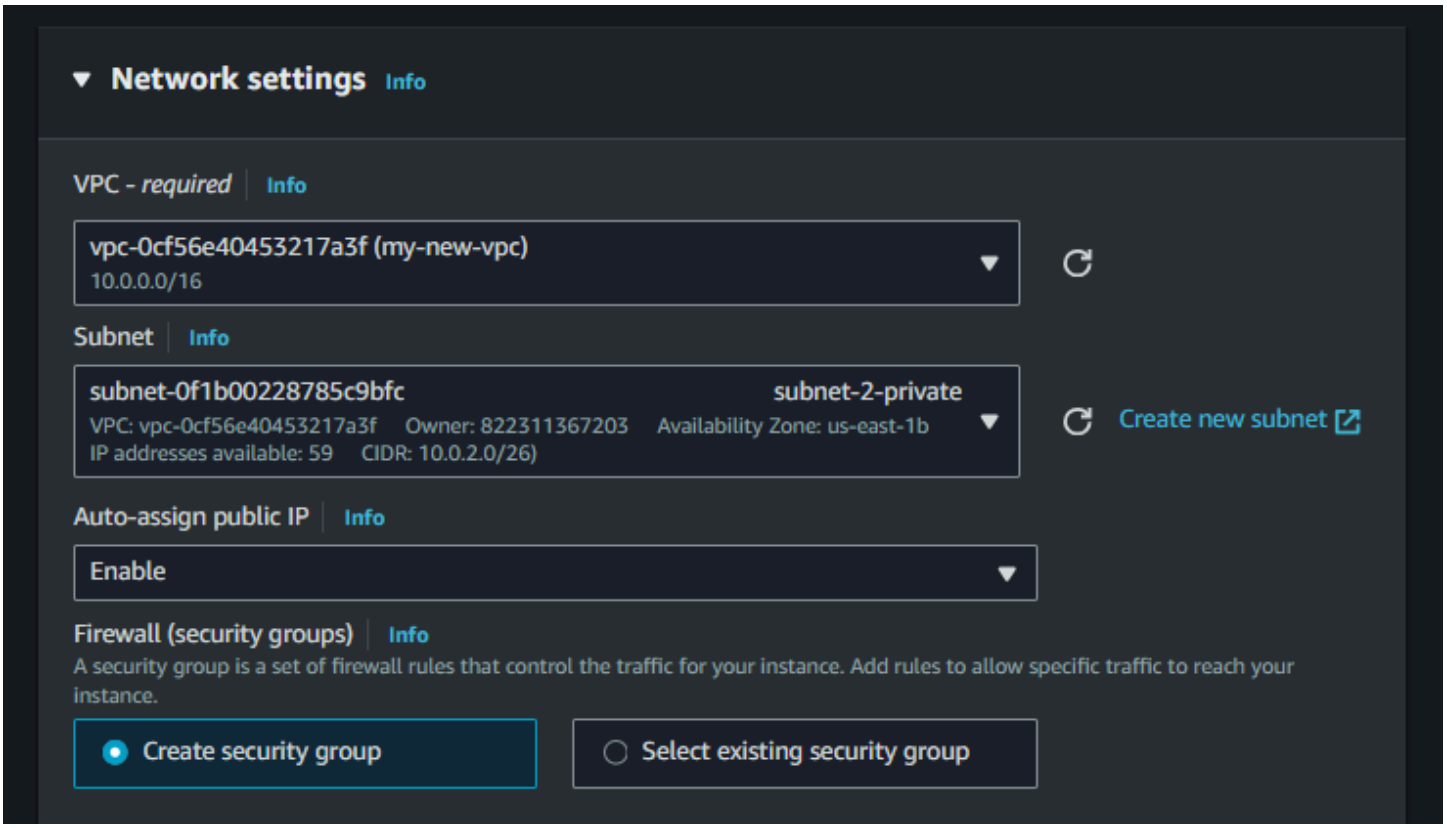


Follow the below 2 images to enable it.



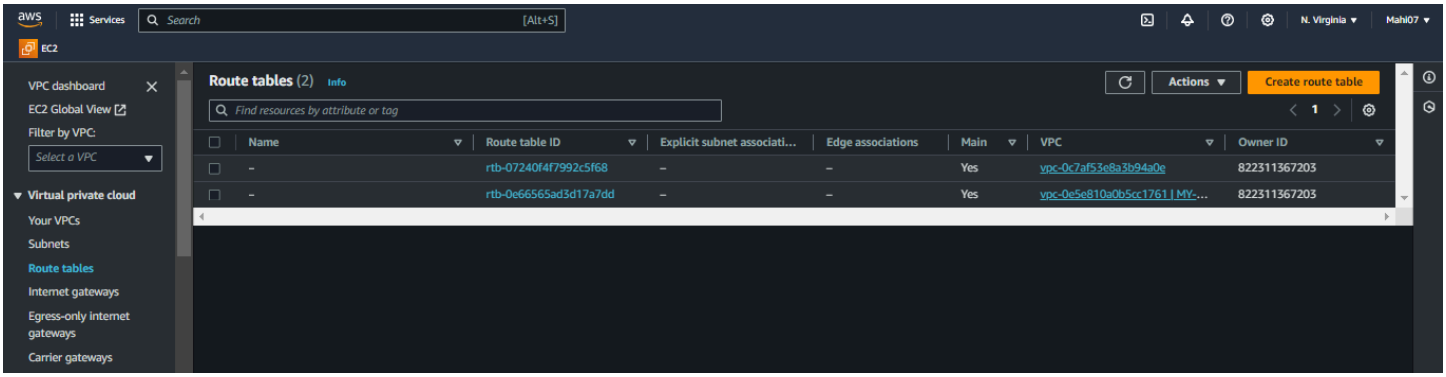


Now you can see the option is enabled.

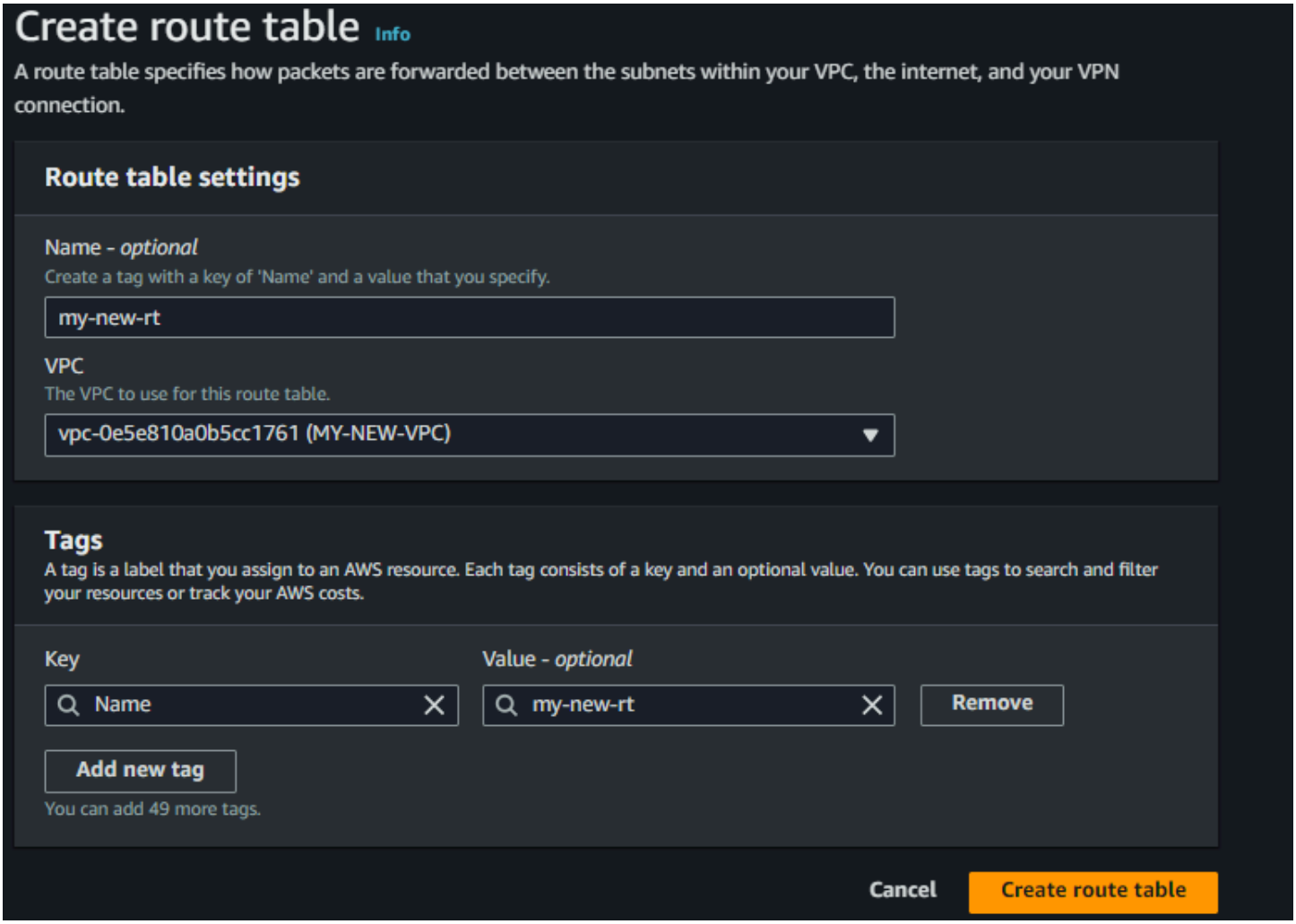


ROUTE TABLES:

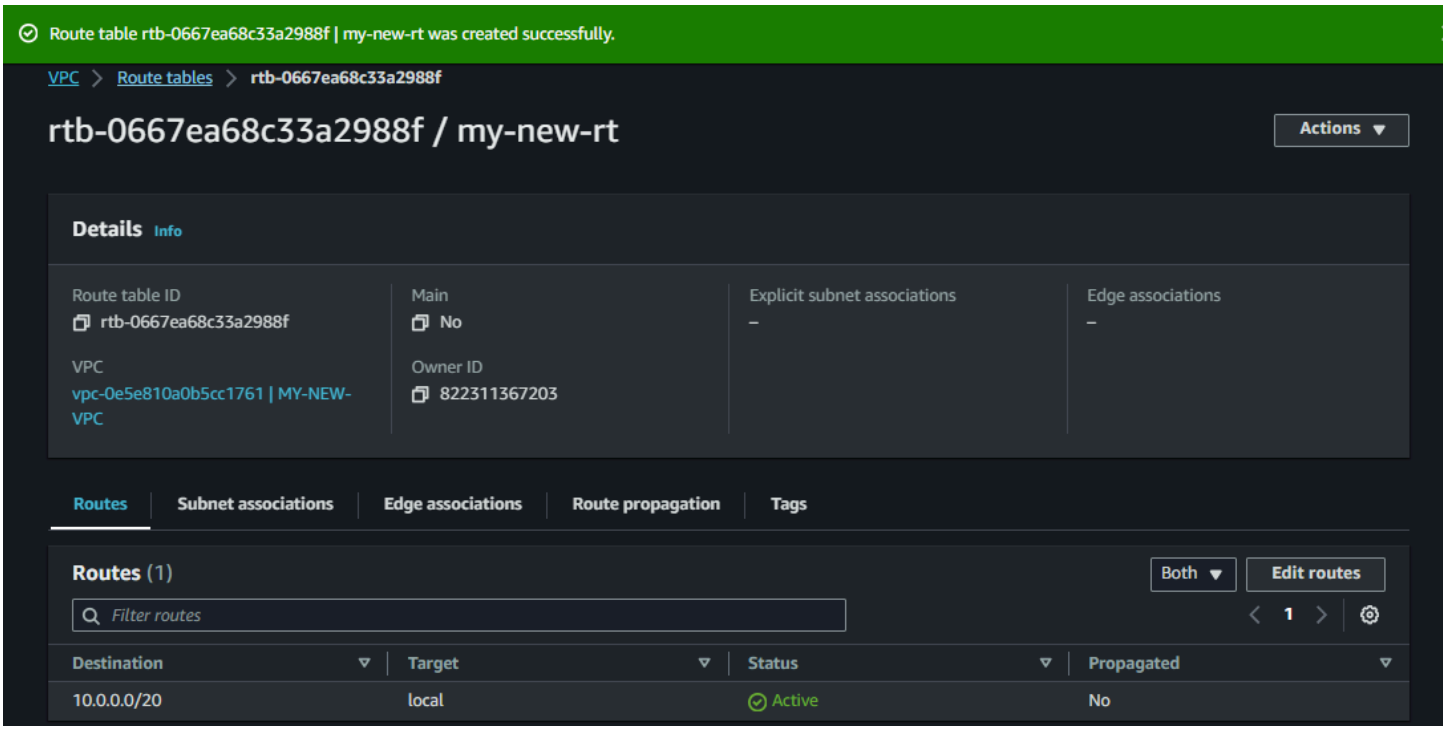
1) Click on the route tables on the left hand side and click on create route table.



2) Give a name to the route table, select the VPC and click on create route table.



3) The route table is created as shown below.

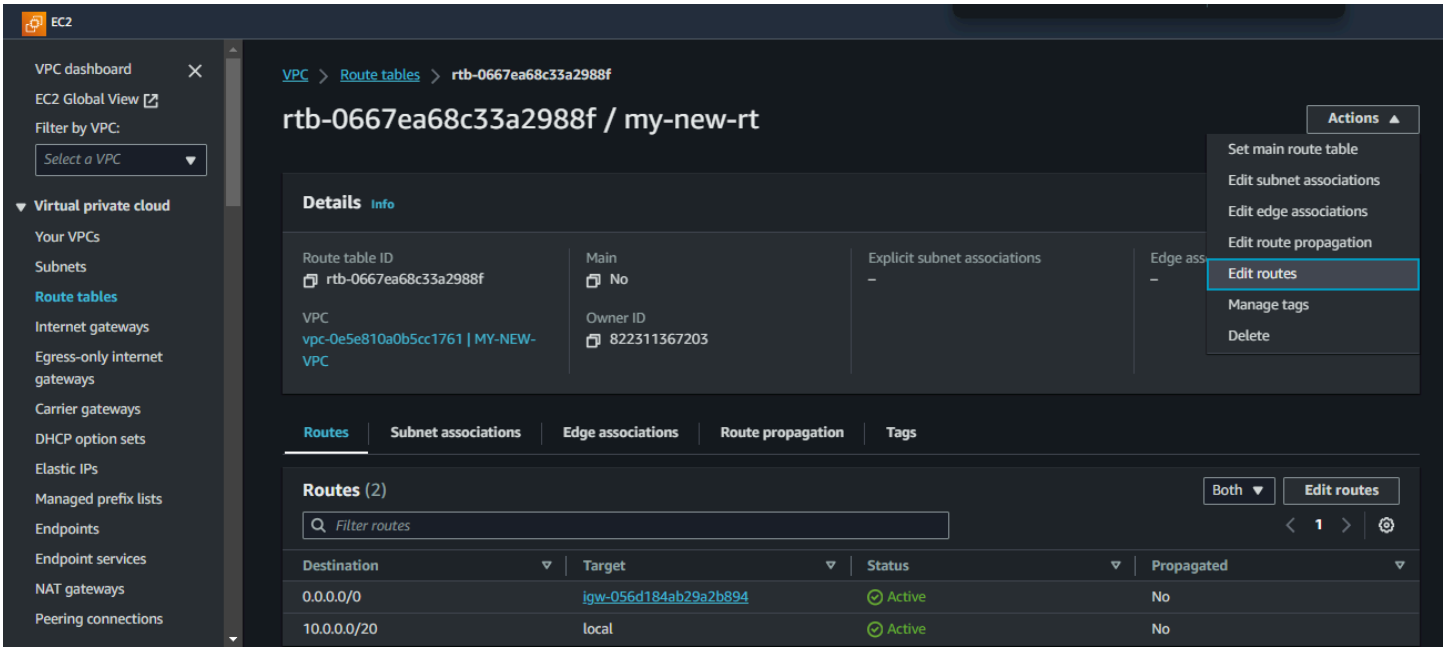


4) If you want to allow a set of people to access or all people to access then you need to add a route with the cidr range.

If only 1 person need to access, mention their IP address and at the end add /32 because 2 power 32-32 = 0 and 2 power 0 = 1. So 32 needs to be mentioned at the end of the IP.

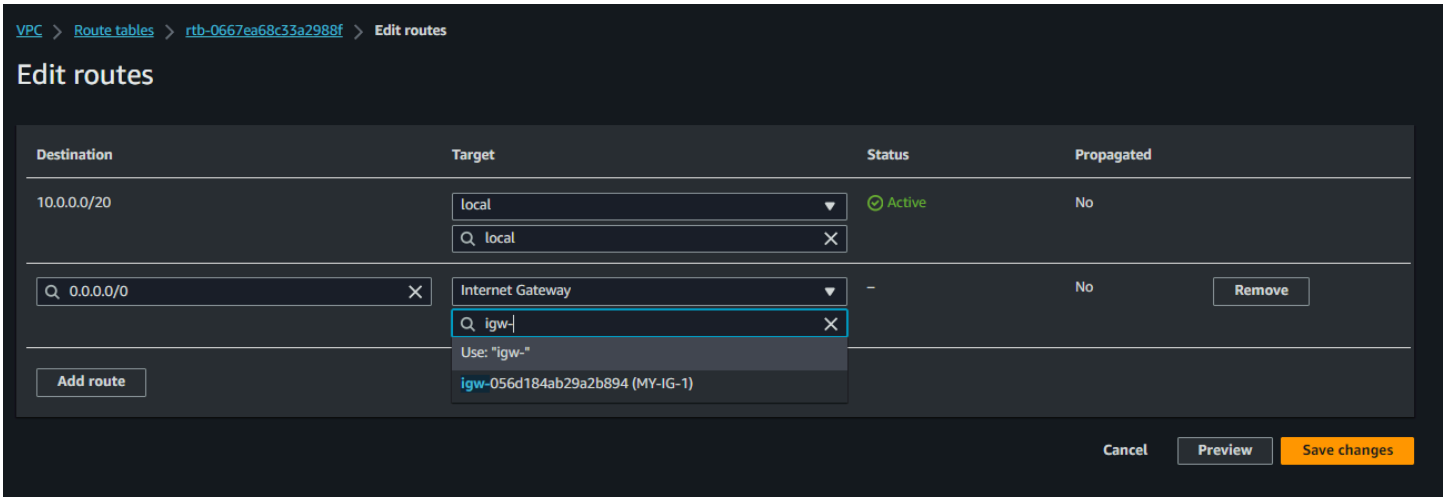
So click on route tables, select the route, click on actions, click on edit routes. Click on 'Add route', mention the cidr range, select the internet gateway from the list of options and select the correct internet gateway you've created and want to attach it for and click on save changes.

a) Selection of route.



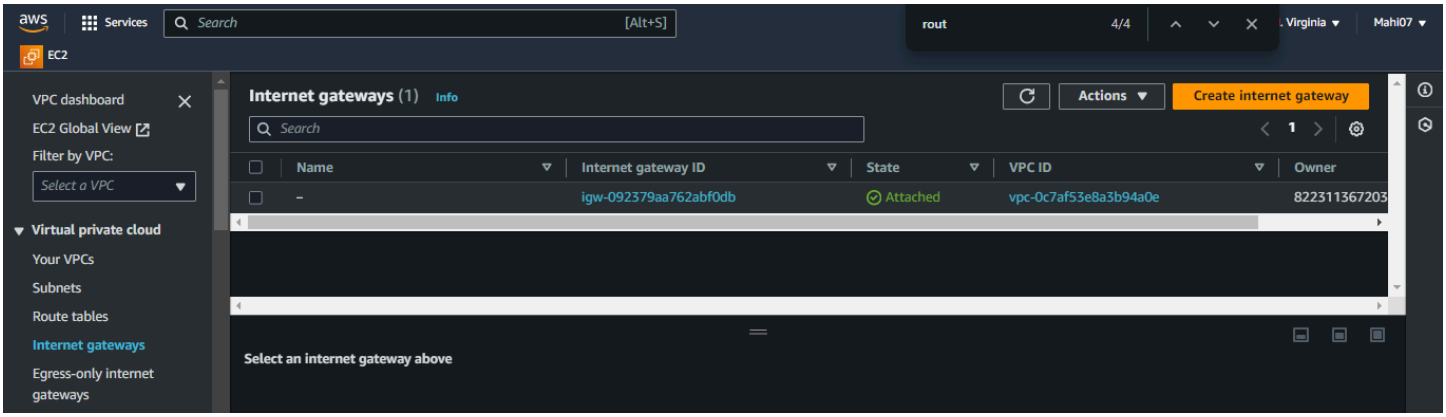
b) Adding route.

NOTE: Until and unless, the internet gateway is attached to VPC you cannot add it to the route table to give access to people who want to access.

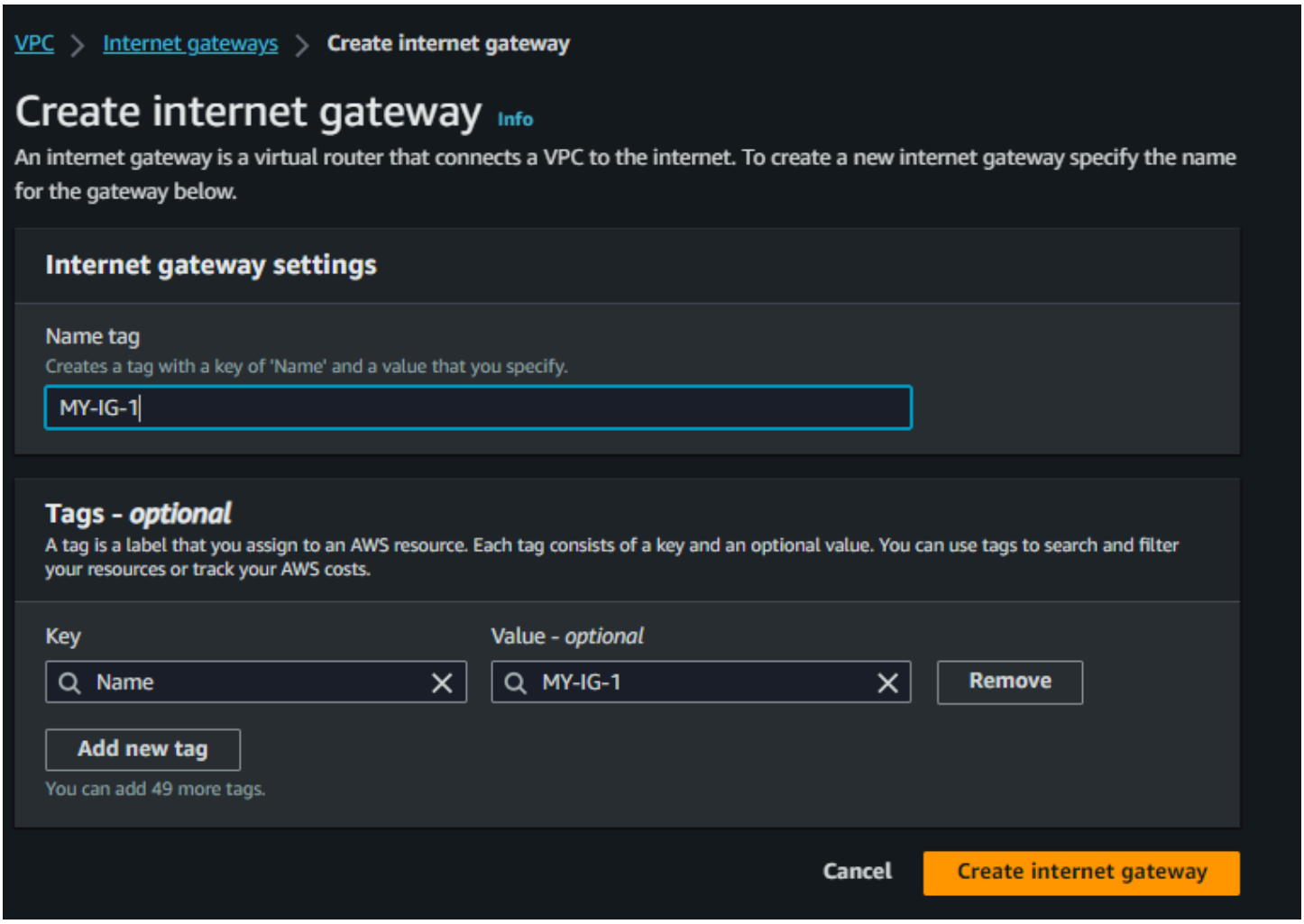


Internet Gateways:

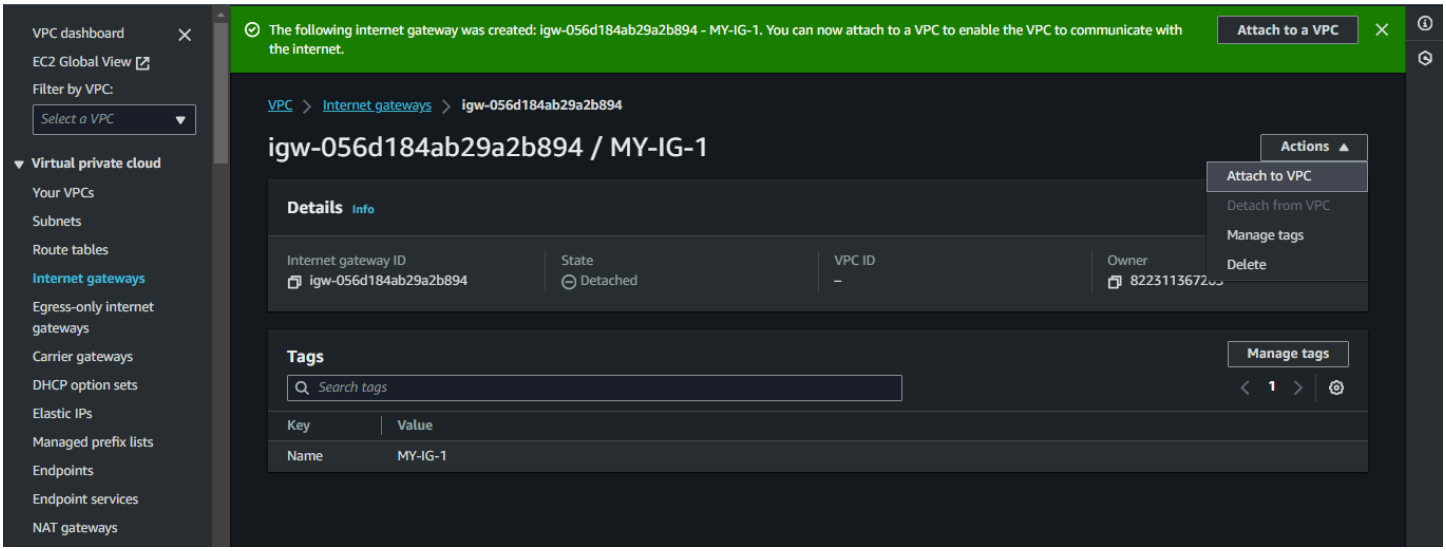
1) Click on internet gateways on the left hand side and click on create internet gateway, you'll be on this page on the aws console.



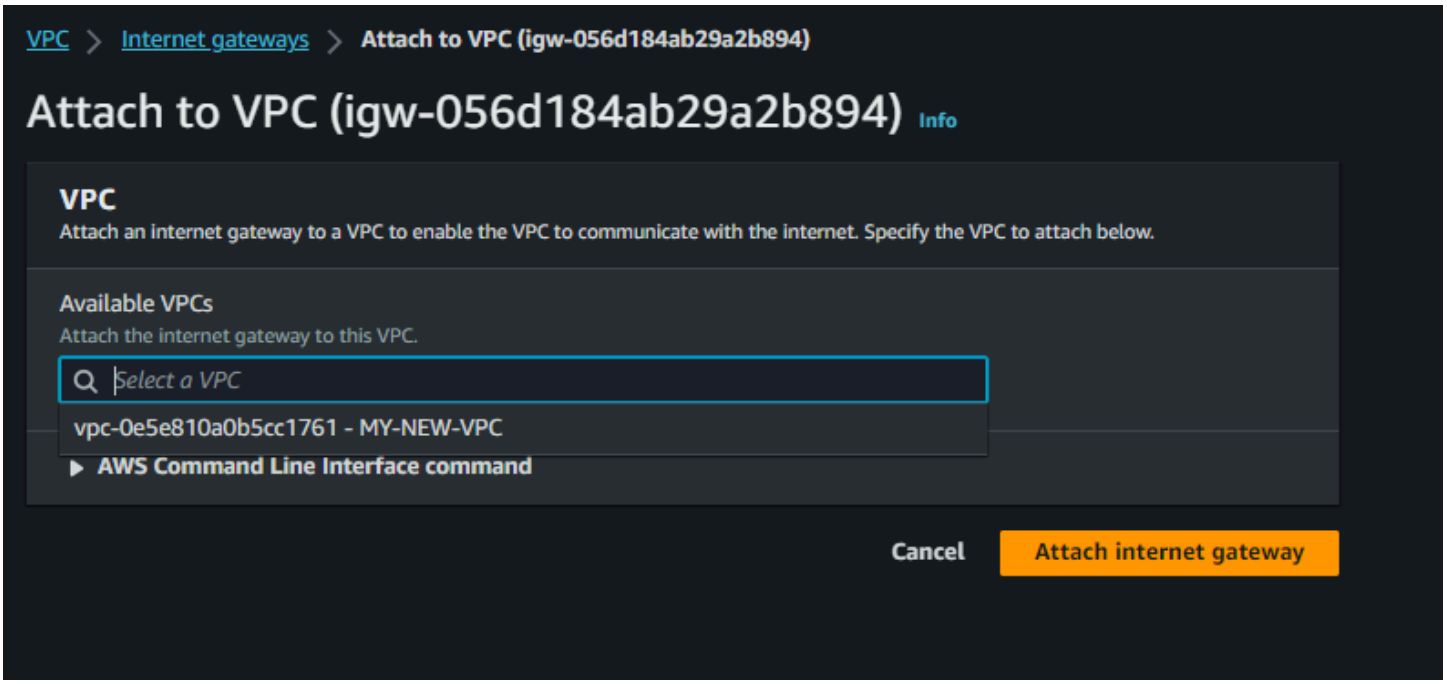
2) You'll be landed on this page, give a name and click on create internet gateway.



3) You can either click on '**Attach to a VPC**' at the top right after the creation of internet gateway or else you can also click on actions and then click on '**Attach to VPC**' as shown below.

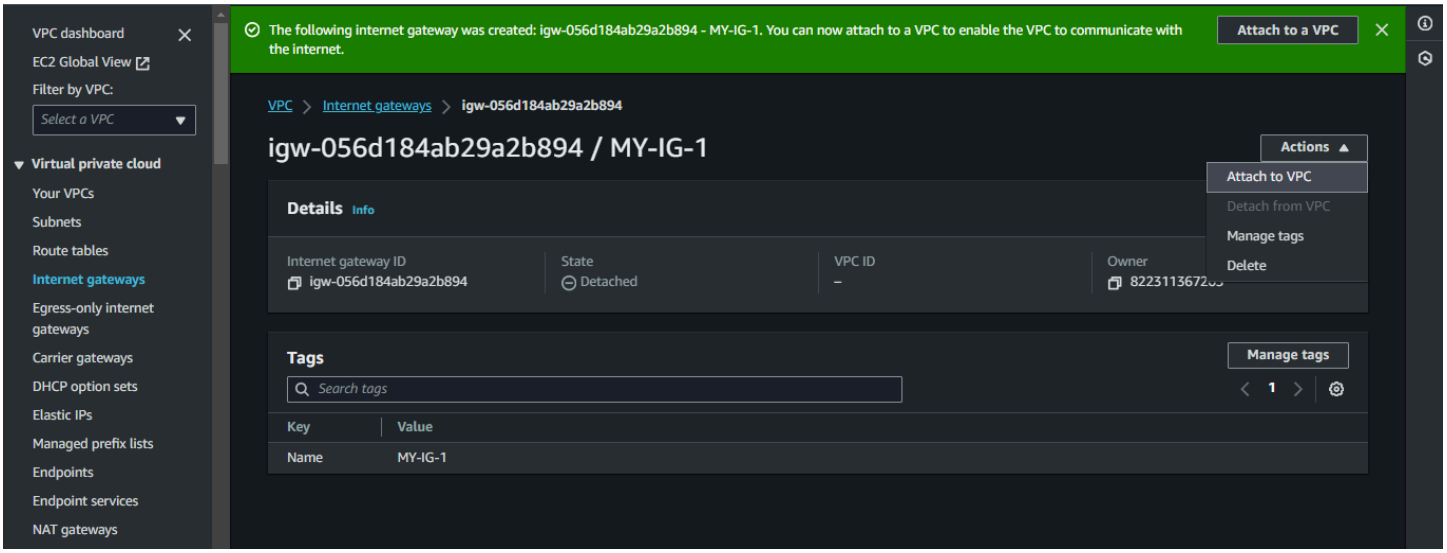


4) Select the VPC that you want to attach the internet gateway and click on '**Attach internet gateway**'.

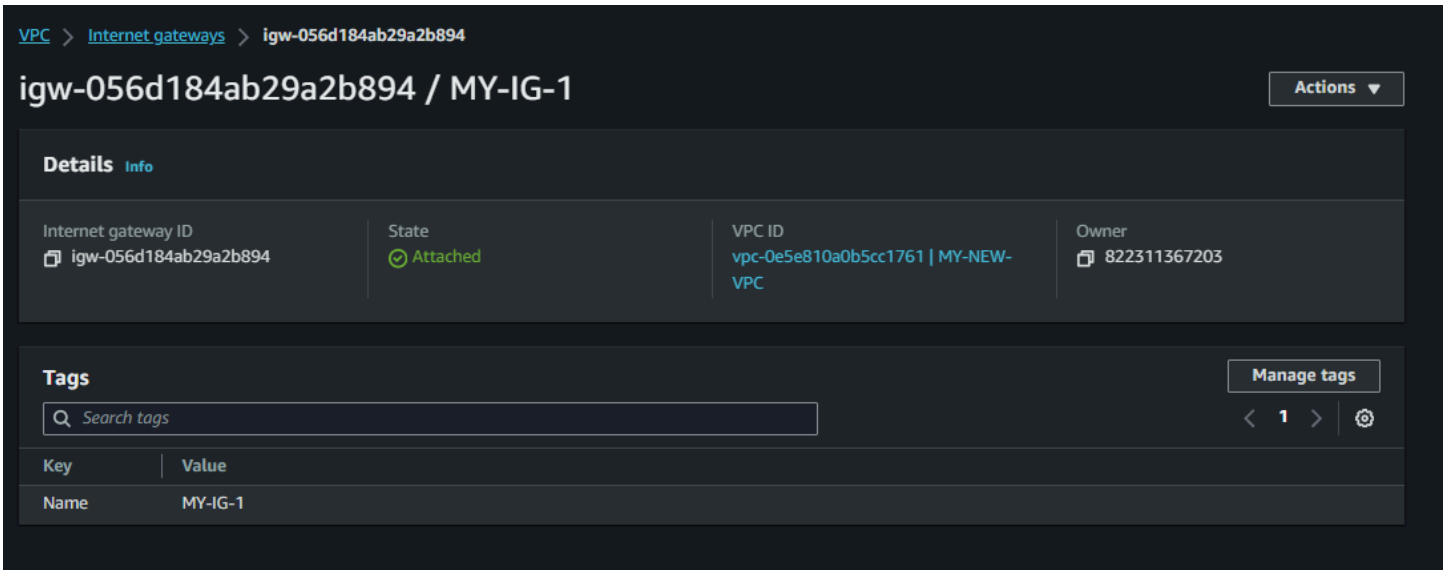


5) Here you can that the VPC ID has been attached to the internet gateway as a confirmation.

a) NO VPC ID attached.



b) VPC ID has been attached.



Peering connection and it's importance:

Definition: Peering connection is a networking connection in aws cloud that offers help to establish a communication between two VPCs so that the resources like instances inside two different VPC networks can communicate with each other.

Tech Def: In other words, this peering connection is a networking connection will help to enable routing between the two VPC networks with the help of the private IP addresses as if they're in the same network.

Example:

If there are two instances that were launched using same VPC then they communicate with each other, where the security is less and aren't isolated.

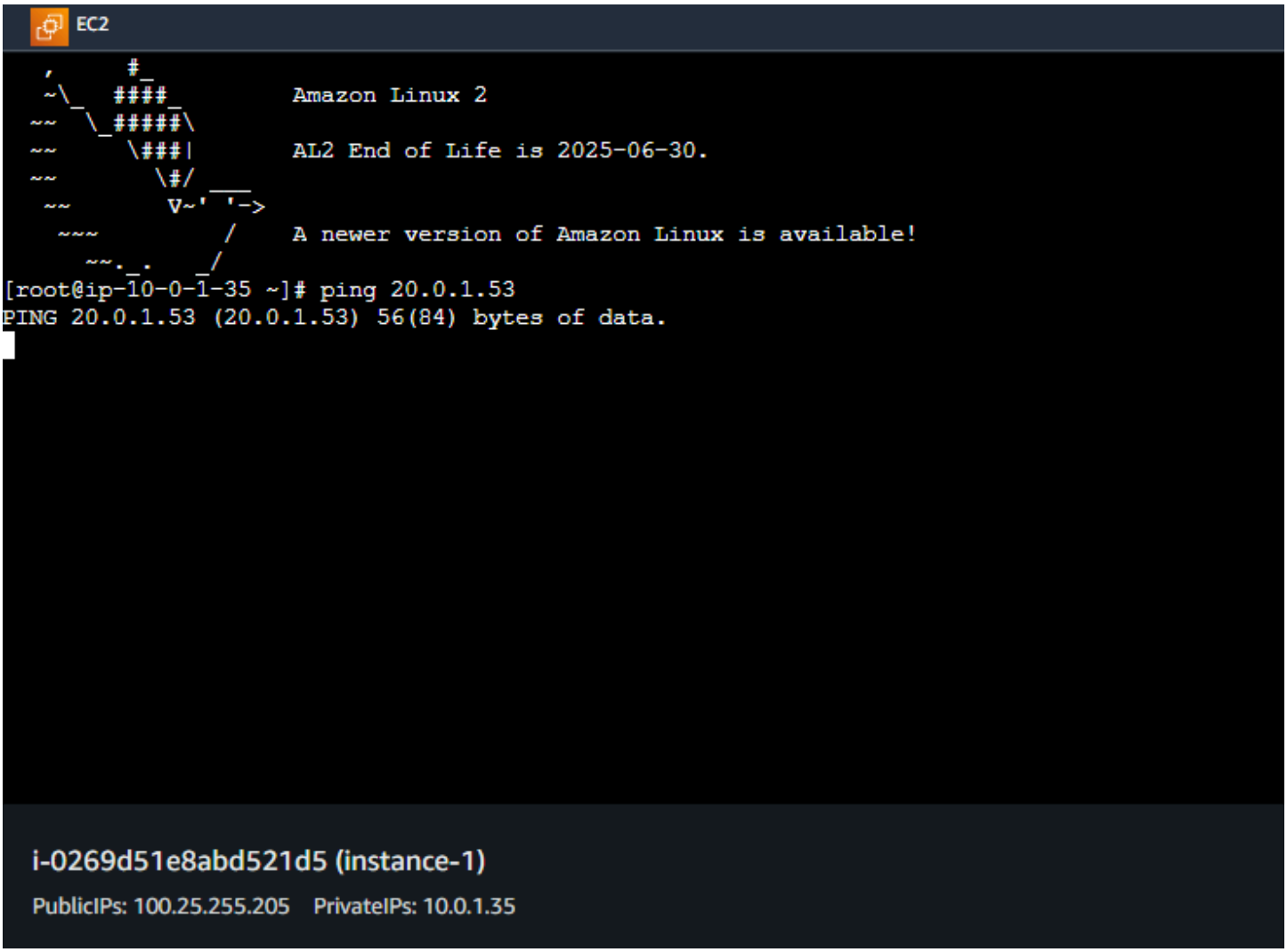
However, if the instances are launched using two different VPCs, they won't communicate with each other and if you want to check if they are communicating or not you can check it with the help of ping command using their IP addresses. In order to communicate, we use peering connection to establish a connection in the VPC level so the instances will also communicate.

You can take the private IP address of instance-1 and check it on instance-2 and vice versa as shown below.

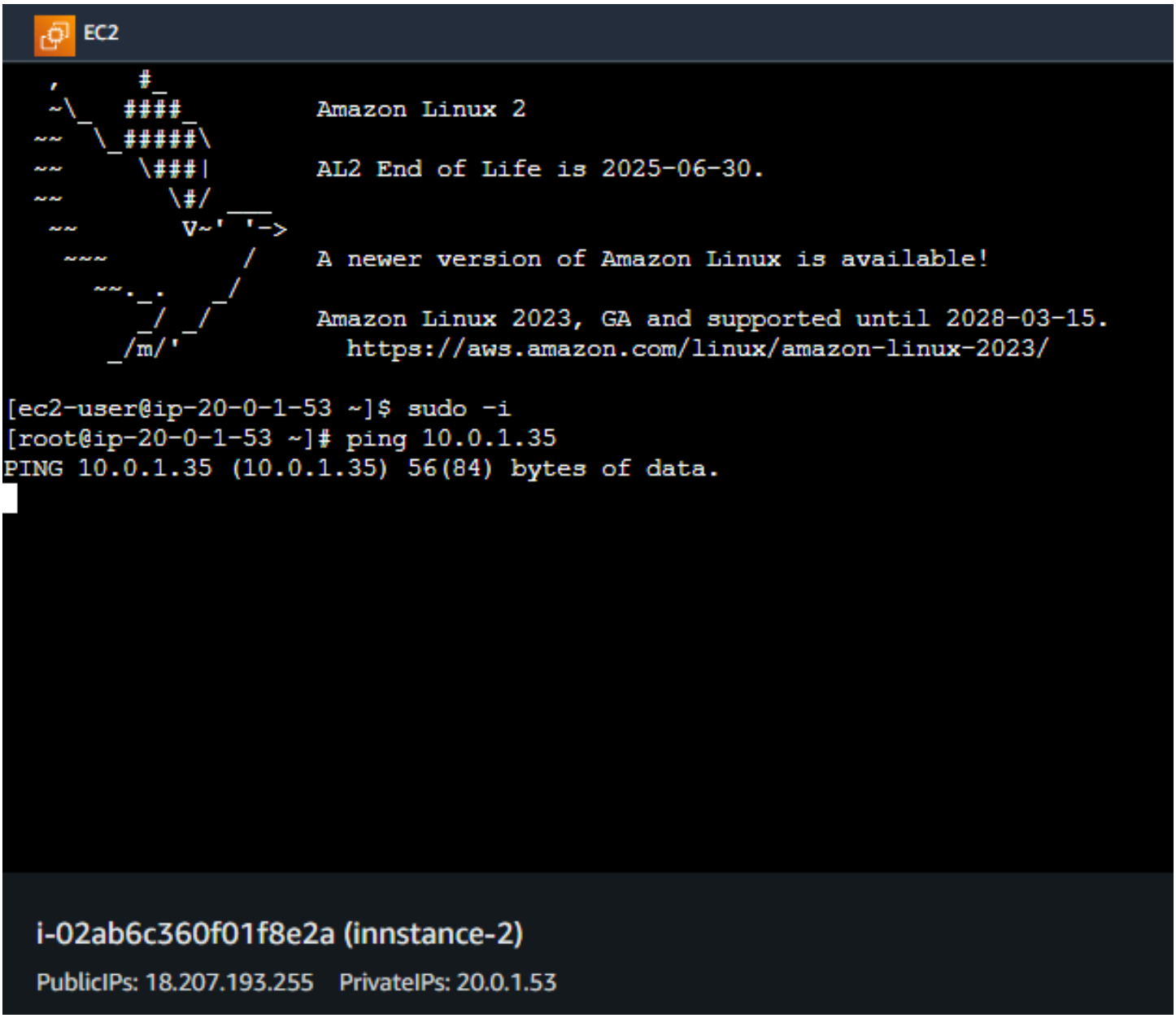
Practical check of peering presentation:

1) Instance 1 and 2, before setting up the peering connection where they're not communicating with each other.

a) I've used the private IP address of instance 2 on instance -1 and it's not communicating as shown below.

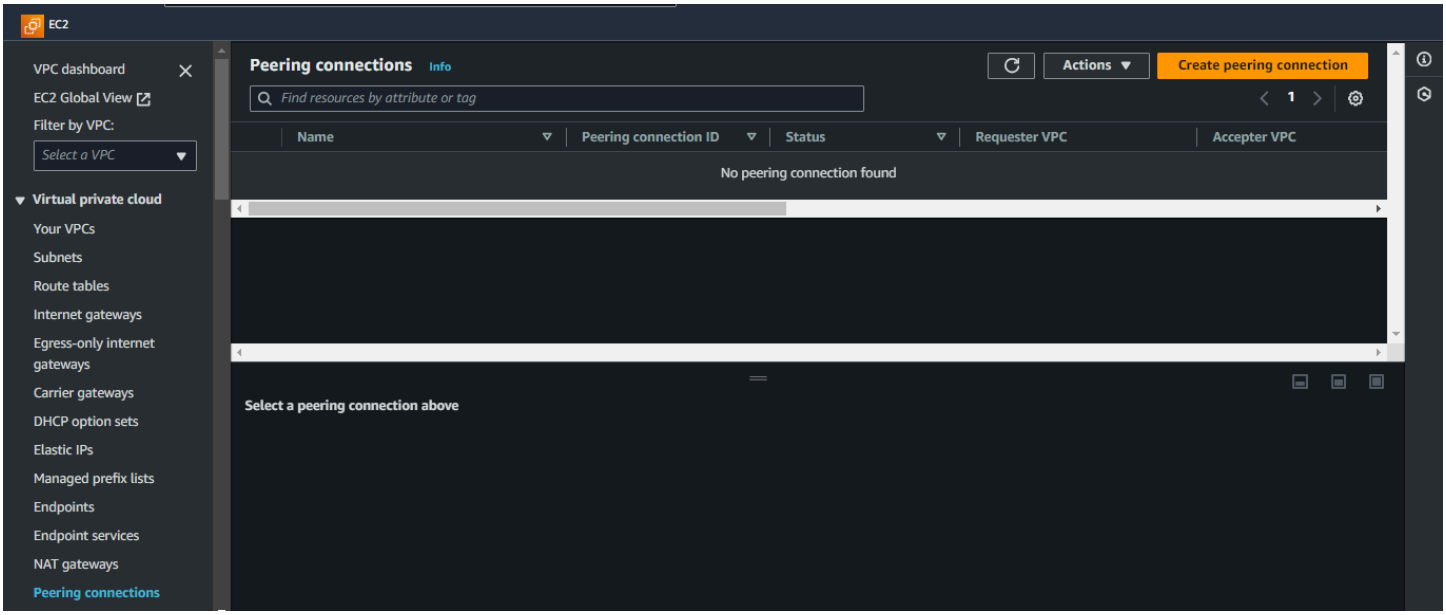


b) I've used the private IP address of instance 1 on instance -2 and it's not communicating as shown below.

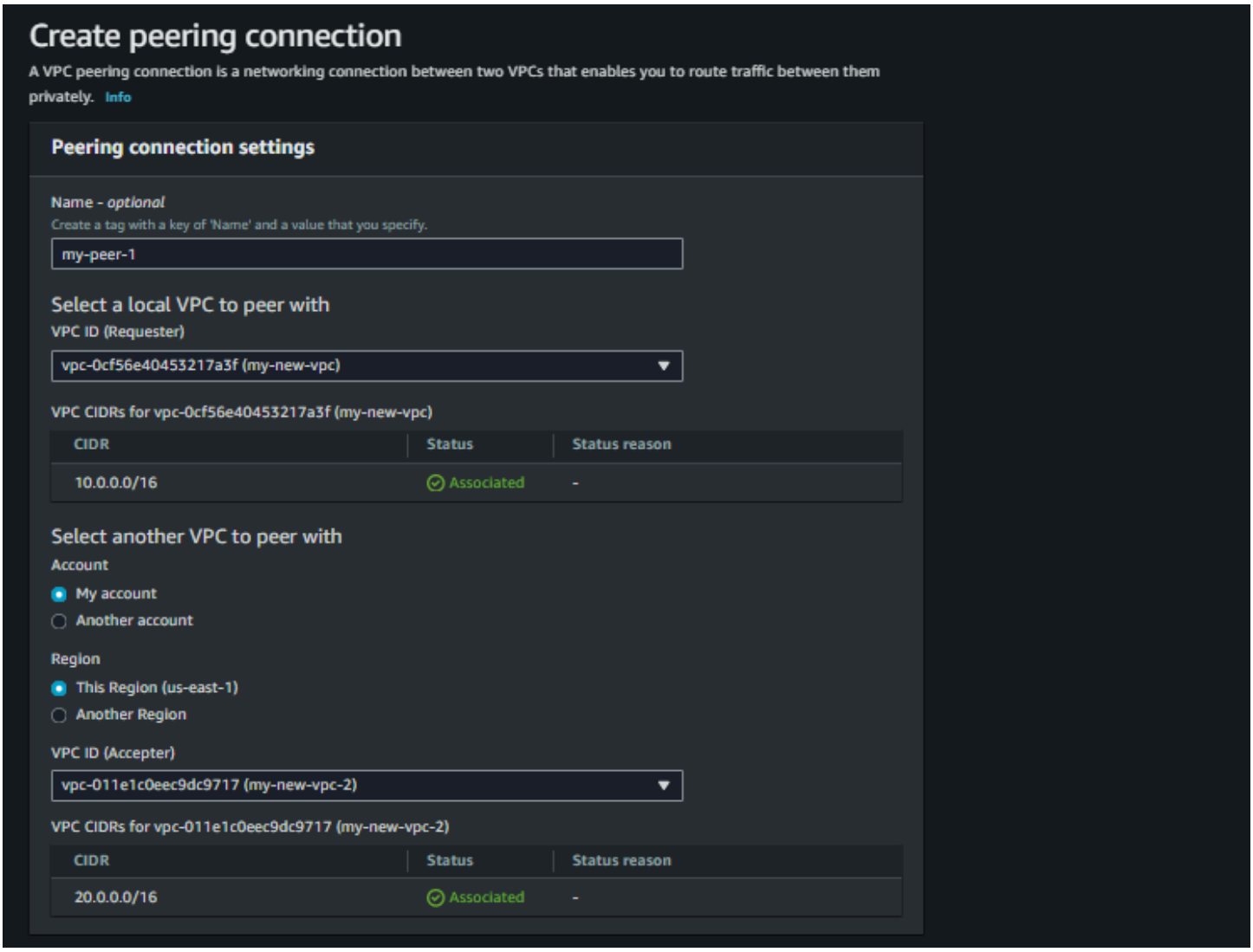


2) Establishing the peering connection.

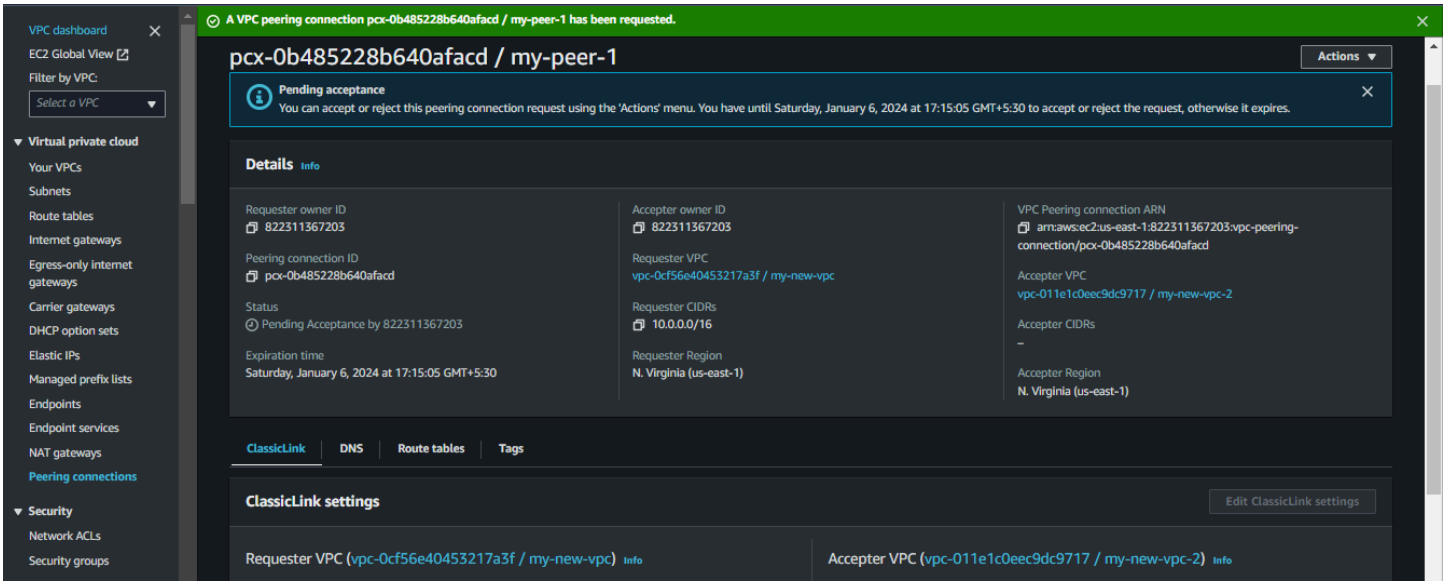
a) Go to the left hand side menu under VPC and click on peering connection and you'll be on this page.

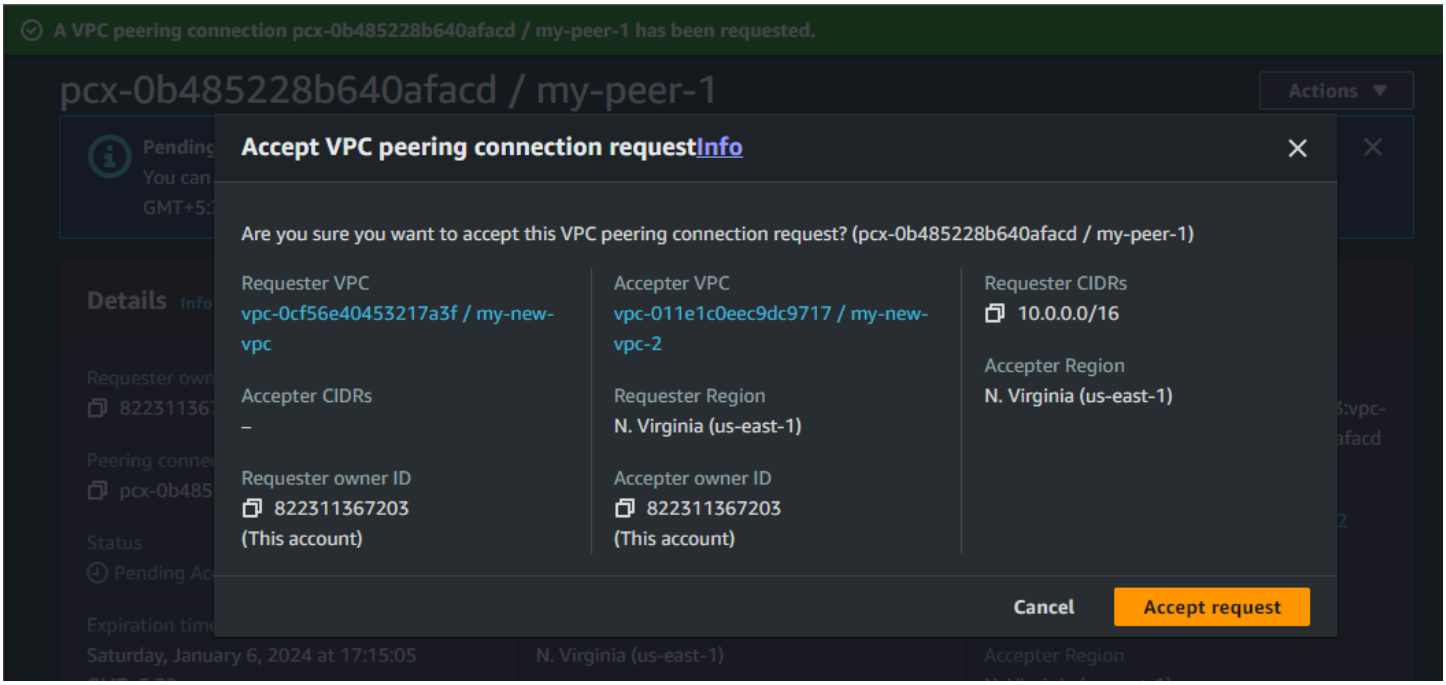


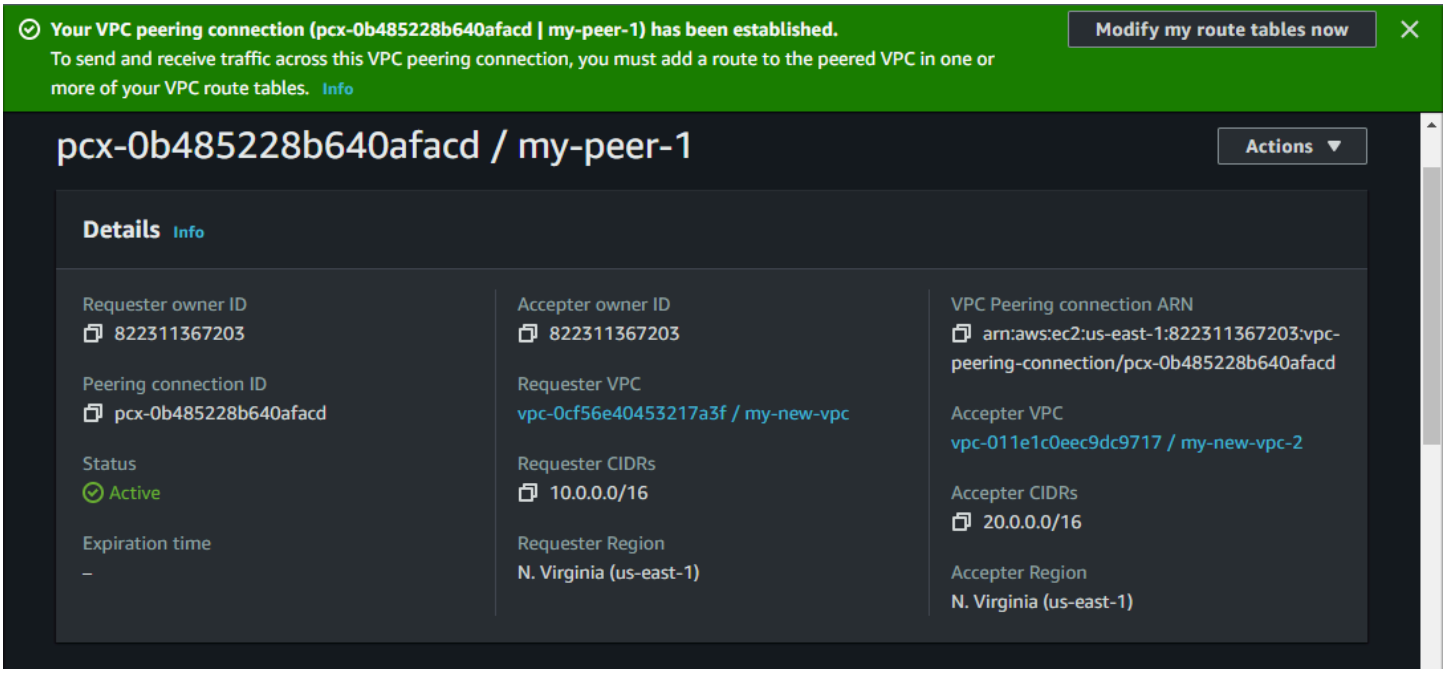
b) Click on create peering connection, give it a name, select the vpc-1 or vpc-2 to peer with and click on create peering connection.



c) After the peering connection is created you need to accept the request. Click on actions and click on accept request.

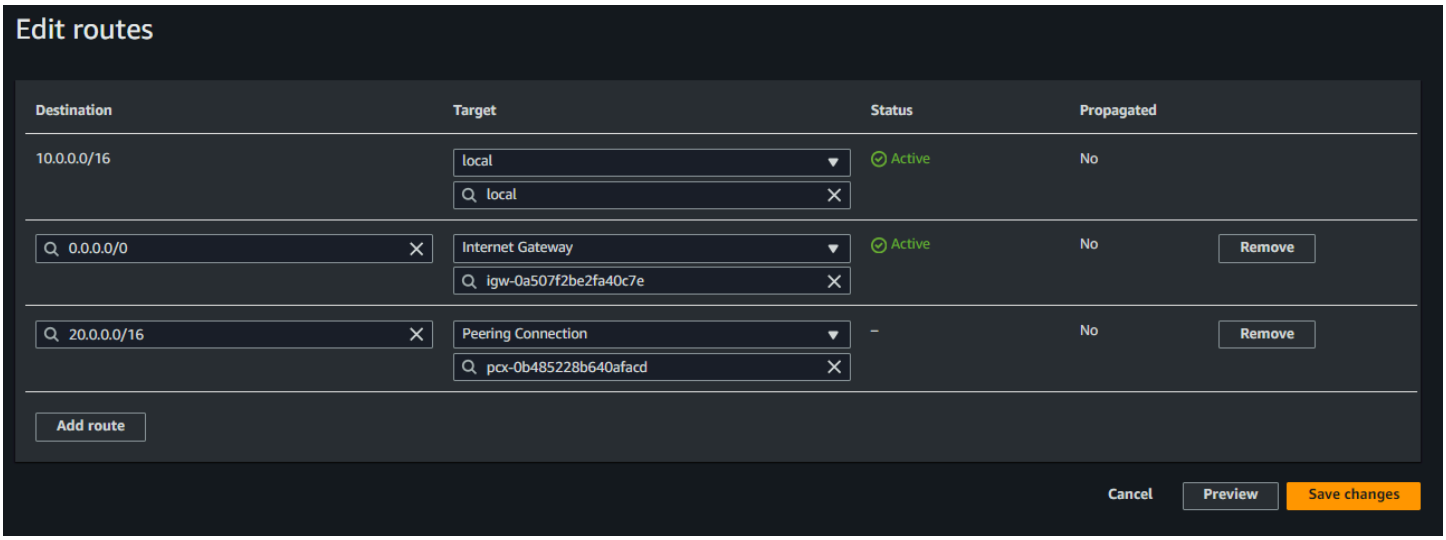




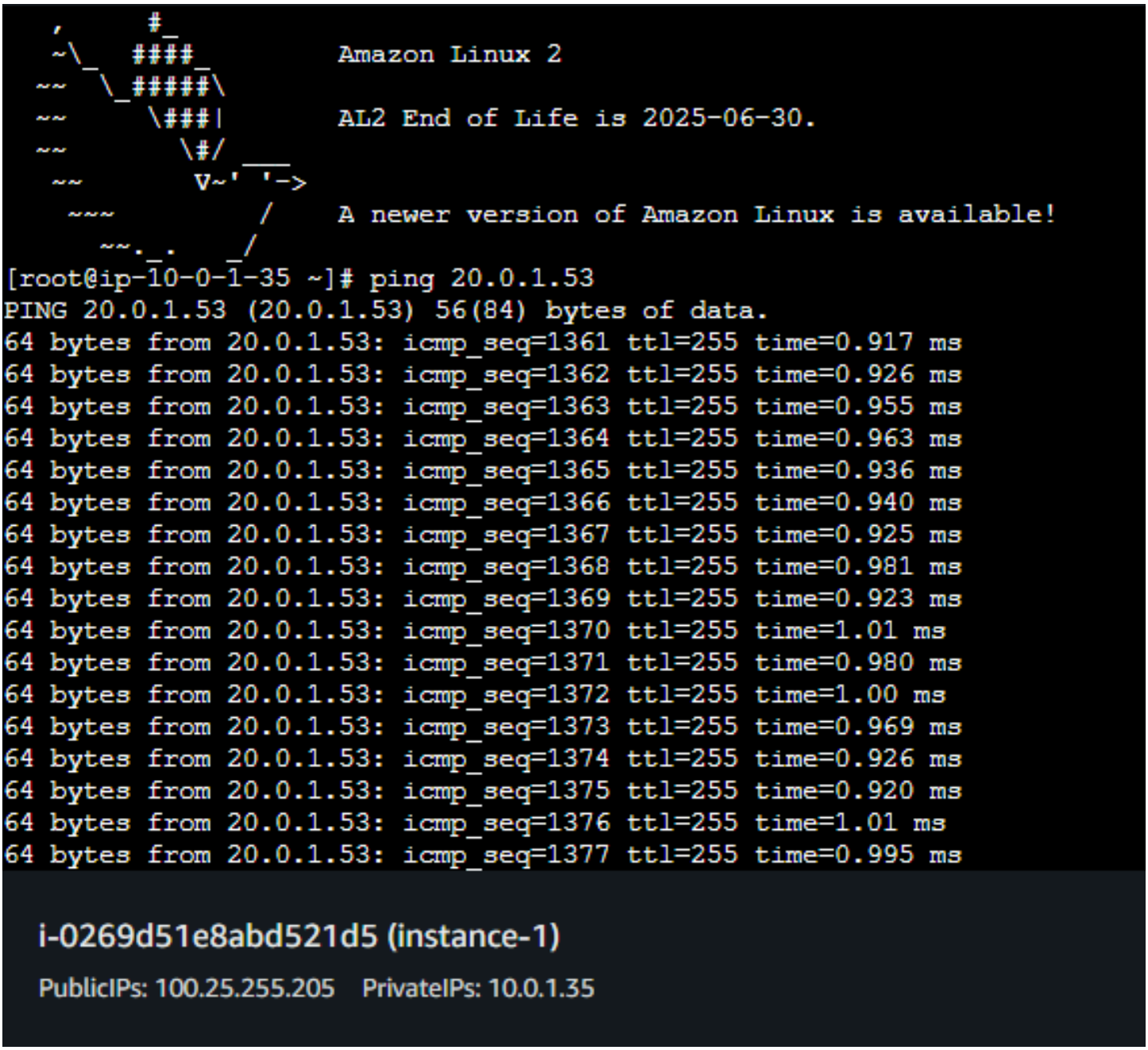


As soon as the request is accepted, the connection isn't fully established because we need to add this peering connection to the route tables.

Go to route tables - -> select one route table - - > click on edit routes - - > click on add route - - > mention the VPC CIDR range - - > select the peering connection from the list of options - - > select the name and click on save changes as shown below. Repeat the same steps for VPC 2 as well to add it in the route table.



As soon as the peering connection is added to the route tables for both the VPCs, we get the packets on instances to demonstrate that the instances are getting communicated as show below.


Amazon Linux 2
AL2 End of Life is 2025-06-30.
A newer version of Amazon Linux is available!
[root@ip-10-0-1-35 ~]# ping 20.0.1.53
PING 20.0.1.53 (20.0.1.53) 56(84) bytes of data.
64 bytes from 20.0.1.53: icmp_seq=1361 ttl=255 time=0.917 ms
64 bytes from 20.0.1.53: icmp_seq=1362 ttl=255 time=0.926 ms
64 bytes from 20.0.1.53: icmp_seq=1363 ttl=255 time=0.955 ms
64 bytes from 20.0.1.53: icmp_seq=1364 ttl=255 time=0.963 ms
64 bytes from 20.0.1.53: icmp_seq=1365 ttl=255 time=0.936 ms
64 bytes from 20.0.1.53: icmp_seq=1366 ttl=255 time=0.940 ms
64 bytes from 20.0.1.53: icmp_seq=1367 ttl=255 time=0.925 ms
64 bytes from 20.0.1.53: icmp_seq=1368 ttl=255 time=0.981 ms
64 bytes from 20.0.1.53: icmp_seq=1369 ttl=255 time=0.923 ms
64 bytes from 20.0.1.53: icmp_seq=1370 ttl=255 time=1.01 ms
64 bytes from 20.0.1.53: icmp_seq=1371 ttl=255 time=0.980 ms
64 bytes from 20.0.1.53: icmp_seq=1372 ttl=255 time=1.00 ms
64 bytes from 20.0.1.53: icmp_seq=1373 ttl=255 time=0.969 ms
64 bytes from 20.0.1.53: icmp_seq=1374 ttl=255 time=0.926 ms
64 bytes from 20.0.1.53: icmp_seq=1375 ttl=255 time=0.920 ms
64 bytes from 20.0.1.53: icmp_seq=1376 ttl=255 time=1.01 ms
64 bytes from 20.0.1.53: icmp_seq=1377 ttl=255 time=0.995 ms
i-0269d51e8abd521d5 (instance-1)
PublicIPs: 100.25.255.205 PrivateIPs: 10.0.1.35

```
   ,     #_
   ~\_  ####_        Amazon Linux 2
  ~~  \_#####\
  ~~     \###|       AL2 End of Life is 2025-06-30.
  ~~       \#/ ___
   ~~       V~' '->
    ~~~         /    A newer version of Amazon Linux is available!
      ~~._.   _/
         _/ _/       Amazon Linux 2023, GA and supported until 2028-03-15.
       _/m/'           https://aws.amazon.com/linux/amazon-linux-2023/

[ec2-user@ip-20-0-1-53 ~]$ sudo -i
[root@ip-20-0-1-53 ~]# ping 10.0.1.35
PING 10.0.1.35 (10.0.1.35) 56(84) bytes of data.
64 bytes from 10.0.1.35: icmp_seq=1179 ttl=255 time=0.945 ms
64 bytes from 10.0.1.35: icmp_seq=1180 ttl=255 time=1.02 ms
64 bytes from 10.0.1.35: icmp_seq=1181 ttl=255 time=0.965 ms
64 bytes from 10.0.1.35: icmp_seq=1182 ttl=255 time=0.963 ms
64 bytes from 10.0.1.35: icmp_seq=1183 ttl=255 time=1.24 ms
64 bytes from 10.0.1.35: icmp_seq=1184 ttl=255 time=0.950 ms
64 bytes from 10.0.1.35: icmp_seq=1185 ttl=255 time=0.929 ms
64 bytes from 10.0.1.35: icmp_seq=1186 ttl=255 time=0.982 ms
64 bytes from 10.0.1.35: icmp_seq=1187 ttl=255 time=0.929 ms
64 bytes from 10.0.1.35: icmp_seq=1188 ttl=255 time=1.92 ms
64 bytes from 10.0.1.35: icmp_seq=1189 ttl=255 time=0.964 ms
64 bytes from 10.0.1.35: icmp_seq=1190 ttl=255 time=0.944 ms
64 bytes from 10.0.1.35: icmp_seq=1191 ttl=255 time=1.02 ms
```

i-02ab6c360f01f8e2a (innstance-2)

PublicIPs: 18.207.193.255 PrivateIPs: 20.0.1.53

Terraform code to create a VPC:

1) Terraform code

a. Provider

```
provider "aws" {
region = "us-east-1"
access_key = "AKIA365MTWIRTVUSQG4O"
secret_key = "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"
}
~
~
~
~
```

b. Resource (VPC). Here "**enable_dns_hostnames**" is essential to generate public IPV4 dns name and private IPV4 dns hostname.

Usually, we get a public IPV4 hostname as "ec2-203-0-113-25.compute-1.amazonaws.com" and private IPV4 dns hostname as "**ip-10-0-1-12.ec2.internal".** So**,** in order to get these names we mention the attribute "**enable_dns_hostnames**" in the terraform code.

To access a web server or service that needs to be accessed over the internet we need public IPV4 dns name but it's difficult to remember the entire public IPV4 dns name to access, so we use Route 53 (aws service) or Go Daddy t o purchase a domain name that's easier for the public to remember and access. To access the resources internally or for internal communication we need private IPV4 dns name.

```
resource "aws_vpc" "one"  {
tags = {
Name = "kalki-vpc"
}
cidr_block = "10.0.0.0/16"
instance_tenancy = "default"
enable_dns_hostnames = "true"
}
~
```

c. Subnet

```
resource "aws_subnet" "two" {
vpc_id = aws_vpc.one.id
tags = {
Name = "my-subnet"
}
availability_zone = "us-east-1b"
cidr_block = "10.0.1.0/22"
}
```

```
resource "aws_subnet" "two" {
vpc_id = aws_vpc.one.id
tags = {
Name = "my-subnet"
}
availability_zone = "us-east-1b"
cidr_block = "10.0.0.0/22"
}
~
```

d. Internet gateway

```
resource "aws_internet_gateway" "three" {
tags = {
Name = "my-igw"
}
vpc_id = aws_vpc.one.id
}
~
~
```

e. Route table, the route table gets created automatically when vpc is created. However, we need to add a route to attach the internet gateway to the route table.

```
resource "aws_route_table" "four" {
tags = {
Name = "my-igw"
}
vpc_id = aws_vpc.one.id

route {
cidr_block = "0.0.0.0/0"
gateway_id = aws_internet_gateway.three.id
}
}
```

2) Terraform plan

```
        }
      + tags_all                                = {
          + "Name" = "kalki-vpc"
        }
    }

Plan: 4 to add, 0 to change, 0 to destroy.
```

The below images show what are the resources that are getting created.

```
Terraform used the selected providers to generate the following execution plan. R
  + create

Terraform will perform the following actions:

  # aws_internet_gateway.three will be created
  + resource "aws_internet_gateway" "three" {
      + arn      = (known after apply)
      + id       = (known after apply)
      + owner_id = (known after apply)
      + tags     = {
          + "Name" = "my-igw"
        }
      + tags_all = {
          + "Name" = "my-igw"
        }
      + vpc_id   = (known after apply)
    }

  # aws_route_table.four will be created
  + resource "aws_route_table" "four" {
      + arn              = (known after apply)
      + id               = (known after apply)
      + owner_id         = (known after apply)
      + propagating_vgws = (known after apply)
      + route            = [
          + {
```

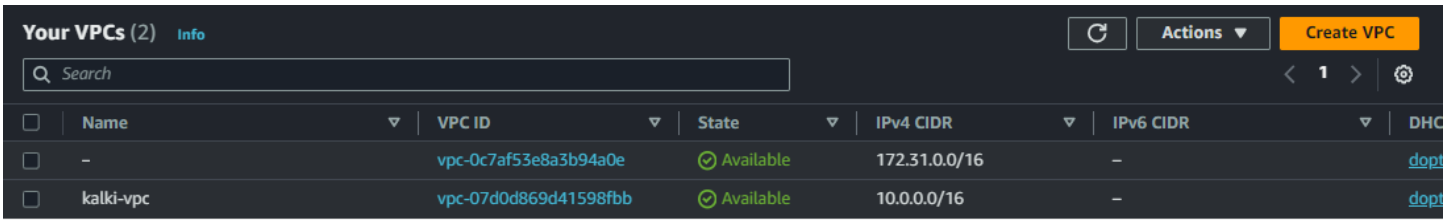
```
# aws_subnet.two will be created
+ resource "aws_subnet" "two" {
    + arn                                            = (known after apply)
    + assign_ipv6_address_on_creation                = false
    + availability_zone                              = "us-east-1b"
    + availability_zone_id                           = (known after apply)
    + cidr_block                                     = "10.0.0.0/22"
    + enable_dns64                                   = false
    + enable_resource_name_dns_a_record_on_launch    = false
    + enable_resource_name_dns_aaaa_record_on_launch = false
    + id                                             = (known after apply)
    + ipv6_cidr_block_association_id                 = (known after apply)
    + ipv6_native                                    = false
    + map_public_ip_on_launch                        = false
    + owner_id                                       = (known after apply)
    + private_dns_hostname_type_on_launch            = (known after apply)
    + tags                                           = {
        + "Name" = "my-subnet"
      }
    + tags_all                                       = {
        + "Name" = "my-subnet"
      }
    + vpc_id                                         = (known after apply)
  }

# aws_vpc.one will be created
+ resource "aws_vpc" "one" {
```

3) Terraform apply.

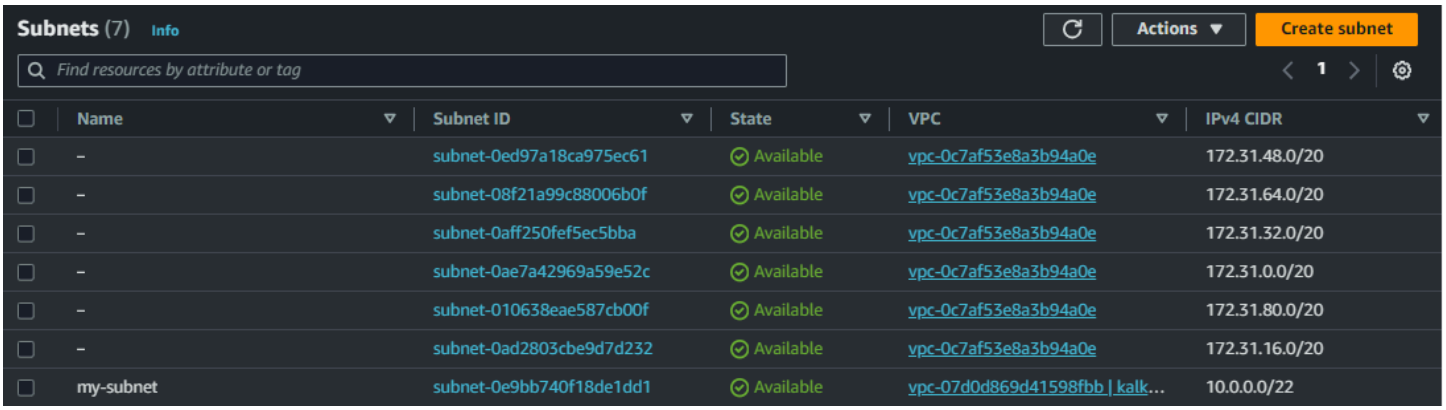
```
Plan: 4 to add, 0 to change, 0 to destroy.
aws_vpc.one: Creating...
aws_vpc.one: Still creating... [10s elapsed]
aws_vpc.one: Creation complete after 11s [id=vpc-07d0d869d41598fbb]
aws_internet_gateway.three: Creating...
aws_subnet.two: Creating...
aws_internet_gateway.three: Creation complete after 0s [id=igw-078c83b8f96a33bf6]
aws_route_table.four: Creating...
aws_subnet.two: Creation complete after 1s [id=subnet-0e9bb740f18de1dd1]
aws_route_table.four: Creation complete after 1s [id=rtb-0da3fb041252e929b]

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
[root@ip-172-31-27-62 ~]#
```

4) Pictorial representation of all resources that are created.

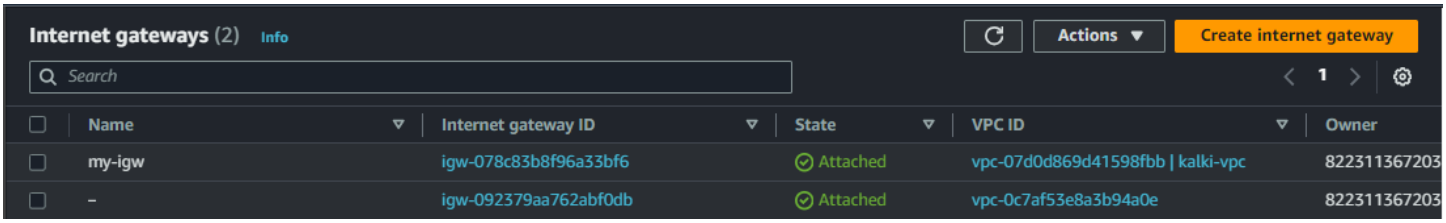
VPC:



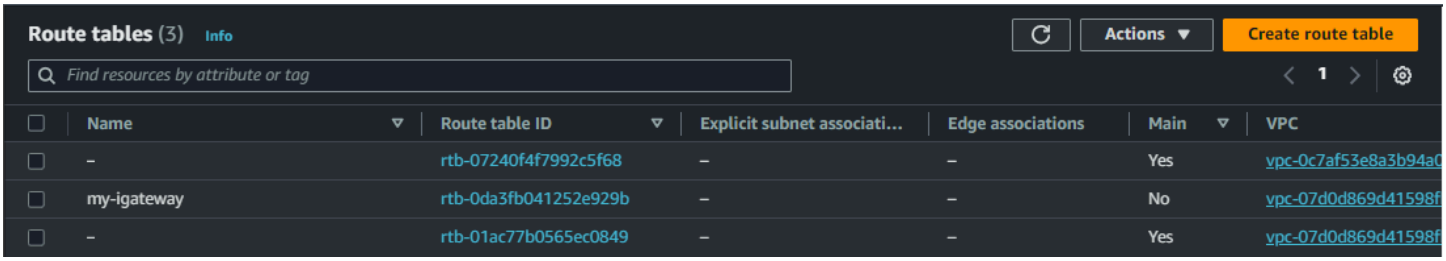
Subnet:



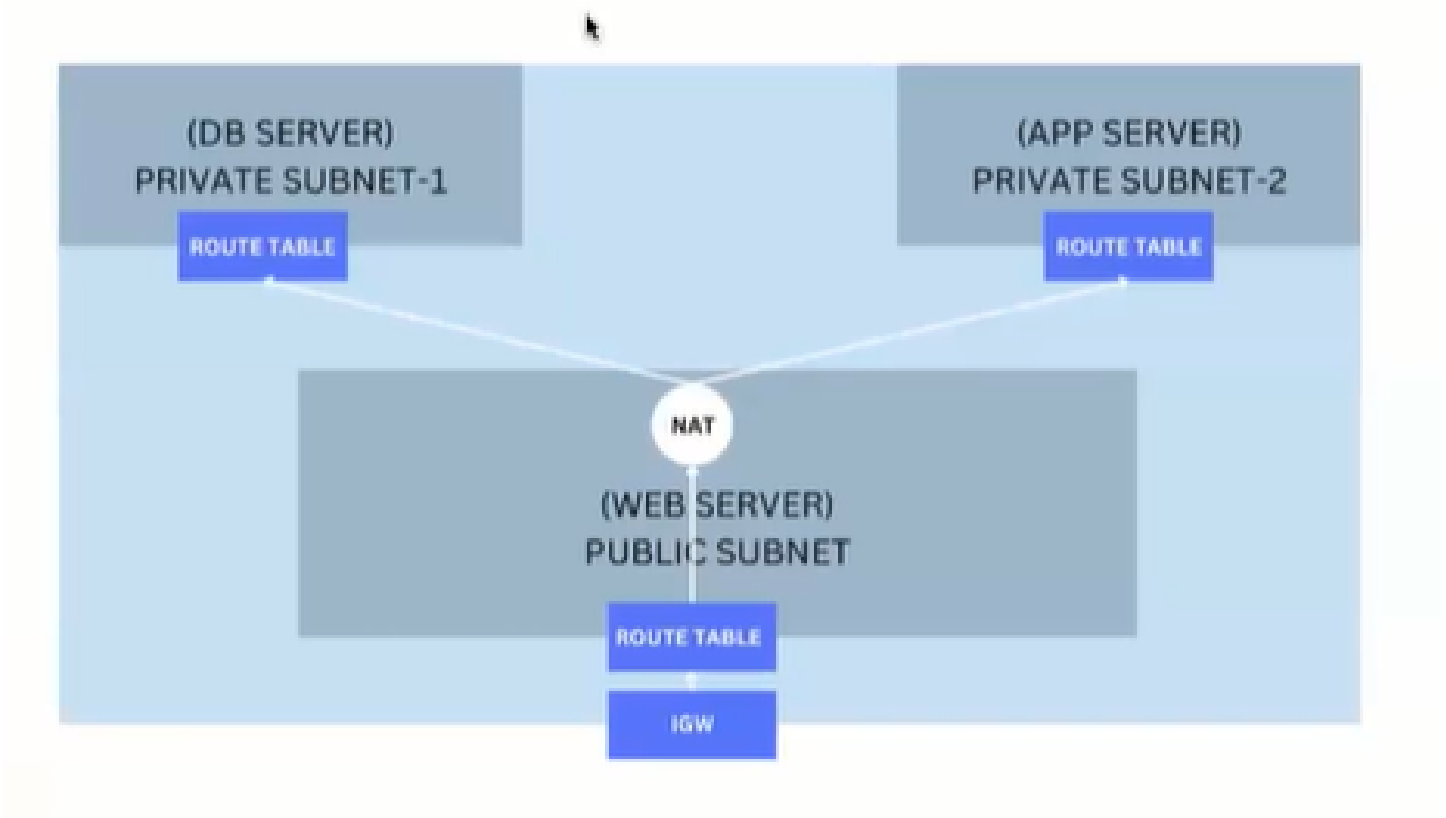
Internet Gateway:

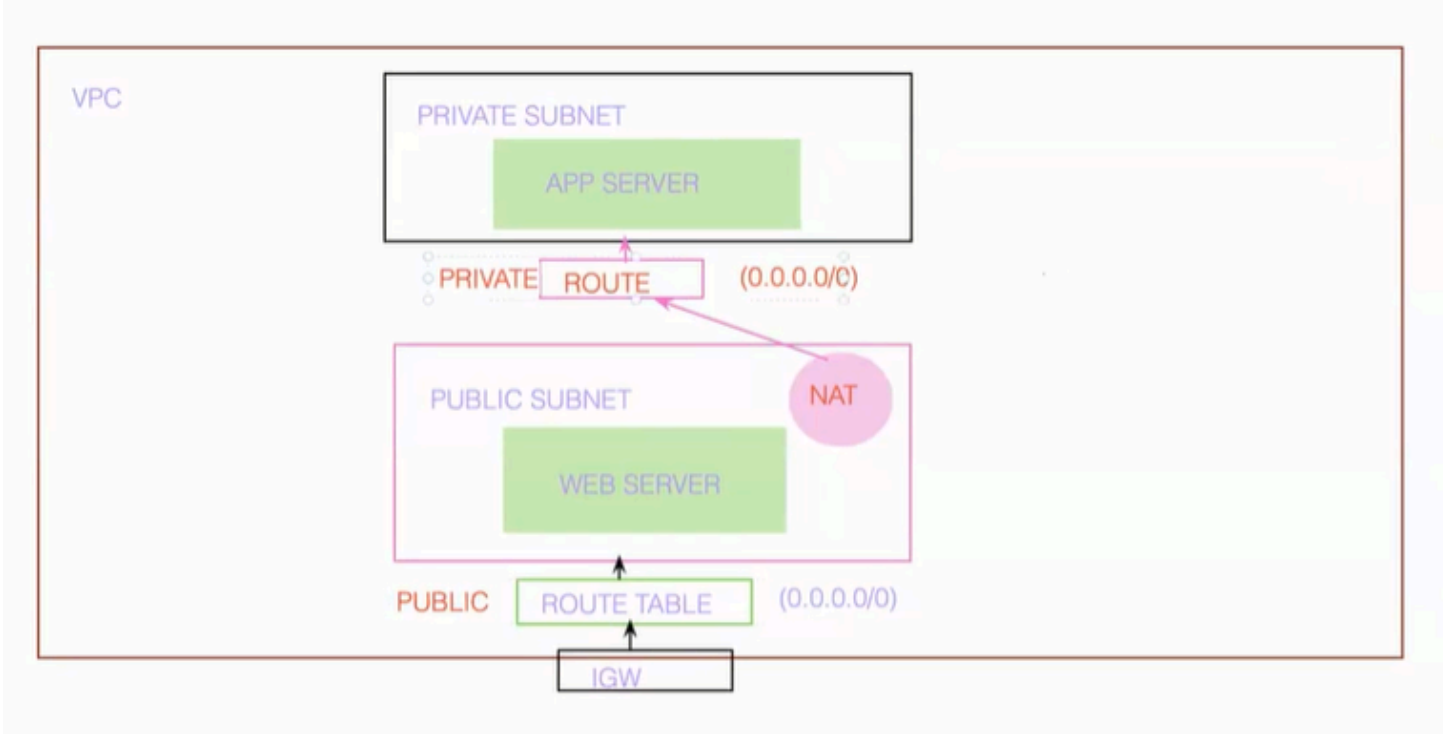


Route table:



TWO TIER ARCHITECTURE OF VPC:





EXPLANATION OF ABOVE ARCHITECTURE:

Components and it's connectivity:

First of all, we've a VPC, it has two subnets in it. One is public subnet and another one is private subnet.

The internet gateway is attached to the VPC and to the public route table. The public route table is attached to the public subnet. We've a NAT Gateway in the public subnet which is attached to the private route table, the private route table is attached to the private subnet.

How each component is interrelated to one another and how it works?

We've a VPC, it has an instance in a public subnet that has a web server installed and assume that the Facebook application is deployed. In order to access the Facebook app, we'll be entering the credentials to login through the home page. The public route table (0.0.0.0/0) has the route added to it to allow the requests of users and with the help of internet gateway attached to the public route table.

If the credentials are correct, then they'd be allowed to the access the resources (Facebook app) in the private subnet which is the backend of the application with the help of a NAT gateway (which is in the public subnet) and with the route defined in the private route table. The private route table has a route that defines that the application can only be accessed by those whose credentials are correct. If the credentials are wrong the APIs that are written by developers will stop entering the users into the private subnet to access the application.

TERRAFORM WORKSPACE

Terraform workspace is a kind of private or an isolated place where you can write different terraform modules for different environments such as dev, test, prod. So in order to isolate the modules of different environments, terraform workspace is useful. Due to workspace, the error finding is easy during the execution to find which environment module has the error to troubleshoot, the prone to the deletion of the infrastructure is solved as it requires to destroy the resources first and then a workspace can be deleted, code reusability is efficient and saves time. It also provides to switch between different environments to execute the terraform modules.

Terraform starts with a single, default workspace named "**default**" that you cannot delete. If you have not created a new workspace, you are using the default workspace in your Terraform working directory.

When you run terraform plan in a new workspace, Terraform does not access existing resources in other workspaces. These resources still physically exist, but you must switch workspaces to manage them.

EBS AND EFS

EBS : EBS is Elastic Block Storage which allows you to create storage volumes and attach them to an EC2 instance. An EBS volume created in an availability zone is '**automatically replicated**' and avoids the loss

of data in case of any hardware component failure. The EBS volumes are persistent, which means it stores the data in it even if the instance is terminated only if you unselect "**Delete on termination**". You can attach an EBS to any other instance in the same availability zone.

You can create a snapshot of the volume and attach the snapshot to another instance so the data in the volume of created snapshot will be reflecting in new instance as well. That's how snapshot is useful.

EFS: It's a Elastic File System is a fully elastic storage that helps you to share the data like used for replication of data from one server to another.

Implementation of data replication between two servers with EFS:

1) Install EFS client on both the instances with sudo yum install -y amazon-efs-utils.
2) First go to AWS console and then search for EFS, create EFS.
3) Before you mount, remove the default security group from your EFS and add the SG of your instance.
4) To mount the EFs to your instance. Click on attach and then use the command from there in both the instances to mount it. Now the data replication takes place from one server to another server.