1) Execute the script shown in video.

terraform provisioners:

Terraform provisioners allow you to execute scripts or actions on a resource after it has been created or destroyed. Provisioners are useful for tasks such as configuring servers, installing packages, or running custom scripts.

Here are the key types of provisioners in Terraform.

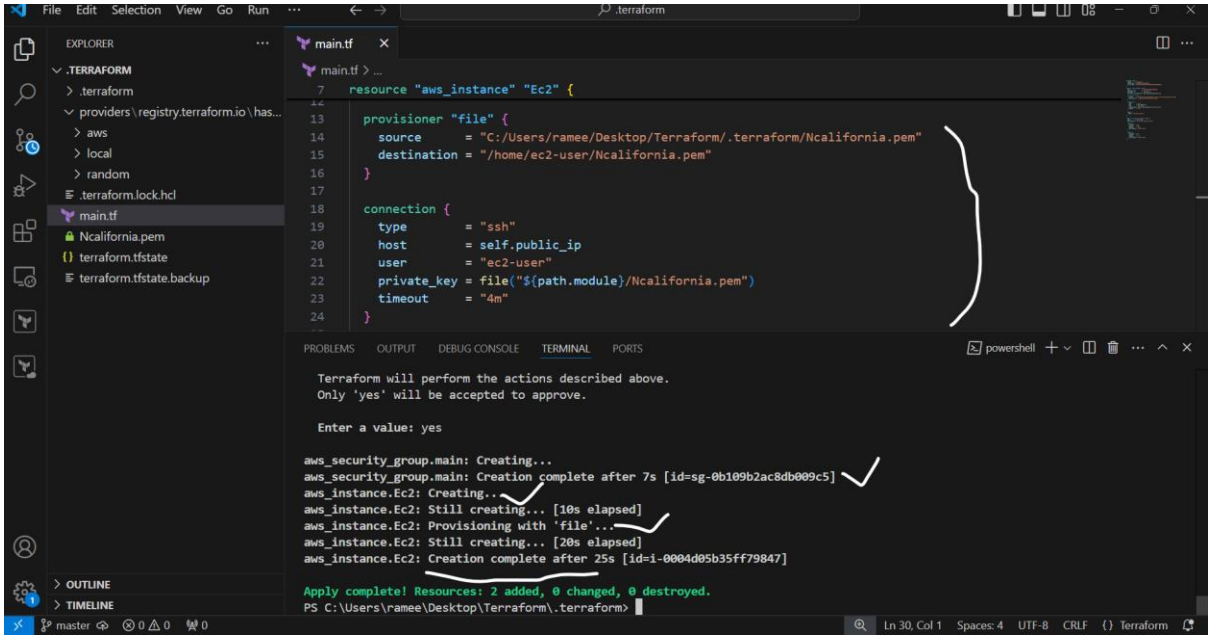
File Provisioner : The File provisioner copies files or directories from your local machine to a newly created instance.

This is script I am executed.

```
provider "aws" {
  region     = "us-west-1"
  access_key = "AKIAW3MD7G7LBGTD56E7"
  secret_key = "hvPqluTyzVWzIBx1vfEORKKo5kVEuQ8YBzN80AsS"
}

resource "aws_instance" "Ec2" {
  ami                    = "ami-0175bdd48fdb0973b"
  instance_type          = "t2.micro"
  key_name               = "Ncalifornia"
  vpc_security_group_ids = [aws_security_group.main.id]

  provisioner "file" {
    source    ="C:/Users/ramee/Desktop/Terraform/.terraform/Ncalifornia.pem"
    destination = "/home/ec2-user/Ncalifornia.pem"
  }

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ec2-user"
    private_key = file("${path.module}/Ncalifornia.pem")
    timeout     = "4m"
  }

  tags = {
    Name = "My_EC2_Instance"
  }
}

resource "aws_security_group" "main" {
  name        = "allow_ssh"
  description = "Allow SSH inbound traffic"
```

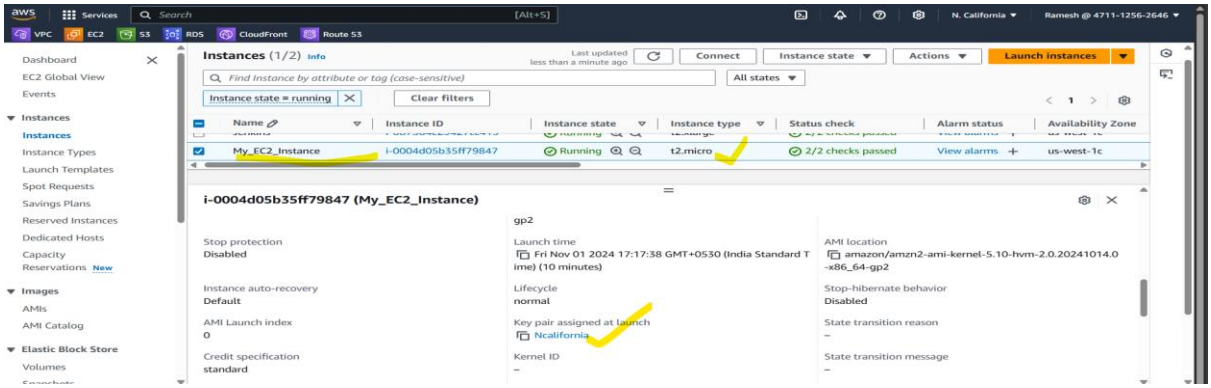
```
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

Here this will execute successful.

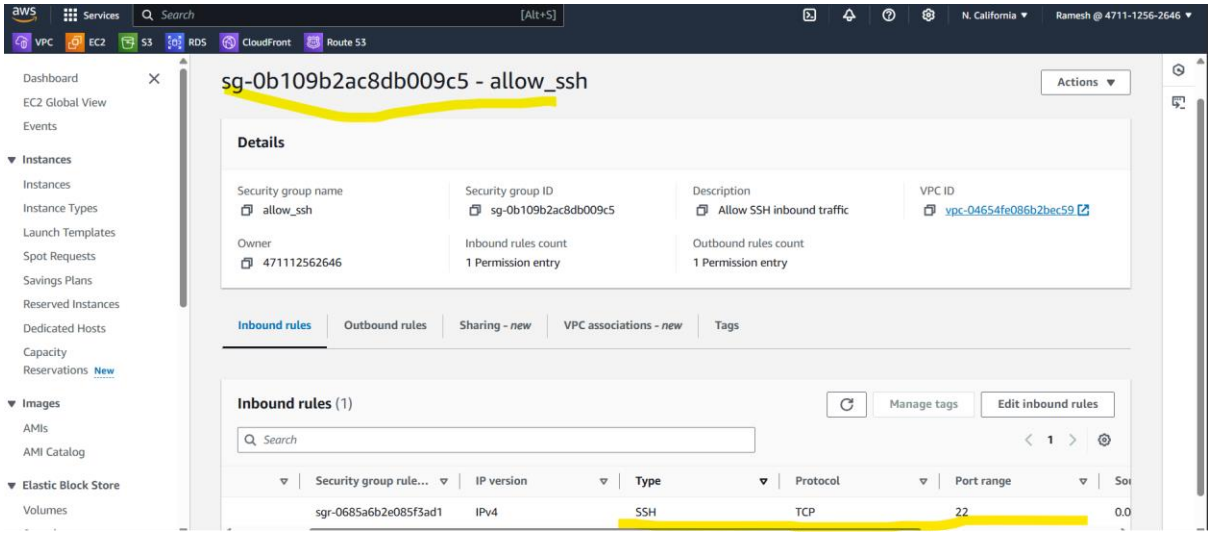


Now we check the file is there or not in instance and also we have to check the security groups.

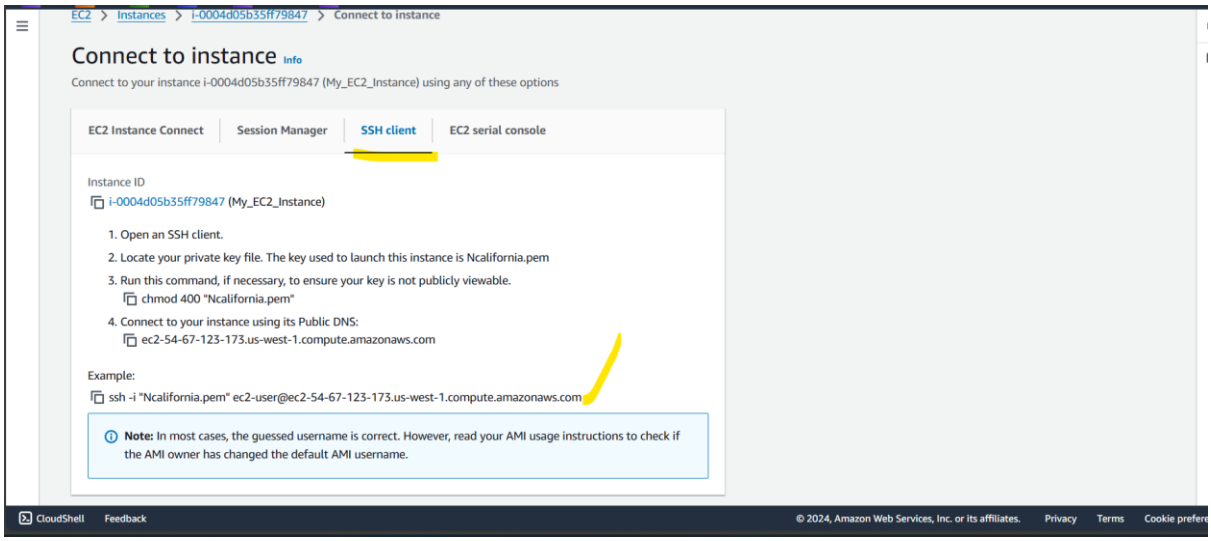
My Ec2 instance is created with Ncalifornia.pem and t2.micro instance type.



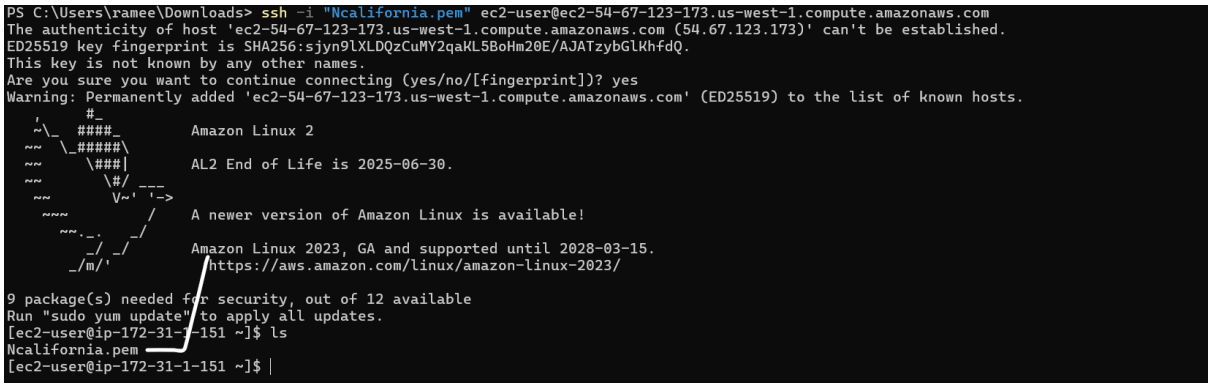
My security group is created



To check we have to copy below example.



Now our file is there.



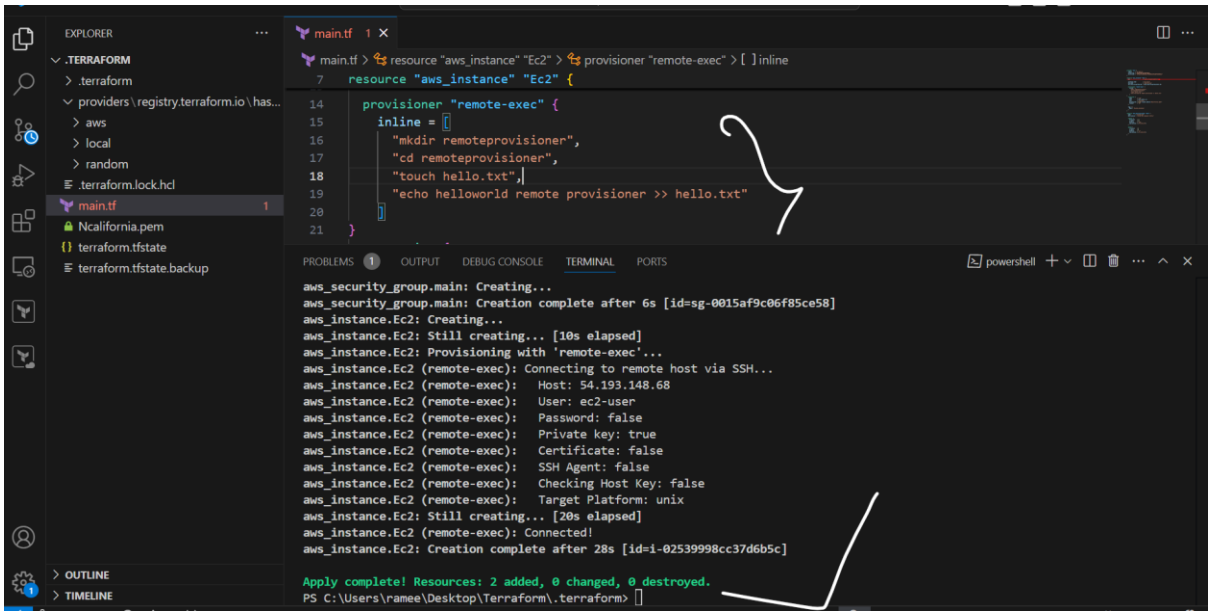
Done.

remote-exec provisioner:

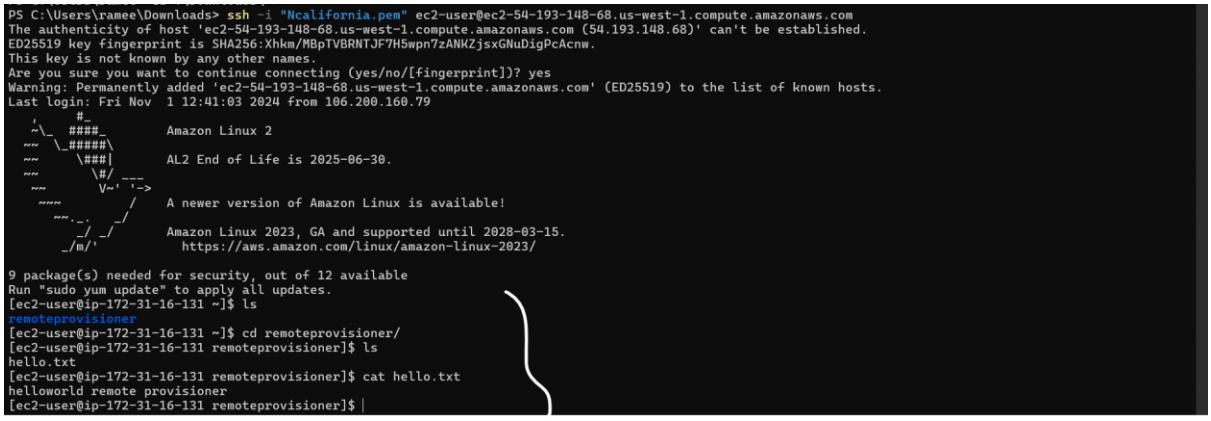
As the name suggests remote-exec it is always going to work on the remote machine. With the help of the remote-exec you can specify the commands of shell scripts that want to execute on the remote machine.

Using the remote-exec provisioner, I am creating a directory. Within this directory, I am creating a hello.txt file and redirecting content into it.

Just here I am changing the script only provisioner block.



Now we want to check server.

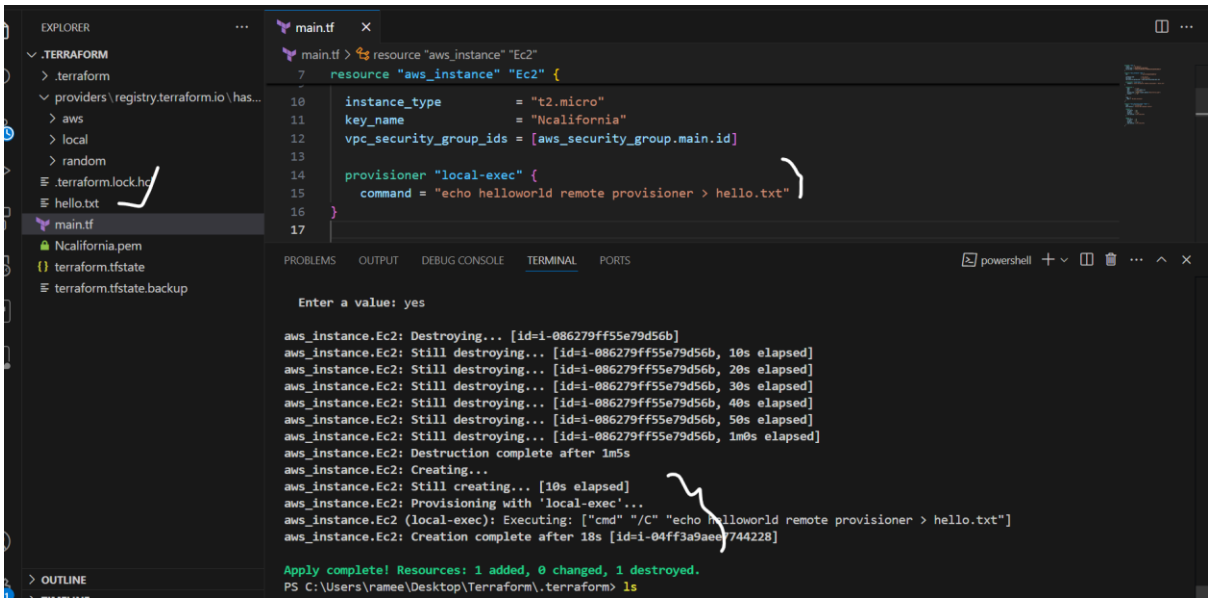


Done.

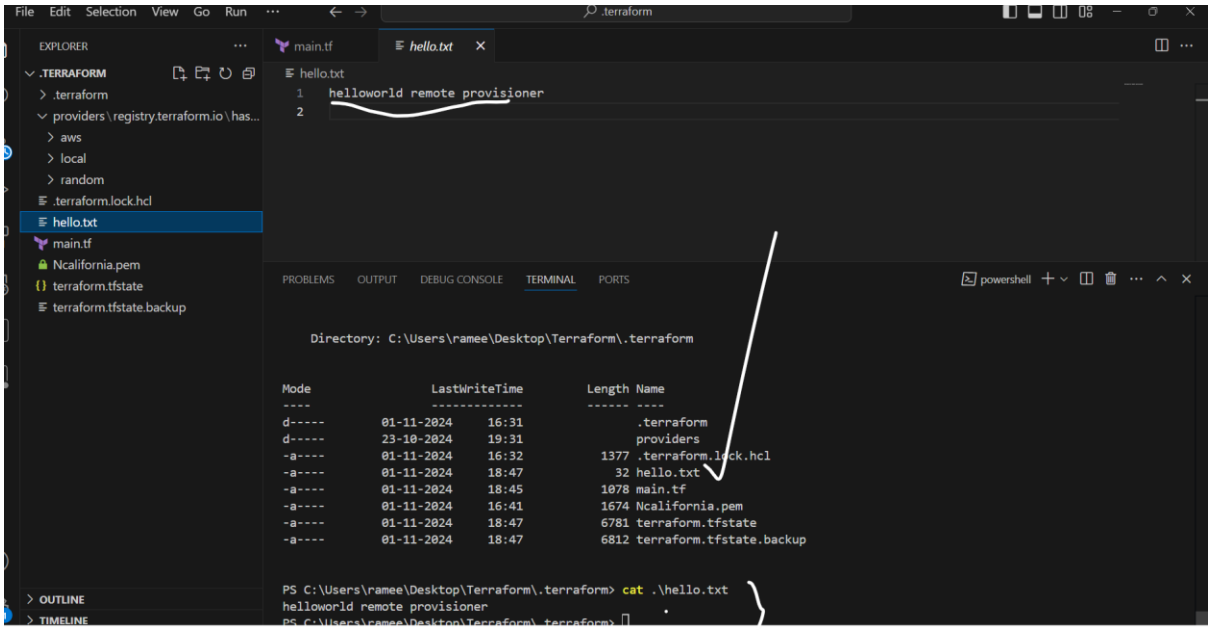
local-exec provisioner:

The next provisioner we are gonna talk about is local-exec provisioner. Basically, this provisioner is used when you want to perform some tasks onto your local machine where you have installed the terraform.

So local-exec provisioner is never used to perform any kind task on the remote machine. It will always be used to perform local operations onto your local machine.



Here local file is created with content.



Done.

Terraform provisioner behaviours:

==========

1) **Default Behavior**:

By default, provisioners run when a resource is created, allowing us to set it up as needed right away.

2) **Destroy Behavior**:

when = destroy allows us to run specific tasks when a resource is being deleted, which helps with cleanup or backup.

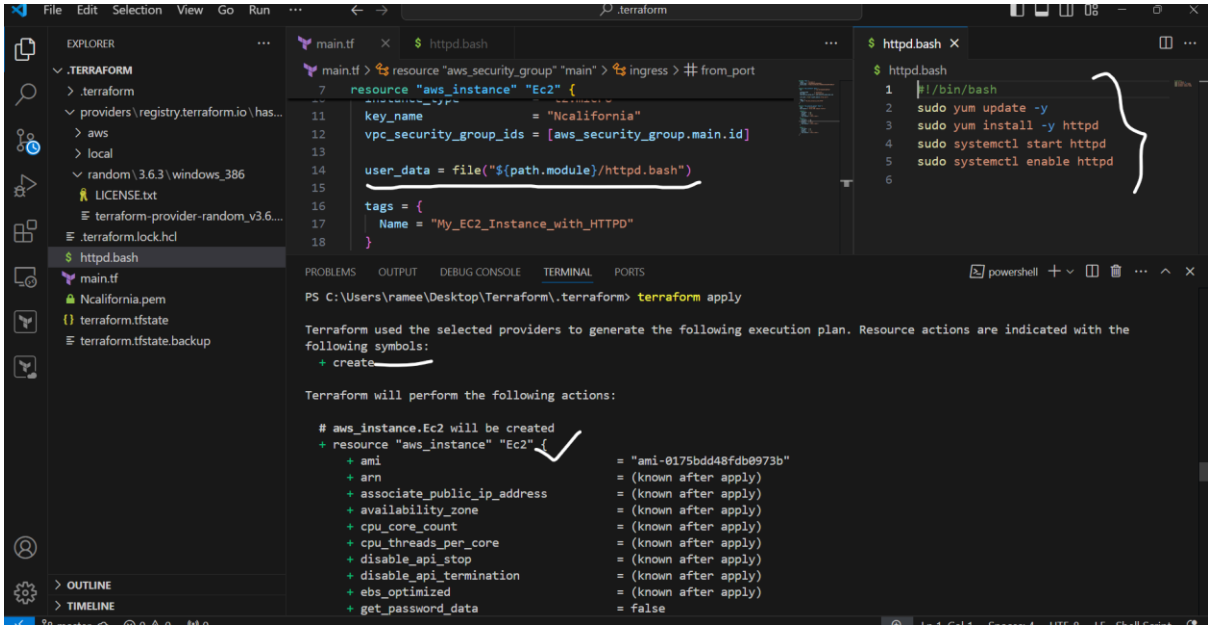
3)on_failure = fail --> to create the resource if the script gets failed.(But terraform will mark the reource as tainted)

4) On_fail = continue --> to create the resource and ignore the changes.

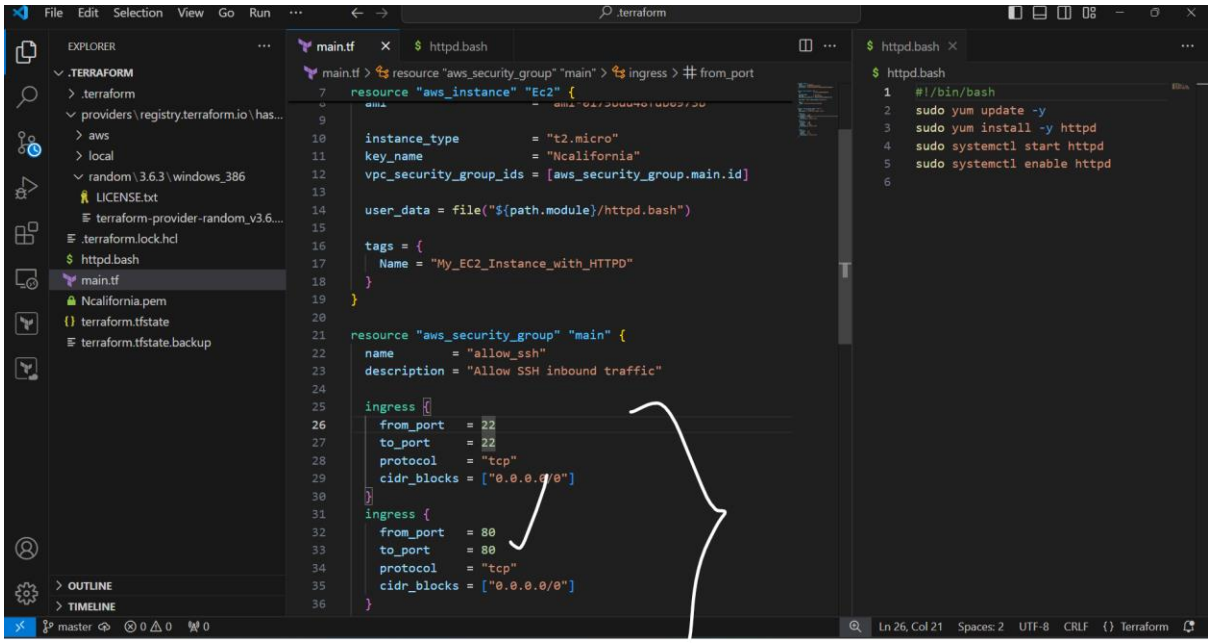
2) Create one ec2 instance with httpd installed using terraform script.

- Terraform configuration file (main.tf) uses the httpd.bash script to automate the setup of an EC2 instance with Apache HTTPD installed.

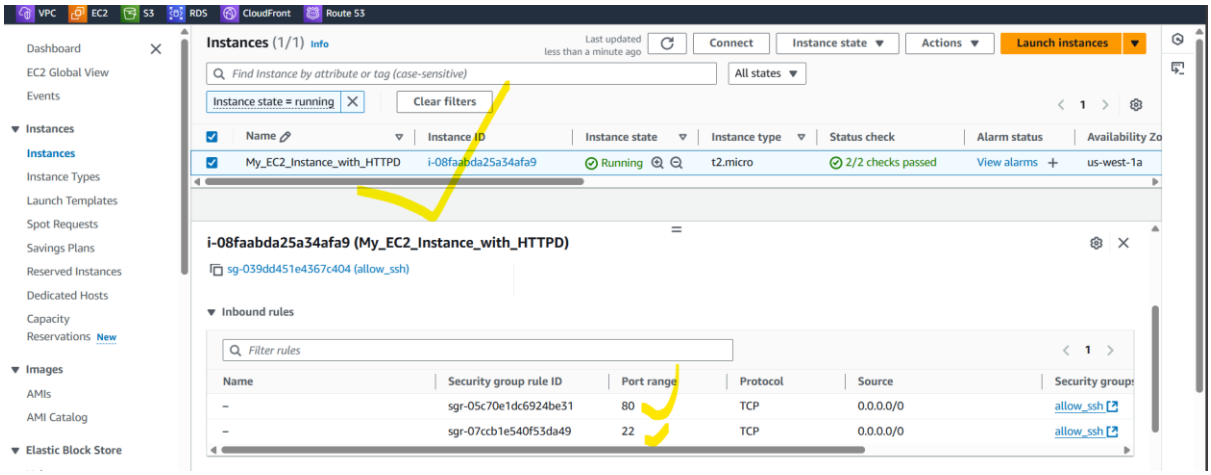
- The user_data = file("${path.module}/httpd.bash") line in main.tf calls httpd.bash as a user data script, which is executed on instance launch to install and start the HTTP server, making the instance ready to serve HTTP requests automatically.



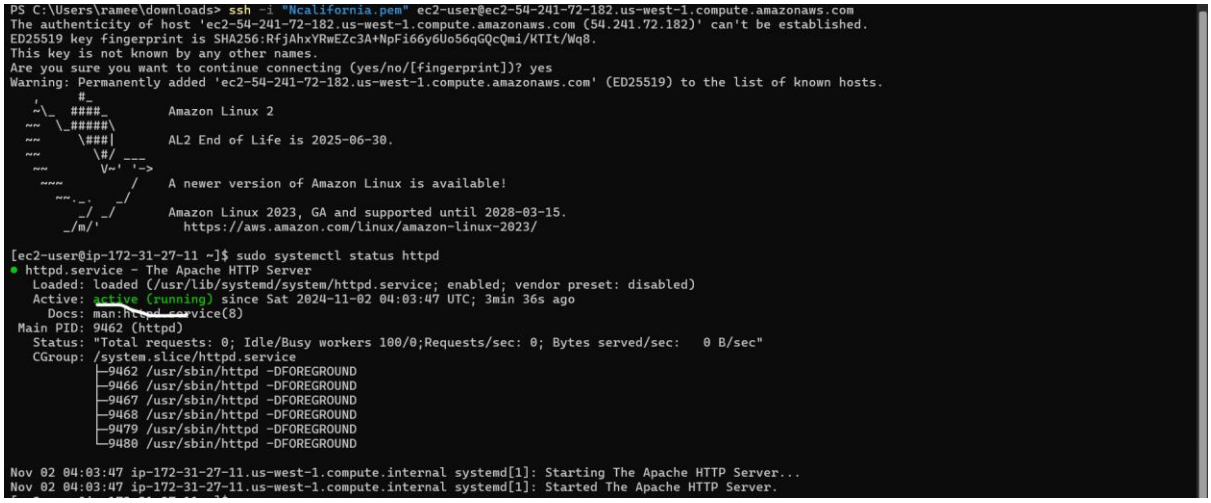
Here I am given the inbound rules port 22 and 80.



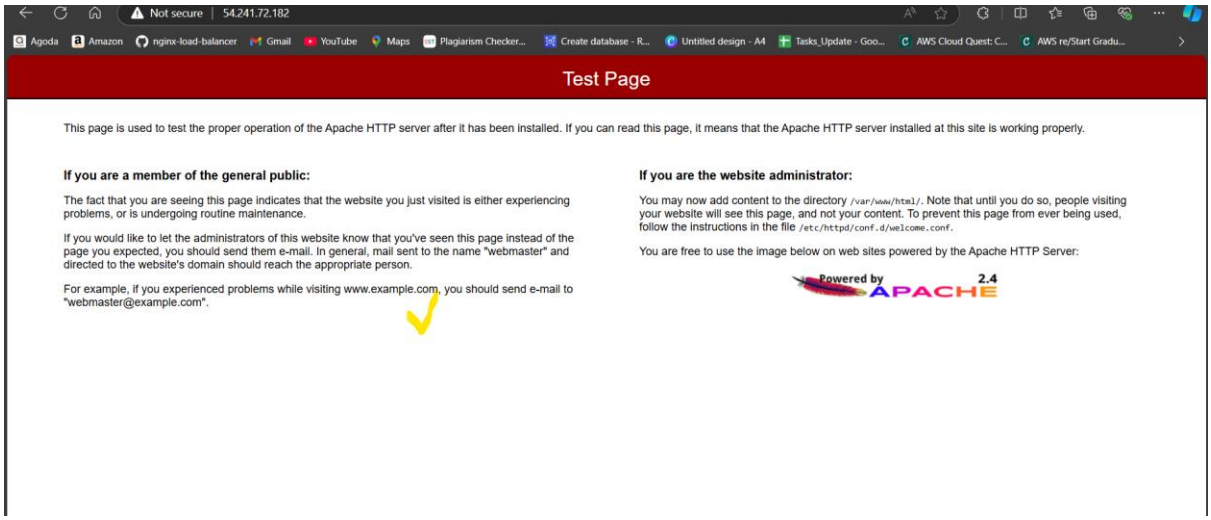
Here our instance is created with port 22 and 80.



Our httpd server status is Running.



Our httpd page also accessible .



Done.

Terraform state file

The Terraform state file is essentially a blueprint that tracks all the infrastructure Terraform manages. It stores information about each resource, including its unique ID, configuration metadata, and the actual state of the infrastructure. This allows Terraform to efficiently check for any differences between what's currently deployed and the desired configuration in the code.

Terraform Remote State and State Locking

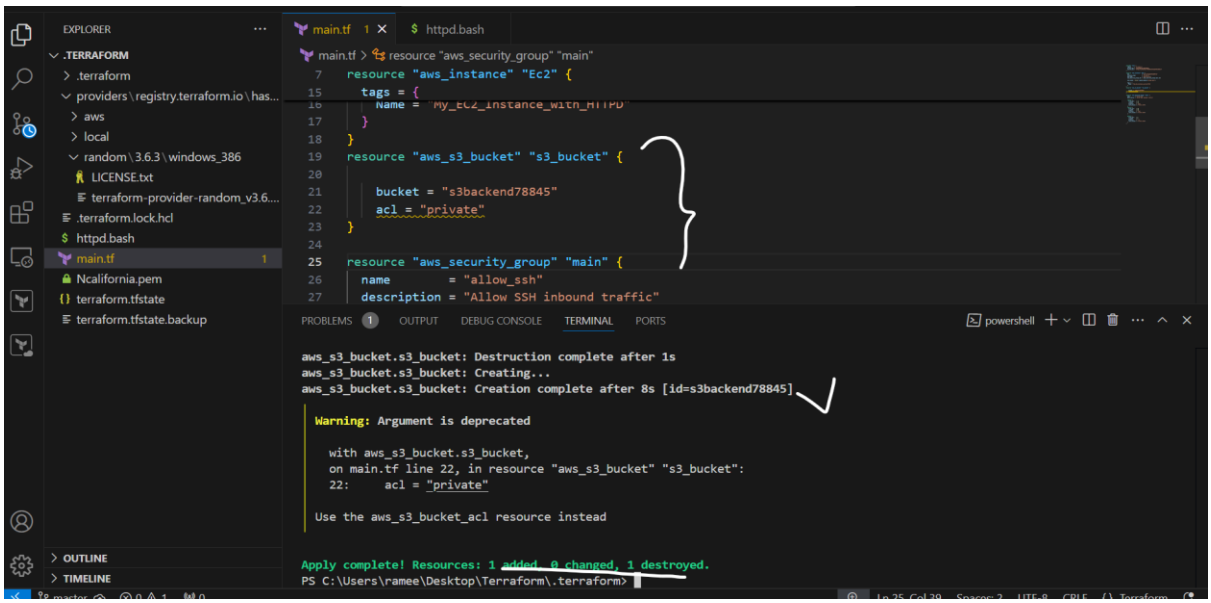
Storing Terraform Files

While you can store Terraform configuration and state files in GitHub, it's not the best practice. Instead, use services like **Amazon S3** or **Terraform Cloud** to keep your files secure.

Benefits of Remote State:

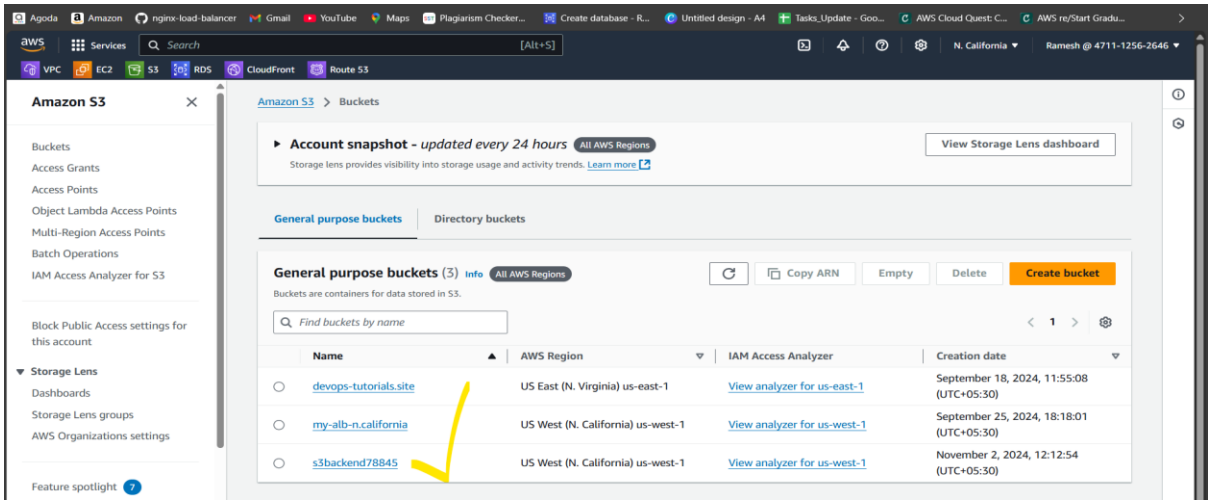
- **Central Access**: Team members can work from the same state file.
- **Better Security**: Services like S3 provide encryption and access controls.
- **Versioning**: Easily track changes and revert to previous states if needed.

1. Setup s3 as backend to the task 3.
   Here using this script I am created one s3 bucket.

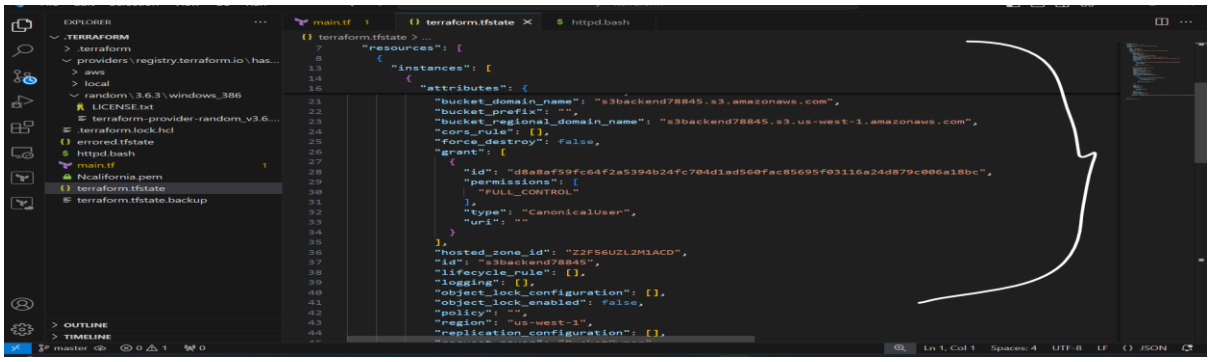


Now we have to check in aws console.

Here our s3 bucket is created.

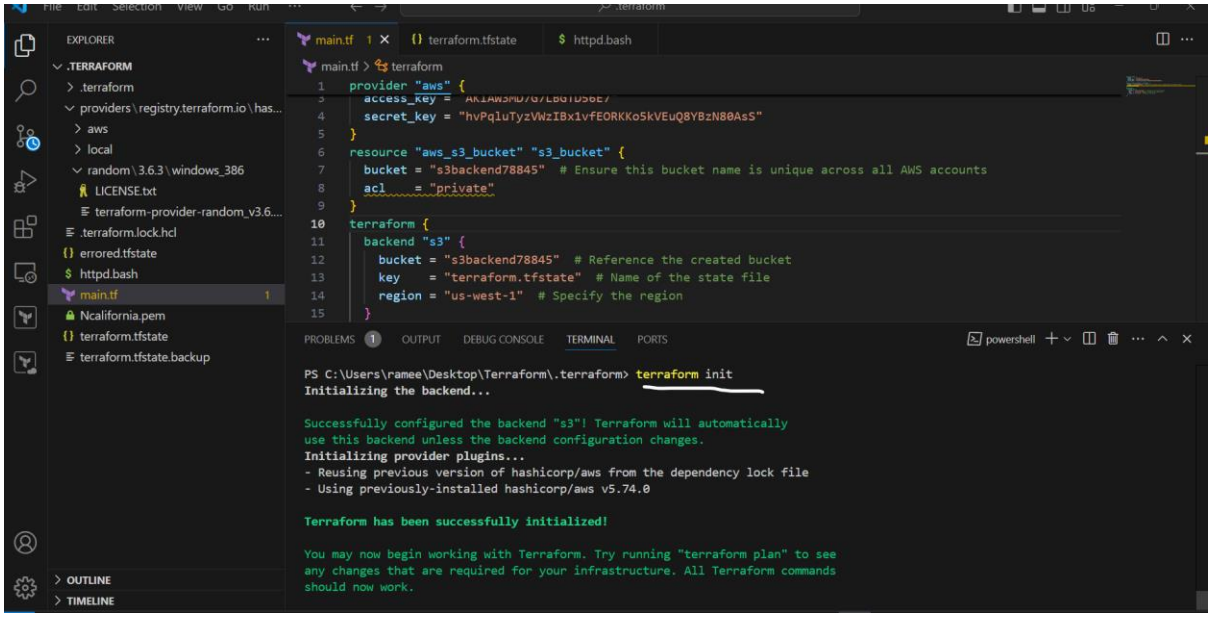


Now check here terraform.tfstate file we see content.



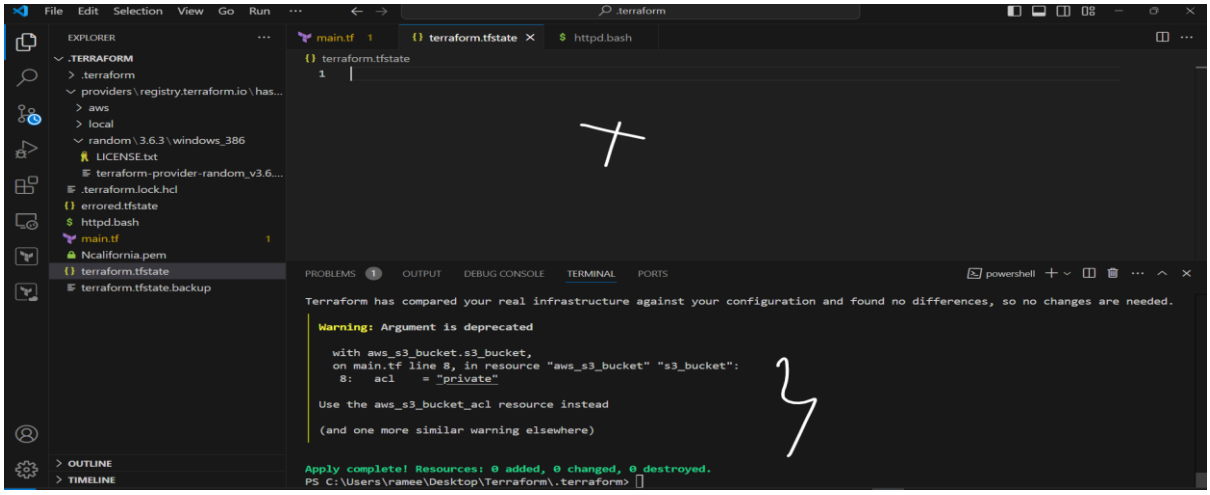
Now we have to setup backend.

First you need to initialize the all dependency's.

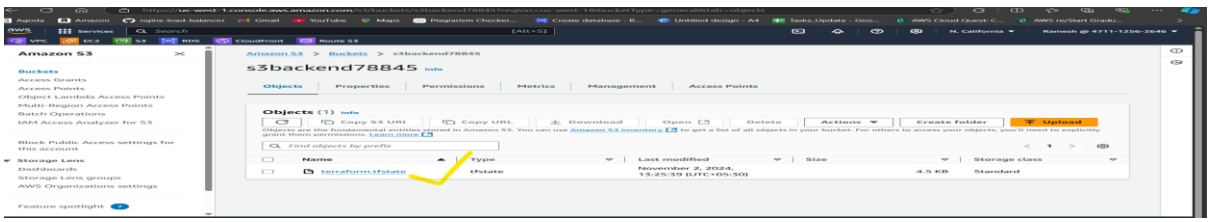


Now try apply the changes using the command terraform init.

After running terraform apply, the terraform.tfstate file is stored in the specified S3 bucket, which allows for remote state management. This means that instead of being visible in the local directory, the state file is securely maintained in S3, enabling collaboration and consistency across different environments and team members.



Now here our terraform.tfstate file is there.



3) Setup dynamo db locking for task3.

What is State Locking?

State locking prevents multiple users from making changes to the infrastructure at the same time. This is important because:

- **Avoid Conflicts**: Only one person can make changes at a time, preventing errors.
- **Maintain Consistency**: The state file always reflects the actual infrastructure.

Using S3 and DynamoDB

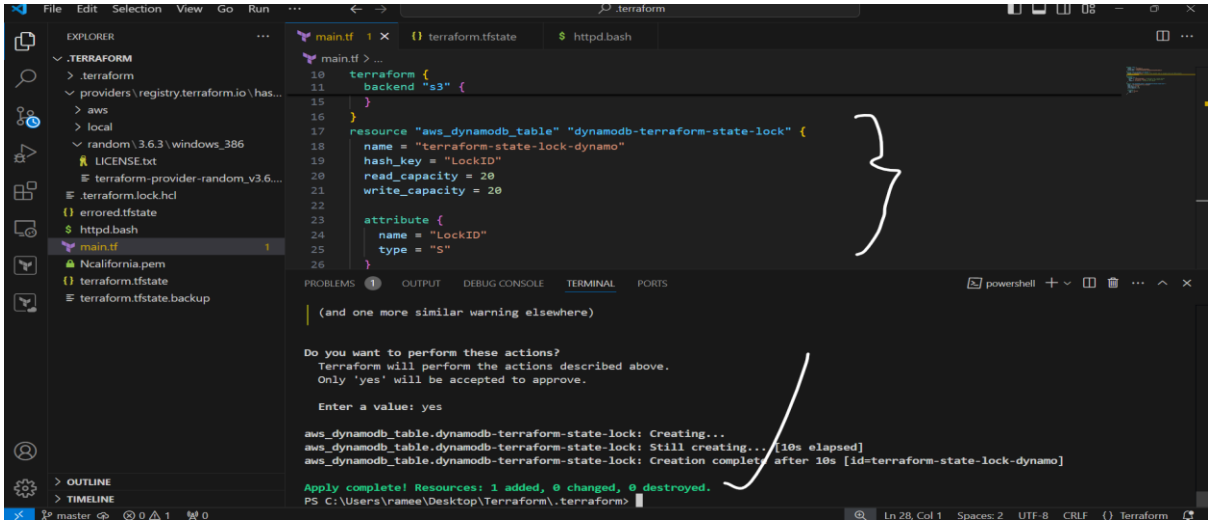
To effectively manage your state, use **Amazon S3** for storing the state file and **DynamoDB** for locking it.

2. **S3**: Stores the state file securely.
3. **Dynamo DB**: Locks the state so no one else can make changes while one user is working.

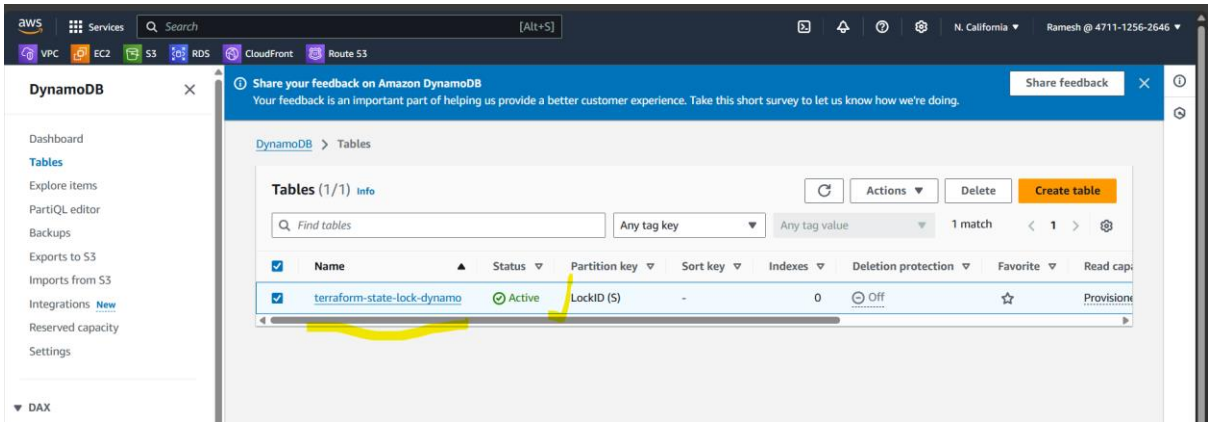
Create dynamo db using terraform:
==========

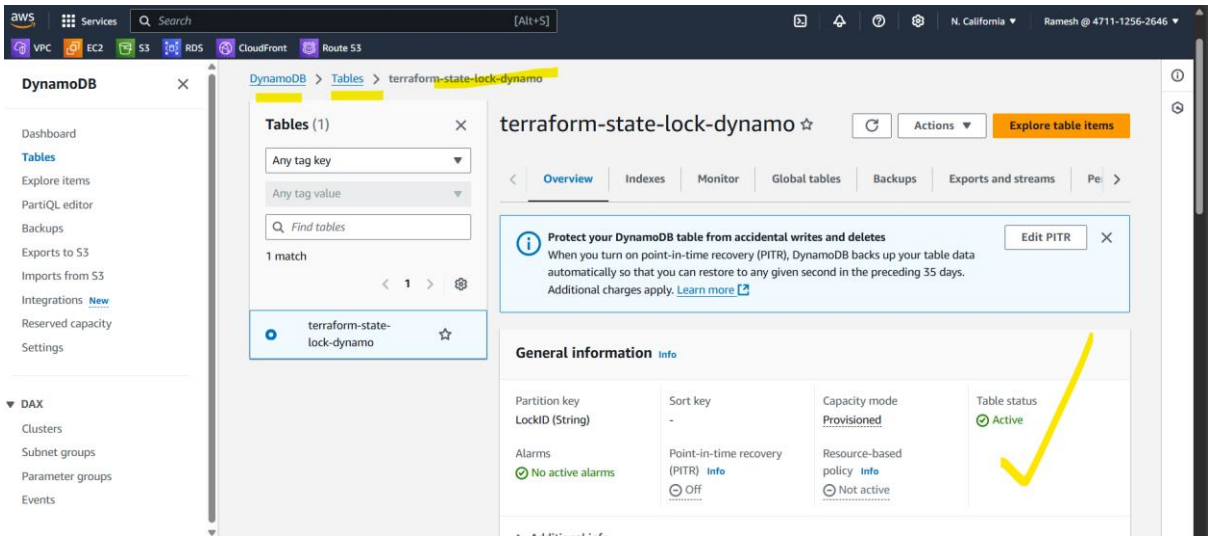
```
resource "aws_dynamodb_table" "dynamodb-terraform-state-lock" {
  name = "terraform-state-lock-dynamo"
  hash_key = "LockID"
  read_capacity = 20
  write_capacity = 20

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

After save the file and initialize the all dependency's.
Then enter terraform apply.



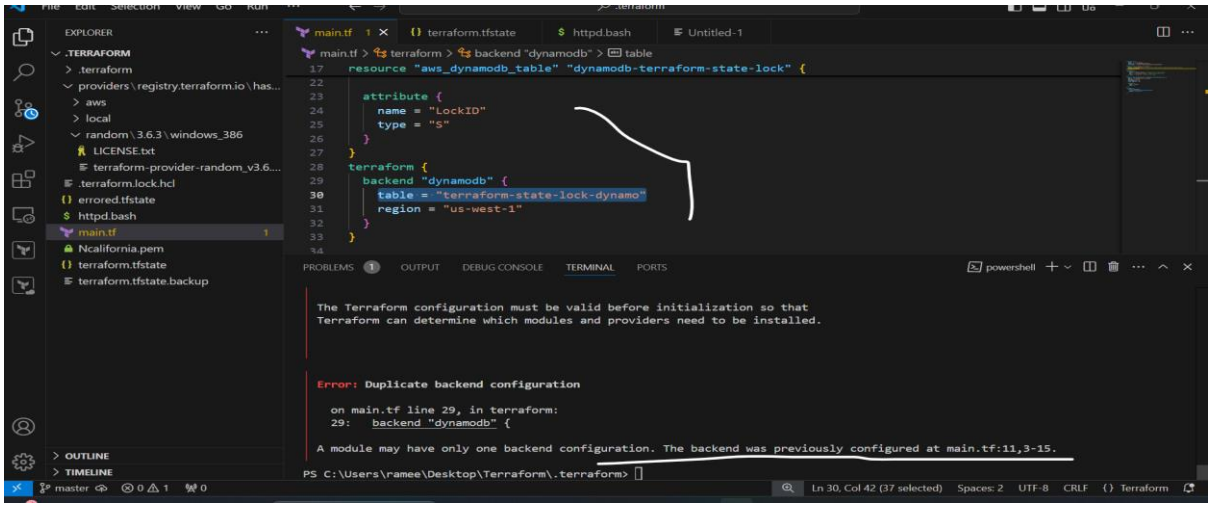
Now we have to check the dynamodb table is created or not.





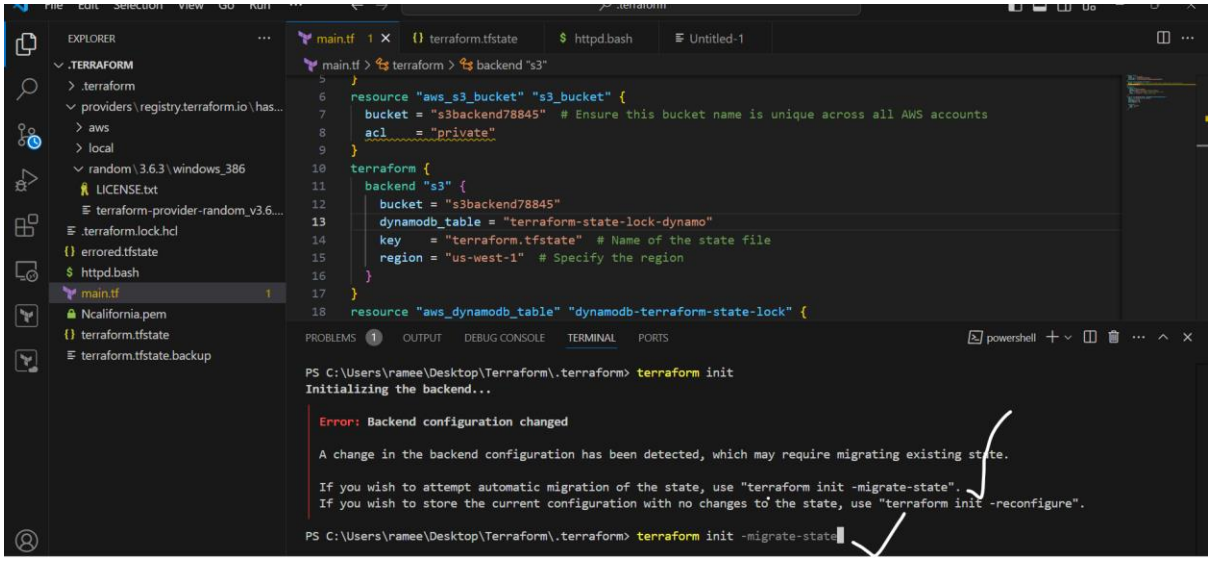
Now I want to Setup dynamo db locking.

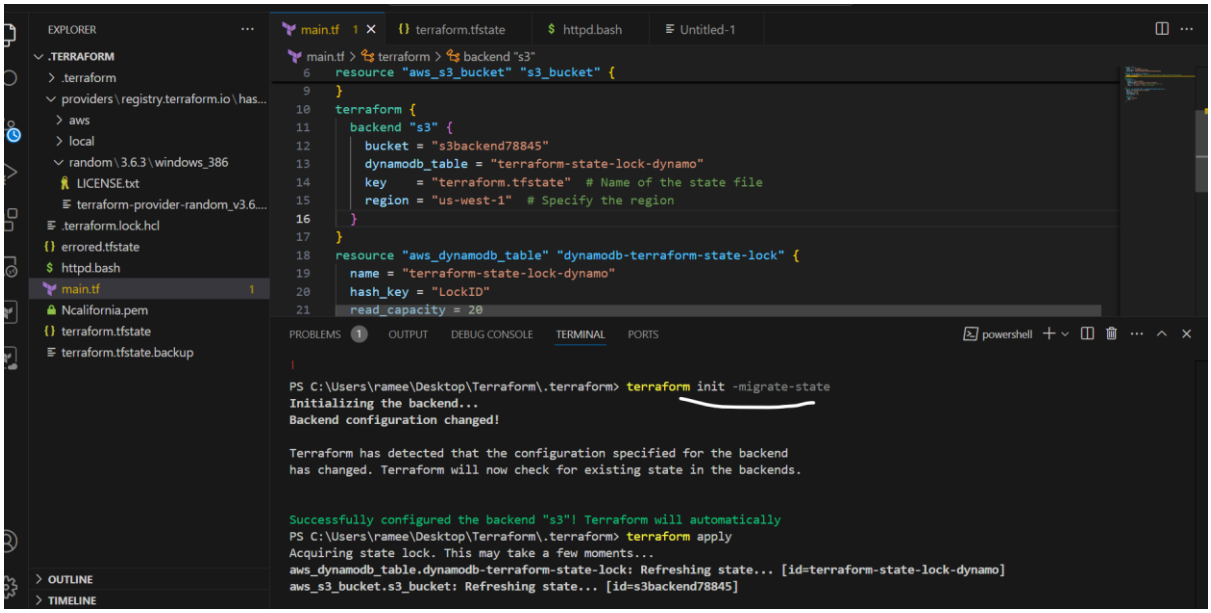
Here I am facing with issue is am already created on backend for s3 bucket. So it's not allowing second dynamo db backend locking.



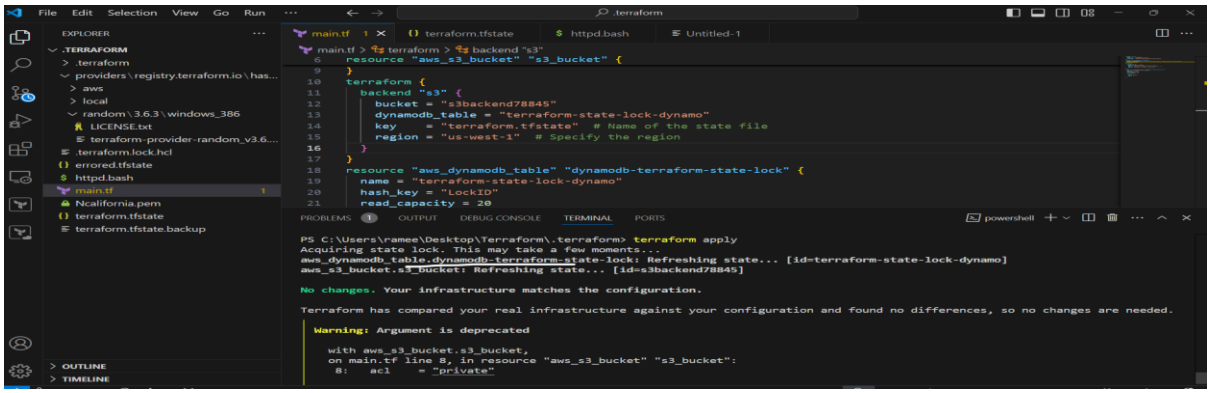
So came s3 backend there I am attached.

Here I am try to initialize it ‘s asking – use terraform init –migrate-state

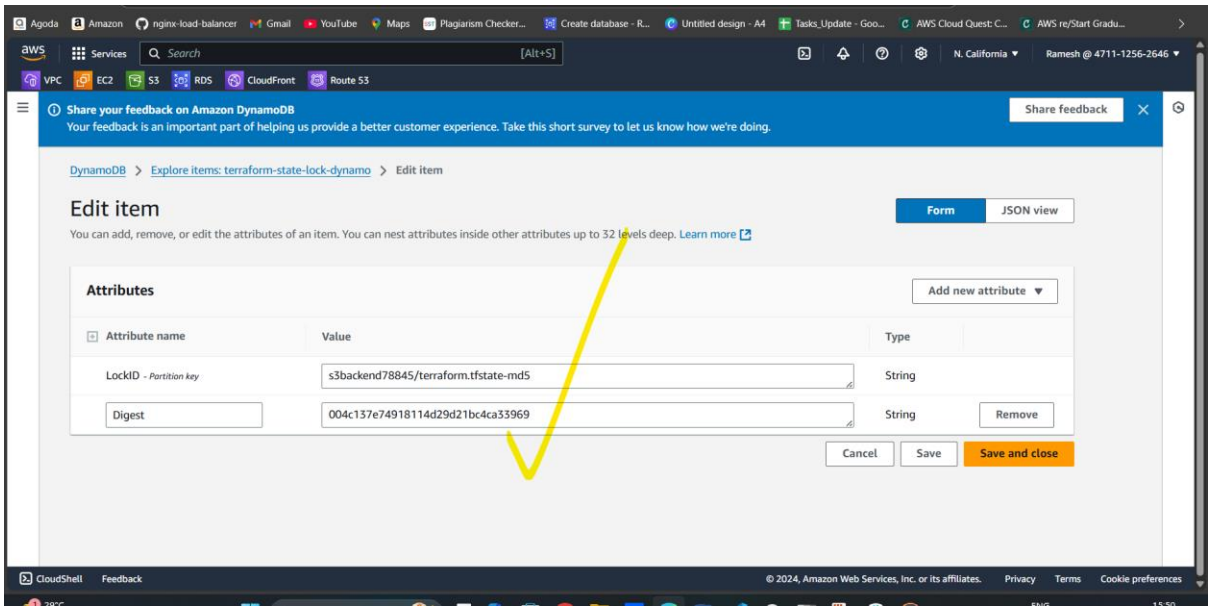




It will success.



Now we want to check it Setup dynamo db locking or not.



Its created.

Terraform provides a range of log levels to help troubleshoot issues, each offering a different depth of information:

1. **INFO** gives basic updates, showing what Terraform is doing, like creating or updating resources.
2. **WARNING** flags potential issues that might not block execution but could lead to unexpected results, helping to catch misconfigurations early.
3. **ERROR** logs critical problems that stop Terraform from proceeding, such as syntax or config errors.
4. **DEBUG** dives deeper, showing detailed internal processing. It’s great for understanding Terraform's internal actions.
5. **TRACE** is the most detailed, logging every action Terraform takes. It’s ideal when I need to trace down to the smallest operation.

To enable any of these, I’d just set the `TF_LOG` environment variable to the required level, like `export TF_LOG=DEBUG`. And if I want to keep the logs, I can set a path with `TF_LOG_PATH` so the logs save to a file for later analysis.