

Terraform

Terraform is an open-source **infrastructure as code (IaC)** tool developed by HashiCorp. It allows you to define and manage your infrastructure resources (like servers, networks, storage) in a human-readable configuration language called HashiCorp Configuration Language (HCL). This code can then be used to provision and manage your infrastructure across different cloud providers, on-premises environments, or even a hybrid combination.

Here are some key benefits of using Terraform:

- **Infrastructure as Code:** Terraform codifies your infrastructure, making it version controlled, reusable, and shareable. This allows for consistent and automated infrastructure deployments, reducing manual errors and improving collaboration within teams.
- **Multi-Cloud Support:** Terraform works with various cloud providers like AWS, Azure, GCP, and many more. You can use the same configuration language to manage your infrastructure across different platforms.
- **Modularity and Reusability:** You can break down your infrastructure into smaller, reusable modules, promoting code organization and simplifying complex deployments.
- **State Management:** Terraform keeps track of the infrastructure resources it manages in a state file. This allows for idempotent operations, meaning you can re-run your Terraform configuration without creating duplicate resources.

Here's a typical workflow using Terraform:

1. **Define Infrastructure:** Write configuration files in HCL to define the desired state of your infrastructure, specifying resources like virtual machines, storage buckets, and networking components.
2. **Plan and Apply:** Run `terraform plan` to preview the changes Terraform will make to your infrastructure based on your configuration. Once satisfied, run `terraform apply` to execute the planned changes.
3. **State Management:** Terraform stores the current state of your infrastructure in a state file. This ensures Terraform knows what resources exist and avoids creating duplicates.

Here are some common use cases for Terraform:

- **Provisioning Cloud Infrastructure:** Automatically create and configure cloud resources like servers, storage, and networking in different cloud platforms.
- **Managing on-premises infrastructure:** Automate the configuration and management of on-premises servers, network devices, and other infrastructure components.
- **Multi-Cloud Deployments:** Manage infrastructure across multiple cloud providers using consistent configuration files, simplifying hybrid deployments.
- **Disaster Recovery:** Use Terraform to define and deploy backup infrastructure configurations for disaster recovery scenarios.

Overall, Terraform is a powerful tool that simplifies infrastructure management and provisioning across various environments. It promotes infrastructure as code principles, enabling automation, consistency, and collaboration in building and managing your infrastructure.

Advantages of Terraform:

- **Infrastructure as Code (IaC):** Terraform allows you to define your infrastructure in code, making it version controlled, reusable, and shareable. This leads to several benefits:
 - **Consistency:** Ensures consistent infrastructure deployments across environments, reducing manual errors and promoting repeatability.
 - **Collaboration:** Enables teams to easily collaborate on infrastructure changes by sharing and reviewing the code.
 - **Version Control:** Allows tracking changes to your infrastructure over time and reverting to previous configurations if needed.
- **Multi-Cloud Support:** Terraform works with a wide range of cloud providers (AWS, Azure, GCP, etc.) and on-premise environments. You can use the same configuration language to manage infrastructure across different platforms, simplifying multi-cloud deployments.
- **Modularity and Reusability:** Terraform configurations can be broken down into smaller, reusable modules. This promotes code organization and simplifies complex infrastructure deployments. These modules can be used across different projects, saving time and effort.
- **State Management:** Terraform keeps track of the infrastructure resources it manages in a state file. This allows for idempotent operations, meaning you can re-run your Terraform configuration without creating duplicate resources. It also provides a single source of truth for your infrastructure configuration.
- **Automation:** Terraform enables infrastructure provisioning and management to be automated, reducing manual work and improving efficiency.

Disadvantages of Terraform:

- **Learning Curve:** Learning Terraform requires understanding HCL syntax and infrastructure concepts. This can have a learning curve, especially for beginners.
- **Security Considerations:** Terraform configurations manage sensitive infrastructure resources. Improper access control or mistakes in configuration can lead to security vulnerabilities. It's crucial to implement proper security practices when using Terraform.
- **Debugging:** Debugging errors in Terraform configurations can be challenging, especially for complex setups. Understanding the error messages and troubleshooting can require expertise.
- **Vendor Lock-In:** While Terraform supports multiple providers, some advanced features or functionalities might be specific to a particular provider. Switching providers might require significant configuration changes.

- **State Management Complexity:** Terraform state files are essential but can become complex for large or distributed deployments. Managing and keeping them consistent across environments requires extra care.

Overall, Terraform is a powerful tool for infrastructure management, offering significant advantages for automation, consistency, and collaboration. However, it's essential to be aware of the learning curve, security considerations, and potential state management complexities when adopting Terraform.

Variables in terraform

In Terraform, variables serve as placeholders for values that can be input into your configuration at runtime. They allow you to parameterize your infrastructure code, making it more flexible and reusable across different environments. Variables can represent various data types, and Terraform supports several types of variables. Here's an explanation of variables and their types in Terraform:

Variables:

Declaration:

Variables are declared using the variable block within a Terraform configuration file.

Syntax:

```
variable "variable_name" {  
  type    = <data_type>  
  default = <default_value>  
}
```

Usage:

Variables can be used throughout your Terraform configuration by referencing them using the `${var.variable_name}` syntax.

Override:

You can override variable values during Terraform operations using the `-var` flag or by specifying variable values in a `.tfvars` file.

Scope:

Variables have scope within the Terraform configuration where they are declared. They can be scoped globally or within specific modules.

Types of Variables in Terraform:

String Variables:

Represent textual data such as names, identifiers, or labels.

```
variable "region" {  
  type    = string  
  default = "us-west-2"  
}
```

Number Variables:

Represent numeric values, including integers and floating-point numbers.

```
variable "instance_count" {  
  type    = number  
  default = 3  
}
```

List Variables:

Store an ordered collection of values of the same type.

```
variable "availability_zones" {  
  type    = list(string)  
  default = ["us-west-1a", "us-west-1b", "us-west-1c"]  
}
```

Map Variables:

Store a collection of key-value pairs, where keys and values can be of different types.

```
variable "tags" {  
  type    = map(string)  
  default = {  
    Name          = "Example"  
    Environment    = "Production"  
  }  
}
```

Boolean Variables:

Represent true or false values.

```
variable "enable_monitoring" {  
  type    = bool  
  default = true  
}
```

Object Variables:

Allow you to define a complex data structure with multiple attributes.

```
variable "subnet" {  
  type = object({  
    cidr_block      = string  
    availability_zone = string  
    tags            = map(string)  
  })  
  default = {  
    cidr_block      = "10.0.1.0/24"  
    availability_zone = "us-west-1a"  
    tags = {  
      Name = "Subnet-1"  
    }  
  }  
}
```

Understanding and effectively using variables in Terraform is essential for creating dynamic and reusable infrastructure as code. They enable you to customize your deployments while maintaining consistency and manageability.

Terraform commands:

Initialization

terraform init: Initializes a Terraform configuration within a directory. This command is typically the first step in a Terraform workflow. It downloads the necessary provider plugins and initializes the backend, setting up the directory for Terraform to manage infrastructure.

Planning and Applying

terraform plan: Creates an execution plan based on the Terraform configuration files in the directory. This plan outlines what actions Terraform will take to achieve the desired state specified in the configuration. It does not make any changes to the infrastructure but provides insights into what will happen when terraform apply is executed.

terraform apply: Applies the changes required to reach the desired state of the configuration. This command executes the plan generated by terraform plan, making the necessary changes to infrastructure resources to match the configuration.

Destroying

terraform destroy: Destroys the infrastructure managed by Terraform. This command removes all resources defined in the Terraform configuration, effectively tearing down the infrastructure provisioned by Terraform.

State Management

terraform state: Manages the Terraform state file. This command has subcommands like list, mv, pull, push, rm, and show to inspect and manipulate the state of resources.

terraform refresh: Updates the state file with the latest information from the real infrastructure. This ensures that Terraform has an accurate view of the current state of resources.

Formatting and Validation

terraform fmt: Formats the Terraform configuration files according to the standard style. This command helps maintain consistency in code formatting across multiple contributors.

terraform validate: Validates the Terraform configuration files, checking for syntax errors and other issues. It ensures that the configuration is properly written and adheres to Terraform's syntax and structure.

Workspaces

terraform workspace: Manages workspaces, which are isolated environments for different instances of a configuration. This command allows you to create, select, list, delete, and show workspaces.

Modules

terraform get: Downloads and installs modules specified in the configuration. Modules are reusable components that encapsulate Terraform configurations.

terraform module: Manages modules in the configuration. This command allows you to list, show, and create modules.

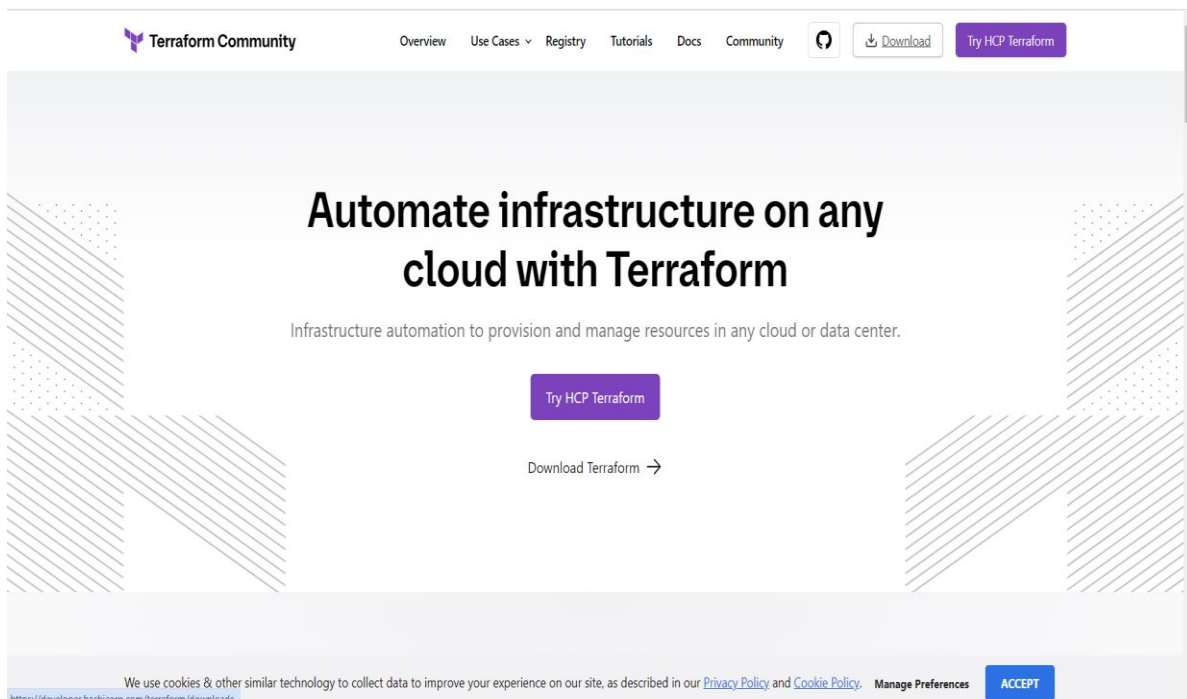
Providers

terraform providers: Lists the providers used in the configuration along with their versions. It provides information about the providers required to manage infrastructure resources.

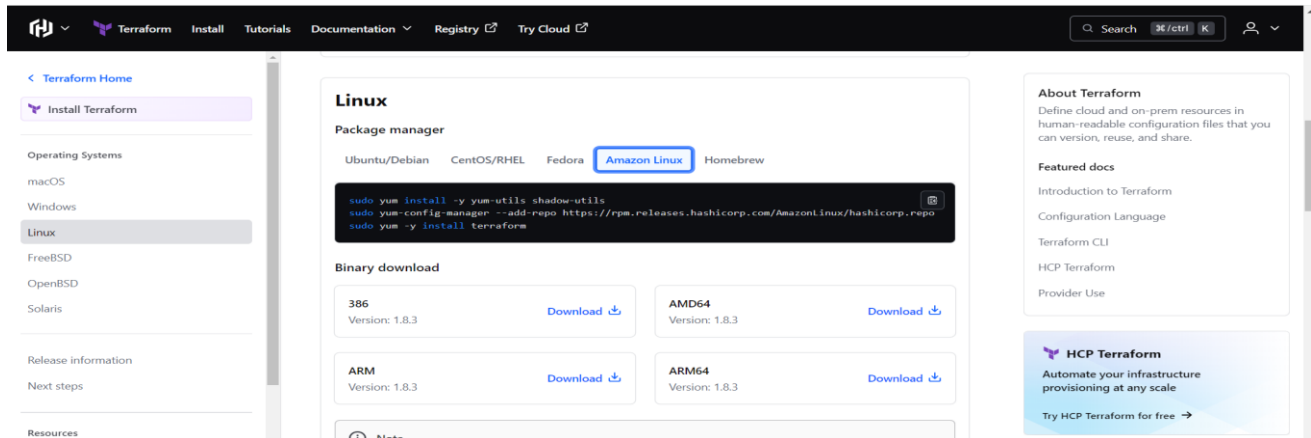
These key commands form the core of the Terraform workflow, enabling users to initialize, plan, apply, and manage infrastructure as code. They provide the necessary tools for provisioning, updating, and destroying infrastructure resources in a controlled and repeatable manner.

TERRAFORM INSTALLATION

- Login to aws account and connect to your ec2 instance
- Come to root user
- Sudo -i
- Open chrome and type terraform download or click on this link <https://www.terraform.io/>
- Click on download button at the top right corner



- Navigate to the “amazon linux” tab in the linux module
 - `sudo yum install -y yum-utils shadow-utils`
 - `sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo`
 - `sudo yum -y install terraform`



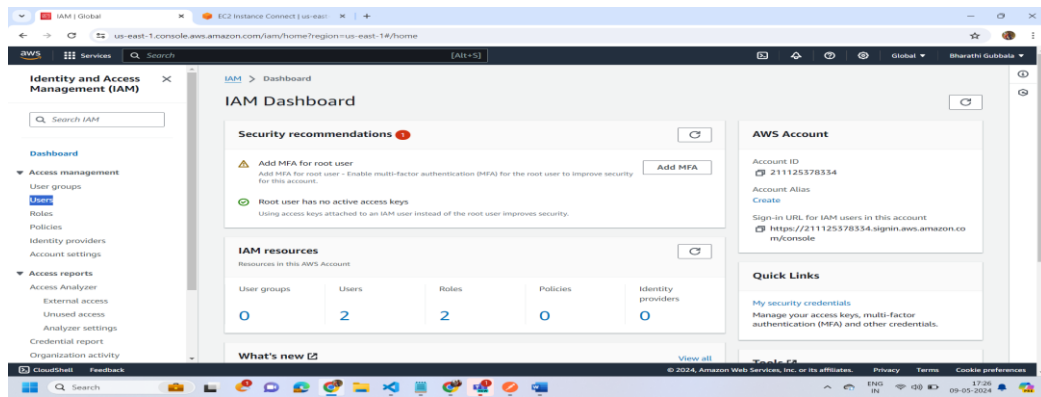
- Execute the commands one by one in the ec2 terminal

```
[root@ip-172-31-27-154 ~]# yum install -y yum-utils shadow-utils
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
amzn2-core
Package yum-utils-1.1.31-46.amzn2.0.1.noarch already installed and latest version
Package 2:shadow-utils-4.1.5.1-24.amzn2.0.3.x86_64 already installed and latest version
Nothing to do
[root@ip-172-31-27-154 ~]# yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
adding repo from: https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
grabbing file https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo to /etc/yum.repos.d/hashicorp.repo
repo saved to /etc/yum.repos.d/hashicorp.repo
[root@ip-172-31-27-154 ~]# yum -y install terraform
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
hashicorp
hashicorp/x86_64/primary
hashicorp
Resolving Dependencies
--> Running transaction check
--> Package terraform.x86_64 0:1.8.3-1 will be installed
--> Processing Dependency: git for package: terraform-1.8.3-1.x86_64
--> Running transaction check
--> Package git.x86_64 0:2.40.1-1.amzn2.0.2 will be installed
--> Processing Dependency: git-core = 2.40.1-1.amzn2.0.2 for package: git-2.40.1-1.amzn2.0.2.x86_64
--> Processing Dependency: git-core-doc = 2.40.1-1.amzn2.0.2 for package: git-2.40.1-1.amzn2.0.2.x86_64
--> Processing Dependency: perl-Git = 2.40.1-1.amzn2.0.2 for package: git-2.40.1-1.amzn2.0.2.x86_64
```

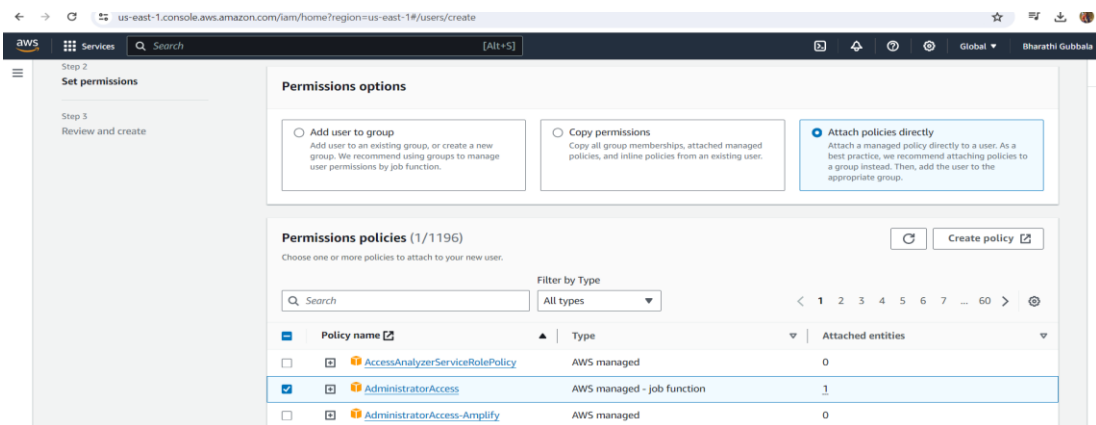
- Terraform installed successfully, if you want to check the terraform installed or not use below command “terraform –version”

Creating Instance Using Terraform

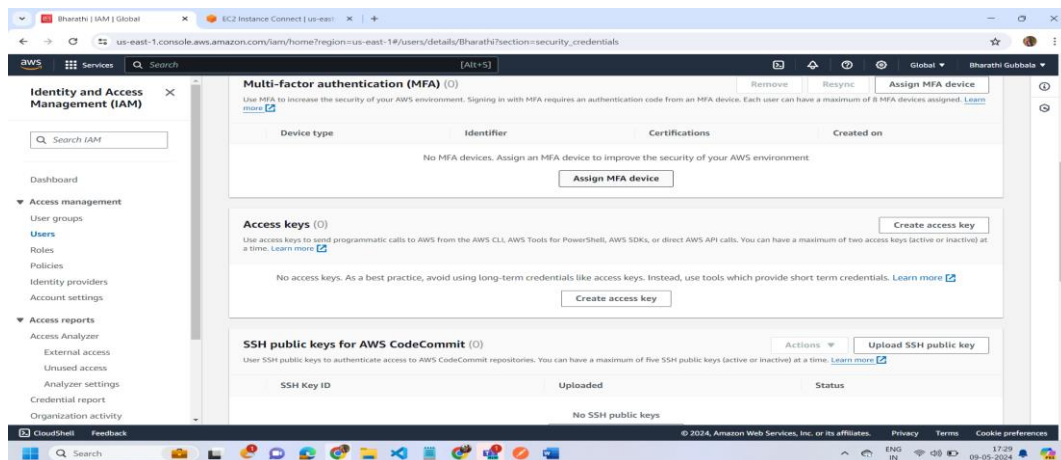
- Log in to your AWS Management Console.
- Navigate to the EC2 service by clicking on "Services" in the top-left corner, then selecting "EC2" under the "Compute" section.
- In the EC2 Management Console, you can search for "IAM" in the search bar located at the top of the page.
- Once IAM appears in the search results, click on it to open the IAM console.
- In the left sidebar, click on "Users" to manage IAM users.



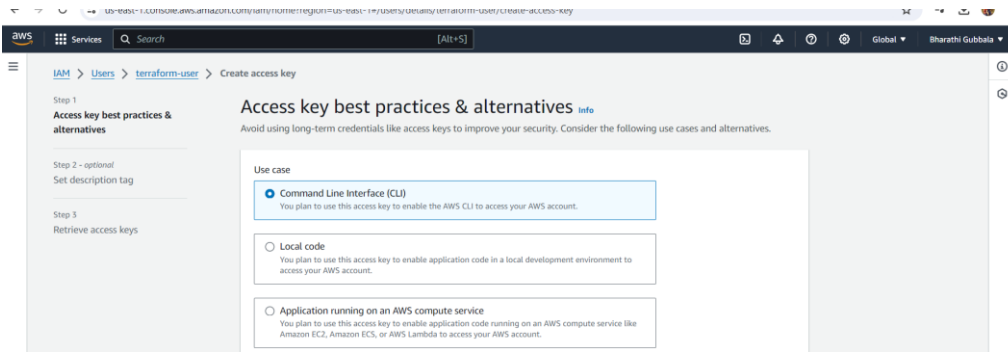
- Click on the "Add user" button.
- Provide a username for the user and click next
- Select the "Attach policies directly" option.
- Search for and select the "AdministratorAccess" policy .



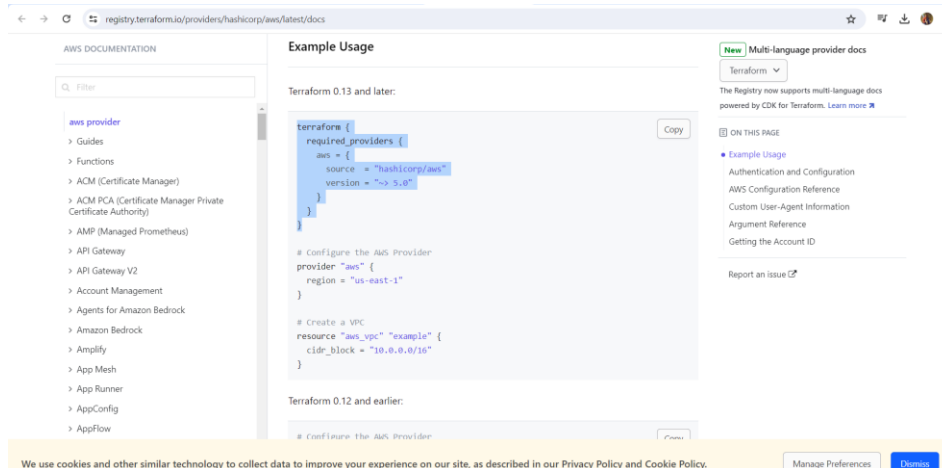
- Click on the Next button.
- Click on the "Create user" button to create the user.
- After creating the user, click on the "Security credentials" tab for the newly created user.
- Under the "Access keys" section, click on the "Create access key" button.



- Select "Command Line Interface " radio button



- and click on the checkbox then click next button give description then click on create a access key button it will create access key and secret access key
- Make sure to copy and securely store the access key ID and secret access key provided.
- Open new tab in chrome and goto official documentation of terraform providers or click on this link <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>



- Copy the code as shown above
- Goto EC2 instance using terminal.
- Switch to the root user with `sudo -i`.
- Create a folder named "terraform" (name is optional) using `mkdir terraform`.
- Create a file named main.tf (name is optional) in the terraform directory.
- Open the main.tf file and add your Terraform configuration code. Make sure to include the correct AMI ID for your EC2 instance.
- Paste the code here and give aws configuration also

```
aws
Services Search [Alt+S] N. Virginia

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = ">= 5.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  access_key = "AKIA7CKANTUPIX604NCA"
  secret_key = "oo+WRvDJWrP9jGcQMpegkt0JUyCcvsRj3kp3Yonu"
}
```

- To launch ec2 instance we have to give resource of aws instance or click on this link <https://registry.terraform.io/providers/hashicorp/aws/2.36.0/docs/resources/instance>
- Copy the code and paste it in the terminal

The screenshot shows the Terraform Registry page for the `aws_instance` resource. The page is divided into three main sections: a left sidebar with navigation links, a central content area showing the resource definition, and a right sidebar with additional links.

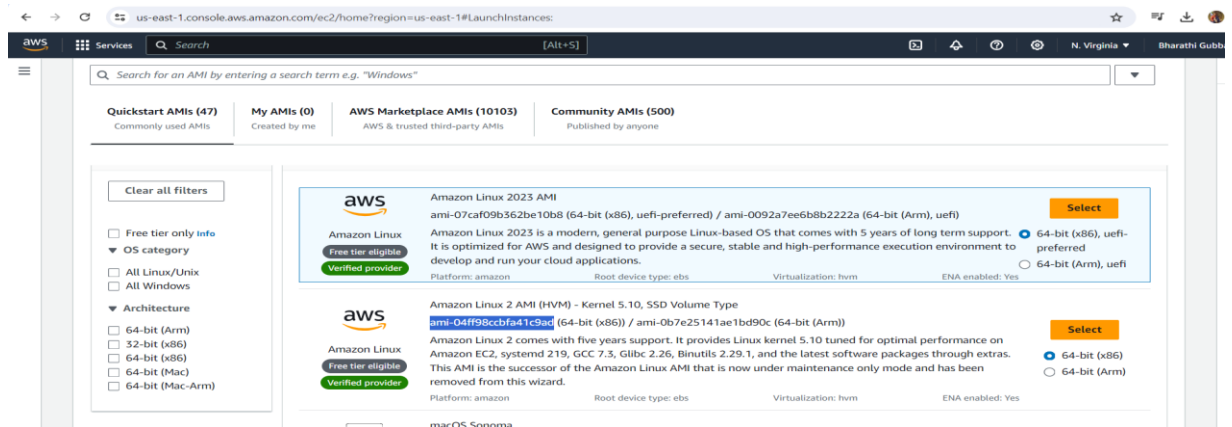
Left Sidebar: Contains a search bar and a list of navigation links including "aws provider", "Guides", "Data Sources", "ACM", "ACM PCA", "API Gateway", "AppMesh", "AppSync", "Application Autoscaling", "Athena", "Autoscaling", "Backup", "Batch", "Budgets", "Cloud9", and "CloudFormation".

Central Content Area: Displays the Terraform configuration for the `aws_instance` resource. The configuration includes a `name` argument, a `values` list, a `filter` block, and an `owners` list. The `aws_instance` resource is highlighted with a blue box, showing its arguments: `ami`, `instance_type`, and `tags`.

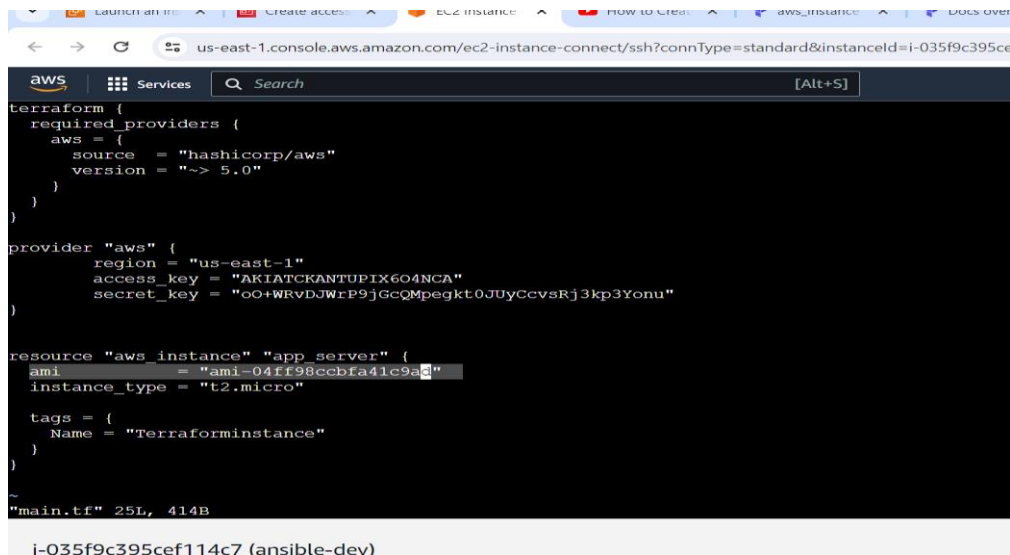
Right Sidebar: Contains a section titled "ON THIS PAGE" with links to "Example Usage", "Argument Reference", "Attributes Reference", and "Import". There is also a "Report an issue" link.

Argument Reference: A section titled "Argument Reference" with the text "The following arguments are supported:".

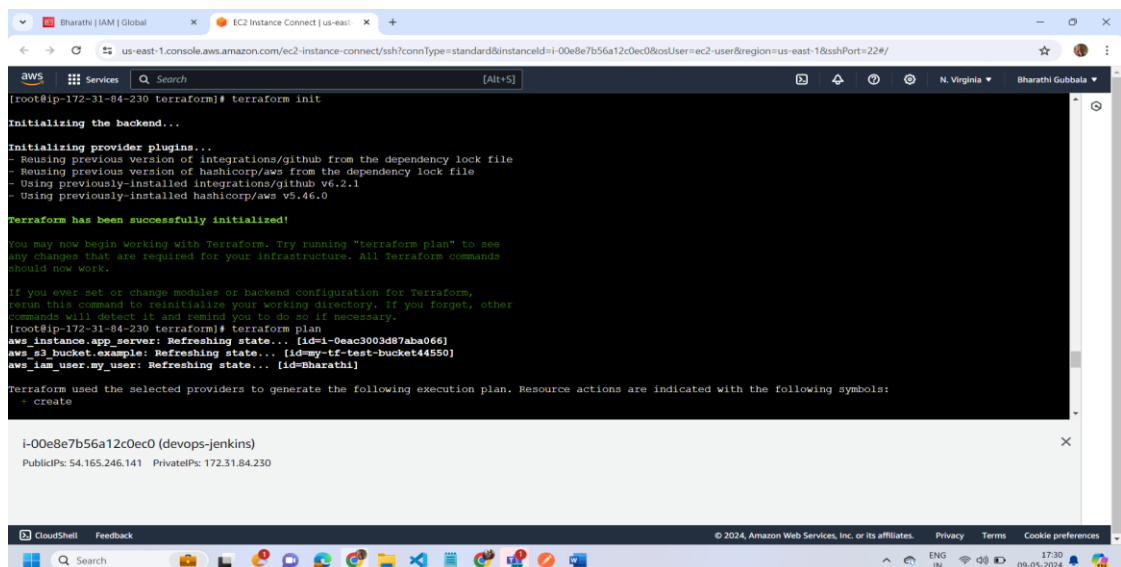
- Now we have to change the ami so goto the instances page click on launch instance
- Then in the application and os images module select the quick start tab, here we will see browse more ami's click on it, select second amazon linux option copy the code given below



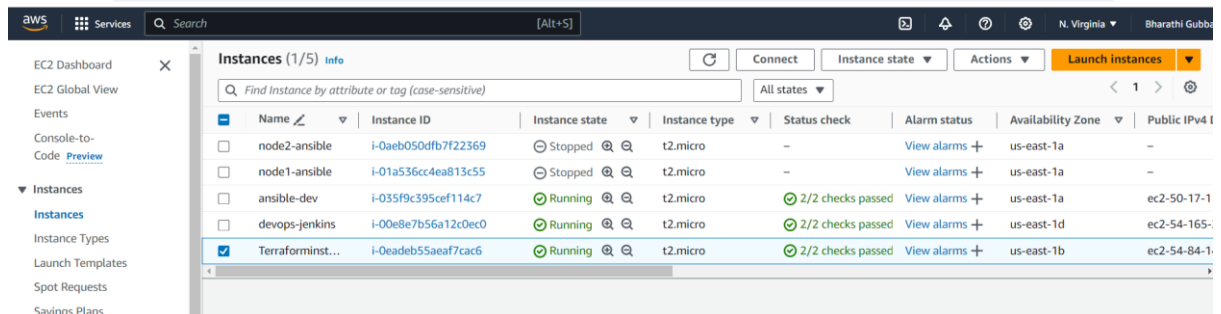
- Paste the code in “ami” and change the name (in tags)(optional)
- Provide the instance type too and save it



- Navigate to the terraform directory.
- Run “terraform init” to initialize the directory.



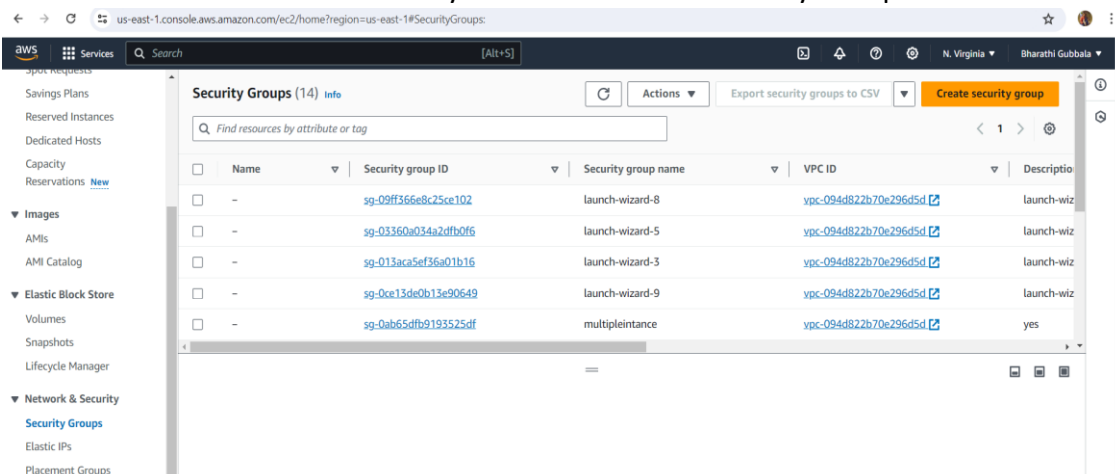
- Run “terraform plan” to see the execution plan.
- If the plan looks good, run “terraform apply” to apply the configuration and create the resources.



- To destroy an instance created using Terraform, you can use the “terraform destroy” command.

Creating multiple instances using variable, security groups and key pair

- Log in to the AWS Management Console.
- Navigate to the EC2 service and launch a new instance.
- If Terraform is not installed, download and install it on your machine.
- Create a new file with a .tf extension, for example, multiinstance.tf.
- Once the file is created, define variables for the instance names and security group IDs.
- To generate the security group ID, follow these steps:
- Navigate to the EC2 instance.
- Go to "Network & Security" and click on the "Security Groups" tab.



- Click on "Create Security Group", provide the necessary details (name, description), and click "Create Security Group".

Create security group [Info](#)

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a new security group, complete the fields below.

Basic details

Security group name [Info](#)

Name cannot be edited after creation.

Description [Info](#)

VPC [Info](#)

Inbound rules [Info](#)

This security group has no inbound rules.

- Note the security group ID generated for future use in the Terraform code.

```
variable "security_group_ids" {
  type    = list(string)
  default = ["sg-0ab65dfb9193525df"]
}
```

- Define a list variable to assign names to the instances.

```
variable "instance_name" {
  type    = list(string)
  default = ["instance1", "instance2", "instance3"]
}
```

- To include a key pair in the instances:
- Navigate to the "Key Pairs" tab in "Network & Security" within the EC2 instance.
- Click on "Create Key Pair", provide a name and select the key pair type, then click "Create Key Pair".

Key pairs (9) [Info](#)

<input type="checkbox"/>	Name	Type	Created	Fingerprint	ID
<input type="checkbox"/>	node2Pair	rsa	2024/05/01 13:36 GMT+5:30	37:3d:53:d5:43:11:35:48:4f:56:e6:3...	key-092b5f4085302880e
<input type="checkbox"/>	node2	rsa	2024/05/01 18:23 GMT+5:30	6d:f3:0d:9e:18:05:eb:01:b4:a0:bc:6...	key-0a54926b239c6df1d
<input type="checkbox"/>	ansidev	rsa	2024/05/02 11:39 GMT+5:30	84:c4:9f:be:52:d4:71:db:d0:a2:80:9...	key-01e14693f4b7c317c
<input type="checkbox"/>	keypairforinstances	rsa	2024/05/17 14:56 GMT+5:30	58:7a:c0:e4:e6:3a:cf:e6:8b:13:a3:81...	key-01cef48c6b5fb5189
<input type="checkbox"/>	node-1	rsa	2024/05/01 18:17 GMT+5:30	32:82:64:71:4e:6b:2f:2e:81:64:e0:7...	key-0500be7c8d4d6222a
<input type="checkbox"/>	nodePair	rsa	2024/05/01 13:29 GMT+5:30	97:ee:25:51:7f:76:f0:96:ce:d4:f3:17...	key-0f4b2839bfbeffa9b
<input type="checkbox"/>	node1	rsa	2024/05/01 18:15 GMT+5:30	df:58:bc:7c:93:1d:00:a5:46:17:40:5...	key-05d4305164e3b08e9
<input type="checkbox"/>	dev	rsa	2024/05/01 18:25 GMT+5:30	14:2db3:bd:0bc1:e9:9d:1f:15:73:1...	key-09b04b41173a93c89
<input type="checkbox"/>	devansiblekey	rsa	2024/05/01 18:14 GMT+5:30	fe:26:0b:3a:f6:8e:8c:53:9b:f4:e5:ef...	key-0e7cd68ed72deddb3

- Copy the name of the key pair and paste it into the Terraform code.

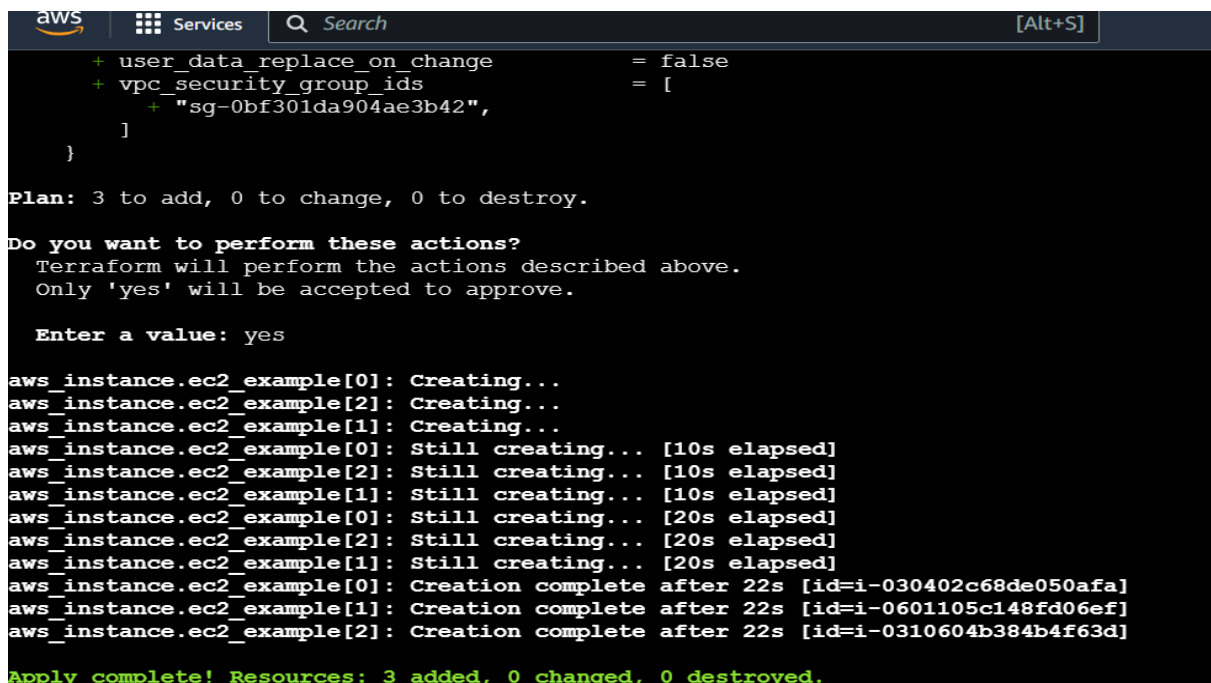
```
variable "instance_name" {
  type    = list(string)
  default = ["instance1", "instance2", "instance3"]
}

variable "security_group_ids" {
  type    = list(string)
  default = ["sg-0ab65dfb9193525df"]
}

resource "aws_instance" "ec2_example" {
  ami                = "ami-04ff98ccbf41c9ad"
  key_name           = "keypairforinstances"
  instance_type      = "t2.micro"
  count              = 3
  vpc_security_group_ids = var.security_group_ids

  tags = {
    Name = var.instance_name[count.index]
  }
}
```

- Save the file.
- Run the command “terraform init” to initialize Terraform.
- Run the command terraform plan to review the execution plan.
- Run the command “terraform apply” to apply the changes and create the instances.



The screenshot shows a terminal window with the AWS CLI interface. At the top, there's a header bar with the AWS logo, 'Services', a search bar, and a keyboard shortcut '[Alt+S]'. Below this, the Terraform plan output is displayed, showing changes to 'user_data_replace_on_change' and 'vpc_security_group_ids'. The plan indicates 3 instances to be added. A prompt asks for confirmation to perform the actions, which is answered 'yes'. The subsequent output shows the creation of three EC2 instances, with status updates every 10-20 seconds, and finally, the successful completion of the apply process, adding 3 resources.

```
aws
Services Search [Alt+S]

+ user_data_replace_on_change = false
+ vpc_security_group_ids      = [
  + "sg-0bf301da904ae3b42",
]

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_instance.ec2_example[0]: Creating...
aws_instance.ec2_example[2]: Creating...
aws_instance.ec2_example[1]: Creating...
aws_instance.ec2_example[0]: Still creating... [10s elapsed]
aws_instance.ec2_example[2]: Still creating... [10s elapsed]
aws_instance.ec2_example[1]: Still creating... [10s elapsed]
aws_instance.ec2_example[0]: Still creating... [20s elapsed]
aws_instance.ec2_example[2]: Still creating... [20s elapsed]
aws_instance.ec2_example[1]: Still creating... [20s elapsed]
aws_instance.ec2_example[0]: Creation complete after 22s [id=i-030402c68de050afa]
aws_instance.ec2_example[1]: Creation complete after 22s [id=i-0601105c148fd06ef]
aws_instance.ec2_example[2]: Creation complete after 22s [id=i-0310604b384b4f63d]

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

- If you want to delete the instances then run the command “terraform destroy”


```

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

aws_instance.ec2_example[2]: Destroying... [id=i-0310604b384b4f63d]
aws_instance.ec2_example[1]: Destroying... [id=i-0601105c148fd06ef]
aws_s3_bucket.example: Destroying... [id=my-tf-test-bucket34567]
aws_instance.ec2_example[0]: Destroying... [id=i-030402c68de050afa]
aws_s3_bucket.example: Destruction complete after 0s
aws_instance.ec2_example[2]: Still destroying... [id=i-0310604b384b4f63d, 10s elapsed]
aws_instance.ec2_example[1]: Still destroying... [id=i-0601105c148fd06ef, 10s elapsed]
aws_instance.ec2_example[0]: Still destroying... [id=i-030402c68de050afa, 10s elapsed]
aws_instance.ec2_example[1]: Still destroying... [id=i-0601105c148fd06ef, 20s elapsed]
aws_instance.ec2_example[2]: Still destroying... [id=i-0310604b384b4f63d, 20s elapsed]
aws_instance.ec2_example[0]: Still destroying... [id=i-030402c68de050afa, 20s elapsed]
aws_instance.ec2_example[2]: Still destroying... [id=i-0310604b384b4f63d, 30s elapsed]
aws_instance.ec2_example[1]: Still destroying... [id=i-0601105c148fd06ef, 30s elapsed]
aws_instance.ec2_example[0]: Still destroying... [id=i-030402c68de050afa, 30s elapsed]
aws_instance.ec2_example[1]: Still destroying... [id=i-0601105c148fd06ef, 40s elapsed]
aws_instance.ec2_example[0]: Still destroying... [id=i-030402c68de050afa, 40s elapsed]
aws_instance.ec2_example[2]: Still destroying... [id=i-0310604b384b4f63d, 40s elapsed]
aws_instance.ec2_example[2]: Destruction complete after 40s
aws_instance.ec2_example[0]: Destruction complete after 40s
aws_instance.ec2_example[1]: Destruction complete after 40s

Destroy complete! Resources: 4 destroyed.

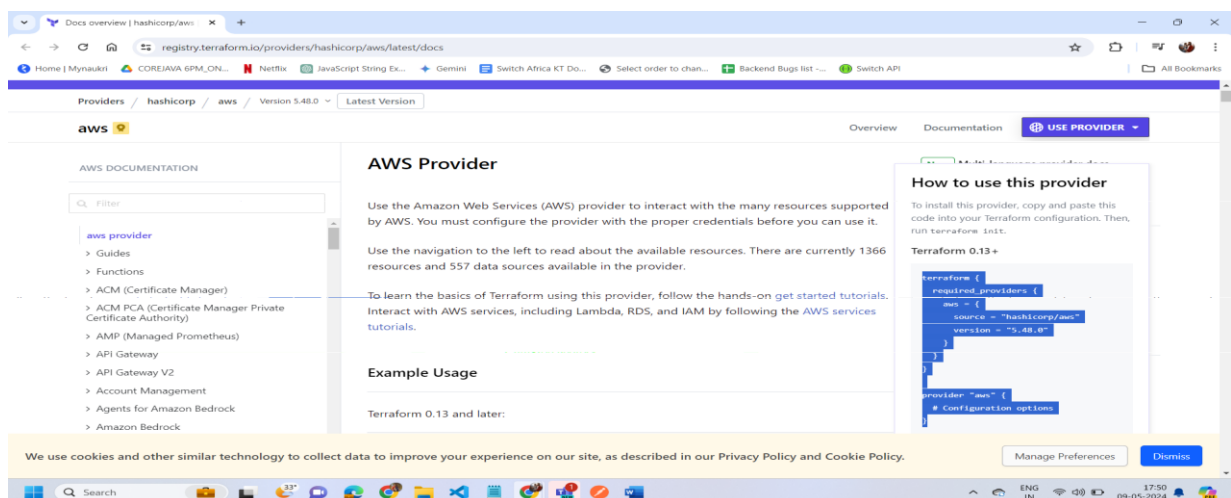
```

The terraform destroy command is used to destroy all the resources that were created by Terraform based on the configuration files.

Note: Providers are required for every task that uses a region, access key, and secret key. If you omit them, you will encounter an error during execution.

Creating A S3 Bucket Using Terraform

- Goto the terraform website using <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>
- Click on “use provider” and copy the configuration



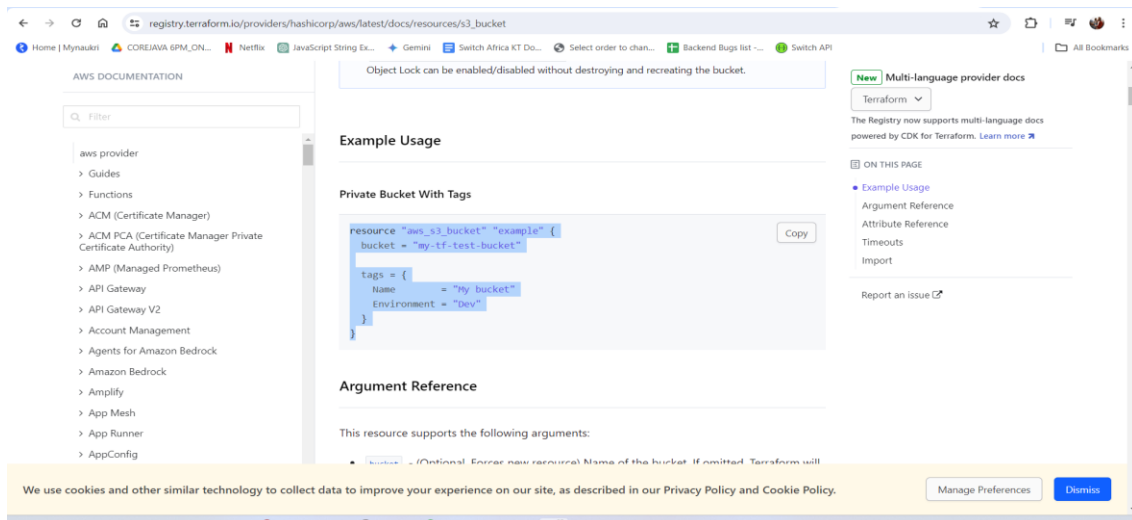
- Goto ec2 instance
- Sudo -i
- Goto terraform folder (cd terraform)
- Create a file with name provider.tf (if "aws" providers are already given no need to provide gain)
- Paste the code here and save it and also mention the region with access key ,secret key

Note:(If you have not created an access key and secret key yet, refer to the section on creating an EC2 instance for instructions on how to do so)

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.49.0"
    }
  }
}

provider "aws" {
  access_key = "AKIATCKANTUPBCU3NLV4"
  secret_key = "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"
  region = "us-east-1"
}
```

- Create a one more file for the s3bucket (name of the file is optional) s3bucket.tf
- Creating an s3bucket we need to provide the resource so goto the official documentation of terraform for the s3bucket or click on this link https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket
- Copy the code as shown below to configure the s3bucket and paste it in the file s3bucket.tf

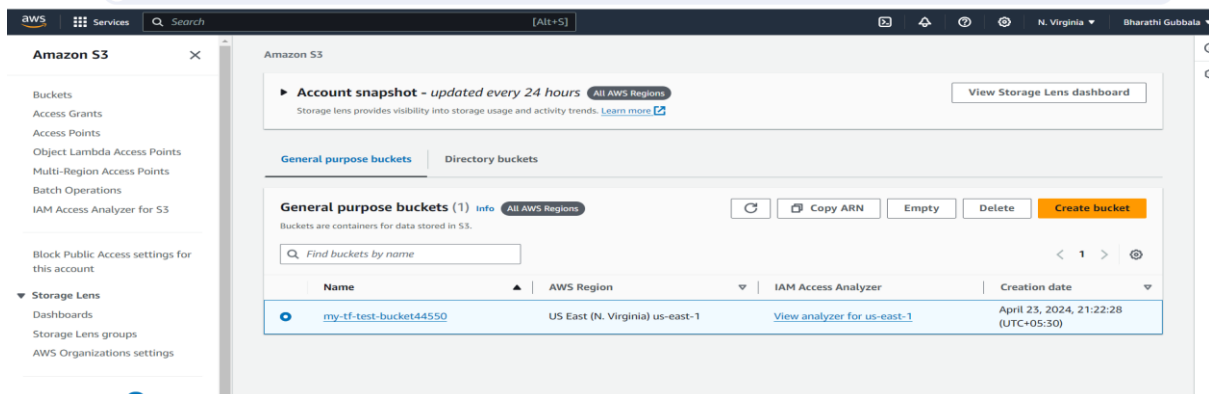


- Make sure that the bucket name should be unique example (bucket = "my-tf-test-bucket44550")

```
resource "aws_s3_bucket" "examplebucket" {
  bucket = "my-tf-test-bucket33333"

  tags = {
    Name       = "My bucket"
    Environment = "Dev"
  }
}
```

- Run command "terraform init"
- Run command "terraform plan"
- Run command "terraform apply"
- The s3 bucket will be created successfully
- To see s3 bucket created or not , type s3 in the search bar click on the s3 option it will redirect to the S3 bucket



Creating Multiple S3 Buckets Using Terraform

- Log in to the AWS Management Console.
- Navigate to the EC2 service and launch a new instance.
- If Terraform is not installed, download and install it on your machine.
- Create a new file with a .tf extension, for example, multibucket.tf
- Declares a variable
- Provides a description for the variable.
- Specifies that the variable is a list of strings.
- Sets the default value of the variable to a list containing three S3 bucket names.

```
variable "s3_bucket_names" {  
  description = "Creating Buckets as per requirement"  
  type        = list(string)  
  default     = ["terraform-s3bucket01one", "terraform-s3bucket02two", "terraform-s3bucket03three"]  
}
```

- Defines an AWS S3 bucket resource
- the count meta-argument to create multiple instances of the resource based on the length of the list variable.
- Set the bucket attribute of each S3 bucket resource to the value of the corresponding element in the list. The count.index is used to dynamically select each bucket name based on the current iteration index.

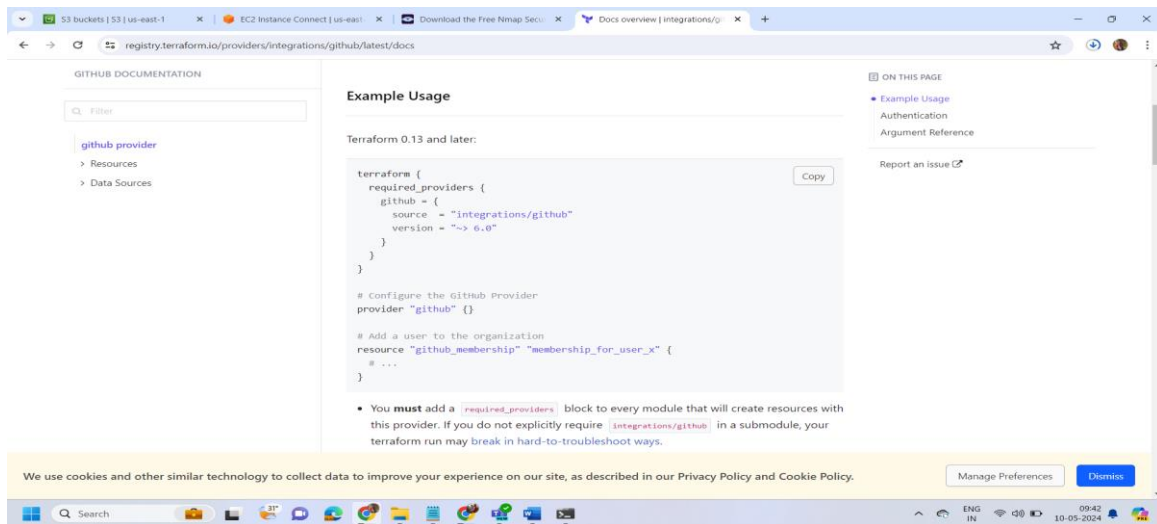
```
variable "s3_bucket_names" {  
  description = "Creating Buckets as per requirement"  
  type        = list(string)  
  default     = ["terraform-s3bucket01one", "terraform-s3bucket02two", "terraform-s3bucket03three"]  
}  
  
resource "aws_s3_bucket" "bucketexample" {  
  count = length(var.s3_bucket_names)  
  bucket = var.s3_bucket_names[count.index]  
}
```

- Save the file
- Run command "terraform init"
- Run command "terraform plan"
- Run command "terraform apply"
- The s3 bucket will be created successfully
- To see s3 bucket created or not , type s3 in the search bar click on the s3 option it will redirect to the s3 bucket

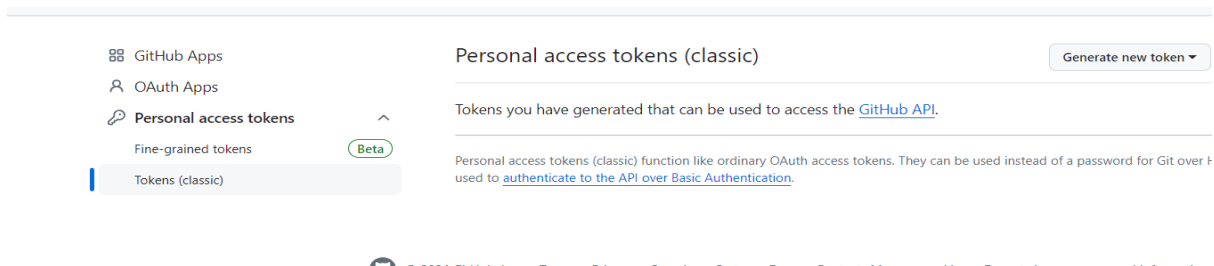
Creating A Git Repo Using Terraform

- Login to aws account and connect to your ec2 instance
- Come to root user
- Sudo -i
- Create a folder using command “mkdir Terraform”
- Create a file at the aws instance in terraform folder “vi github.tf”
- Open a new tab in chrome type “terraform git hub provider” or click on this link for terraform github providers

“<https://registry.terraform.io/providers/integrations/github/latest/docs>”



- Copy the code shown above paste it in github.tf file
- In github provider we have to give the token so login to github account
- Goto settings then select developer settings
- Here you are able to see two options shown below click on the personal access token then select token(classic)

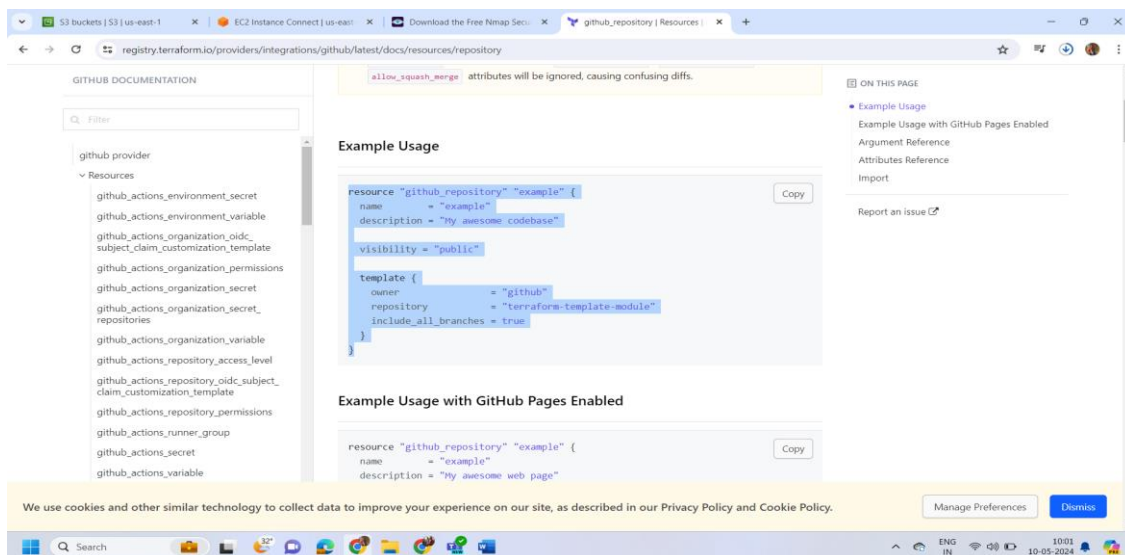


- Click on generate new token give necessary permissions then click on “generate token”
- Token will be created then copy that token paste it in the github.tf file as shown below

```
terraform {  
  required_providers {  
    github = {  
      source = "integrations/github"  
      version = "~> 6.0"  
    }  
  }  
}  
  
# Configure the GitHub Provider  
provider "github" {  
  
  token = "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"  
  
}
```

- Goto chrome type terraform hithub repository and click on the link which is related to the terraform or click on this link

<https://registry.terraform.io/providers/integrations/github/latest/docs/resources/repository>



- Copy this code, paste it in github.tf file and modify the code as shown below, save it

```

terraform {
  required_providers {
    github = {
      source = "integrations/github"
      version = "~> 6.0"
    }
  }
}

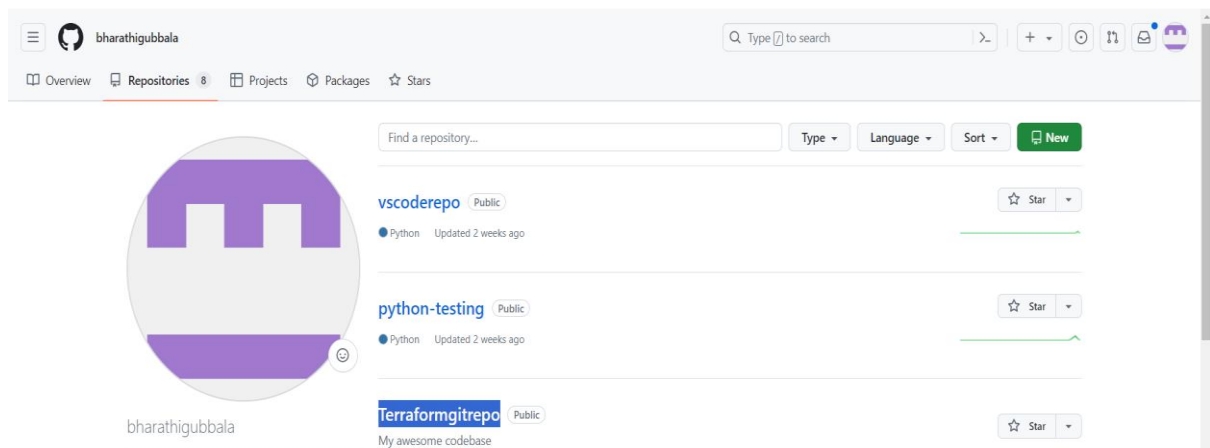
# Configure the GitHub Provider
provider "github" {

  token = "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"
}

resource "github_repository" "Terraformgitrepo" {
  name          = "Terraformgitrepo"
  description   = "My awesome codebase"
  visibility    = "public"
}

```

- Run "terraform init"
- Run "terraform plan"
- Run "terraform apply"
- Goto github repository, the repository will be created in the github



Creating IAM User Using Terraform

- Search IAM user creation using terraform in google or click on this link for terraform documentation to create a user
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/iam_user
- To create an IAM user using Terraform, you can follow these steps:
- Initialize Terraform in your project directory, If you haven't already done so, initialize Terraform by running terraform init in your project directory. This will set up your Terraform environment.
- Write the Terraform configuration file, Create a .tf file (e.g., iam_user.tf) where you'll define the resources needed to create the IAM user. Below is an example configuration to create an IAM user named "example_user":

```
provider "aws" {  
  region = "us-east-1" # Specify your desired AWS region  
}  
  
resource "aws_iam_user" "example_user" {  
  name = "example_user"  
}
```

- provider block specifies the AWS provider and the region where the resources will be created.
- resource block defines an IAM user named "example_user".
- Apply the Terraform configuration: Run terraform apply in your project directory. Terraform will read your configuration, create an execution plan, and prompt you to confirm the plan before applying it. If everything looks good, type "yes" to proceed.
- Verify the IAM user: Once Terraform applies the configuration successfully, you can verify the creation of the IAM user in the AWS Management Console.

Note: Providers are required for every task that uses a region, access key, and secret key. If you omit them, you will encounter an error during execution.

Deploying MySQL Database on AWS RDS using Terraform

- Before using Terraform to provision resources on AWS, you need to set up your AWS credentials. This typically involves configuring your AWS access key ID and secret access key.
- If you haven't already initialized Terraform in your project directory, run the following command:
- `terraform init`
- Create a `.tf` file (e.g., `main.tf`) where you'll define the Terraform configuration for the MySQL RDS instance. Or follow this link to see the documentation https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_instance
- Write all necessary fields that are required to create the database

The screenshot shows the Terraform Registry documentation page for the `aws_db_instance` resource. The page is titled "Example Usage" and "Basic Usage". It contains two code snippets: a basic usage example and a custom Oracle usage example with a replica.

Basic Usage

```
resource "aws_db_instance" "default" {
  allocated_storage = 10
  db_name           = "mydb"
  engine            = "mysql"
  engine_version    = "8.0"
  instance_class    = "db.t3.micro"
  username          = "foo"
  password          = "foobarbaz"
  parameter_group_name = "default.mysql8.0"
  skip_final_snapshot = true
}
```

RDS Custom for Oracle Usage with Replica

```
# Lookup the available instance classes for the custom engine for the region
data "aws_rds_orderable_db_instance" "custom-oracle" {
  engine            = "custom-oracle-ee" # CEV engine to be used
  engine_version    = "19.c.ee.002"     # CEV engine version to be used
  license_model      = "bring-your-own-license"
  storage_type       = "ssd"
}
```

The page also includes a sidebar with navigation links, a search bar, and a "Multi-language provider docs" section.


```

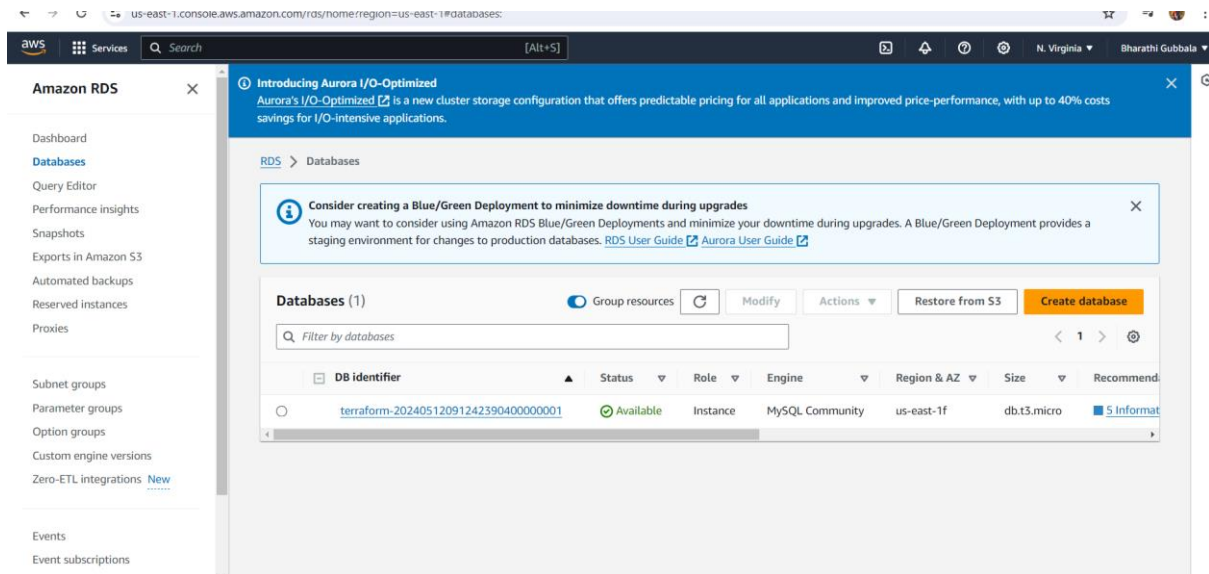
provider "aws" {
  region = "us-east-1" # Specify your desired A
}

resource "aws_db_instance" "default" {
  allocated_storage    = 10
  db_name              = "mydb"
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t3.micro"
  username             = "foo"
  password             = "foobarbaz"
  parameter_group_name = "default.mysql8.0"
  skip_final_snapshot = true

  tags = {
    Name = "MyRDS"
  }
}

```

- Review your Terraform configuration to ensure that all settings, such as the instance class, database name, engine, engine version, storage type, username, and password, are correctly defined and meet your requirements.
- Run `terraform plan` to see the execution plan, which will show you what Terraform will do when you apply the configuration. This step is optional but recommended for reviewing changes before applying them.
- `terraform plan`
- Once you're satisfied with the plan, apply the Terraform configuration to create the MySQL RDS instance:
- `terraform apply`
- Terraform will show you a summary of the changes it will make. Type "yes" to confirm and apply the changes.
- Terraform will provision the MySQL RDS instance on AWS. This process may take a few minutes.
- Once the Terraform apply command completes successfully, verify the deployment by checking the AWS Management Console or using AWS CLI commands to confirm that the MySQL RDS instance has been created with the specified configuration.
- If you want to delete the MySQL RDS instance later, you can run `terraform destroy`:
- `terraform destroy`
- Confirm by typing "yes" to delete the resources provisioned by Terraform.
- That's it! You've now deployed a MySQL database on AWS RDS using Terraform.



Creating VPC Using Terraform

- Before you can interact with AWS using Terraform, you need to set up your AWS credentials. You can do this by creating an IAM user in the AWS Management Console and generating an access key ID and secret access key.
- Create a new directory for your Terraform project and navigate into it.
- Create a new file with a .tf extension (e.g., main.tf) to define your Terraform configuration.
- Open the file .
- Start by defining the AWS provider block in your Terraform configuration. This block specifies the AWS region and your AWS credentials.

```

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.49.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  access_key = "AKIATCKANTUPBCU3NLV4"
  secret_key = "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"
}

```

- Define the VPC using the `aws_vpc` resource type. Specify the CIDR block (IP address range) for the VPC and optionally provide tags for identification.

```
# 1: Create a VPC
resource "aws_vpc" "myvpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "MyVPC"
  }
}
```

- Define a public subnet within the VPC using the `aws_subnet` resource type. Specify the VPC ID where the subnet will be created, the CIDR block for the subnet, and the availability zone.

```
# 2: Create a public subnet
resource "aws_subnet" "PublicSubnet" {
  vpc_id            = aws_vpc.myvpc.id
  availability_zone = "us-east-1a"
  cidr_block        = "10.0.1.0/24"
}
```

- If you want to define the private subnet and internet gateway you can include as shown below, which are optional.

```
# 3: Create a private subnet
resource "aws_subnet" "PrivSubnet" {
  vpc_id      = aws_vpc.myvpc.id
  cidr_block  = "10.0.2.0/24"
}

# 4: Create an internet gateway
resource "aws_internet_gateway" "myIgw" {
  vpc_id = aws_vpc.myvpc.id
}
```

- Define a route table using the `aws_route_table` resource type. Specify the VPC ID and create a route with the destination CIDR block `0.0.0.0/0` pointing to an internet gateway.

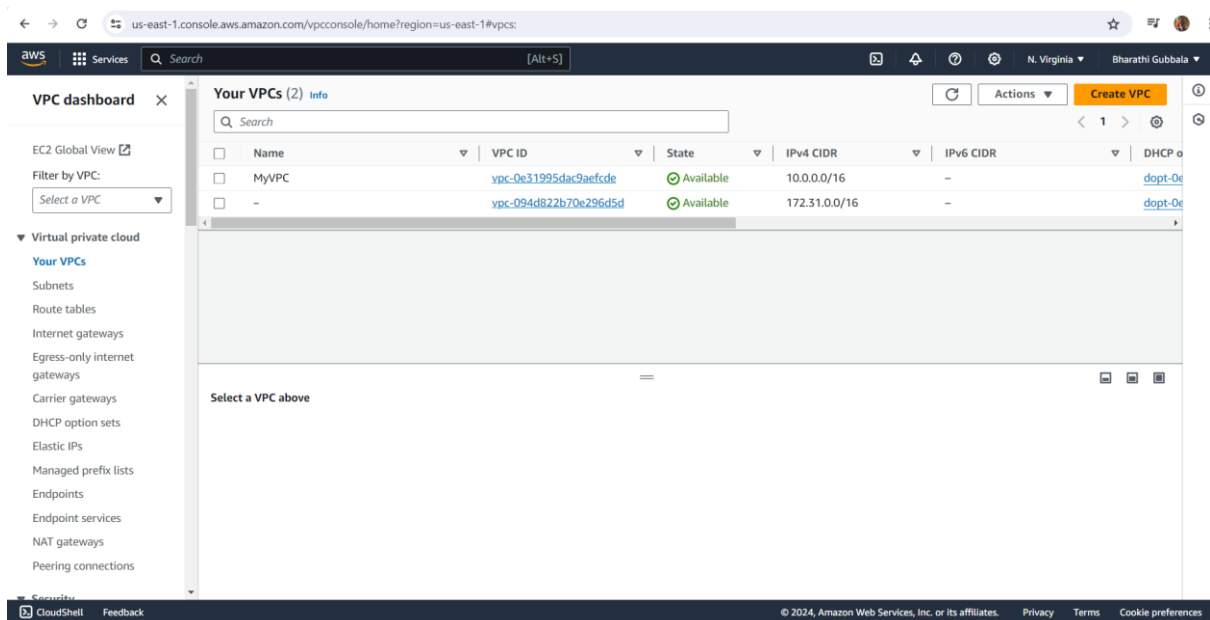
```
# 5: Create a route table for public subnet
resource "aws_route_table" "PublicRT" {
  vpc_id = aws_vpc.myvpc.id

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

- Associate the route table with the public subnet using the `aws_route_table_association` resource type. Specify the subnet ID and the route table ID.

```
# 6: Associate route table with public subnet
resource "aws_route_table_association" "PublicRTAssociation" {
  subnet_id      = aws_subnet.PublicSubnet.id
  route_table_id = aws_route_table.PublicRT.id
}
```

- Save your Terraform configuration file (e.g., `vpc.tf`) and run `terraform init` to initialize Terraform in the directory containing your configuration file.
- Then, run `terraform plan` to review the planned changes, followed by `terraform apply` to apply the configuration and create the VPC, public subnet, route table, and association.



Interview questions for Terraform

1. What is Terraform, and how does it differ from other infrastructure as code (IaC) tools?

Terraform is an open-source infrastructure as code (IaC) tool created by HashiCorp. It allows users to define and provision infrastructure using declarative configuration files. Unlike other IaC tools, Terraform is cloud-agnostic, meaning it can manage infrastructure across multiple cloud providers and on-premises environments using the same configuration language

2. Explain the core components of Terraform and their roles in infrastructure management.

- **Providers:** Responsible for interacting with APIs of various infrastructure platforms (e.g., AWS, Azure, Google Cloud).
- **Resources:** Represent infrastructure components such as virtual machines, networks, databases, etc., and define their configuration.
- **Variables:** Allow dynamic input values to be passed into Terraform configurations.
- **Outputs:** Provide a way to extract and display useful information about the infrastructure after it's been created.
- **State:** Stores the current state of infrastructure managed by Terraform, enabling it to perform updates and manage resources effectively

3. What is the purpose of the Terraform state file, and how is it managed?

The Terraform state file keeps track of the existing resources in your infrastructure. It's used to map your real-world resources to your configuration and is crucial for understanding the changes Terraform needs to make. Terraform can manage the state file locally or remotely, depending on the configuration.

4. How does Terraform ensure idempotent infrastructure deployments?

Terraform ensures idempotency by comparing the current state of infrastructure (stored in the state file) with the desired state defined in the configuration. It only makes necessary changes to achieve the desired state, ensuring that running the same configuration multiple times results in the same infrastructure state.

5. Describe the difference between Terraform's declarative syntax and imperative syntax.

- **Declarative syntax:** In Terraform, you define the desired end state of your infrastructure without specifying the steps needed to achieve it. Terraform handles the execution order and dependencies automatically.

- **Imperative syntax:** In imperative programming, you specify the exact steps and commands to achieve a particular outcome. Terraform's declarative syntax abstracts away these implementation details.

6. What are Terraform providers, and why are they important?

Terraform providers are plugins responsible for managing resources in various infrastructure platforms (e.g., AWS, Azure, Kubernetes). They abstract the API interactions required to create, update, and delete resources, making Terraform cloud-agnostic and flexible.

7. Explain the difference between Terraform modules and resources.

- **Resources:** Represent individual infrastructure components (e.g., EC2 instance, S3 bucket) and define their configuration within a Terraform configuration file.
- **Modules:** Reusable packages of Terraform configurations that represent a set of resources and can be called from other configurations. Modules allow for abstraction, encapsulation, and reuse of infrastructure configurations.

8. How does Terraform handle dependencies between resources?

Terraform automatically manages dependencies between resources based on the configuration. It determines the dependency graph by analyzing resource relationships and ensures resources are created, updated, or destroyed in the correct order to satisfy these dependencies.

9. What are the benefits of using remote state in Terraform?

- **Concurrency:** Allows multiple users to work on the same infrastructure concurrently without conflicts.
- **Security:** Remote state can be stored securely, reducing the risk of sensitive information exposure.
- **State Locking:** Remote backends provide built-in state locking mechanisms to prevent concurrent modifications.
- **Collaboration:** Facilitates collaboration by centralizing the state and making it accessible to the entire team.
- **State Versioning:** Many remote backends offer versioning capabilities, enabling rollbacks and historical tracking of infrastructure changes.

10. What strategies can you implement for handling sensitive data (secrets) in Terraform configurations?

- Utilize Terraform's built-in mechanisms like `terraform.tfvars` files for storing sensitive data.
- Leverage environment variables to pass sensitive data securely.

- Use external secret management systems such as HashiCorp Vault or AWS Secrets Manager.
- Encrypt sensitive data using tools like KMS and decrypt them during runtime.

11. How does Terraform support infrastructure drift detection and remediation?

Terraform can detect infrastructure drift by comparing the current state of deployed resources with the state described in Terraform configurations. Remediation involves running `terraform plan` and `terraform apply` to reconcile the desired state with the actual state.

12. What are the advantages of using Terraform in a CI/CD pipeline?

- Enables infrastructure as code, allowing version control and code review.
- Facilitates automated provisioning of infrastructure.
- Ensures consistency and repeatability across environments.
- Integrates seamlessly with CI/CD tools for continuous deployment.

13. Explain the concept of Terraform workspaces and when you might use them.

Terraform workspaces enable managing multiple environments within a single configuration. They allow you to maintain separate state files for each environment, making it easier to manage configurations for dev, staging, and production environments.

14. What are some common best practices for organizing Terraform configurations and modules?

- Modularize configurations into reusable modules.
- Separate environments using workspaces or directories.
- Use variables and outputs for parameterization and abstraction.
- Version control configurations using Git.
- Implement a folder structure that reflects the organization's architecture.

15. Describe the process of creating custom Terraform providers?

- Define provider schema using the Provider SDK.
- Implement CRUD operations for resources using the SDK.
- Compile provider code into a binary.
- Distribute the provider binary and register it with Terraform.

16. How can you manage multiple environments (e.g., dev, staging, production) using Terraform?

- Use Terraform workspaces to maintain separate state files.
- Utilize variable files or environment variables for configuration parameterization.
- Employ conditional logic within configurations to handle environment-specific settings.

17. What are the limitations or challenges you've encountered when using Terraform, and how did you address them?

- State management complexities: Addressed by using remote state backends.
- Limited support for certain cloud provider features: Mitigated by extending Terraform with custom providers or leveraging native cloud tools.
- Dependencies between resources: Managed by careful resource ordering and using Terraform features like `depends_on`.

18. What security considerations should you take into account when using Terraform to provision infrastructure?

- Protect sensitive data using encryption and secure storage mechanisms.
- Limit access to Terraform configuration files and state files.
- Implement least privilege principles when configuring IAM roles and policies.
- Regularly audit and review Terraform configurations for security vulnerabilities.

19. How do you handle state locking in Terraform to prevent concurrent modifications?

Use Terraform's remote state backends, which provide built-in locking mechanisms to prevent concurrent modifications. This ensures that only one Terraform operation can modify the state at a time.

20. Can you explain the difference between Terraform `apply`, `plan`, and `destroy` commands?

- `terraform plan`: Generates an execution plan showing what actions Terraform will take when `terraform apply` is executed.
- `terraform apply`: Applies the changes described in the Terraform configuration to the actual infrastructure.
- `terraform destroy`: Destroys all the resources described in the Terraform configuration, effectively deleting the infrastructure.

