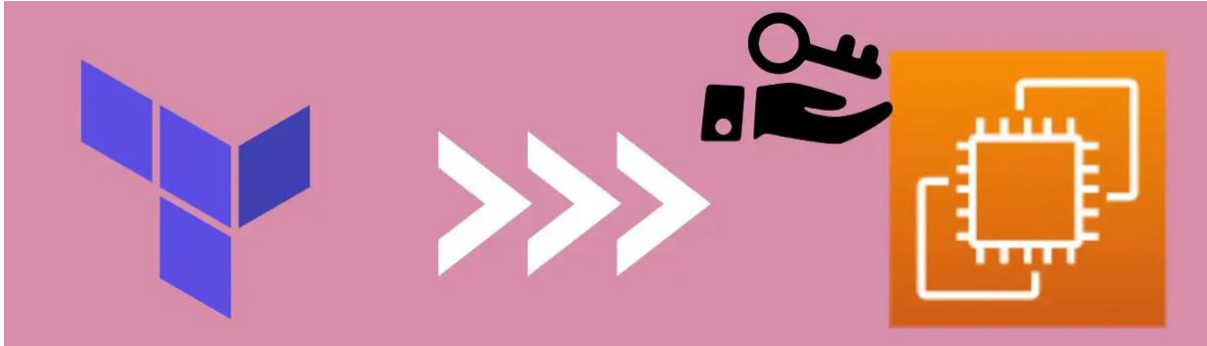


# Automating EC2 Instance Launch by using Terraform

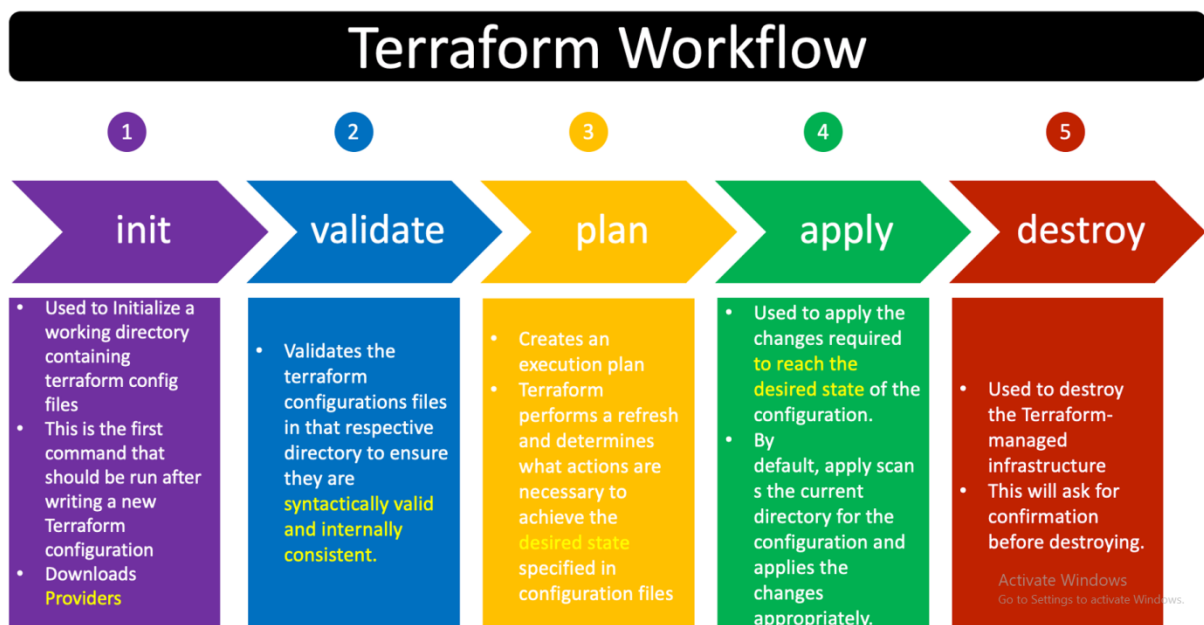


## Introduction:-

### 1.What is Terraform?

Terraform is an open-source infrastructure as code (IaC) tool created by HashiCorp. It is used for building, changing, and versioning infrastructure efficiently. Terraform allows you to define your infrastructure in a declarative configuration language and then create, update, or delete infrastructure resources to match that configuration.

### 2.What is the workflow of Terraform?



### 3.Steps to launch a AWS instance by using Terraform

#### a>Login to AWS Account and Create EC2 Instance

EC2 > Instances > Launch an instance

Launch an instance

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags

Name

terraform-1

Add additional tags

Application and OS Images (Amazon Machine Image)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images

Summary

Number of instances

1

Software Image (AMI)

Amazon Linux 2023 AMI 2023.2.2...read more

ami-0dbc3d7bc646e8516

Virtual server type (instance type)

t2.micro

Firewall (security group)

New security group

Storage (volumes)

1 volume(s) - 8 GiB

Cancel

Launch instance

Review commands

#### b>Select ubuntu and launch the instance

Amazon Linux

macOS

Ubuntu

Windows

Red Hat

SUSE Li

Browse more AMIs

Amazon Machine Image (AMI)

Ubuntu Server 22.04 LTS (HVM), SSD Volume Type

ami-0fc5d935ebf8bc3bc (64-bit (x86)) / ami-016485166ec7fa705 (64-bit (Arm))

Virtualization: hvm ENA enabled: true Root device type: ebs

Free tier eligible

Description

Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2023-09-19

Architecture

64-bit (x86)

AMI ID

ami-0fc5d935ebf8bc3bc

Verified provider

Instances (1/2)

Find Instance by attribute or tag (case-sensitive)

Connect Instance state Actions Launch instances

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
terraform-1	i-0b0facab089d6583b	Running	t2.micro	Initializing	No alarms	us-east-1b	ec2-34-229-145-18
linux-demo	i-04b215f5b72534eda	Stopped	t2.micro	-	No alarms	us-east-1b	-

c>After creating the instance please click on connect

The screenshot shows the 'EC2 Instance Connect' interface in the AWS Management Console. It has four tabs: 'EC2 Instance Connect' (selected), 'Session Manager', 'SSH client', and 'EC2 serial console'. Under 'EC2 Instance Connect', the 'Instance ID' is 'i-0b0facab089d6583b (terraform-1)'. The 'Connection Type' section has two options: 'Connect using EC2 Instance Connect' (selected) and 'Connect using EC2 Instance Connect Endpoint'. The 'Public IP address' is '34.229.145.185'. The 'User name' field contains 'ubuntu'. A note at the bottom states: 'Note: In most cases, the default user name, ubuntu, is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.' At the bottom right are 'Cancel' and 'Connect' buttons.

d>Now please follow the commands as below

1.Update your package list to ensure you have the latest information about available packages.

```
# sudo apt update
```

2.Ensure that your system is up to date and you have installed the gnupg, software-properties-common, and curl packages installed. You will use these packages to verify HashiCorp's GPG signature and install HashiCorp's Debian package repository.

```
# sudo apt-get update && sudo apt-get install -y gnupg software-properties-common
```

3.Install the HashiCorp GPG key.

```
# wget -O- https://apt.releases.hashicorp.com/gpg | \
```

```
gpg --dearmor | \
```

```
sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg
```

4. Verify the key's fingerprint

```
# gpg --no-default-keyring \
```

```
--keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg \
```

```
--fingerprint
```

#####

The gpg command will report the key fingerprint:

/usr/share/keyrings/hashicorp-archive-keyring.gpg

-----

pub rsa4096 XXXX-XX-XX [SC]

AAAA AAAA AAAA AAAA

uid [ unknown] HashiCorp Security (HashiCorp Package Signing)  
<security+packaging@hashicorp.com>

sub rsa4096 XXXX-XX-XX [E]

#####

5. Add the official HashiCorp repository to your system. The `lsb_release -cs` command finds the distribution release codename for your current system, such as `buster`, `groovy`, or `sid`.

```
# echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
```

```
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
```

```
sudo tee /etc/apt/sources.list.d/hashicorp.list
```

6. Download the package information from HashiCorp

```
# sudo apt update
```

7. Install Terraform from the new repository.

```
# sudo apt-get install terraform
```

Note:- please copy commands properly or refer below link

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

```
ubuntu@ip-172-31-23-194:~$ terraform --version
Terraform v1.6.2
on linux_amd64
ubuntu@ip-172-31-23-194:~$
```

e>After installing terraform please create a directory called terraform {any name} enter the directory .

```
ubuntu@ip-172-31-23-194:~$ mkdir terraform
ubuntu@ip-172-31-23-194:~$ ls
terraform
ubuntu@ip-172-31-23-194:~$ cd terraform/
```

f>Here after entering we need to export the aws access key and secret key to connect with AWS providers.

```
ubuntu@ip-172-31-23-194:~/terraform$ export AWS Access key id
ubuntu@ip-172-31-23-194:~/terraform$ export AWS secret key id
```

#export AWS\_ACCESS\_KEY\_ID= [.....]

#export AWS\_SECRET\_ACCESS\_KEY= [.....]

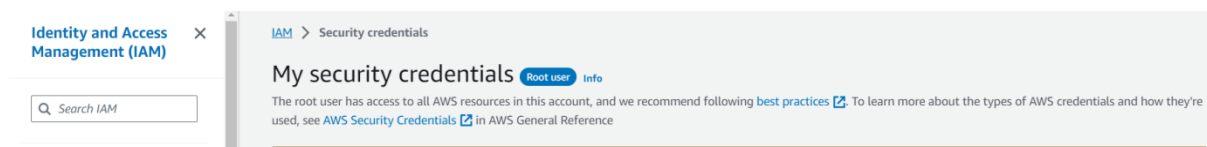
\*We can use above commands to export the AWS keys and you can give access keys and give them in place of this [.....] as mentioned above\*

Just for the information:-

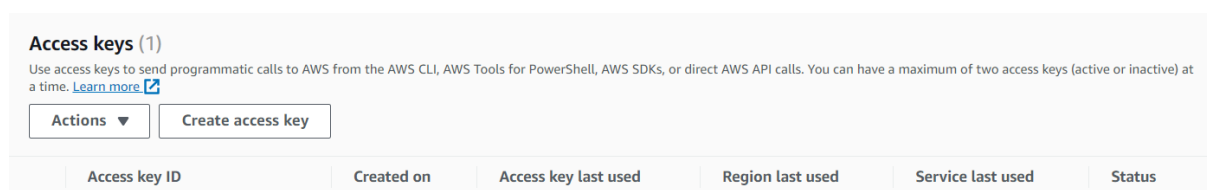
<> To create keys{Access & Secret}



\$.Go to console click on your account and select security credentials or search IAM and go to >security credentials.



\$. In that page please select create new access keys and after using that you can delete that.



After creating access keys you will get the new keys [access and secret] .

g>After giving keys please create file inside the directory as main.tf

\*terraform file extension is .tf

```
ubuntu@ip-172-31-23-194:~/terraform$ vi main.tf
```

f>After enter into file press “i” key in keyboard and paste the below configuration

```
terraform {  
  required_providers {  
    aws = {  
      source  = "hashicorp/aws"  
      version = "~> 4.16"  
    }  
  }  
  
  required_version = ">= 1.2.0"  
}  
  
provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "app_server" {  
  ami           = "ami-830c94e3"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "ExampleAppServerInstance"  
  }  
}
```

\* if required please click below link so that you can get the configuration file\*

<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-build>

## g>Next step initialize

When you create a new configuration — or check out an existing configuration from version control — you need to initialize the directory with `terraform init`.

Initializing a configuration directory downloads and installs the providers defined in the configuration, which in this case is the aws provider.

Initialize the directory.

```
ubuntu@ip-172-31-23-194:~/terraform$ vi main.tf
ubuntu@ip-172-31-23-194:~/terraform$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 4.16"...
- Installing hashicorp/aws v4.67.0...
- Installed hashicorp/aws v4.67.0 (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.
```

We recommend using consistent formatting in all of your configuration files. `#terraform fmt` command automatically updates configurations in the current directory for readability and consistency.

Format your configuration. Terraform will print out the names of the files it modified, if any. In this case, your configuration file was already formatted correctly, so Terraform won't return any file names.

You can also make sure your configuration is syntactically valid and internally consistent by using the `# terraform validate` command.

Validate your configuration. The example configuration provided above is valid, so Terraform will return a success message.

```
ubuntu@ip-172-31-23-194:~/terraform$ terraform fmt
main.tf
ubuntu@ip-172-31-23-194:~/terraform$ terraform validate
Success! The configuration is valid.
```

h>After that please enter # terraform plan enter and followed by # terraform apply

```
ubuntu@ip-172-31-23-194:~/terraform$ 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_instance.app_server will be created
+ resource "aws_instance" "app_server" {
  + ami                        = "ami-830c94e3"
  + 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_state             = (known after apply)
```

```
ubuntu@ip-172-31-23-194:~/terraform$ terraform apply

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

Terraform will perform the following actions:

# aws_instance.app_server will be created
+ resource "aws_instance" "app_server" {
  + ami                        = "ami-830c94e3"
  + 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_state             = (known after apply)
  + instance_type              = "t2.micro"
```

```
    + "Name" = "ExampleAppServerInstance"
  }
  + 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.

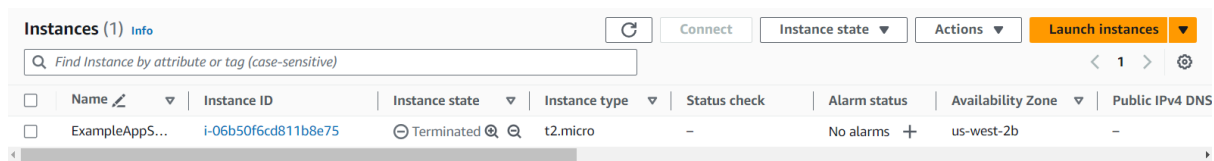
  Enter a value: yes

aws_instance.app_server: Creating...
aws_instance.app_server: Still creating... [10s elapsed]
aws_instance.app_server: Still creating... [20s elapsed]
aws_instance.app_server: Still creating... [30s elapsed]
aws_instance.app_server: Still creating... [40s elapsed]
aws_instance.app_server: Creation complete after 43s [id=i-06b50f6cd811b8e75]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
ubuntu@ip-172-31-23-194:~/terraform$
```



i>Now you can go to AWS console and check the region where you wished to create instance will be created. And verify that can destroy that instance by using # **terraform destroy**



	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
<input type="checkbox"/>	ExampleAppS...	i-06b50f6cd811b8e75	Terminated	t2.micro	-	No alarms	us-west-2b	-

j>Some useful other commands in terraform

### #**terraform taint**

Command: `terraform taint RESOURCE_ADDRESS`

Purpose: The terraform taint command is used to mark a specific resource as tainted. Tainting a resource tells Terraform that it needs to be recreated the next time you apply your Terraform configuration. This can be useful when you want to force the recreation of a resource due to changes in its configuration or dependencies. Tainting is often used when you want to trigger the replacement of a resource without making any other changes in your Terraform code.

### #**terraform untaint**

Command: `terraform untaint RESOURCE_ADDRESS`

Purpose: The terraform untaint command is used to remove the taint from a previously tainted resource. This means that the resource will not be recreated during the next terraform apply. Untainting is typically used when you no longer want to force the replacement of a resource that was previously tainted.

### #**terraform show**

Command: `terraform show [PLAN_FILE]`

Purpose: The terraform show command is used to display the current state of your infrastructure as managed by Terraform. It can show the current resource attributes and their values, making it useful for inspecting the state after applying your configuration. You can provide an optional PLAN\_FILE argument if you want to show a plan file's content.

### #**terraform refresh**

Command: `terraform refresh`

Purpose: The terraform refresh command is used to update the Terraform state file to reflect the real-world state of your infrastructure. This command queries the actual resources in your cloud provider and updates the state file with their current attributes. It

can be helpful in situations where the state file becomes out of sync with the actual infrastructure, possibly due to changes made outside of Terraform.

## #terraform state

Command: terraform state [SUBCOMMAND] [ARGS]

Purpose: The terraform state command is used to perform various operations on Terraform's state file, which stores information about your managed resources. You can use it to inspect, modify, and manage resource states.