Terraform create EC2 Instance on AWS

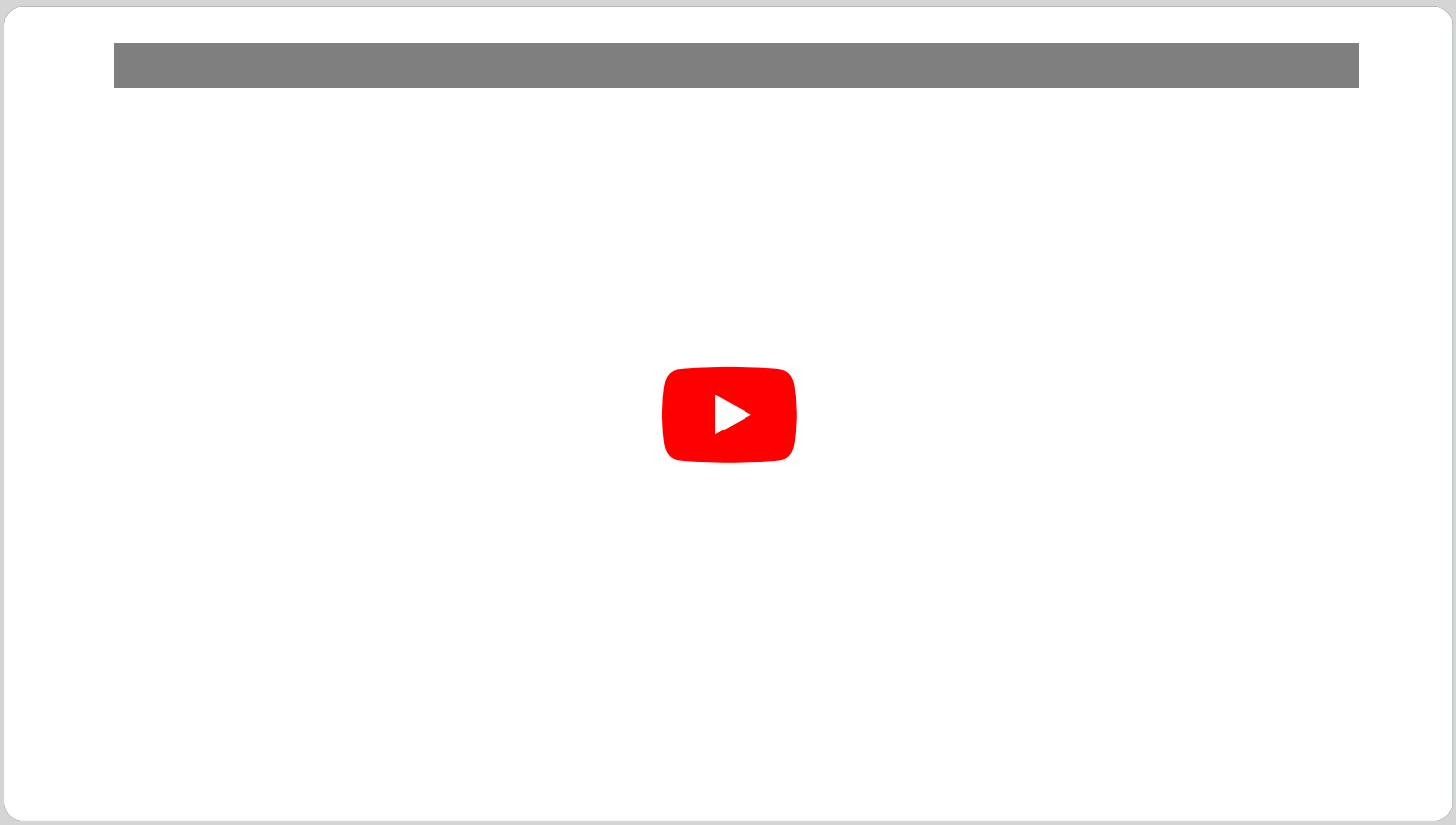
Mar 19, 2021 · 12 min read · Share on:

This guide will help you to **Create your first AWS EC2 using terraform**.

When it comes to IAC(Infrastructure As Code) Terraform is always the first choice of DevOps although there are many alternatives available in the market such as Ansible, Chef, Puppet, SaltStack, CloudFormation but due the fact that -

1. Terraform is really easy install
2. Terraform has very good API documentation
3. It is widely adopted in the DevOps community
4. Great support for a popular cloud service provider such as Google Cloud Platform, AWS.

(To know more on How to setup Virtual Machine on Google Cloud - Click Here)



1. The only prerequisite is - You must install Terraform before jumping to Google Cloud setup.

Table of Content

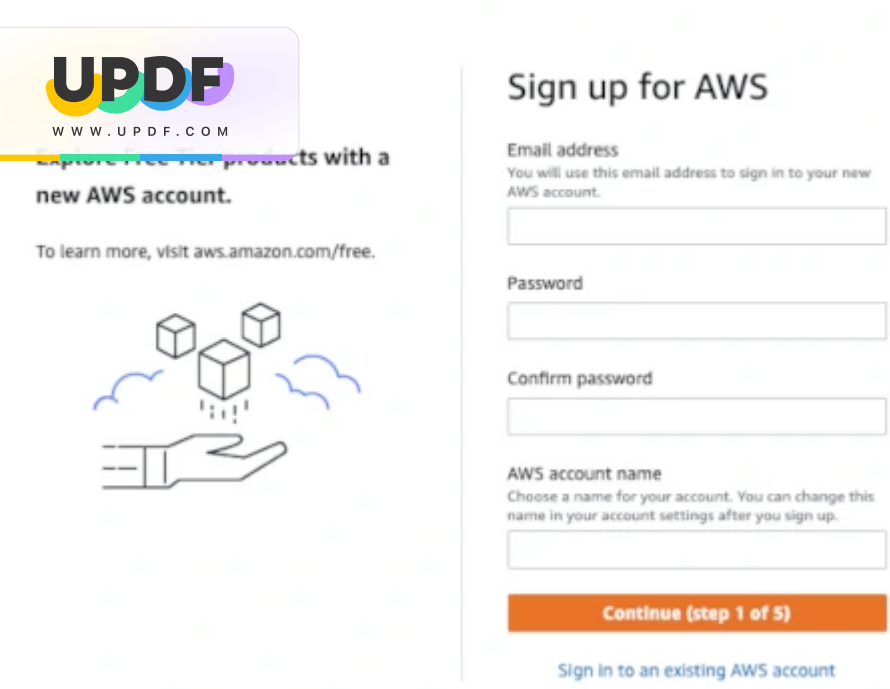
1. Setup AWS Account

As our aim of this article to setup an AWS EC2 instance the first step would be to create an AWS account.

If you are a beginner and want to learn the Terraform then AWS provides you free tier - *12 months or 750 Hours/month*, where you can experiment.

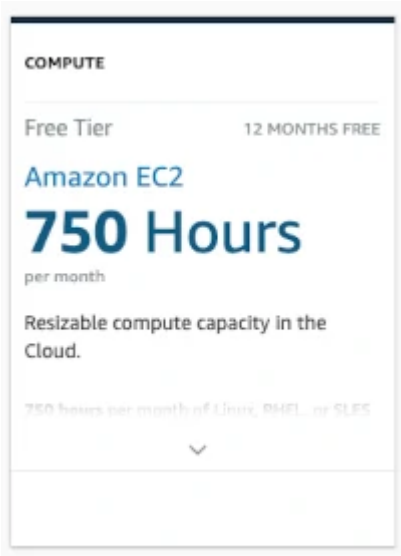
1.1 Sign up for AWS account

Goto https://aws.amazon.com/ and click on Complete sign-up



AWS Sing up page

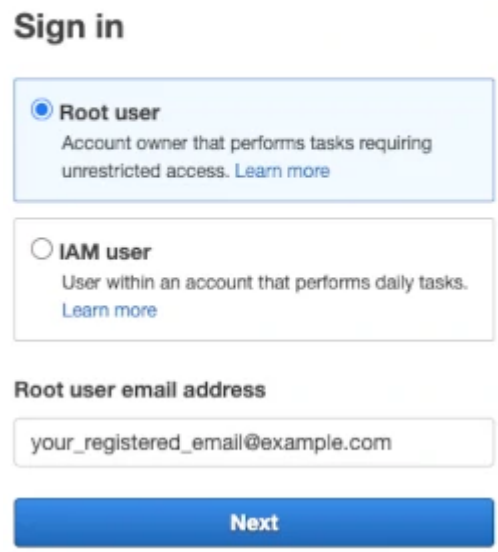
You need to choose **AMAZON EC2 Free Tier**. This tier is sufficient enough for learning purposes.



AWS EC2 Free tier

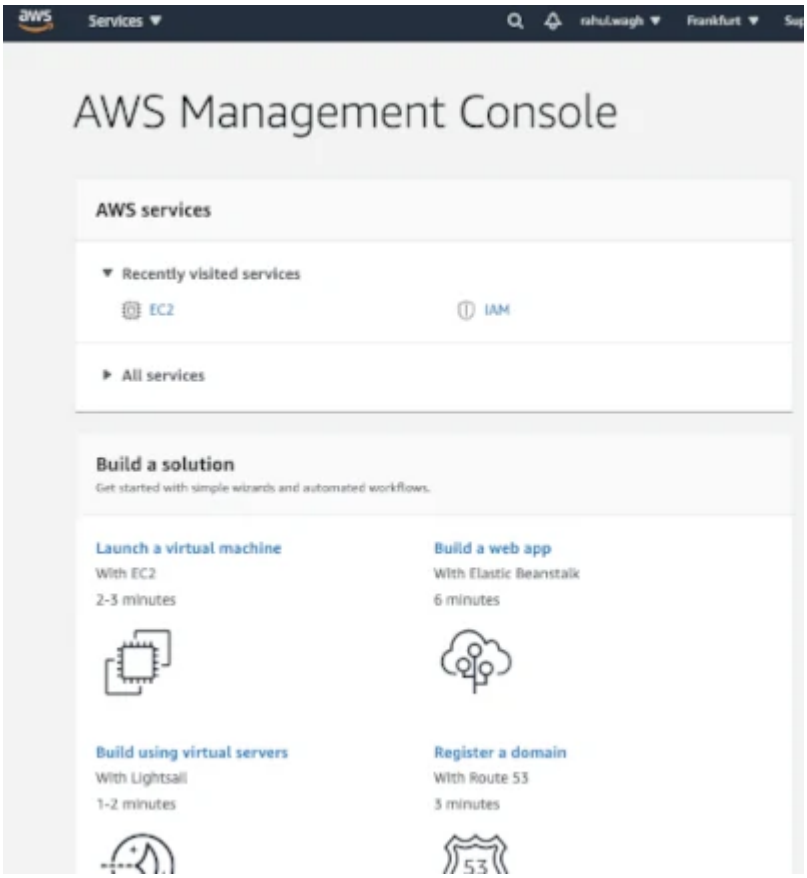
Amazon will ask for your Credit card number to complete the sign-up process, AWS will debit around 1$ so that they can verify your card details. Amazon will refund the amount after the authorization.

1.2 After Signup LogIn as ROOT user



AWS console login as root user

Once you login into your AWS account you should see a dashboard, I know the dashboard can be little overwhelming for the first time user but do not worry we are gonna take one item at a time.

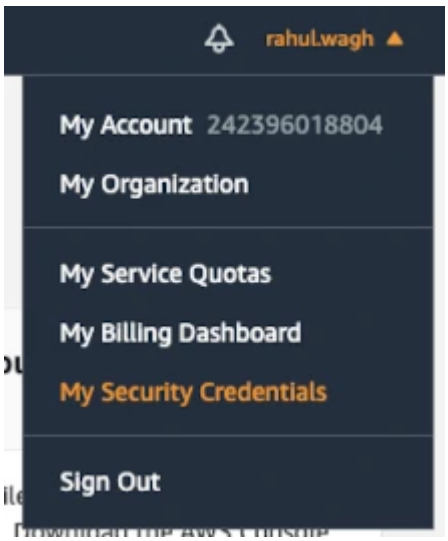


2. Generate Access keys (access key ID and secret access key)

Terraform installed on your Desktop/Laptop needs to communicate with AWS and to make this communication terraform needs to be authenticated.

For authentication, we need to generate **Access Keys (access key ID and secret access key)**. These access keys can be used for making - programmatic calls to AWS from the AWS CLI, Tools for PowerShell, AWS SDKs, or direct AWS API calls.

1. Goto *My Security Credentials*



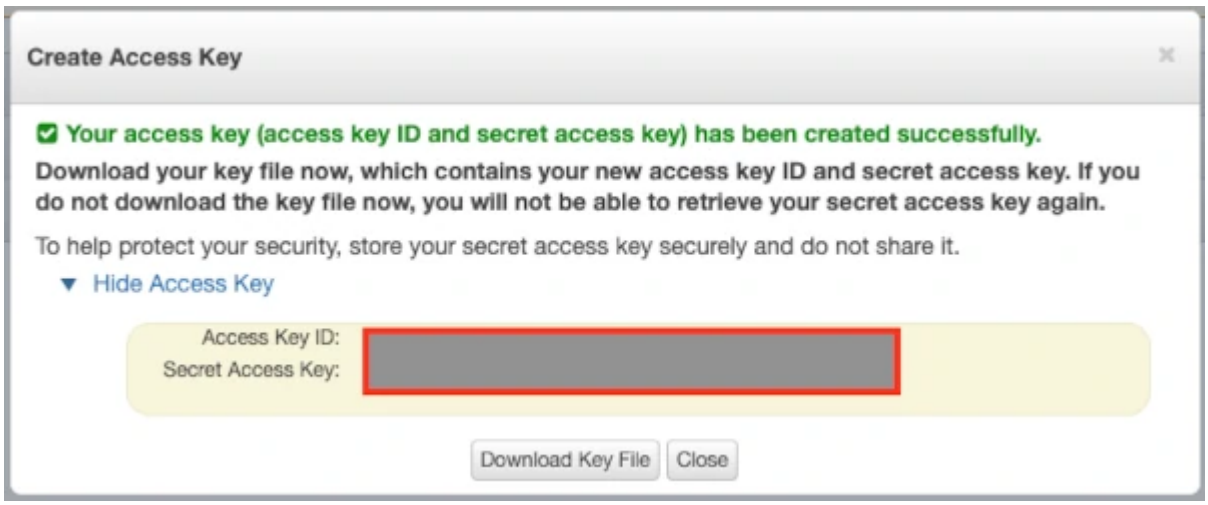
AWS security credentials

2. On *Your Security Credentials* page click on *Access keys (access key ID and secret access key)*

AWS access key create new access key

3. Click on **Create New Access key**

4. Copy the *Access Key ID* and *Secret Access Key* (*Note:- You can view the Secret Access Key only once, so make sure to copy it.*)



AWS access key id and secret access key generated

3. Create your first Terraform infrastructure (main.tf)

Before we start writing terraform script, the first thing to learn over here is - *"You need to save your configuration with .tf extension"*

We will start by creating an empty `main.tf` file.

3.1 Provider

The first line of code in which we are going to write is *provider*.

We need to tell terraform which cloud provider we are going to connect .e.g - AWS, Google, or Azure

As this article is focused on **AWS**, so we are going to mention **AWS** as our provider.

Here is the basic syntax for the provider

```yaml
resource "<PROVIDER>_<TYPE>" "<NAME>" {
 [CONFIG …]
}
```

1. "PROVIDER _ TYPE" - aws, google
2. "NAME" - You can define your name.

This is how our final `main.tf` will look like for AWS -

```yaml
provider "aws" {
  region     = "eu-central-1"
  access_key = "XXXXXXXXXXXXXXXXXXX"
  secret_key = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
}
```

Note: You can copy access_key and secret_key from Step-2

3.2 resource - "aws_instance"

So what do you mean by resource?

Resource - It is something that we are going to provision/start on AWS.

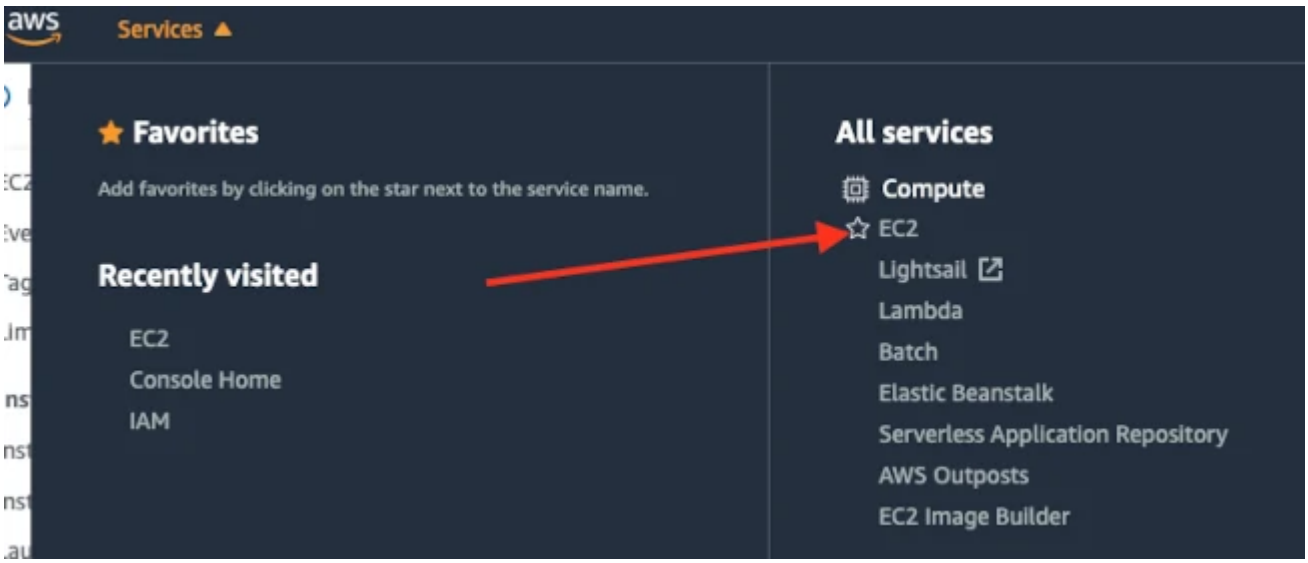
Now for this article, we are going to provision EC2 instance on AWS.

But before we provision the EC2 instance, we need to gather few points -

1. **ami** = you need to tell Terraform which AMI(Amazon Machine Image) you are going to use. Is it going to be Ubuntu, CentOS or something else
2. **instance_type** = Also based on your need you have to choose the instance_type and it can be t2.nano, t2.micro, t2. small etc.

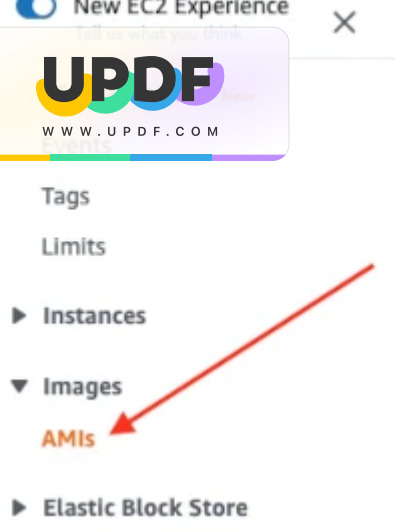
3.3 How to find ami(Amazon Machine Image)

1. To find the correct *ami* you need to Goto: `Services -> EC2`



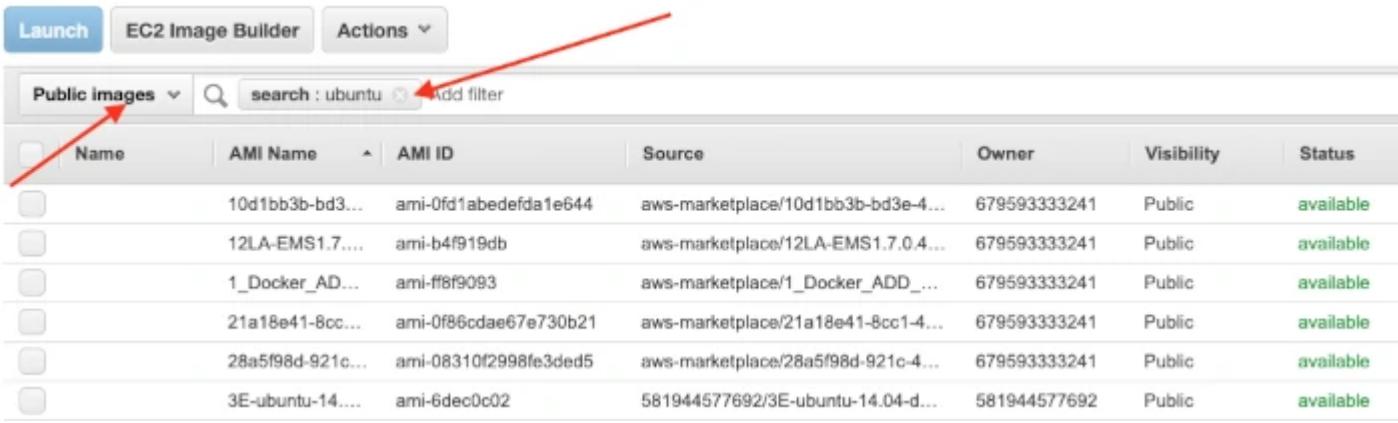
AWS services EC2

2. In the left Navigation you will find `Images -> AMIs`



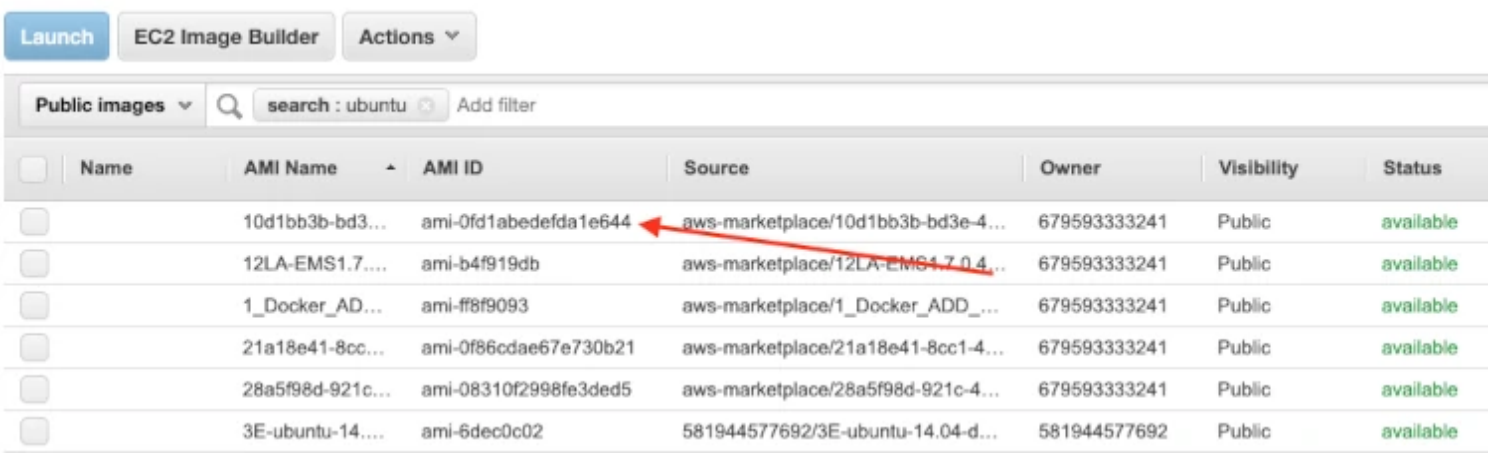
AWS AMIs option in the left navigation

3. On the search menu click on `public images` and then type `Ubuntu`.



AWS access key id and secret access key generated

4. Copy the AMI ID from the search result.



AWS access key id and secret access key generated

3.4 How to find correct instance_type

You can find the correct `instance_type` by visiting this page.

Since I am looking for a very basic `instance_type` not production level instance, so I choose `t2.micro`

Here is the `aws_instance` configuration -

```BASH
resource "aws_instance" "ec2_example" {
    ami = "ami-0767046d1677be5a0"
    instance_type = "t2.micro"
    tags = {
        Name = "Terraform EC2"
    }
}
```

4. terraform commands

Alright, now we have completed all the pre-requisites for provisioning our first `ec2` instance on the AWS.

4.1 terraform plan

The first command which we are going to run is -

```BASH
terraform init
```

```
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v3.32.0...
- Installed hashicorp/aws v3.32.0 (signed by HashiCorp)

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
```

The terraform init command is responsible for downloading all the dependencies which are required for the provider `AWS`.

Once you issue the `terraform init` command it will download all the provider's dependencies on your local machine.

4.2 terraform plan

This command will help you to understand how many resources you are gonna add or delete.

Here is the command -

```bash
terraform plan
```

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.ec2_example will be created
```

```
      + 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)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + outpost_arn                  = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
      + private_dns                  = (known after apply)
      + private_ip                   = (known after apply)
      + public_dns                   = (known after apply)
      + public_ip                    = (known after apply)
      + secondary_private_ips        = (known after apply)
      + security_groups              = (known after apply)
      + source_dest_check            = true
      + subnet_id                    = (known after apply)
      + tags                         = {
          + "Name" = "Terraform EC2"
        }
      + tenancy                      = (known after apply)
      + vpc_security_group_ids       = (known after apply)

      + ebs_block_device {
          + delete_on_termination = (known after apply)
          + device_name           = (known after apply)
          + encrypted             = (known after apply)
          + iops                  = (known after apply)
          + kms_key_id            = (known after apply)
          + snapshot_id           = (known after apply)
          + tags                  = (known after apply)
          + throughput            = (known after apply)
          + volume_id             = (known after apply)
...
```

As you can see the output of `terraform plan`, at the end it will show all the resources adde and deleted.

4.3 Terraform apply

This command will do some real stuff on AWS. Once you will issue this command, it will be going to connect to AWS and then finally going to provision AWS instance.

Here is the command -

```bash
terraform apply
```

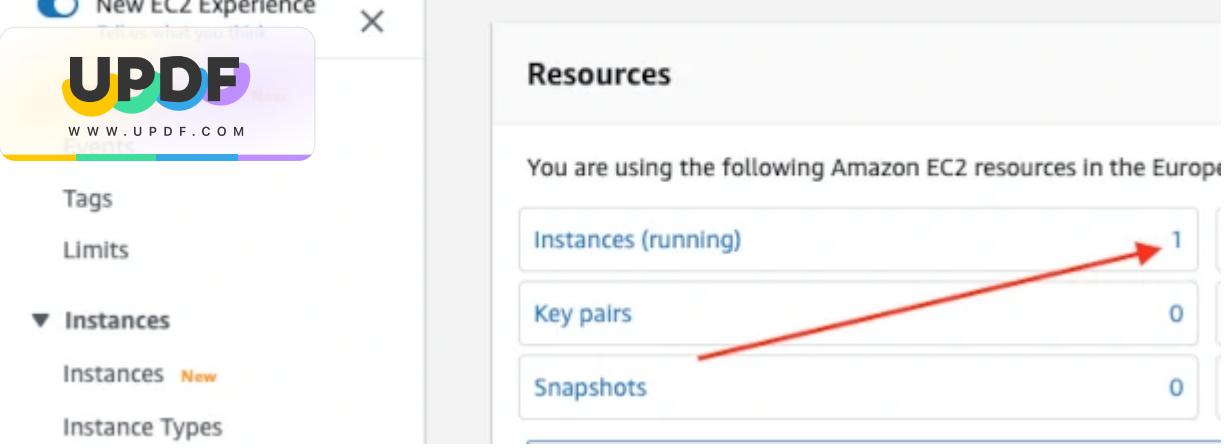
```bash
Plan: 1 to add, 0 to change, 0 to destroy.

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.ec2_example: Creating...
aws_instance.ec2_example: Still creating... [10s elapsed]
aws_instance.ec2_example: Still creating... [20s elapsed]
aws_instance.ec2_example: Still creating... [30s elapsed]
aws_instance.ec2_example: Creation complete after 33s [id=i-0a948ac635a2010f1]

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

As you can see the log output has created `t2.micro` instance.

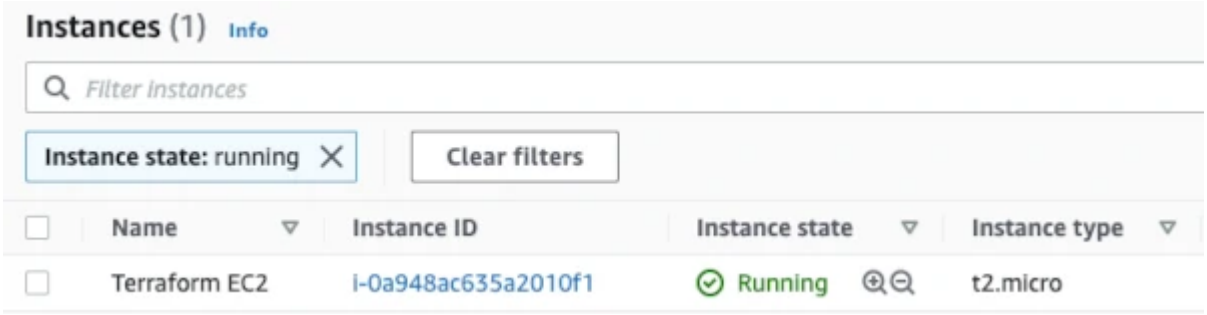
4.4 Verify the EC2 setup

Let's verify the setup by going back to AWS console.

`Goto -> Services -> EC2` you should see 1 instance running.



AWS ec2 running instance



AWS ec2 running instance with more details

You can also see the **Tag name - Terraform EC2** which we mentioned in the terraform script.

4.4 terraform destroy

Now we have seen how to write your terraform script and how to provision your EC2 instance.

Let see how to remove or delete everything from AWS.

We are going to use the command -

```bash
terraform destroy
```

```
Do you 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: Destroying... [id=i-0a948ac635a2010f1]
aws_instance.ec2_example: Still destroying... [id=i-0a948ac635a2010f1, 10s elapsed]
aws_instance.ec2_example: Still destroying... [id=i-0a948ac635a2010f1, 20s elapsed]
aws_instance.ec2_example: Still destroying... [id=i-0a948ac635a2010f1, 30s elapsed]
aws_instance.ec2_example: Still destroying... [id=i-0a948ac635a2010f1, 40s elapsed]
aws_instance.ec2_example: Destruction complete after 40s

Destroy complete! Resources: 1 destroyed.
```

It will remove all the running EC2 Instances.

> The previous steps were very basic Terraform examples for setting up an EC2 instance. But in the actual project I think you need to do a little more than just basic ***EC2 Instance*** setup. The remaining part of this blog post will help you to do some advanced setup with your ***EC2 Instance*** using Terraform.
>
> **For example -**
>
> Setup up a custom startup script for an Amazon Elastic Compute Cloud (EC2) using Terraform
>
> Copy and Execute script from local machine to remote EC2 instance using file and remote-exec provisioner

5. Setup up a custom startup script for an Amazon Elastic Compute Cloud (EC2) using Terraform

To set up a custom startup script for an Amazon Elastic Compute Cloud (EC2) instance using Terraform, you can use the user_data attribute of the aws_instance resource. The user_data

Here is an example of how to use the `user_data` attribute to specify a custom startup script for an EC2 instance in Terraform. In this example -

1. First we will setup the EC2 instance
2. Secondly we are going to install Apache Web Server
3. At last we are going to set up a very basic HTML page and deploy it on the Apache web server

BASH

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "<INSERT_YOUR_ACCESS_KEY>"
   secret_key = "<INSERT_YOUR_SECRET_KEY>"
}

resource "aws_instance" "ec2_example" {

    ami = "ami-0767046d1677be5a0"
    instance_type = "t2.micro"
    key_name= "aws_key"
    vpc_security_group_ids = [aws_security_group.main.id]

  user_data = <<-EOF
      #!/bin/sh
      sudo apt-get update
      sudo apt install -y apache2
      sudo systemctl status apache2
      sudo systemctl start apache2
      sudo chown -R $USER:$USER /var/www/html
      sudo echo "<html><body><h1>Hello this custom page built with Terraform User Data<
      EOF
}
```

6. Copy and Execute the script from local machine to remote EC2 instance using file and remote-exec provisioner

In case of large and complex scripts, you should use `file` as well as `remote-exec` provisioner. *(For more in-depth tutorial please refer to this blog post on - What is terraform provisioner)*

file provisioner- First we will use `file` provisioner to copy the file to the remote EC2 instance.

remote-exec provisioner- To execute the script copied using `file` provisioner

Here is an example where you can also use the `file provisioner` in Terraform to copy a script file from the local machine to the instance and execute it.

```
#main.tf

provider "aws" {
   region     = "eu-central-1"
   access_key = "<INSERT_YOUR_ACCESS_KEY>"
   secret_key = "<INSERT_YOUR_SECRET_KEY>"
}

resource "aws_instance" "ec2_example" {

    ami = "ami-0767046d1677be5a0"
    instance_type = "t2.micro"
    key_name= "aws_key"
    vpc_security_group_ids = [aws_security_group.main.id]

  # file provisioner -
  # It will copy the "startup.sh" to remote machine
  provisioner "file" {
    source      = "/home/rahul/Jhooq/startup.sh"
    destination = "/home/ubuntu/startup.sh"
  }

  # connection -
  # This block will be used for ssh connection to initiate the copy
  connection {
      type        = "ssh"
      host        = self.public_ip
      user        = "ubuntu"
      private_key = file("/home/rahul/Jhooq/keys/aws/aws_key")
      timeout     = "4m"
   }
}
```

```
    provisioner "remote-exec" {
      inline = [
        "chmod +x /home/ubuntu/startup.sh",
        "/home/ubuntu/startup.sh"
      ]
    }

resource "aws_security_group" "main" {
  egress = [
    {
      cidr_blocks      = [ "0.0.0.0/0", ]
      description      = ""
      from_port        = 0
      ipv6_cidr_blocks = []
      prefix_list_ids  = []
...
```

Read More - Terragrunt -

1. How to use Terragrunt?

Posts in this Series

- Managing strings in Terraform: A comprehensive guide
- How to use terraform depends_on meta argument?
- What is user_data in Terraform?
- Why you should not store terraform state file(.tfstate) inside Git Repository?
- How to import existing resource using terraform import comand?
- Terraform - A detailed guide on setting up ALB(Application Load Balancer) and SSL?
- Testing Infrastructure as Code with Terraform?
- How to remove a resource from Terraform state?
- What is Terraform null Resource?
- In terraform how to skip creation of resource if the resource already exist?
- How to setup Virtual machine on Google Cloud Platform