Terraform

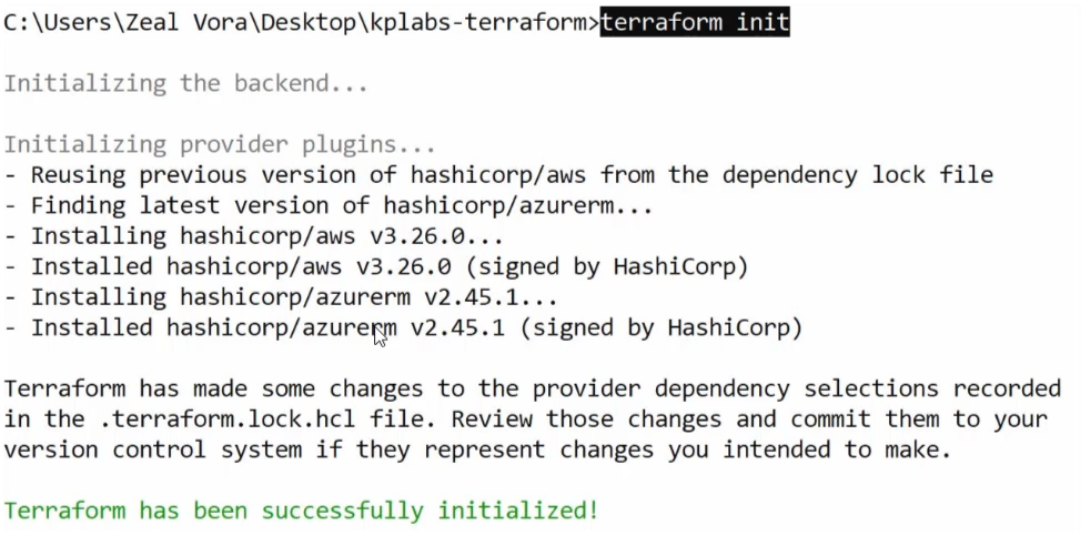
Introduction :

- **Terraform** is an **IAC** tool used for **provisioning and managing infrastructure**.
- IAC stands for **Infrastructure as Code**, allowing infrastructure management through code.
- **Terraform** is **free** and **open-source**.
- Terraform automates the **provisioning of servers** and **infrastructure**.
- It supports **multiple cloud providers** like **AWS, Azure, GCP**, and tools like **Kubernetes, Ansible**, etc., with **hundreds of supported providers**.
- Terraform can integrate with **configuration management tools** like **Ansible** to install and configure applications, and manage dependencies inside the server.
- Terraform files use the `.tf` extension.
- Terraform code is written in **HCL** (HashiCorp Configuration Language).
- Terraform code is **reusable** and supports **version control** for efficient management.
- It is **easily extensible** using **plug-ins** to add functionality.

Terraform AWS document link

Provider & Resource :

- **Provider**: Connects to the platform (e.g., AWS) to manage infrastructure. Adding a new provider we should run the `terraform init` command it will **download** the **plug-ins associated** with the **provider** (AWS, Azure, GCP).



- **Resource**: The actual infrastructure components (e.g., EC2 instance, S3 bucket) you want to create or manage.

Creating **EC2 instance** in 0.12 version of terraform :

```
provider "aws" {
    region = "us-east-1"
    access_key = "***************IM3"
    secret_key = "************************d1s6"
}

resource "aws_instance" "myec2" {
    ami = "ami-0182f373e66f89c85"
    instance_type = "t2.micro"
    key_name = "linux"    // your key-pair
    tags = {
      name = "surendhar"
    }
}
```

In Administrator PowerShell, we need to initialize the directory that contains `terraform.exe` using,

```
terraform init
```

Terraform plan shows the changes required to create, update, or destroy these resources based on the current state.

```
terraform plan
```

To run the script, we need to apply the Terraform configuration using,

```
terraform apply
```

Creating a **GitHub repository** :

```
terraform {
  required_providers {
    github = {
        source = "integrations/github"
        version = "6.3.0"
    }
```

```
  }
}

provider "github" {
    token = "**********************"
}

resource "github_repository" "surendhar" {
    name = "terraform_repo"
    visibility = "public"
}
```

```
terraform apply
```

Now, our github repo will created successfully.

Destroy all the resource that we created,

```
terraform destroy
```

If we want to **create** or **destroy only one resource** will use **-target** flag.

```
terraform destroy -target <resource_name>.<local_resource_name>
```

e.g. I have created an EC2 instance, and now I want to destroy that instance. Since other resources are running together, I will use the **-target** flag to destroy only the specific resource by specifying its **resource_name** and **local_resource_name** (e.g., **aws_instance.myec2**).

Auto approve :

To **auto approve** the **create state & destroy state** will use,

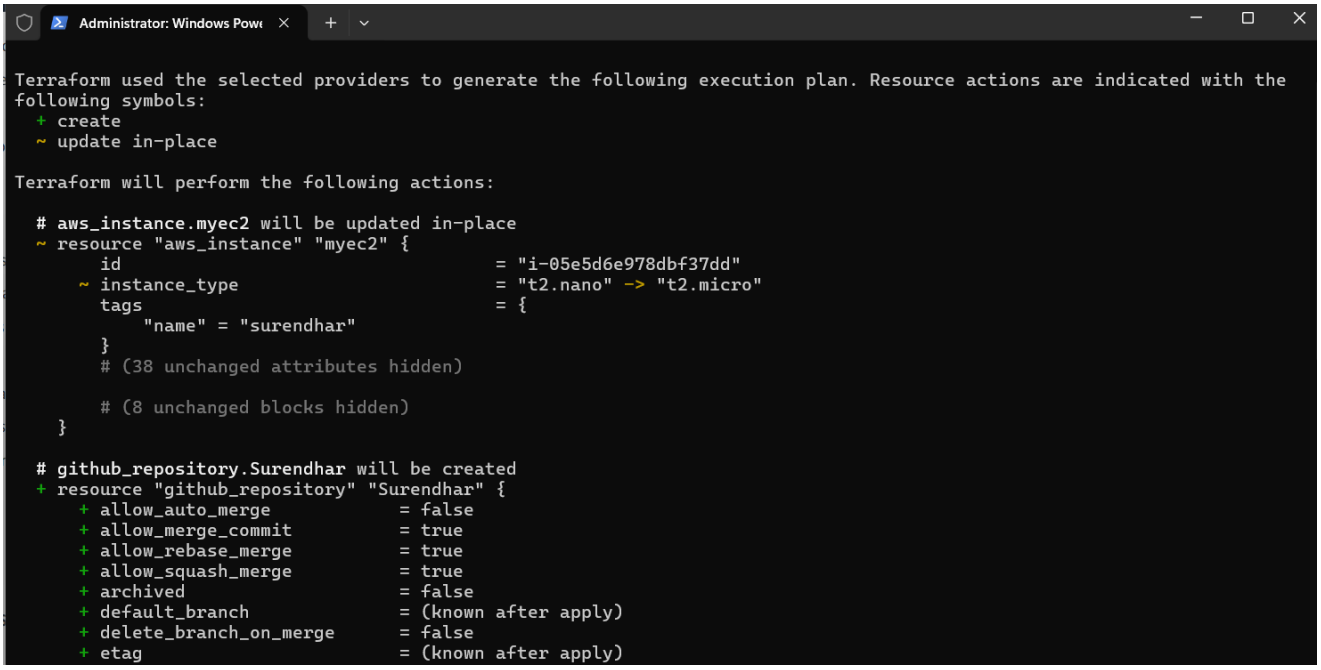
```
terraform apply -auto-approve
```

```
terraform apply -target <resource_name>.<local_resource_name> -auto-approve
```

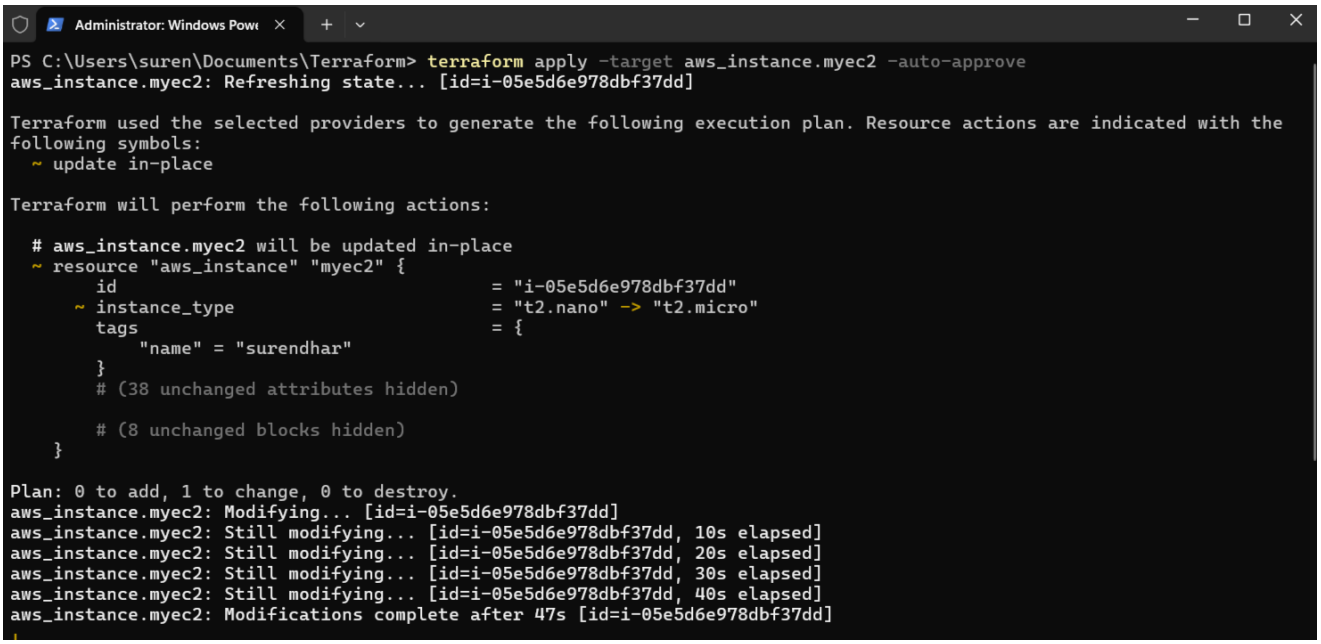
Desired State & Current State :

- **Desired State**: This is the configuration you want for your infrastructure. It defines how you want your resources (like servers, databases, networks) to be set up and managed. You specify this in your Terraform configuration files.
- Terraform creates or destroys infrastructure resources as defined by the **desired state**
- The infrastructure is described by the desired state.
- **Desired state** refers to the configuration or setup you want to achieve.
- **Current State**: This refers to the actual state of your infrastructure at any given moment. Terraform keeps track of what resources are currently deployed and their configurations, usually in a state file.
- The current state is the actual state of a resource that is currently deployed.
- **Current state** refers to the present configuration of the infrastructure.

Terraform concludes that the **current state matches the desired state**.



I manually changed the instance type to `t2.nano`, so it is showing that the desired state is `t2.micro` and the current state is `t2.nano`. If we want to apply the desired state, we need to run `terraform apply` to update it.



Now, the instance type has been changed from `t2.nano` to `t2.micro`, as described by the **desired state**.

Terraform Variables :

- Terraform **variables are placeholders** used to **store values** that can be **reused** throughout the configuration, making it **easier to manage** and **update infrastructure settings** in **one place**.
- Terraform variables store values that can be referenced throughout our code. This makes our work easier, as we **only need** to **edit the variable values instead** of **changing** each occurrence in the **code**.

Create **Security Group** :

```
provider "aws" {
  region = "us-east-1"
  access_key = "****************FIM3"
  secret_key = "****************************d1s6"
}

resource "aws_security_group" "surey" {
  name        = "ALLOW-TCP-V4"
  description = "Allow TLS inbound traffic and all outbound traffic"

  ingress {
    from_port        = 80
    to_port          = 80
```

```
    protocol         = "tcp"
    cidr_blocks      = var.vpn_ip      // add the variable which holds the
value of the cidr_blocks
  }

  ingress {
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = var.vpn_ip        // add the variable which holds the
value of the cidr_blocks
  }

  ingress {
    from_port        = 22
    to_port          = 22
    protocol         = "tcp"
    cidr_blocks      = var.vpn_ip        // add the variable which holds the
value of the cidr_blocks
  }

  ingress {
    from_port        = 8080
    to_port          = 8080
    protocol         = "tcp"
    cidr_blocks      = var.vpn_ip        // add the variable which holds the
value of the cidr_blocks
  }

  tags = {
    Name = "Terraform_TCP"
  }
}
```

Variable for the `cidr_blocks` to be added to the Security Group :

```
variable "vpn_ip" {
    default = ["1.8.99.44/32"]
}
```

We need to specify the variable `vpn_ip` in the `cidr_block` for the Security Group, instead of adding individual IPs for all the `cidr_blocks`.

Multiple Approaches of variable assignment :

- Variable default
- Command line flag
- From a file
- Environment variable

1. Variable default :

```
resource "aws_instance" "cloud" {
    ami = "ami-0182f373e66f89c85"
    instance_type = var.instancetype
    key_name = "windows"

    tags = {
      name = "surendhar"
    }
}
```

```
variable "instancetype" {
    default = "t2.small"
}
```

I will define the `instancetype` variable in a separate file named `variables.tf` and set its default value. Then, I will reference this variable in the `main.tf` file as `var.instancetype`. This approach will streamline our configuration and improve efficiency.

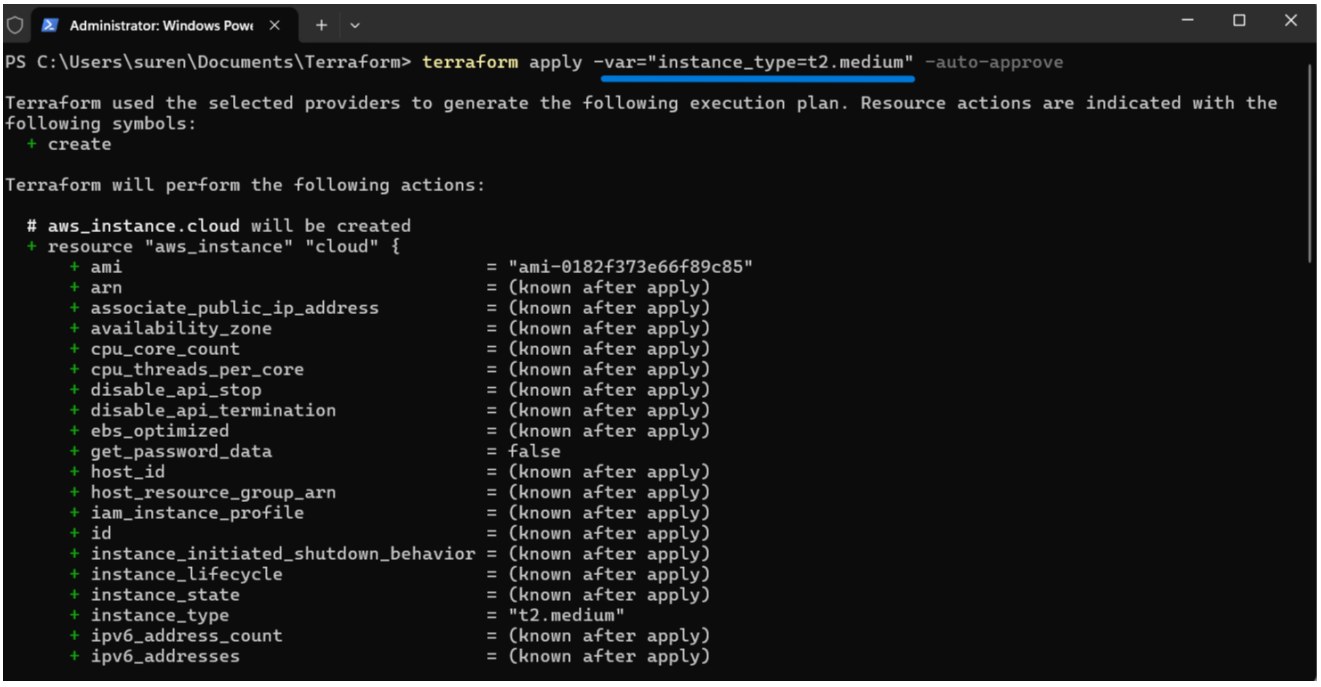
- Resource in `main.tf`
- Variable in `variable.tf`

2. Command line flag :

```
resource "aws_instance" "cloud" {
    ami = "ami-0182f373e66f89c85"
    instance_type = var.instancetype
    key_name = "windows"

    tags = {
      name = "surendhar"
    }
}
```

```
variable "instancetype" {
    default = "t2.small"
}
```



I use the `-var` flag in the CLI to change the `instance_type` to an explicit value. If a default value is already defined in the **variable file** and I provide a **new value** using the `-var` flag, the CLI value will take **precedence over the default value**.

- Resource in `main.tf`
- Variable in `variable.tf`
- PowerShell `terraform apply -var="instance_type=t2.medium" -auto-approve`

3. From a file :

To map values to variables in Terraform, define variables in `variables.tf`, assign values in `terraform.tfvars`, and reference them in `main.tf` to create resources dynamically based on the variable inputs.

- Resource in `main.tf` instance
- Variable in `variable.tf` variable
- Value in `terraform.tfvars` map values to variables.

Variable.tf file :

```
variable "instance_type" {
  description = "The type of instance to use"
```

```
  type        = string
  default     = "t2.micro"
}
```

terraform.tfvars file :

```
instance_type = "t2.micro"
```

If the file didn't work will use, this command :

```
terraform apply -var-file="terraform.tfvars"
```

4. Environment Variable

```
setx TF_VAR_instancetype t2.large
```

```
set MY_VARIABLE=some_value && terraform apply -
var="my_variable=%MY_VARIABLE%"
```

set the environment variable and execute the Terraform apply command.