Table of Contents

AWS Resource Creation

- Below are the AWS Resources that are created using AWS Console.

```
AWS:
- EC2 Service
    - EC2 instance is a resource
- S3
    - S3 Bucket is resource
- VPC
    - subnet, igw, route tables, nat, vpc endpoint
- IAM
```

```
    - IAM Role (sts trust relationship document), IAM Users, IAM Groups, IAM
Policy

- RDS
    - RDS Instance ( Instance Type, DB Type )
- Athena
- CloudWatch
- CloudTrail
- SNS
- SSM
- Lambda
```

IAC

- As of now, we have deployed many Resources in AWS, but manually (Using AWS Console).
- It becomes difficult to reproduce the same set of resources in:
    - In Different Region
    - In Different AWS Account
    - Multiple environments (dev/qa/prod) in multiple regions in multiple AWS account.
- Wouldn't it be great, if all our infrastructure was code?. That code would be deployed and ***create/update/delete*** our infrastructure.
- IAC is DevOps practice to treat infrastructure the same way as developers treat code.
- Here, IAC Scripts are definition of Infrastructure that could be on Cloud.
- Code is stored in a **Version Control System** that logs a history of code development, changes, and bug fixes.

Issues with configuring infrastructure manually

- In an ideal scenario, in cloud setups, there are three environments majorly categorized as **dev/qa/prod**.
- These Environments comprises of **AWS VPC, EC2, S3, IAM, RDS etc** resources that are used for any Application or Data Processing.
- A new feature change is developed in **dev** setup, tested in **qa** setup.
- Sometimes it happens that, some feature is working in one environment and does not worked on other.
- This could be due to that, resources in **dev** are configured differently than **qa and prod** as there are provisioned manually.
- Ideally all environments should be in sync. It is also a lot of manual work to constantly apply changes to each environment.
- Having environments with different infrastructure causes a number of issues such as you only find issues on a certain environment and make development hard as you are never testing against production like infrastructure.

--

- Configuring infrastructure manually is very error prone. If you want to try out a new infrastructure configuration then you have to make the change to an environment manually. If the change is what you want then you have to remember what steps are involved to make the change and then manually apply them to all of your other environments.
- If you do not want that change, then you have to remember how to roll the environment back to how it was.
- As the process is manual, often the changes are not made exactly the same to each environment which is one of the reasons that environments end up differing.
- It is very time consuming to make the changes manually. If you have several environments and the change is quite complex it can take days to roll that change to each environment.
- Once you have an environment, when you no longer need it, destroying all the resources manually again is a tedious task.
- For starters you have to destroy the infrastructure in the correct order as often you cannot destroy a piece of infrastructure if another piece depends on it.

Terraform to the rescue

- Terraform helps in **Creating, Updating, Destroying infrasturcture**.
- Terraform solves all of these problems because your infrastructure is defined in code.
- The code represents the state of your infrastructure. When your run Terraform against your code it will update your environment to be exactly how you have specified it in code.
- **Reproducible every time. All of your environments are identical.**
- A change is made to the code, then instantly you can get Terraform to update every environment simultaneously to include the new change.
- As your infrastructure is now defined in code, this can be also stored in **source control system**.
- This means that you can make a change to your code, roll it into an environment using Terraform and try it out.
- If the change is no good then you can simply go back to the previous version of the code and run Terraform again.
- Then Terraform handles putting the environment back to how it was.
- If the change is good you can roll it into all of your other environments.

--

Terraform

- `Terraform` is a tool for ***building,changing,& versioning infrastructure***. Terraform can manage services for multiple clouds.

- Terraform has become one of the popular **Infrastructure-as-code (IaC) tool**, getting used by DevOps team worldwide to automate infrastructure provisioning and management.

- Terraform is an open-source, cloud-agnostic provisioning tool developed by ***HashiCorp*** and written in ***GO language***.

- Terraform is actually split into two parts.

    - One part is the **Terraform engine** that knows how to get from the state your infrastructure is currently in to how you want your infrastructure to be.
    - The other part is the **provider** which is the part that talks to the infrastructure to find out the current state.

- Meaning you can use Terraform to configure infrastructure in **AWS, Azure, GCP, Oracle Cloud Platform and just about any other cloud**.

Terraform Installation

- Navigate to Terraform Downloads Page and Download the Package as per **Linux/Windows**
- For **Windows**:
    - Extract the **terraform.exe** file under a location like **"C:\terraform"**
    - Configure environment variables for terraform
        - Navigate to **Advanced System Settings** > **Environment variables**.
        - In system variables, select **path** and click on **edit** and **add path** of terraform executable i.e **"C:\terraform"**
- Execute the **terraform version** on ***CMD Prompt*** and ***Git Bash***:

```
C:\Users\Admin>terraform -version
Terraform v1.3.1
on windows_amd64
```

```
$ terraform -version
Terraform v1.3.1
on windows_amd64

$ which terraform
/c/terraform/terraform
```

--

- Just enter `terraform` to list all terraform commands

```
$ terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
```

```
validate      Check whether the configuration is valid
plan          Show changes required by the current configuration
apply         Create or update infrastructure
destroy       Destroy previously-created infrastructure
```

--

- For **Linux**:
  - Terraform Install CLI
  - For Amazon Linux 2:

```
# Install yum-config-manager to manage your repositories.
sudo yum install -y yum-utils
# Use yum-config-manager to add the official HashiCorp Linux
repository.
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
# sudo yum -y install terraform
sudo yum -y install terraform
[ec2-user@ip-172-31-4-240 ~]$ terraform version
Terraform v1.3.1
on linux_amd64
```

  - Enable tab completion:
    - If you use either **Bash** or Zsh, we can enable tab completion for Terraform commands. To enable autocomplete, first ensure that a config file exists for your chosen shell.

```
touch ~/.bashrc
terraform -install-autocomplete
```

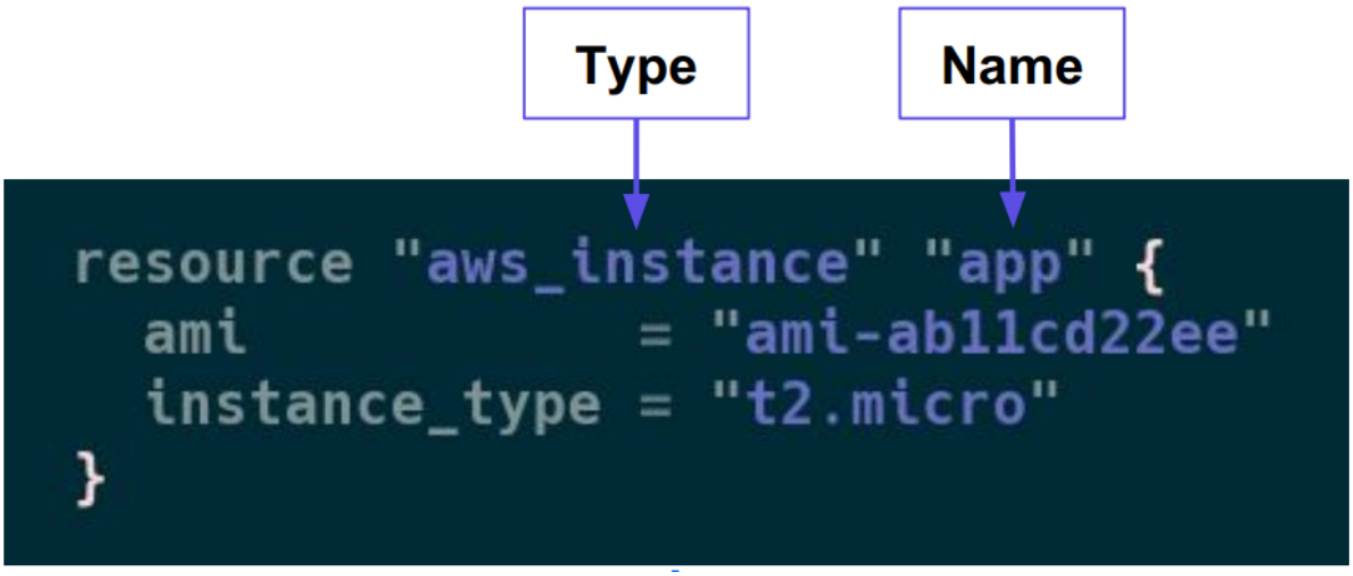
> When Running Terraform Commands from EC2 instance, attach IAM Role to this Instance, and terraform will use policies attached to this Role for Resource Creation/Updation/Deletion.

HCL Basics-Introduction to HCL

- HCL is **HashiCorp Configuration Language**.
- ***Key Elements of HCL***
  - HCL syntax comprises blocks that define a variety of configurations available to Terraform.
  - Blocks are comprised of **key = value** pairs.
  - ***Terraform accepts values of type string, number, boolean, map, and list.***
  - Single line comments start with **#**, while **multi-line comments** use an opening /* and a closing */.
  - An interpolated variable reference is constructed with the "${var.region}" syntax.This example references a variable named **region**, which is prefixed by **var**.

- The opening **${** and closing **}** indicates the start of interpolation syntax.
- **Strings** are wrapped in double quotes.
- **Lists** of primitive types (string, number, and boolean) are wrapped in square brackets: **["s3", "data", "athena", "event", "us-east-1"]**.
- **Maps** use curly braces **{}** and **colons :**, for example: **{ "password" : "my_password", "db_name" : "wordpress" }**.

--



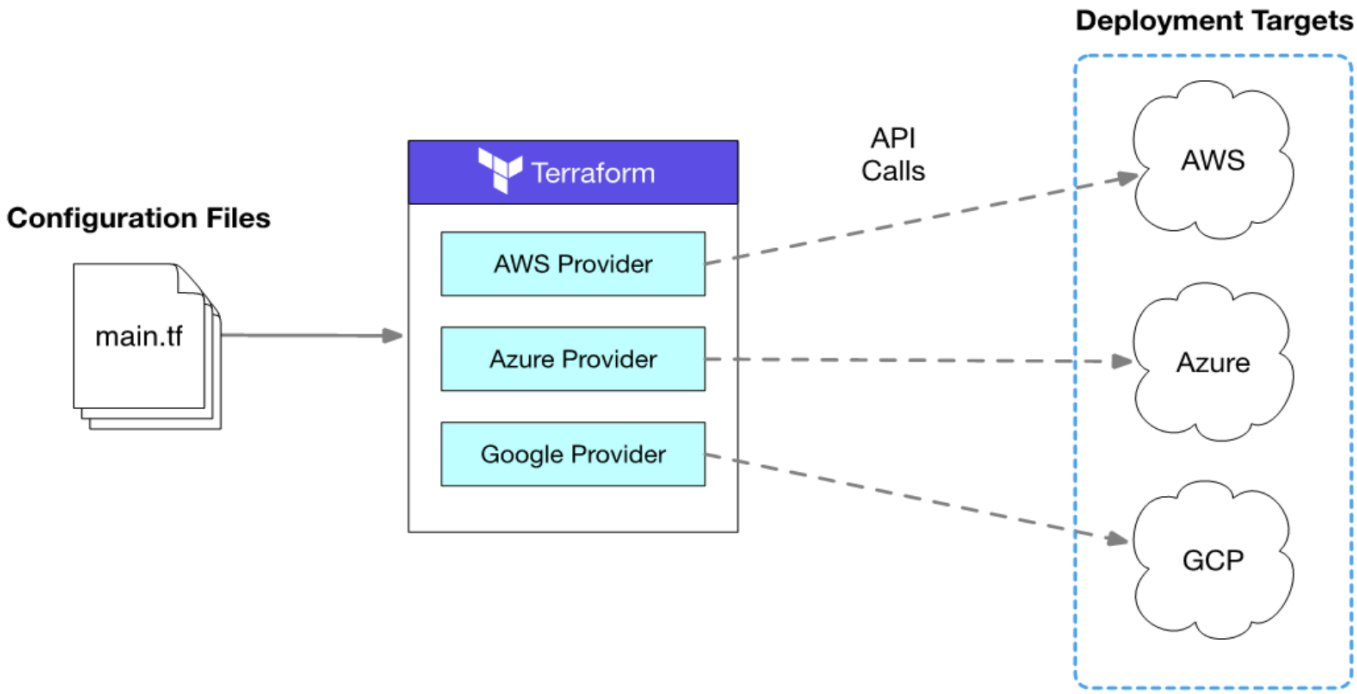
- Introduction-to-HCL

Provider

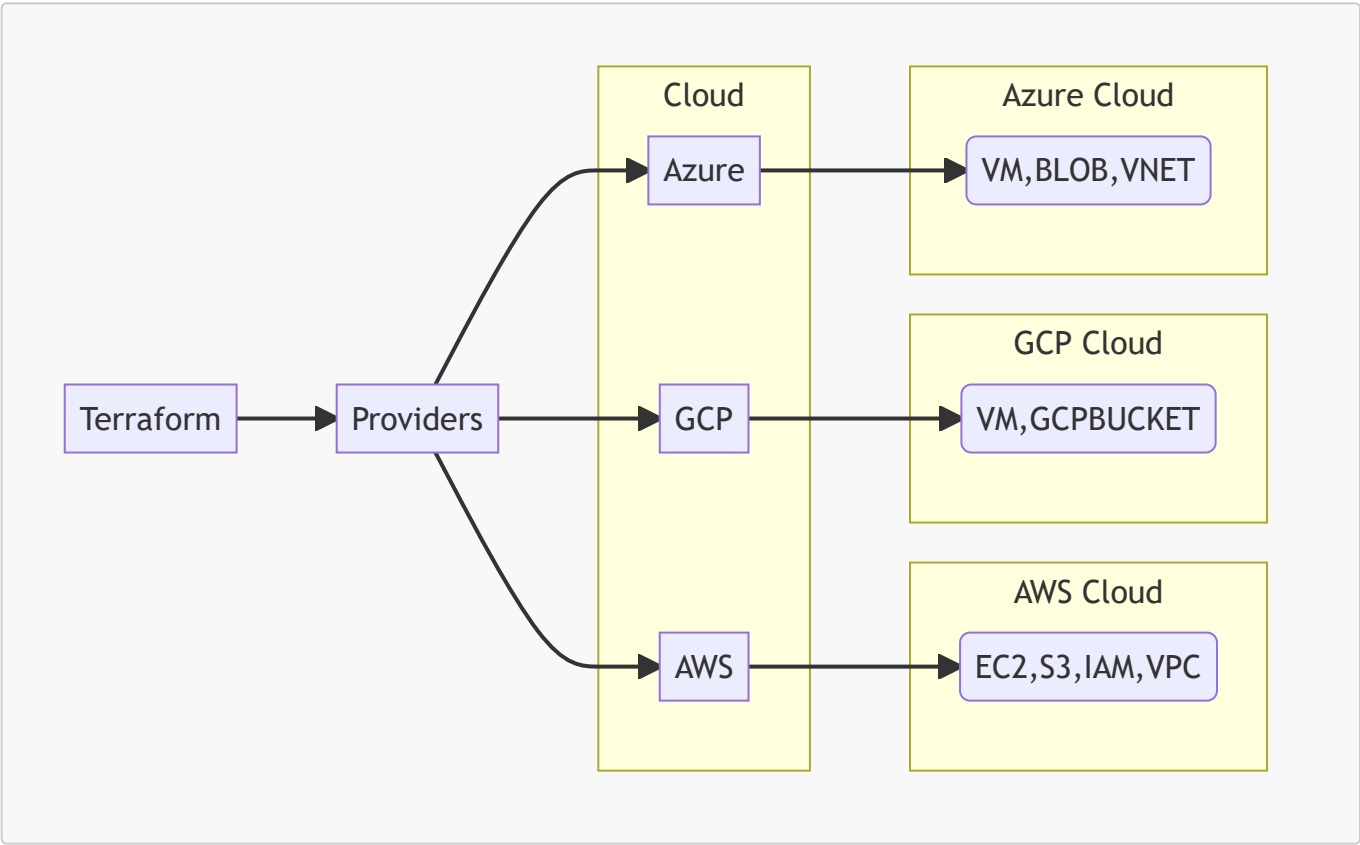
- A `provider` is responsible for understanding API interactions and exposing resources.
- Most of the available providers correspond to cloud infrastructure platform that offers resource types.
- Simply this could be a EC2 Service API interaction for launching/configuration of resource components in that service.

--

- **TF Providers**



--



- Provider Requirements
- Terraform AWS Provider

--

- To make a provider available on Terraform, execute ***terraform init***, these commands download any plugins we need for our providers and store in local folder as **.terraform**
- If you need to copy the plugin directory manually, we can do it, moving the files to **.terraform.d/plugins**

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

If the plugin is already installed, **terraform init** will not download again unless to upgrade the version, run ***terraform init -upgrade***.

--

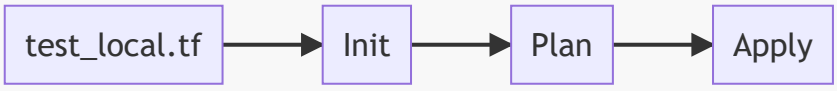
- Multiple Providers
    - There can be multiple configurations for the same provider, and there can be a logic to select which one to use on a **per-resource** or **per-module** basis.

```
#default configuration
provider "aws" {
  region = "us-east-1"
}
# reference this as `aws.west`.
provider "aws" {
  alias = "west"
  region = "us-west-2"
}
```

`test_local.tf`

```
resource "local_file" "countries" {
  filename = "/tmp/countries.txt"
  content = "India"
}
```

- Above code has below file components
    - ***Resource Type*** : `local_file`
    - ***Resource Name*** : `countries`
    - ***Arguments*** : Here `filename` and `content` are arguments
    - ***Block Name*** : `resource`



- For more details on **local_file** provider click here

--

aws_vpc.tf

```
resource "aws_vpc" "terraform_test_vpc" {
  cidr_block         = "172.31.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name = "terraform_test_vpc"
  }
}
```

- For more details on **aws_vpc** provider click here

aws_ec2.tf

```
#default configuration
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "ec2_web_server" {
  ami = "ami-0c2f66c1f66a1ff8d"
  instance_type = "t2.micro"
}
```

- For more details on **aws_instance** provider click here

--

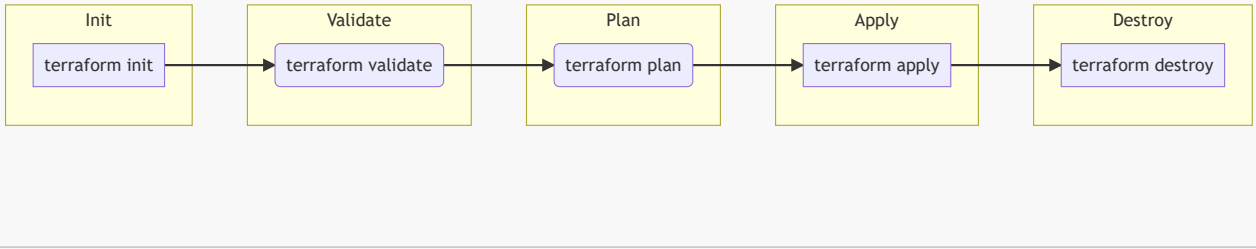
aws_s3.tf

```
#default configuration
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "my_data" {
  bucket = "webserver-bucket-bucket"
  acl = "private"
}
```

- For more details on **aws_s3_bucket** provider click here

Terraform Workflow



--

Terraform Init

- The `terraform init` command is used to initialize a working directory containing **Terraform configuration files**.
- This command will also perform backend initialization, storage for terraform state file, modules installation.
- During `init`, the configuration is searched for module blocks, & the source code for referenced modules is retrieved from the locations given in their source arguments.
- Terraform must initialize the ***provider*** before it can be used. This downloads and installs the provider's plugin so that it can later be executed. Initializes the backend configuration.
- terraform init

```
# initialize directory, pull down providers
terraform init

<<comment
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.36.1...
- Installed hashicorp/aws v4.36.1 (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.
comment

# Check for folder .terraform
ls -altR .terraform

# initialize directory, do not download plugins
terraform init -get-plugins=false
```

--

Terraform Plan

- The ***terraform plan*** command is used to create an `execution plan` by scanning all `*.tf` files in the directory.
- It will not modify things in infrastructure.
- Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the **desired state** specified in the configuration files.
- This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.

```
terraform plan

<<comment
| Error: error configuring Terraform AWS Provider: no valid credential sources for
Terraform AWS Provider found.

| Please see https://registry.terraform.io/providers/hashicorp/aws
| for more information about providing credentials.
|
| Error: failed to refresh cached credentials, no EC2 IMDS role found, operation
error ec2imds: GetMetadata, request send failed, Get
"http://169.254.169.254/latest/meta-data/iam/security-credentials/": dial tcp
169.254.169.254:80: connectex: A socket operation was attempted to an unreachable
network.
|
|   with provider["registry.terraform.io/hashicorp/aws"],
|   on ec2-instances.tf line 12, in provider "aws":
|   12: provider "aws" {
comment

# The word "comment" in bash script can be anything but it should be the same for
ending the comment block.
```

--

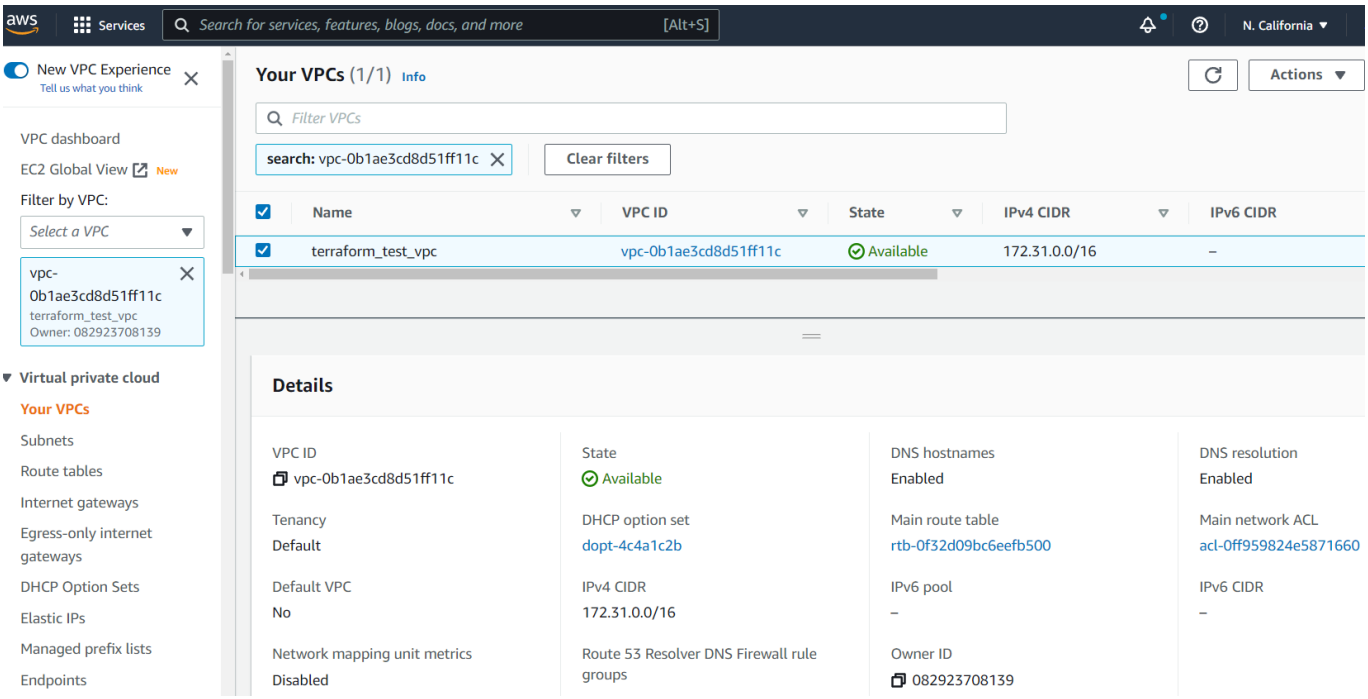
Terraform Apply

> When you run terraform apply, The Terraform will assume IAM Credentials configured

> In case of executing terraform commands on the EC2 instance, IAM Role attached to the EC2 instance should have policies attached for resource creation using Terraform.

- The ***terraform apply*** command is used to apply the changes required to reach the desired state of the configuration.
- Terraform apply will also write data to the terraform **.tfstate** file.
- Once the ***terraform apply*** is completed, resources are immediately available and can be verified in AWS Console.

--



--

Terraform State Files

- Terraform stores information about your infrastructure in a **state file**. This **state file** keeps track of resources created by your configuration and maps them to real-world resources.
- This state is stored by default in a local file named ***terraform.tfstate***, but it can also be stored remotely.
- This is how terraform is aware of the provisioned resources when you run **terraform plan** or the resource that need to be deleted.
- Terraform compares your configuration with the state file and your existing infrastructure to create plans and make changes to your infrastructure.
- When you run **terraform apply** or **terraform destroy** against your initialized configuration, Terraform writes metadata about your configuration to the state file and updates your infrastructure resources accordingly.
- You can see the state file in your terraform initialised directory:

--

- **Understanding Desired state and Current State**
    - The **desired state** is described in the Terraform Configuration files (the Manifests) and the function of terraform is to create modify and destroy infrastructure to match this state.

- The **current state** is the live state of the infrastructure which may not be the same as the desired state.

terraform.tfstate


```
{
  "version": 4,
  "terraform_version": "1.3.1",
  "serial": 2,
  "lineage": "1bcb0e05-3a0c-7474-4ad1-1685aa26e064",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_vpc",
      "name": "terraform_test_vpc",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "arn": "arn:aws:ec2:us-west-1:082923708139:vpc/vpc-0b1ae3cd8d51ff11c",
            "assign_generated_ipv6_cidr_block": false,
            "cidr_block": "172.31.0.0/16",
            "default_network_acl_id": "acl-0ff959824e5871660",
            "default_route_table_id": "rtb-0f32d09bc6eefb500",
            "default_security_group_id": "sg-05a9b93083dcab3ec",
            "dhcp_options_id": "dopt-4c4a1c2b",
            "enable_classiclink": false,
            "enable_classiclink_dns_support": false,
            "enable_dns_hostnames": true,
            "enable_dns_support": true,
            "enable_network_address_usage_metrics": false,
            "id": "vpc-0b1ae3cd8d51ff11c",
            "instance_tenancy": "default",
            "ipv4_ipam_pool_id": null,
            "ipv4_netmask_length": null,
            "ipv6_association_id": "",
            "ipv6_cidr_block": "",
            "ipv6_cidr_block_network_border_group": "",
            "ipv6_ipam_pool_id": "",
            "ipv6_netmask_length": 0,
            "main_route_table_id": "rtb-0f32d09bc6eefb500",
            "owner_id": "082923708139",
            "tags": {
              "Name": "terraform_test_vpc"
            },
            "tags_all": {
              "Name": "terraform_test_vpc"
            }
          },
          "sensitive_attributes": [],
```

```
            "private": "eyJzY2hlbWFfdmVyc2lvbiI6IjEifQ=="
          }
        ]
      }
    ],
    "check_results": []
}

# Show all resources and information
terraform show

terraoform state list
aws_vpc.terraform_test_vpc

terraoform state show aws_vpc.terraform_test_vpc
```

--

> You should not manually change information in your state file in a real-world situation ***to avoid unnecessary drift between your Terraform configuration, state, and infrastructure***.

> Any change in state could result in your infrastructure being destroyed and recreated at your next terraform apply.

> Do not manually modify state files.

Terraform Destroy

- The **terraform destroy** command is used to destroy the Terraform-managed infrastructure.
- **terraform destroy** command is not the only command through which infrastructure can be destroyed.
- You can remove the resource block from the configuration and run **terraform apply** this way you can destroy the infrastructure.

```
terraform destroy --auto-approve
# This command will take backup of terraform.tfstate as terraform.tfstate.backup.
# The existing "terraform.tfstate" file contains blank resources output as the
resources created are now deleted.
# Navigate to AWS Console and validate the VPC Created in previous step is now
deleted.
```

`terraform.tfstate`

```
{
  "version": 4,
  "terraform_version": "1.3.1",
```

```
  "serial": 4,
  "lineage": "1bcb0e05-3a0c-7474-4ad1-1685aa26e064",
  "outputs": {},
  "resources": [],
  "check_results": []
}
```

Terraform Remote State

- Terraform Local State file can be stored to Terraform Cloud.

Pre-requisite

- Sign In/Sign Up to Terraform Cloud Account : Sign in to Terraform Cloud
- Create a new organization and provide a name.
- Under this Organization, ***Create a new Workspace > CLI-Driven Workspace > Provide Workspace Name as test_tf_workspace***
- Once Workspace is created, an example block of code will be provided.
- Add this code in the below file.

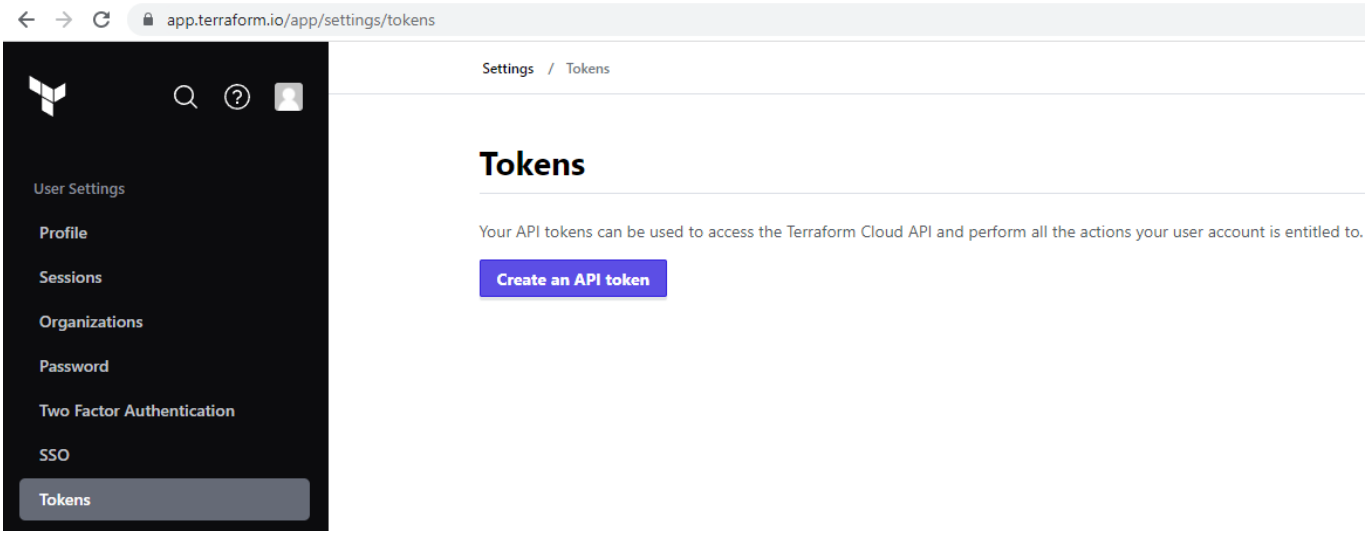
`backends.tf`

```
terraform {
  cloud {
    organization = "ORGNAME"

    workspaces {
      name = "test_tf_workspace"
    }
  }
}
```

- Under the Workspace Settings, modify the **Execution mode: Remote** to **Execution mode: Local** > Click on **Save Settings.**
- Navigate to **User Settings > Tokens > Create an API Token > Add Description > Create API Token**.

--



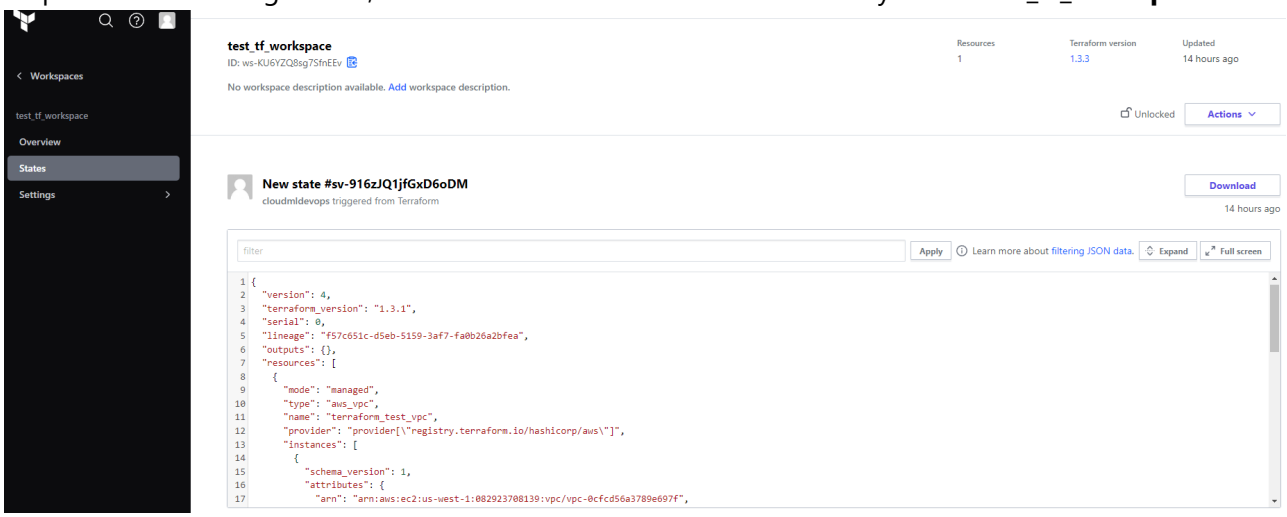
- Copy the token.
- Enter command ***terraform login***
    - Enter the token created in the previous step.
- Execute below commands once terraform login command is completed.

```
terraform init

terraform apply
# navigate to the state files that are getting uploaded
```

--

- As per backend configuration, the State File will now be stored remotely in the **test_tf_workspace**



--

Terraform Validate

- The **terraform validate** command validates the configuration files in a directory.

- This command check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.
- Validate runs checks that verify whether a configuration is syntactically valid, verifies reusable modules, also validates correctness of attribute names, value types.
- It can run before the ***terraform plan***.
- Validation requires an initialized working directory with any referenced plugins and modules installed.

```
# validate configuration files for syntax
terraform validate

# Success! The configuration is valid.

# validate code skip backend validation
terraform validate -backend=false
```

--

Terraform Refresh

- The ***terraform refresh*** command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.
- This does not modify infrastructure but does modify the state file.

Terraform Variables

- Variables play an important part in Terraform configuration when you want to manage infrastructure.
- Terraform resource creation can be parameterized with variables.
- Variables can have a **default** value.
- Terraform CLI defines the following optional arguments for variable declarations:
    - **default** - A default value which then makes the variable optional.
    - **type** - This argument specifies what value types are accepted for the variable.
    - **description** - This specifies the input variable's documentation.
- For more details on **variables** provider click here

--

Input Variable

- This Serve as parameters for a Terraform code, so users can enter values without editing the source.
- Each input variable accepted by a module must be declared using a **variable block** in a file:

`variables.tf`

```
```

```
variable "vpc_cidr" {
  type    = string
  description = "Enter the VPC CIDR Value."
}
```

- Within the module that declared a variable, its value can be accessed from within expressions as `var.<NAME>`, where `<NAME>` matches the label given in the declaration block i.e `var.vpc_cidr`

`main.tf`

```
resource "aws_vpc" "terraform_test_vpc" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name = "terraform_test_vpc"
  }
}
```

Passing Variables in Terraform

Specify variable via argument

- If only a few variables are defined or if existing values have to be explicitly overwritten, these can be specified as an argument of the corresponding Terraform command.
- Use ***-var"variable_name=variable_value"*** while executing the ***terraform commands***.

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -
var="instance_type=t2.micro"
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-
def456"}'
```

Variable with .tfvars file

- If a large number of variables have to be specified, it is more convenient to specify their values in a variable definitions file (with a filename ending in either ***.tfvars*** or ***.tfvars.json***) and then specify that file on the command line with ***-var-file***:

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

--

- A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable name assignments:

```
image_id = "ami-abc123"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c",
]
```

--

Auto tfvars files

- Terraform also automatically loads a number of variable definitions files where:
  - Files named exactly ***terraform.tfvars*** or ***terraform.tfvars.json***.
  - Any files with names ending in ***.auto.tfvars*** or ***.auto.tfvars.json***

Terraform Environment Variables

- Environment variables can be used to set variables. Terraform searches the environment of its own process for environment variables named **TF_VAR_name** followed by the name of a declared variable.
- The environment variables must be in the format ***TF_VAR_name*** and this will be checked last for a value.
- To use environment variables with Terraform they must have the ***TF_VAR*** prefix.

```
export TF_VAR_region=us-west-1
export TF_VAR_ami=ami-049d8641
export TF_VAR_alist='[1,2,3]'
export TF_VAR_amap='{ foo = "bar", baz = "qux" }'
```

--

- The variable definition for above environment variables passed will be as:

```
variable "region" {
    type        = string
    description = "region value is used as from environment variables."
}
variable "ami" {
    type        = string
    description = "ami value is used as from environment variables."
}
```

--

Variable Types

- The `type` argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable. If no type constraint is set then a value of any `type` is accepted.
- The supported `type` keywords are:
    - `string`
    - `Numbers`
    - `Boolean`
    - `list`
    - `map`

--

List

- **List** can store multiple values and the first value is the **0th index position**.
- For example, to access value of the item at the 0th position use : **var.test_list[0]**

```
variable "test_list" {
type = list(string)
default = ["Value1", "Value2"]
}
```

Map

- `Map` is a Key-Value pair. Key is needed to access the value.

```
variable "ami_ids" {
  type = map
  default = {
  "eu-west-1" = "ami-0713f98de93617bb4"
  "us-east-1" = " ami-062f7200baf2fa504"
  "us-east-2" = "ami-07a0844029df33d7d"
  }
}
```

- Use **var.ami_ids["us-east-1"]** to fetch the corresponding value.

--

Variable Definition Precedence

- Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

1. Environment variables
2. The ***terraform.tfvars*** file, if present.
3. The ***terraform.tfvars.json*** file, if present.

4. Any ***.auto.tfvars*** or ***.auto.tfvars.json*** files, processed in lexical order of their filenames.
5. Any ***-var*** and ***-var-file*** options on the command line, in the order they are provided.