Terraform

What is Terraform?

Terraform is an Infrastructure as Code (IaC) tool created by HashiCorp. It lets you define cloud and on-prem infrastructure using declarative configuration files.

Terraform Installation:

Visit official installation documentation and follow the steps.

https://developer.hashicorp.com/terraform/install

Basic Components of Terraform:

1. **Providers:**
    a. Providers are plugins that allow Terraform to interact with external APIs — like AWS, Azure, Google Cloud, or even DNS providers.
    b. They define what kind of infrastructure you can create/manage and how Terraform talks to those services.

    Example:

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

    c. This tells Terraform: "I want to create/manage AWS resources in us-east-1."

2. **Resources:**
    a. Resources are the actual infrastructure components you want to create/manage — EC2 instances, S3 buckets, VPCs, databases, etc.
    b. Resources define what infrastructure you want.

    Example:

```
resource "aws_instance" "my_server" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

General Folder Structure:

```
terraform-ec2/
│
├── main.tf          # Main config
├── variables.tf     # Input variables
├── outputs.tf       # Output values
├── terraform.tfvars # Values for variables
├──modules           # Folder for reusable resource files
   ├── ec2           # resource file
   ├── vpc
```

1. main.tf:
    a. The primary Terraform configuration file where you define your provider, resources, modules, and any other configs.
    b. It's often the starting point for a Terraform project, main entry for your infrastructure definitions.
2. variables.tf:

a. File where you declare variables that your Terraform configuration will use.
b. Variables allow you to parameterize your configs for flexibility.

Example:

```
variable "region" {
  description = "AWS region to deploy"
  type        = string
  default     = "us-east-1"
}

variable "instance_type" {
  description = "EC2 instance type"
  type        = string
}
```

3. outputs.tf:
   a. File where you define outputs — values Terraform will print/display after apply.
   b. Outputs let you easily extract useful info like instance IPs, resource IDs, or URLs.

Example:

```
output "instance_id" {
  value = aws_instance.my_server.id
}

output "instance_public_ip" {
  value = aws_instance.my_server.public_ip
}
```

4. terraform.tfvars:
    a. A file to store input variable values.
    b. To keep variables (like region, instance type, AMI ID) separate from the code, making your configs reusable and easier to maintain.

    Example:

    ```
    region = "us-east-1"
    instance_type = "t2.micro"
    ami = "ami-12345678"
    ```

5. Modules:
    a. Modules are containers for multiple resources that are used together. Think of them like reusable building blocks or templates.
    b. They help you organize code, re-use infrastructure patterns, and keep your configurations clean.

    Example:

    Instead of writing all resources inline, you can create a vpc module and call it in your main config.

    ```
    module "vpc" {
      source = "./modules/vpc"
      cidr_block = "10.0.0.0/16"
    }
    ```

    The module folder "./modules/vpc" contains a set of resources related to a VPC.

6. terraform.tfstate:
    a. A file (usually terraform.tfstate) that keeps track of the current state of your infrastructure managed by Terraform.

b. Terraform uses this state file to know what exists, what changed, and what to update. Without it, Terraform would not know what infrastructure it controls.
c. **Important**:
   - The state file contains resource IDs, metadata, and other info.
   - Should be protected (don't commit to public repos).
   - Can be stored remotely (S3, Terraform Cloud) for team collaboration.

Terraform commands:

Command	Description
terraform init	Initializes a Terraform working directory: downloads provider plugins and sets up backend.
terraform validate	Validates the Terraform configuration files for syntax errors and correctness.
terraform fmt	Formats Terraform code to a canonical style (auto-indent, spacing, etc.).
terraform plan	Creates an execution plan, showing what changes Terraform will make without applying them.
terraform apply	Applies the changes required to reach the desired state of the configuration (creates/updates).

Command	Description
terraform destroy	Destroys all resources managed by the current Terraform configuration (tears down infrastructure).
terraform show	Shows the current state or a saved plan in a human-readable format.
terraform output	Displays the output values defined in your configuration (useful for getting instance IPs, etc).
terraform state	Commands to interact with the Terraform state file (list, show, rm, mv resources).
terraform import	Imports existing infrastructure into your Terraform state to manage it going forward.
terraform workspace	Manages multiple workspaces (like different environments — dev, prod).

As terraform is creating infrastructures using AWS or any other cloud service, it needs credentials to use our account and there are multiple ways to do so. For AWS services we need “access key” and “secret key” which can be generated from AWS Console.

How to provide credentials to Terraform AWS provider?

A. Environment Variables (Recommended)

Set credentials in your shell environment:

```
export AWS_ACCESS_KEY_ID="your-access-key-id"
export AWS_SECRET_ACCESS_KEY="your-secret-access-key"
```

```
export AWS_SESSION_TOKEN="your-session-token"
# only if using temporary credentials
```

Terraform automatically picks these up when running terraform apply or terraform plan.

B. Shared Credentials File (~/.aws/credentials)

If you use AWS CLI, you probably have credentials stored here:

```
[default]
aws_access_key_id = your-access-key-id
aws_secret_access_key = your-secret-access-key
```

Terraform can use this file automatically if no explicit credentials are provided.

C. Explicitly in Terraform config (not recommended for security reasons)

You can specify credentials directly in the provider block, but **this is not recommended** because you risk leaking secrets:

```
provider "aws" {
  region     = "us-east-1"
  access_key = "your-access-key-id"
  secret_key = "your-secret-access-key"
}
```

D. Assume Role

If you want to assume a role (e.g., in cross-account scenarios):

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

```
    role_arn = "arn:aws:iam::123456789012:role/myrole"
  }
}
```

Note: Terraform resource or provider blocks can be copied from terraform registry and used directly with customization.

Practical: Launch an AWS ec2 instance in ap-south-1 region

We will not use credentials explicitly so add access key and secret key to environment variables or command palette if using VS Code.

I am using VS Code for the practical.

Steps:

- Check if terraform is installed.
  terraform -v
- Create a folder structure like:

```
terraform-ec2/
├── main.tf
├── variables.tf
├── terraform.tfvars
├── outputs.tf
```

- Open main.tf and enter provider block:

```
provider "aws" {
  region = var.region
}
```

- Resource block to create an ec2 instance:

```
resource "aws_instance" "my_ec2" {
  ami           = var.ami
  instance_type = var.instance_type
  key_name      = aws_key_pair.terraform_key.key_name

  tags = {
    Name = "TerraformEC2"
  }
}
```

- Generate a new SSH key pair locally:

```
resource "tls_private_key" "generated_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}
```

- Create an AWS key pair using the public key:

```
resource "aws_key_pair" "terraform_key" {
  key_name   = "terraform-key"  # This will be created in AWS
  public_key =tls_private_key.generated_key.public_key_openssh
}
```

- Save the private key to a local file (so you can SSH into the instance)

```
resource "local_file" "private_key_pem" {
  content = tls_private_key.generated_key.private_key_pem
  filename             = "terraform-key.pem"
  file_permission      = "0400"
```

```
}
```

- Save main.tf and open variables.tf to mention the variables.
- Enter variables.

```
variable "region" {
  description = "AWS region to deploy resources in"
  type        = string
  default     = "ap-south-1"
}

variable "ami" {
  description = "AMI ID for the EC2 instance"
  type        = string
}

variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"
}
```

- Save it and open terraform.tfvars.
- Enter ami-id for A. Linux or ubuntu instance from AMI Catalog of ap-south-1 region and keep instance type as t2.micro to use free-tier resources.

```
ami           = "ami-03bb6d83c60fc5f7c"
instance_type = "t2.micro"
```

- Save it and open outputs.tf, it will have blocks to show instance id and public ip address on successful creation.

```
output "instance_public_ip" {
  description = "Public IP of the EC2 instance"
  value       = aws_instance.my_ec2.public_ip
}

output "instance_id" {
  description = "ID of the created EC2 instance"
  value       = aws_instance.my_ec2.id
}
```

Before applying terraform commands lets breakdown the file contents to know what's happening:

1. Resource block:

a. ***`resource "aws_instance" "my_ec2" {`***

- "aws_instance" is the Terraform resource type. It represents an EC2 instance in AWS.
- "my_ec2" is the Terraform local name for this resource.
- It's how you refer to this EC2 instance elsewhere in your code.

Example: `aws_instance.my_ec2.public_ip` to get the public IP.

- Inside the Block:

b. ***`ami           = var.ami`***

- AMI ID used to boot the instance.
- This comes from a variable so you can reuse the config with different images.

c. ***`instance_type = var.instance_type`***

- EC2 instance size like t2.micro, t3.medium, etc.

d. `key_name = aws_key_pair.terraform_key.key_name`

- Tells EC2 what key pair to use for SSH.
- Refers to the aws_key_pair resource we created (terraform_key).

e.
```
tags = {
  Name = "TerraformEC2"
}
```

- Tag helps identify the instance in the AWS Console.

✓ ***my_ec2, generated_key, terraform_key etc. are all local variables declared for resources created so that they can be referenced in other blocks.***

✓ ***Every time you define a resource block, this local name needs to be mentioned.***

2. Variable:

➢ In variable block we mention variable block with word "variable" and name of the variable in quotes.

➢ The details of variable are written in "{}" and the attributes declared are type of variable such as string, int, etc, description of the variable and if needed then default value too.

➢ If any value is not mentioned for the variable in tfvars file then this default value is used.

3. tfvars:

➢ As mentioned earlier the values of type mentioned in variable are declared here. For example, we have put both ami ID and instance type in string format.

4. Output:

- In this block we are defining the output we want with the name we want it to be shown.
- Value attribute parses the values using the default resource block names and local names of them.
- Using this an output appears on the console once the infrastructure is created.

Let's use terraform to create the infrastructure from scratch:

1. If using VS code open terminal with path of the folder that has these files and in Linux terminal go to that folder.

```
PS D:\Terraform_Practicals> cd .\practical_01\
```

2. Run **`terraform init`** to initialize our formation. It will create a hidden folder ".terraform" and a file ".terraform.lock.hcl" which is created by HashiCorp and we are not supposed to do anything with it.

```
PS D:\Terraform_Practicals\practical_01> terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Finding latest version of hashicorp/tls...
- Finding latest version of hashicorp/local...
- Installing hashicorp/aws v5.98.0...
- Installed hashicorp/aws v5.98.0 (signed by HashiCorp)
- Installing hashicorp/tls v4.1.0...
- Installed hashicorp/tls v4.1.0 (signed by HashiCorp)
- Installing hashicorp/local v2.5.3...
- Installed hashicorp/local v2.5.3 (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.
PS D:\Terraform_Practicals\practical_01>
```

3. Run **`terraform fmt`** to format the documents in correct format (not needed every time).
4. Run terraform validate to check for validation of our configuration files if they are correctly configured then "Success" message will appear.

```
PS D:\Terraform_Practicals\practical_01> terraform validate
Success! The configuration is valid.
```

5. Run **`terraform plan`** and it will show the creation plan of the infrastructure.

```
PS D:\Terraform_Practicals\practical_01> 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.my_ec2 will be created
  + resource "aws_instance" "my_ec2" {
      + ami                                  = "ami-0af9569868786b23a"
```

6. Run **`terraform apply`** to create the infrastructure. If we use only **`terraform apply`** then it will prompt to enter a value as "yes" to proceed and to avoid that we can use **`terraform apply -auto-approve`**.

```
PS D:\Terraform_Practicals\practical_01> 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.my_ec2 will be created
  + resource "aws_instance" "my_ec2" {
      + ami                                  = "ami-0af9569868786b23a"
```

```
Plan: 4 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_id        = (known after apply)
  + instance_public_ip = (known after apply)

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
```

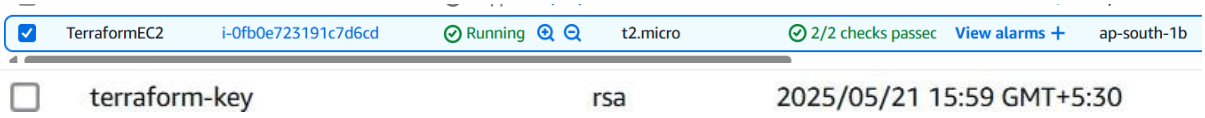
```
tls_private_key.generated_key: Creating...
tls_private_key.generated_key: Creation complete after 1s [id=391600c952b7fd0d5b516938f64ff6146c49f31e]
local_file.private_key_pem: Creating...
local_file.private_key_pem: Creation complete after 0s [id=72ea51ef406bcbfeaaab5e1747c277c1fcb0534c]
aws_key_pair.terraform_key: Creating...
aws_key_pair.terraform_key: Creation complete after 1s [id=terraform-key]
aws_instance.my_ec2: Creating...
aws_instance.my_ec2: Still creating... [10s elapsed]
aws_instance.my_ec2: Still creating... [20s elapsed]
aws_instance.my_ec2: Creation complete after 21s [id=i-0fb0e723191c7d6cd]

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

instance_id = "i-0fb0e723191c7d6cd"
instance_public_ip = "35.154.30.2"
PS D:\Terraform_Practicals\practical_01>
```

- As can be seen the creation plan is shown again, it shows "4 to add" then prompts to enter "yes" and at the end "Apply complete" with "4 resources added".
- The output is also visible as "instance id" and "Its public IP".
- The resources like key-pair and instance can be seen on AWS console.



- We asked to save the key pair locally so it is also saved in the same directory with the same name "terraform-key".
- The other important thing is "terraform.tfstate" file is also created and if we run **`terraform apply`** again the it will check this file and if there are no changes then it will just refresh the resources.

```
PS D:\Terraform_Practicals\practical_01> terraform apply --auto-approve
tls_private_key.generated_key: Refreshing state... [id=391600c952b7fd0d5b516938f64ff6146c49f31e]
local_file.private_key_pem: Refreshing state... [id=72ea51ef406bcbfeaaab5e1747c277c1fcb0534c]
aws_key_pair.terraform_key: Refreshing state... [id=terraform-key]
aws_instance.my_ec2: Refreshing state... [id=i-0fb0e723191c7d6cd]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

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

Outputs:

instance_id = "i-0fb0e723191c7d6cd"
instance_public_ip = "35.154.30.2"
```

- Let's destroy these resources using **`terraform destroy -auto-approve`**.

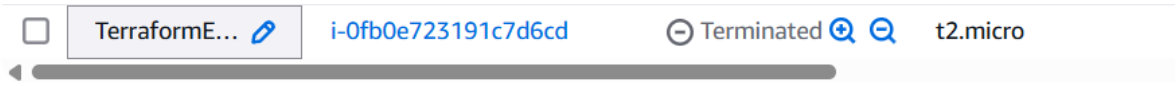
- Plan will show “4 to destroy”, all those will be destroyed and “Destroy Complete” message will be displayed.

```
Plan: 0 to add, 0 to change, 4 to destroy.

Changes to Outputs:
  - instance_id        = "i-0fb0e723191c7d6cd" -> null
  - instance_public_ip = "35.154.30.2" -> null
local_file.private_key_pem: Destroying... [id=72ea51ef406bcbfeaaab5e1747c277c1fcb0534c]
local_file.private_key_pem: Destruction complete after 0s
aws_instance.my_ec2: Destroying... [id=i-0fb0e723191c7d6cd]
aws_instance.my_ec2: Still destroying... [id=i-0fb0e723191c7d6cd, 10s elapsed]
aws_instance.my_ec2: Still destroying... [id=i-0fb0e723191c7d6cd, 20s elapsed]
aws_instance.my_ec2: Still destroying... [id=i-0fb0e723191c7d6cd, 30s elapsed]
aws_instance.my_ec2: Destruction complete after 30s
aws_key_pair.terraform_key: Destroying... [id=terraform-key]
aws_key_pair.terraform_key: Destruction complete after 0s
tls_private_key.generated_key: Destroying... [id=391600c952b7fd0d5b516938f64ff6146c49f31e]
tls_private_key.generated_key: Destruction complete after 0s

Destroy complete! Resources: 4 destroyed.
PS D:\Terraform_Practicals\practical_01>
```

- Check AWS Console of EC2 instance and that instance will be terminated.



- Similarly, the key pair will disappear from console as well as local folder.
- Another important change that can be seen is, the “terraform.tfstate” file does not have the content that was there when apply was done but that information is saved in “terraform.tfstate.backup” file.

In this way, we covered basics of Terraform and also performed a simple practical to create an instance, key-pair for that and saved it to the local folder.

Terraform is a very powerful tool for Infrastructure creation and its cloud version is used to maintain the version control and avoid explicit usage of credentials in local machine.

♣♣♣