Terraform

❓ Terraform is an `IaaC` tool that can define & provide Infrastructure configuration using a declarative configuration language known as HashiCorp Configuration Language, or optionally JSON.

▼ Terraform State Files

📌 Terraform is an open-source infrastructure as a code software tool that allows users to define and provision infrastructure using a simple, human-readable language.

- In TF, a state file is a file that stores information about the current state of your infrastructure. It keeps track of the resources that Terraform is managing, as well as their current properties and settings.
- The state file is used by Terraform to determine the changes that need to be made to the infrastructure in order to reach the desired state specified in the Terraform configuration files.
- The state file is also used to store metadata about the resources that Terraform is managing, such as the ID of a resource, etc.., that was created by Terraform.

- When you run Terraform commands such as `terraform plan`, `terraform apply`, or `terraform destroy`, Terraform will read and update the state file to reflect the changes that are being made to the infrastructure.
- The terraform state file, by default, is named `terraform.tfstate` and is held in the same directory where Terraform is run **&** It is created after running `terraform apply` command.
- It's important that Terraform state files should be kept safe and be treated as sensitive information. It should be kept in a secure place like a private S3 bucket or a vault, and should be versioned along with your terraform code.

▼ Files Creation

- When you run Terraform commands especially `terraform init`, the state files that are created include:

1. `.terraform` : This directory contains the plugins and configuration files that Terraform uses to interact with the various providers (e.g. AWS, Azure, GCP) and modules.
2. `terraform.tfstate`: This file contains the current state of the infrastructure managed by Terraform. It includes information about the resources that Terraform is managing, such as their type, ID, and current properties.
3. `terraform.tfstate.backup` : This file is a backup of the previous state file, created before Terraform makes changes to the infrastructure. It can be used to roll back to a previous state in case of errors.
4. `terraform.tfstate.lock` : This file is used to lock the state file to prevent multiple Terraform processes from modifying it at the same time. It contains information about the process that currently holds the lock, such as its process ID and hostname.
5. `.tf` : These are the Terraform configuration files that define the infrastructure you want to create.
6. `modules/`: This directory contains the state files for any modules that are used in the configuration. Each module has its own directory with a `terraform.tfstate` file that contains the state for that specific module.

- Please note that these are the files that are typically created by the `terraform init` command. Depending on the configuration, additional files and directories may also be created.

▼ Configure Access & Secret Keys in CLI

- we can configure ACCESS & SECRET key using CLI this helps to avoid hardcoding these credentials in the terraform scripts.
- for that, we need AWS CLI, we can install from official docs.

```
aws configure
```

▼ Terraform Commands

- `terraform init`: Initializes a Terraform working directory. This command is typically run before any other Terraform commands and is used to prepare a directory for use with Terraform.
- `terraform validate`: Validates the Terraform configuration files. This command is used to check the syntax and basic correctness of the Terraform.
- `terraform plan`: Generates and shows an execution plan. This command is used to see what changes Terraform will make before making them. Example: **`terraform plan -out=tfplan`**.
- `terraform apply`: Applies the changes required to reach the desired state of the configuration. This command creates or modifies resources in the infrastructure. Example: **`terraform apply tfplan`**.
- `terraform apply --auto-approve`: approving “yes” at the time of applying.
- `terraform destroy`: Destroys Terraform-managed infrastructure. This command is used to remove resources that were created by Terraform. Example: **`terraform destroy -auto-approve`**.
- `terraform state list`: Lists all resources that Terraform is managing. This command is used to get a list of all the resources that Terraform is currently managing. Example: **`terraform state list`**.
- `terraform show`: Shows the details of a specific resource in the Terraform state file. This command is used to view the details of a specific resource that Terraform is managing. Example: **`terraform state show aws_instance.example`**
- **`terraform import`**: Imports an existing resource into the Terraform state. This command is used to import resources that were created outside of Terraform into the Terraform state file. Example: **`terraform import aws_vpc.my_vpc vpc-12345678`**

- `terraform state pull`: Outputs the current state of a Terraform project as a JSON file. This command is used to export the current state of a Terraform project to a file for backup or for use with other tools. Example: `terraform state pull > state.tf`
- `terraform state mv`: Move a resource in the state file to a new address. This command is used to change the name or address of a resource in the Terraform state file without creating a new resource. Example: `terraform state mv aws_instance.example aws_instance.example_new`.
- `terraform fmt`: Formats the Terraform configuration files. This command is used to format and clean up the Terraform configuration files, making them easier to read and understand. Example: `terraform fmt`
- `terraform taint`: Marks a specific resource for destruction and recreation. This command is used to mark a specific resource for destruction and recreation on the next `terraform apply`. Example: `terraform taint aws_instance.example`
- `terraform graph`: Generates a visual representation of the Terraform resources. This command is used to generate a visual representation of the Terraform resources and their dependencies, which can be helpful in understanding complex configurations. Example: `terraform graph | dot -Tpng > graph.png`
- `terraform workspace`: Manages Terraform workspaces. This command is used to create, switch between, and manage different workspaces in a Terraform project. Example: `terraform workspace select dev`
- `terraform output`: Reads an output from a state file. This command is used to read the value of an output variable from the Terraform state file, which can be useful for passing data between Terraform modules. Example: `terraform output public_ip`
- `terraform providers`: Shows the providers used in the configuration. This command is used to list the providers used in the Terraform configuration and their versions. Example: `terraform providers`
- `terraform debug`: Allows to debug a specific resource. This command is used to run a specific resource through the Terraform interpreter in order to diagnose issues or print internal values. Example: `terraform debug resource aws_instance.example`

▼ Terraform Graph

- install **Graphviz** in your server, here are some examples for reference but kindly install from official docs only.

```
choco install graphviz

or

winget install graphviz

or

sudo apt install graphviz
```

```
terraform graph | dot -Tpng > graph.png

or

terraform graph -type=plan | dot -Tpng > graph.png

or

terraform graph -type=plan | dot -Tpng -o graph.png
```

▼ Importing Existing Infrastructure into Terraform

▼ Create main.tf file with Terraform provider AWS.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region     = "us-west-2"
  access_key = "[ACCESS KEY]"
  secret_key = "[SECRET KEY]"
}
```

- `terraform init` to initialize state files with template.

▼ Now provide few attributes (may vary based on requirements).

```
resource "aws_instance" "TomcatServer" {
 ami           = "unknown"
 instance_type = "unknown"
}
```

▼ After adding above piece of code, the whole template will look like this.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region     = "us-west-2"
  access_key = "[ACCESS KEY]"
  secret_key = "[SECRET KEY]"
}

resource "aws_instance" "TomcatServer" {
 ami           = "unknown"
 instance_type = "unknown"
}
```

▼ We will import our existing Infrastructure

```
terraform import aws_instance.myvm <Instance ID>
```

- running this `terraform plan` command will create a **terraform.tfstate** file where we can observe the meta data or all the configuration details of our existing Infrastructure server.
- try running `terraform plan` again will show a replacement in the terminal but don't go ahead with the further process this will replace the existing infrastructure with template configuration since there is no server's config updated in the template.
- but before running `terraform plan` command try changes **ami = "unknown" —> ami = "AMI ID from terraform.tfstate file"**, now running `terraform plan` will not replace the Infrastructure instead shows to update with other attributes.

▼ Reference links

- https://letslearndevops.com/2018/08/23/terraform-get-latest-centos-ami/
- https://www.terraform.io/language/data-sources
- https://registry.terraform.io/modules/terraform-aws-modules/s3-bucket/aws/latest
- https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform