



What is Terraform?

Terraform by HashiCorp is a popular IaC tool that uses a **declarative language** (HashiCorp Configuration Language, HCL) to manage infrastructure. It allows provisioning, scaling, and version-controlling infrastructure across cloud providers and on-premises systems.

Why is Terraform widely used?

- **Multi-cloud support:** AWS, Azure, GCP, and others.
- **State tracking:** Tracks deployed infrastructure via .tfstate files, enabling updates and drift detection.
- **Idempotence:** Ensures running the same script multiple times yields the same results.
- **Extensibility:** Modular design supports reusable components.

How Terraform Works

1. **Define:** Write resource definitions in .tf files.
 - Example: Define an AWS EC2 instance with its type, name, and AMI.
2. **Initialize:** Use terraform init to download required provider plugins.
3. **Plan:** Run terraform plan to preview the changes Terraform will make to match the desired state.
4. **Apply:** Execute the plan using terraform apply to provision or update resources.
5. **State Management:** Terraform tracks resource states in a .tfstate file for future reconciliation.

What is IaC?

Infrastructure as Code (IaC) is a practice in DevOps where infrastructure (servers, networks, storage, etc.) is managed and provisioned using code instead of manual processes. It ensures consistency, scalability, and repeatability by treating infrastructure configurations as version-controlled files.

Types of IaC:

1. **Declarative (Desired State):**

Specifies **what** the infrastructure should look like. The tool figures out **how** to achieve it.

Example: Terraform, CloudFormation.

- Example: Define an EC2 instance with a specific configuration, and Terraform ensures it's created exactly as described.

2. **Imperative (Step-by-Step):**

Specifies **how** to build and manage infrastructure with detailed commands. Example:

Ansible, Chef.

- Example: Run commands to install software and configure servers sequentially.

3. **Mutable vs. Immutable:**

- **Mutable:** Infrastructure is updated in place (e.g., Ansible).
- **Immutable:** Infrastructure is replaced entirely for updates (e.g., Packer with Terraform).
- Use immutable for better consistency and less drift.

Terraform Core Components (Detailed)

1. **Providers:**

- Plugins that interact with cloud APIs (e.g., AWS, Azure, Kubernetes).
- Examples: aws, google, azurearm.

2. **Resources:**

- Basic building blocks representing infrastructure (e.g., an EC2 instance, S3 bucket).
- Example: hcl

```
resource "aws_instance" "example" {  
  ami = "ami-123456"  
  instance_type = "t2.micro" }
```

3. State:

- Terraform keeps a **state file (terraform.tfstate)** that tracks resources under its control.
- Enables:
 - Detecting drift between the real-world state and defined state.
 - Incremental updates to infrastructure.

4. Modules:

- Logical groupings of resources that can be reused.
- Example: A VPC module with subnets, routes, and gateways.

5. Variables:

- Input variables (variables.tf): Parameterize configurations for flexibility.
- Output variables (outputs.tf): Pass data between modules or display key information.

Terraform Directory Structure (Detailed)

A typical Terraform project setup:

project/

```
|— main.tf      # Primary configuration file with resource definitions.
|— variables.tf # Input variable definitions and types.
|— outputs.tf   # Outputs to display after `terraform apply`.
|— provider.tf  # Cloud provider credentials and configuration.
|— terraform.tfvars # Variable values for the `variables.tf`.
└— terraform.tfstate # State file generated after applying changes.
```

what are plugins

In Terraform, **plugins** are essential components that extend its functionality by enabling interaction with various providers (e.g., AWS, Azure) or resources (e.g., EC2, S3 buckets). They handle the communication between Terraform and the external APIs of cloud providers, SaaS platforms, or other systems.

Key Concepts of Plugins

1. Types of Plugins:

- **Provider Plugins:** These manage access to specific services (e.g., AWS, GCP, Azure).
- **Provisioner Plugins:** Handle actions on resources after they are created (e.g., running scripts with local-exec or remote-exec).
- **Custom Plugins:** Written by users to interact with custom APIs or systems not supported natively by Terraform.

2. Examples of Popular Provider Plugins:

- **AWS** (hashicorp/aws): Manage AWS resources like EC2, S3, RDS, etc.
- **Azure** (hashicorp/azurerm): Manage resources on Microsoft Azure.
- **Kubernetes** (hashicorp/kubernetes): Manage Kubernetes clusters and workloads.

How Plugins Work

1. Initialization:

- Plugins are downloaded during terraform init based on the providers defined in your configuration.
- Example: If your configuration uses AWS, Terraform downloads the hashicorp/aws provider plugin.

2. Execution:

- Terraform uses plugins to communicate with the target API and manage resources.
- Plugins perform CRUD operations (Create, Read, Update, Delete) based on your .tf configuration.

3. Dependency Management:

- Plugins are versioned and stored in the .terraform directory of your project.

- You can specify versions in your configuration for compatibility.

```
Ex:-provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "example" {  
  ami           = "ami-0c55b159cbfaffe1f0"  
  instance_type = "t2.micro"  
}
```

Here, the **AWS plugin** translates this configuration into API calls to AWS.

Benefits of Plugins

1. Extensibility:

- Plugins allow Terraform to work with a wide range of providers and services.

2. Modularity:

- You can enable or disable specific providers without affecting others.

3. Customization:

- Create plugins for internal tools or proprietary APIs.

Where Plugins Are Stored

1. Local Plugins:

- Terraform stores downloaded plugins in a `.terraform/plugins` or `.terraform/providers` directory within the project.

2. Remote Plugins:

- Terraform pulls plugins from the Terraform Registry or other remote registries.

What is the .tfstate File?

- **Definition:** It is a JSON file where Terraform keeps a record of the current state of the resources it manages.
 - **Purpose:** Tracks the mapping between your configuration (.tf files) and the real-world infrastructure.
-

Why is it Important?

1. State Tracking:

- Ensures Terraform knows which resources it created, updated, or deleted.
- Helps reconcile differences between the desired state and the actual state during plan or apply.

2. Incremental Updates:

- If you modify your .tf configuration, Terraform references the .tfstate to determine what needs to change, avoiding re-provisioning everything.

3. Drift Detection:

- Identifies discrepancies if resources are modified outside Terraform (e.g., manually in the AWS Console).

Detecting Drift Between Real-World State and Defined State

- **What is Drift?**

"Drift" happens when the actual state of your infrastructure differs from the state defined in your Terraform configuration (.tf files) or recorded in the .tfstate file.

- Example: You use the AWS Console to manually stop an EC2 instance managed by Terraform. This change creates a "drift" because Terraform doesn't know about it.

- **How Does Terraform Handle Drift?**

- When you run terraform plan, Terraform checks the **real-world state** (by querying the cloud provider) and compares it with:
 1. Your .tf configuration (desired state).
 2. The .tfstate file (record of the previous state).
- It detects mismatches and displays them as actions to "fix the drift."

Infrastructure as Code(IaC)

Before the advent of IaC, infrastructure management was typically a manual and time-consuming process. System administrators and operations teams had to:

1. **Manually Configure Servers:** Servers and other infrastructure components were often set up and configured manually, which could lead to inconsistencies and errors.
2. **Lack of Version Control:** Infrastructure configurations were not typically version-controlled, making it difficult to track changes or revert to previous states.
3. **Documentation Heavy:** Organizations relied heavily on documentation to record the steps and configurations required for different infrastructure setups. This documentation could become outdated quickly.
4. **Limited Automation:** Automation was limited to basic scripting, often lacking the robustness and flexibility offered by modern IaC tools.
5. **Slow Provisioning:** Provisioning new resources or environments was a time-consuming process that involved multiple manual steps, leading to delays in project delivery.

IaC addresses these challenges by providing a systematic, automated, and code-driven approach to infrastructure management. Popular IaC tools include Terraform, AWS CloudFormation, Azure Resource Manager templates others.

These tools enable organizations to define, deploy, and manage their infrastructure efficiently and consistently, making it easier to adapt to the dynamic needs of modern applications and services.

Why Terraform ?

There are multiple reasons why Terraform is used over the other IaC tools but below are the main reasons.

1. **Multi-Cloud Support:** Terraform is known for its multi-cloud support. It allows you to define infrastructure in a cloud-agnostic way, meaning you can use the same configuration code to provision resources on various cloud providers (AWS, Azure, Google Cloud, etc.) and even on-premises infrastructure. This flexibility can be beneficial if your organization uses multiple cloud providers or plans to migrate between them.
2. **Large Ecosystem:** Terraform has a vast ecosystem of providers and modules contributed by both HashiCorp (the company behind Terraform) and the community. This means you can find pre-built modules and configurations for a wide range of services and infrastructure components, saving you time and effort in writing custom configurations.
3. **Declarative Syntax:** Terraform uses a declarative syntax, allowing you to specify the desired end-state of your infrastructure. This makes it easier to understand and maintain your code compared to imperative scripting languages.

4. **State Management:** Terraform maintains a state file that tracks the current state of your infrastructure. This state file helps Terraform understand the differences between the desired and actual states of your infrastructure, enabling it to make informed decisions when you apply changes.
5. **Plan and Apply:** Terraform's "plan" and "apply" workflow allows you to preview changes before applying them. This helps prevent unexpected modifications to your infrastructure and provides an opportunity to review and approve changes before they are implemented.
6. **Community Support:** Terraform has a large and active user community, which means you can find answers to common questions, troubleshooting tips, and a wealth of documentation and tutorials online.
7. **Integration with Other Tools:** Terraform can be integrated with other DevOps and automation tools, such as Docker, Kubernetes, Ansible, and Jenkins, allowing you to create comprehensive automation pipelines.
8. **HCL Language:** Terraform uses HashiCorp Configuration Language (HCL), which is designed specifically for defining infrastructure. It's human-readable and expressive, making it easier for both developers and operators to work with.

Getting Started

To get started with Terraform, it's important to understand some key terminology and concepts. Here are some fundamental terms and explanations.

1. **Provider:** A provider is a plugin for Terraform that defines and manages resources for a specific cloud or infrastructure platform. Examples of providers include AWS, Azure, Google Cloud, and many others. You configure providers in your Terraform code to interact with the desired infrastructure platform.
2. **Resource:** A resource is a specific infrastructure component that you want to create and manage using Terraform. Resources can include virtual machines, databases, storage buckets, network components, and more. Each resource has a type and configuration parameters that you define in your Terraform code.
3. **Module:** A module is a reusable and encapsulated unit of Terraform code. Modules allow you to package infrastructure configurations, making it easier to maintain, share, and reuse them across different parts of your infrastructure. Modules can be your own creations or come from the Terraform Registry, which hosts community-contributed modules.

4. **Configuration File:** Terraform uses configuration files (often with a .tf extension) to define the desired infrastructure state. These files specify providers, resources, variables, and other settings. The primary configuration file is usually named main.tf, but you can use multiple configuration files as well.
5. **Variable:** Variables in Terraform are placeholders for values that can be passed into your configurations. They make your code more flexible and reusable by allowing you to define values outside of your code and pass them in when you apply the Terraform configuration.
6. **Output:** Outputs are values generated by Terraform after the infrastructure has been created or updated. Outputs are typically used to display information or provide values to other parts of your infrastructure stack.
7. **State File:** Terraform maintains a state file (often named terraform.tfstate) that keeps track of the current state of your infrastructure. This file is crucial for Terraform to understand what resources have been created and what changes need to be made during updates.
8. **Plan:** A Terraform plan is a preview of changes that Terraform will make to your infrastructure. When you run terraform plan, Terraform analyzes your configuration and current state, then generates a plan detailing what actions it will take during the apply step.
9. **Apply:** The terraform apply command is used to execute the changes specified in the plan. It creates, updates, or destroys resources based on the Terraform configuration.
10. **Workspace:** Workspaces in Terraform are a way to manage multiple environments (e.g., development, staging, production) with separate configurations and state files. Workspaces help keep infrastructure configurations isolated and organized.
11. **Remote Backend:** A remote backend is a storage location for your Terraform state files that is not stored locally. Popular choices for remote backends include Amazon S3, Azure Blob Storage, or HashiCorp Terraform Cloud. Remote backends enhance collaboration and provide better security and reliability for your state files.

These are some of the essential terms you'll encounter when working with Terraform. As you start using Terraform for your infrastructure provisioning and management, you'll become more familiar with these concepts and how they fit together in your IaC workflows.

Variables.tf v/s .tfvars

variables.tf:

- Purpose: Defines the variables used in the Terraform configuration.
- Content: Includes the variable name, type, default value, and description.
Example:

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

terraform.tfvars:

- Purpose: Assigns values to the variables declared in variables.tf.
- Content: Contains specific values for variables without needing to declare their structure.
Example:

```
instance_type = "t3.medium"  
region        = "us-west-2"
```

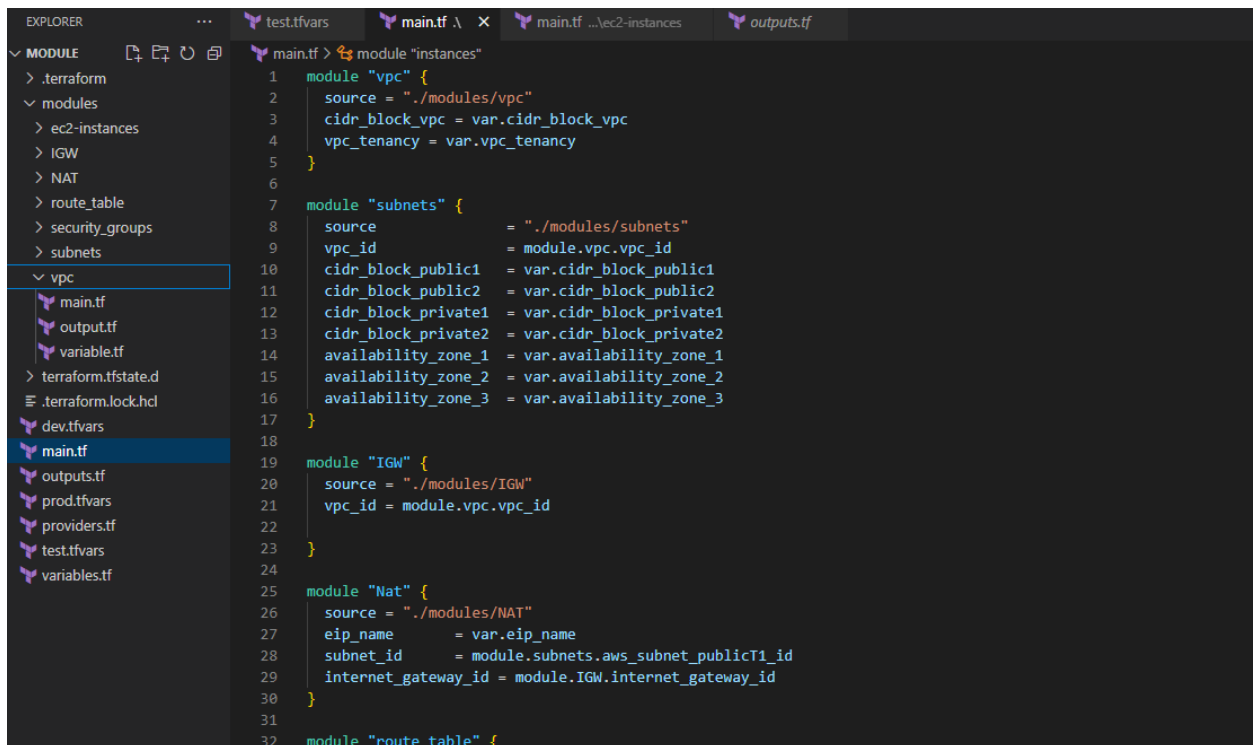
Key Differences:

Feature	variables.tf	terraform.tfvars
Definition vs. Assignment	Defines variable structure	Assigns values to variables
Optional/Required	Mandatory for variable definition	Optional (can use other methods for input)
File Name Standard	Convention is variables.tf	Convention is terraform.tfvars

MODULES

Modules are containers for multiple resources that are used together to accomplish a specific task. They help organize and reuse infrastructure code by grouping related resources and configurations.

```
.
├── main.tf      # Root module calling child modules
├── variables.tf # Root module variables
├── outputs.tf   # Root module outputs
├── vpc/         # Child module directory
│   ├── main.tf
│   ├── variables.tf
│   ├── outputs.tf
│   └── ...
└── ec2/        # Another child module
    ├── main.tf
    ├── variables.tf
    ├── outputs.tf
    └── ...
```



```
EXPLORER
├── .terraform
├── modules
│   ├── ec2-instances
│   ├── IGW
│   ├── NAT
│   ├── route_table
│   ├── security_groups
│   └── subnets
│       ├── main.tf
│       ├── output.tf
│       └── variable.tf
├── terraform.tfstate.d
├── .terraform.lock.hcl
├── dev.tfvars
├── main.tf
├── outputs.tf
├── prod.tfvars
├── providers.tf
├── test.tfvars
└── variables.tf

main.tf
1 module "instances"
2   module "vpc" {
3     source = "../modules/vpc"
4     cidr_block_vpc = var.cidr_block_vpc
5     vpc_tenancy = var.vpc_tenancy
6   }
7   module "subnets" {
8     source = "../modules/subnets"
9     vpc_id = module.vpc.vpc_id
10    cidr_block_public1 = var.cidr_block_public1
11    cidr_block_public2 = var.cidr_block_public2
12    cidr_block_private1 = var.cidr_block_private1
13    cidr_block_private2 = var.cidr_block_private2
14    availability_zone_1 = var.availability_zone_1
15    availability_zone_2 = var.availability_zone_2
16    availability_zone_3 = var.availability_zone_3
17  }
18
19  module "IGW" {
20    source = "../modules/IGW"
21    vpc_id = module.vpc.vpc_id
22  }
23
24
25  module "Nat" {
26    source = "../modules/NAT"
27    eip_name = var.eip_name
28    subnet_id = module.subnets.aws_subnet_publicT1_id
29    internet_gateway_id = module.IGW.internet_gateway_id
30  }
31
32  module "route_table" {
```

*** Root Module:**

- Every Terraform configuration is inherently a module (the root module).
- Consists of all .tf files in the working directory.

***Child Modules:**

- Modules that are called by the root module or other modules.
- Can be stored locally or fetched from a remote source (e.g., Terraform Registry, GitHub, S3).

*** Advantages of Modules:**

- **Reusability:** Write once and use across multiple projects.
- **Consistency:** Standardize infrastructure components.
- **Abstraction:** Hide implementation details while exposing inputs and outputs.

(EXPLORE LITTLE MORE BY YOURSELF)

TERRAFORM IMPORT

Terraform Import is a command that allows you to bring existing infrastructure into Terraform's state file, enabling Terraform to manage that resource. This command does not generate a configuration file for the imported resource; you need to write the .tf configuration manually.

Key Features of Terraform Import

1. **State Synchronization:**
 - Links existing infrastructure with Terraform's state, making Terraform aware of it.
2. **Resource Compatibility:**
 - Supports many resource types, including instances, buckets, and databases.
3. **Configuration Flexibility:**
 - Requires writing corresponding .tf files for the imported resource manually.

4. Resource-Specific Identifiers:

- The import process uses resource-specific unique identifiers (e.g., instance ID for AWS EC2).

5. Non-Destructive:

- Does not alter the existing infrastructure during the import process.

When to Use Terraform Import

1. Adopting Terraform for Existing Infrastructure:

- When migrating an already-deployed infrastructure to Terraform for management.

2. Recovering State:

- Re-establishing Terraform state if the state file is lost or corrupted.

3. Hybrid Infrastructure Management:

- Gradually transitioning some resources under Terraform management without redeploying them

Command syntax

terraform import RESOURCE_TYPE.NAME RESOURCE_ID

```
main.tf > ...
1  resource "aws_instance" "imported-ec2" {
2      ami = "" #ami id of importing instances
3      instance_type = "" #type of importing instances
4
5  }
6
7  |
```

```
egoud@LAPTOP-6MJB0998 MINGW64 ~/OneDrive/Documents/Desktop/import
$ terraform import aws_instances.imported-ec2 id-2nje322234
```

(you need to create a empty resource and import after that add the remaining attributes)

TERRAFORM PROVISIONERS

The provisioners are used to performing certain custom actions & tasks either on the **local machine** or on the **remote machine**

There are two types in provisioners

1. Generic provisioners (independent)
2. vendor specific

1: → *file provisioner

*local provisioner

* remote provisioner

2: → *chef

*puppet

* habitat

Refere this link for more document (<https://jhooq.com/terraform-provisioner/>)

TERRAFORM COMMANDS

- **terraform init**
Prepares your working directory for Terraform operations by downloading plugins and initializing configurations.
- **terraform plan**
Creates an execution plan, showing changes Terraform will make to achieve the desired state.
- **terraform apply**
Executes the changes defined in the Terraform configuration to create, update, or destroy resources.
- **terraform destroy**
Removes all infrastructure managed by Terraform in the current workspace.

- **terraform validate**
Checks the Terraform configuration files for syntax and logical errors.
- **terraform fmt**
Formats Terraform configuration files to a standard style for better readability.
- **terraform show**
Displays the details of the Terraform state or a saved plan file.
- **terraform output**
Fetches the outputs defined in your configuration after a successful apply.
- **terraform state**
Manipulates or inspects the Terraform state, such as listing or moving resources.
- **terraform refresh**
Updates the Terraform state with the real-world infrastructure to detect changes.
- **terraform import**
Links existing resources in the cloud provider to Terraform state for management.
- **terraform workspace**
Manages multiple workspaces for isolating environments like dev, staging, and prod.
- **terraform graph**
Generates a visual representation of the Terraform configuration.
- **terraform console**
Opens a REPL console for evaluating expressions and testing configurations.
- **terraform taint**
Marks a resource for destruction and recreation during the next apply.
- **terraform untaint**
Removes the taint marking from a resource.
- **terraform providers**
Displays the providers required for the current configuration.
- **terraform login**
Authenticates Terraform to Terraform Cloud or Enterprise.
- **terraform logout**
Removes Terraform Cloud or Enterprise authentication credentials.
- **terraform state mv**
Moves a resource in the state file from one name or module to another.
- **terraform state rm**
Removes a resource from Terraform state without affecting the actual infrastructure.
- **terraform force-unlock**
Manually removes a lock on the state file if a previous operation was interrupted.
- **terraform plan -destroy**
Previews the destruction of all resources.

TERRAFORM PROVIDERS

1. Officially -managed by Hashicorp (plugins)
2. partner -Third party
3. Community – managed by individual community's

(Also check Terraform state, attributes)

VARIABLES

Multiple approaches to variable assignments

1. env variables
2. cli flags
3. from a file
4. variable default

TERRAFORM STATE COMMAND :-

1. **terraform state list**: Lists all resources tracked in the Terraform state.
2. **terraform state show [resource]**: Displays detailed information about a specific resource in the state.
3. **terraform state mv [src] [dest]**: Moves a resource from one location to another within the state file.
4. **terraform state rm [resource]**: Removes a resource from the state without destroying it in the infrastructure.
5. **terraform state pull**: Fetches the current state from the remote backend and saves it locally.
6. **terraform state push [state file]**: Overwrites the remote state with a local state file.
7. **terraform state replace-provider**: Replaces provider references in the state file (useful during provider migrations).

(These commands are essential for advanced state management and troubleshooting in Terraform projects.)

NULL RESOURCES :-

As in the name you see a prefix ***null*** which means this resource will not exist on your Cloud Infrastructure(AWS, Google Cloud, Azure). The reason is there is no terraform state associated with it, due to you can update the *null_resource* inside your Terraform file as many times as you can.

Terraform *null_resource* can be used in the following scenario -

1. Run shell command
2. You can use it along with [local provisioner and remote provisioner](#)
3. It can also be used with Terraform Module, Terraform count, Terraform Data source, Local variables
4. It can even be used for output block

It is a tool managing dependencies outside direct infrastructure, used to execute provisioners (scripts commands) to define dependency chain between resources

(Reference :- <https://jhooq.com/terraform-null-resource/>)

TERRAFORM WORKSPACE

A Terraform workspace is an environment where Terraform manages a distinct state. It is especially useful for managing multiple environments like dev, test, and prod using the same Terraform configuration but maintaining separate state files.

Key Features:

1. **Isolation:** Each workspace has its own state file, allowing isolation of environments within a single configuration.
2. **Default Workspace:** The initial workspace created automatically in every Terraform setup is named default.
3. **Dynamic Names:** Workspaces can be dynamically named, enabling flexibility for multiple environments.
4. **Remote Workspaces:** In Terraform Cloud, workspaces are the primary units for organizing configurations and states.

Required Commands:

1. Create Workspace:

```
terraform workspace new <workspace_name>
```

Creates a new workspace.

2. Switch Workspace:

```
terraform workspace select <workspace_name>
```

Switches to an existing workspace.

3. List Workspaces:

```
terraform workspace list
```

Displays all available workspaces.

4. Show Current Workspace:

```
terraform workspace show
```

Displays the current workspace name.

5. Delete Workspace:

```
terraform workspace delete <workspace_name>
```

Use Cases:

1. **Multi-Environment Management:** Separate workspaces for dev, test, and prod.
2. **Resource Isolation:** Avoid overwriting resources by using separate state files.
3. **Simplified Configurations:** Maintain a single configuration for all environments, relying on workspace-specific states.

Workspaces provide a simple way to manage infrastructure configurations across multiple environments effectively.

(Reference link:- <https://jhooq.com/terraform-workspaces/> with examples)

TERRAFORM BACKEND

A **Terraform backend** is responsible for storing and managing the state of your infrastructure, enabling Terraform to function as an effective tool for provisioning and managing infrastructure resources.

Types of Terraform Backend

1.local

2.Remote :- s3,aws s3, terraform cloud,...etc

Key Points:

1. **State Storage:** Terraform's state file (usually terraform.tfstate) tracks the resources managed by Terraform. The backend is where this state file is stored.
2. **Remote Backends:** A remote backend stores the state file remotely, often in cloud storage (like **AWS S3**, Azure Blob Storage, etc.), enabling collaboration among team members. This is crucial when using Terraform in team environments as it ensures consistency in the state across multiple users.
3. **Local Backends:** By default, Terraform stores the state file locally in the project directory, which is useful for individual use cases or small-scale projects.
4. **Backend Configuration:** The backend is configured in the terraform block, typically in a backend.tf file or directly in main.tf for remote storage.
5. **Locking and Consistency:** When using remote backends (like S3 with DynamoDB for state locking), Terraform ensures that only one user can modify the state at a time, preventing conflicts.

Benefits of Using Remote Backends:

1. **Collaboration:** Enables multiple developers to work on the same infrastructure, as state is centralized.
2. **State Locking:** Prevents conflicts with state changes, particularly when using resources concurrently.
3. **Security:** State files often contain sensitive information (e.g., secrets, passwords), and remote backends provide encryption and access control.
4. **Consistency:** Ensures the same state is used across different environments and by all team members.

Remote Backend Configuration (using AWS S3 and DynamoDB for locking)

```
dev1 > main.tf > terraform
1  provider "aws" {
2      region = "ap-south-1"
3      access_key = "AKIA3TD2SLYQDKECMQM6"
4      secret_key = "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"
5  }
6
7  terraform {
8      backend "s3" {
9          bucket = "bucket-static-v1"
10         dynamodb_table = "dynamodb-state-locking"
11         key = "key/terraform.tfstate"
12         region = "ap-south-1"
13         encrypt = true
14     }
15 }
```

Types of Backends:

1. **Local Backend:** Default, stores the state file locally.
2. **Remote Backends:** Use cloud storage for state management (e.g., AWS S3, Azure Storage, GCS).
3. **Custom Backends:** Terraform allows the creation of custom backends to suit specific needs.

Note:- * Using backend allows storing the state remotely, enabling collaboration, state locking & better state management)

(Reference link :- <https://jhooq.com/terraform-manage-state/> , <https://jhooq.com/terraform-state-file-locking/>)

The disadvantage of using aws s3 and DynamoDB for storing and locking is

1. high cost
2. needs to be maintained
3. storing and locking is separate

- Before Going to Terraform cloud lets check some Git commands which will help in VCS

GIT COMMANDS

PUSH

1. git init
2. git add <files excluding the statefiles><only code .tf files>
3. git status
4. git commit -m "first commit"
5. git log
6. git checkout -b <branch name>
7. git remote add origin <repo link(create a link for infra)>
8. git push -u origin <branch name>

PULL

1. git init
2. git remote add origin <repo link>
3. git pull origin <branch name>

TERRAFORM CLOUD

1.cli-Driven

2.VCS

Terraform Cloud Overview

Terraform Cloud is a SaaS offering by HashiCorp that simplifies infrastructure management by providing features like:

- **Remote State Management:** Keeps state files in a secure, shared location.
- **Collaboration:** Teams can work together on infrastructure changes with access controls and audit logs.

- **Run Environment:** Executes Terraform plans and applies remotely in Terraform's environment.
- **Policy as Code:** Allows enforcement of organizational policies using Sentinel.
- **VCS Integration:** Integrates with GitHub, GitLab, or Bitbucket for automated runs based on code changes.
- **Notifications and Webhooks:** Alerts or actions triggered by workspace events.

Key Components of Terraform Cloud

1. **Organizations:** A way to group workspaces for a company or project.
2. **Workspaces:** Each workspace manages a unique Terraform state. Workspaces in Terraform Cloud differ from traditional CLI workspaces.
3. **Runs:** A single execution of a Terraform plan or apply within a workspace.
4. **Variables:** You can define environment variables, Terraform input variables, or sensitive variables (e.g., API keys).
5. **Remote Operations:** Terraform Cloud performs terraform plan and terraform apply remotely.

CLI-Driven Workspaces in Terraform Cloud

What are CLI-Driven Workspaces?

CLI-driven workspaces allow Terraform Cloud to act as a **remote backend** for Terraform operations initiated through the CLI. Unlike VCS-driven workspaces, where plans and applies are triggered automatically by VCS changes, CLI-driven workspaces depend on user-initiated actions.

How CLI-Driven Workspaces Work

1. **Terraform Backend Configuration:** Configure Terraform to use Terraform Cloud as the backend for state storage.
2. **CLI Commands:** Users run terraform init, terraform plan, and terraform apply from their local CLI.
3. **State Management:** The state file is automatically stored in Terraform Cloud.
4. **Remote Execution:** Optionally, the actual execution (plan and apply) can occur in Terraform Cloud.

TERRAFORM CLOUD CLI DRIVEN SETUP:-

1. Register in official terraform cloud :- <https://app.terraform.io>
2. Create an organization
3. Create a Workspace (choose CLI -driven)

Choose your workflow

Version Control Workflow

Trigger runs based on changes to configuration in repositories.



Best for those who need traceability and transparency

CLI-Driven Workflow

Trigger runs in a workspace using the Terraform CLI.



Best for those comfortable with Terraform CLI

API-Driven Workflow

Trigger runs using the HCP Terraform API.



Best for those with custom integrations and pipelines

4. Copy the code form cloud and add in local .tf file

```
Backend_cloud.tf > terraform
1 terraform {
2   backend "remote" {
3
4     organization = "Fundoo_3Tire"
5
6     workspaces {
7       name = "PROD"
8     }
9   }
10 }
```

5. Login into the terraform cloud through cli

```
PS C:\Users\egoud\OneDrive\Documents\Desktop\TFC> terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
  C:\Users\egoud\AppData\Roaming\terraform.d\credentials.tfrc.json

Do you want to proceed?
Only 'yes' will be accepted to confirm.

Enter a value: █
```

6. Run and apply form the cli
7. state files and lock will maintained by Terraform cloud

CLI_Creation

01: wv-p2tcl2awpact4q3P

Add workspace description.

Unlocked

Resources 0

Terraform v1.10.1

Updated 9 hours ago

Lock

New run

Triggered via UI

CURRENT

Applied

Estimated cost decrease

↓ \$60.72

Plan & apply duration

2 minutes

Resources changed

+0 -0 -31

erangoud triggered a destroy run from UI 9 hours ago

Run Details

Plan finished

9 hours ago

Resources: 0 to add, 0 to change, 31 to destroy

Cost estimation finished

9 hours ago

Resources: 5 of 5 estimated - \$0.00/mo - \$60.72

Apply finished

9 hours ago

Resources: 0 added, 0 changed, 31 destroyed

Triggered via UI

#sv-cNgmVLu7kjq2vAQ | erangoud triggered from Terraform 9 hours ago | #run-9jPXR7grb5KHsV48

Triggered via CLI

#sv-L9euY8AHD7DKpC7a | erangoud triggered from Terraform 9 hours ago | #run-v4HW9G5y89LFMXzh

What are Sentinel Mocks?

Sentinel mocks are **JSON files** representing the structure and content of Terraform state or plan data. They simulate the state, configuration, or execution context of a Terraform run. These mocks allow policy authors to test and validate their Sentinel policies in a controlled environment without executing real Terraform workflows.

TERRAFORM CLOUD VCS SETUP :-

Terraform Cloud's **Version Control Integration** allows you to connect a **remote Git repository** to manage and automate infrastructure workflows. Below is a step-by-step explanation of how Terraform Cloud integrates with version control and how the workflow operates.

1. High-Level Overview

Terraform Cloud uses version control systems (VCS) like **GitHub**, **GitLab**, **Bitbucket**, or **Azure Repos** to:

1. Automatically detect changes in your Terraform configurations.
2. Trigger Terraform **runs** (plan and apply) when changes are pushed to the repository.
3. Provide a central dashboard for state management and collaborative workflows.

2. Components of Version Control Flow

1. **Workspace:**

- A **Terraform Cloud Workspace** is mapped to a Git repository or a specific branch.
- Each workspace manages its own **Terraform state**.

2. **Version Control System (VCS):**

- Terraform Cloud integrates with Git repositories to automatically trigger runs.

3. **Runs:**

- Terraform Cloud performs **plan** and **apply** runs when changes are pushed to the repository.

4. **Variables:**

- Use Terraform Cloud to securely manage variables (e.g., access keys, configurations) for different environments.

3. Version Control Flow in Terraform Cloud

Step 1: Connect Your Git Repository

1. Go to your **Terraform Cloud Workspace**.
2. Under the **Settings** tab, click **Version Control**.
3. Choose a supported Git provider (e.g., GitHub).
4. Authenticate and authorize Terraform Cloud to access your repository.
5. Select a **repository** and optionally, specify a **branch** to link with the workspace.

Step 2: Initialize the Workspace

Once the Git repository is linked:

1. Terraform Cloud watches the linked branch for changes to .tf files.
2. The workspace is automatically initialized, setting up backend storage for the Terraform state.

Step 3: Push Terraform Code to the Repository

Open a **pull request** (PR) in your Git provider to merge changes into the main branch.

Step 4: Terraform Cloud Detects Changes

1. When the pull request is created or merged, Terraform Cloud triggers a **plan run**:
 - Terraform Cloud pulls the latest code from the branch.
 - Executes terraform init and terraform plan.
 - Displays the **plan summary** in the Terraform Cloud UI.
2. Review the plan output in Terraform Cloud:
 - **Additions (+)**: Resources to be created.
 - **Changes (~)**: Resources to be updated.
 - **Deletions (-)**: Resources to be destroyed.

Step 5: Approve or Apply Changes

1. For non-production environments:
 - If the Terraform plan is correct, Terraform Cloud automatically applies the changes.
2. For production environments:
 - Use **manual approval** to apply the changes:
 - Review the plan in the Terraform Cloud UI.
 - Click **Confirm & Apply** to execute terraform apply.

Step 6: Collaborative Workflow

1. **Collaborators**:
 - Team members can view, comment, and approve Terraform runs directly in Terraform Cloud.
2. **Version Control History**:
 - The Git repository retains the history of changes.

- Terraform Cloud provides detailed logs of plan and apply runs, tied to the specific Git commit.

CLI vs. VCS-Driven Workspaces

Feature	CLI-Driven Workspace	VCS-Driven Workspace
Triggering Runs	User runs commands via CLI	Automatically triggered by VCS changes
State Management	Remote backend in Terraform Cloud	Remote backend in Terraform Cloud
Collaboration	Requires manual coordination	Built-in with version control triggers
Execution Location	Local CLI or remotely in Terraform Cloud	Remotely in Terraform Cloud