Terraform Complete Docs Beginner to Advanced

Topics Covered

What is Iac?

What is Terraform?

Terraform Architecture and Use Cases

Terraform Installation and Guidance

What is HCL?

Basic Syntax of Terraform

Terraform Commands

Terraform folders

What is Resource, provider and Block in terraform?

Meta Arguments

Modules

Custom Modules

Basic Terraform Project step by step

What is Remote backend

What is workspace ?

Introduction to Terraform

➢ What is Infrastructure as Code (IaC)?

Infrastructure as Code (IaC) is the practice of managing and provisioning computer infrastructure, such as virtual machines, networks, and databases, through code written in a human-readable format. This approach allows for the creation, modification, and deletion of infrastructure resources in a consistent, repeatable, and version-controlled manner.

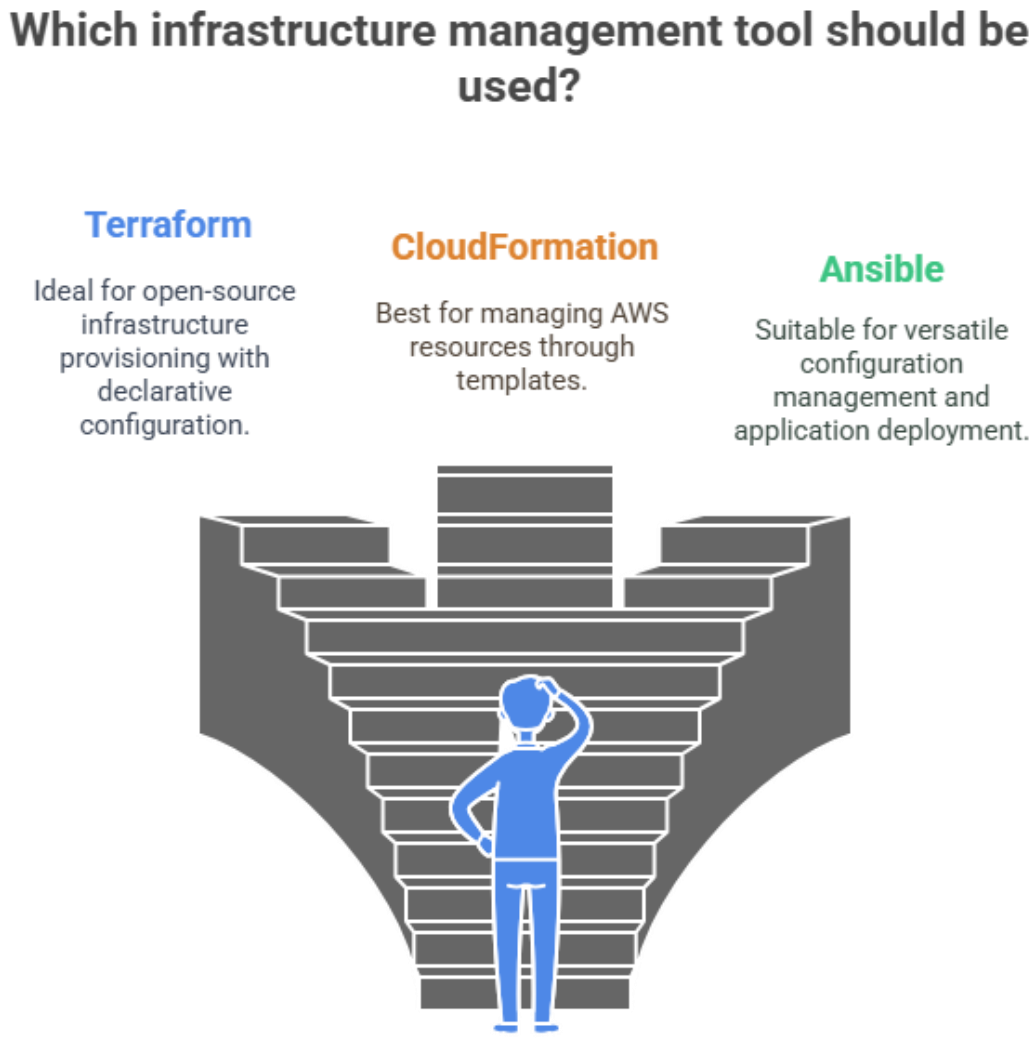
Key Benefits of IaC:

- **Consistency**: IaC ensures that infrastructure is provisioned consistently across different environments, such as development, testing, and production.
- **Repeatability**: IaC enables the recreation of infrastructure resources with a single command, reducing the risk of human error.
- **Version control**: IaC code is stored in a version control system, allowing for tracking changes and auditing.
- **Collaboration**: IaC enables multiple teams to work together on infrastructure provisioning, reducing conflicts and improving communication.

Popular IaC Tools:

- **Terraform**: An open-source IaC tool that supports multiple cloud providers, such as AWS, Azure, and Google Cloud.
- **AWS CloudFormation**: A service that allows users to create and manage infrastructure resources in AWS using a template.
- **Ansible**: An open-source automation tool that can be used for IaC, among other use cases.
- **CloudFormation**: A service offered by AWS that allows users to create and manage infrastructure resources in AWS using a template.

➢ **Terraform vs CloudFormation vs Ansible**

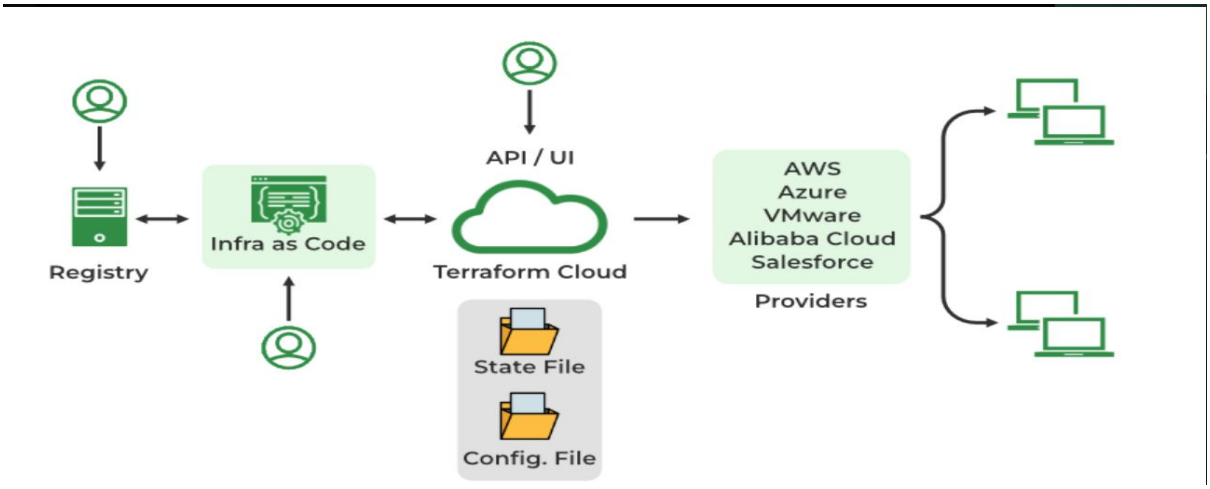


➢ Terraform use cases and architecture

Terraform is a versatile tool that can be used in a variety of scenarios. Here are some common use cases for Terraform:

1. **Cloud Infrastructure Management**: Terraform can be used to manage cloud infrastructure resources such as virtual machines, networks, and storage.
2. **DevOps Automation**: Terraform can be used to automate DevOps tasks such as provisioning, deployment, and configuration management.
3. **Infrastructure as Code (IaC)**: Terraform allows you to define infrastructure resources in a human-readable configuration file, making it easier to manage and version control infrastructure.
4. **Multi-Cloud Management**: Terraform supports multiple cloud providers, making it easier to manage infrastructure across multiple clouds.
5. **Hybrid Cloud Management**: Terraform can be used to manage infrastructure in both cloud and on-premises environments.
6. **Disaster Recovery**: Terraform can be used to automate disaster recovery processes, ensuring that infrastructure is restored quickly in the event of a disaster.
7. **Security**: Terraform can be used to automate security tasks such as configuring firewalls, setting up access control lists, and managing encryption.

➢ Terraform Architecture



Components of Terraform Architecture

1. Terraform Configuration Files

These files contain the definition of the infrastructure resources that Terraform will manage, as well as any input and output variables and modules. The configuration files are written in the HashiCorp Configuration Language (HCL), which is a domain-specific language designed specifically for Terraform.

2. Terraform State File

This file stores the current state of the infrastructure resources managed by Terraform state. The state file is used to track the resources that have been created, modified, or destroyed, and it is used to ensure that the infrastructure resources match the desired state defined in the configuration files.

3. Infrastructure as Code

Terraform allows you to use code to define and manage your infrastructure, rather than manually configuring resources through a user interface. This makes it easier to version, review, and collaborate on infrastructure changes.

4. Cloud APIs or other Infrastructure Providers

These are the APIs or other interfaces that Terraform uses to create, modify, or destroy infrastructure resources. Terraform supports multiple cloud providers, as well as on-premises and open-source tools.

5. Providers

Terraform integrates with a wide range of cloud and infrastructure providers, including AWS, Azure, GCP, and more. These providers allow Terraform to create and manage resources on those platforms.

Overall, the architecture of a Terraform deployment consists of configuration files, a state file, and a CLI that interacts with cloud APIs or other infrastructure providers to create, modify, or destroy resources. This architecture enables users to define and manage infrastructure resources in a declarative and reusable way.

Terraform Private Module Registry

A private module registry is a repository for Terraform modules that is only accessible to a specific group of users, rather than being publicly available. Private module registries are useful for organizations that want to manage and distribute their own infrastructure code internally, rather than using publicly available modules from the Terraform Registry.

Source: https://www.geeksforgeeks.org/what-is-terraform/

Terraform Installation and Guidance

Mac: https://developer.hashicorp.com/terraform/install#darwin

Windows : https://developer.hashicorp.com/terraform/install#windows

Linux : https://developer.hashicorp.com/terraform/install#linux

HashiCorp Configuration Language (HCL)

HCL is a configuration language developed by HashiCorp, the company behind Terraform. It is a human-readable format that allows you to define infrastructure resources in a concise and readable way.

BASIC TERRAFORM SYNTAX

- **Blocks**: A block is a group of attributes that define a resource or a value. Blocks are denoted by curly brackets {}.
- **Attributes**: Attributes are key-value pairs that define the properties of a resource or a value. Attributes are denoted by a key followed by a colon : and a value.
- **Variables**: Variables are values that can be used throughout the configuration file. Variables are denoted by a dollar sign $ followed by a name.
- **Functions**: Functions are reusable blocks of code that can be used to define complex resources or values. Functions are denoted by a function keyword.

.tf and .tfvars files

Terraform configuration files have the following extensions:

- **.tf**: This is the main configuration file that defines the infrastructure resources.
- **.tfvars**: This is a file that contains variable values that can be used throughout the configuration file

Example of Basic Syntax in Terraform

```
# Define a resource
resource "aws_instance" "example" {
  # Define the resource attributes
  ami           = "ami-abc123"
  instance_type = "t2.micro"
}

# Define a variable
variable "region" {
  type        = string
  default     = "us-west-2"
}

# Use the variable in a resource
resource "aws_instance" "example" {
  # Use the variable value
  region = var.region
}
```

Terraform Commands

Terraform provides several commands that can be used to manage infrastructure resources. Here are some of the most commonly used Terraform commands:

- **terraform init**: Initializes a new Terraform working directory.
- **terraform plan**: Creates an execution plan for the current configuration.
- **terraform apply**: Applies the changes described in the execution plan.
- **terraform destroy**: Destroys the infrastructure resources created by Terraform.
- **Terraform validate:** The validate command performs precisely what its name implies. It ensures that the code is internally coherent and examines it for syntax mistake

.terraform Folder

When you run terraform init, Terraform creates a new .terraform folder in the current working directory. This folder contains the following files and subfolders:

- **terraform.tfstate**: The Terraform state file, which contains the current state of the infrastructure resources.
- **terraform.tfstate.backup**: A backup of the Terraform state file.
- **terraform.tfvars**: A file that contains variable values.
- **.terraform.lock.hcl**: A lock file that prevents multiple Terraform processes from accessing the same state file.

Resources

Resources are the core of Terraform, and they represent the infrastructure components that you want to create, manage, or destroy. Resources can be thought of as the "building blocks" of your infrastructure.

Providers

Providers are the plugins that allow Terraform to interact with specific cloud or on-premises infrastructure providers. Providers are responsible for creating, managing, and destroying resources on behalf of Terraform.

Variables

Variables are values that can be used throughout your Terraform configuration. Variables can be used to parameterize your configuration, making it easier to manage and reuse.

Outputs

Outputs are values that are generated by Terraform and can be used by other tools or scripts. Outputs can be used to capture values such as IP addresses, URLs, or other metadata that is generated by Terraform.

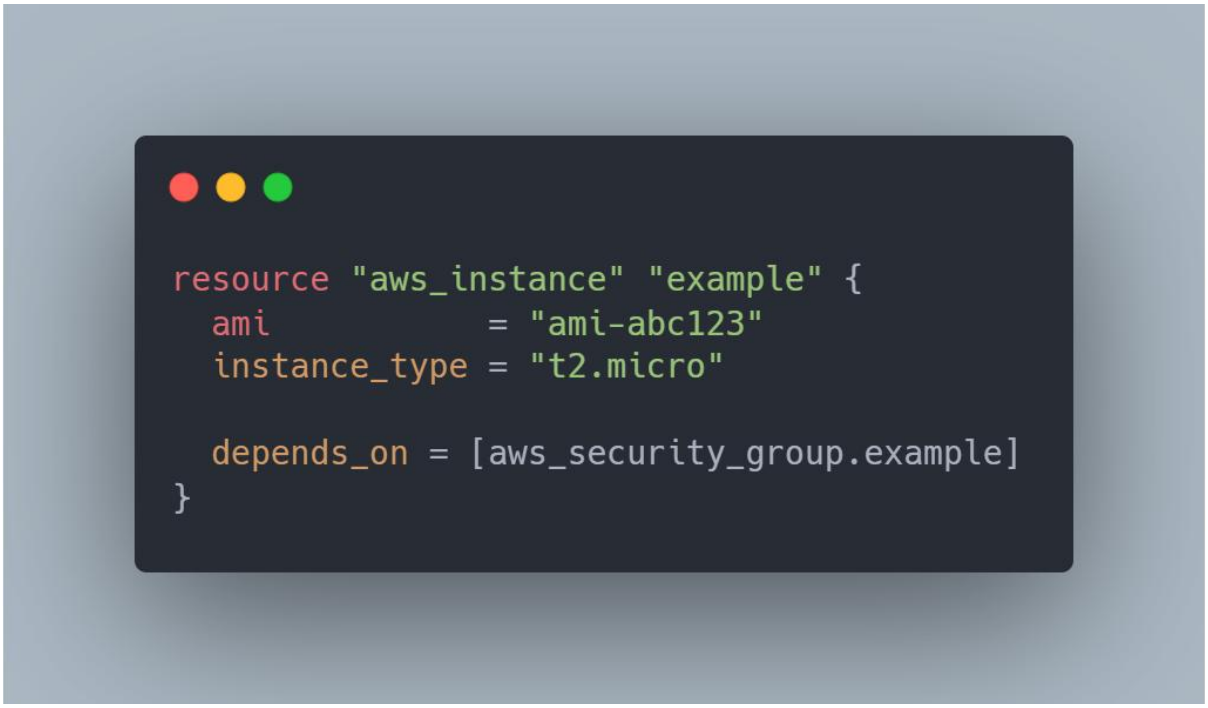
```
# Configure the AWS provider
provider "aws" {
  region = "us-west-2"
}

# Define a variable for the instance type
variable "instance_type" {
  type        = string
  default     = "t2.micro"
}

# Define a resource for an EC2 instance
resource "aws_instance" "example" {
  ami           = "ami-abc123"
  instance_type = var.instance_type
}

# Define an output for the instance IP address
output "instance_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the EC2 instance"
}
```

➢ Meta-Arguments

Meta-arguments are a feature of Terraform that allows you to pass additional arguments to resources and modules. They are used to customize the behavior of resources and modules.

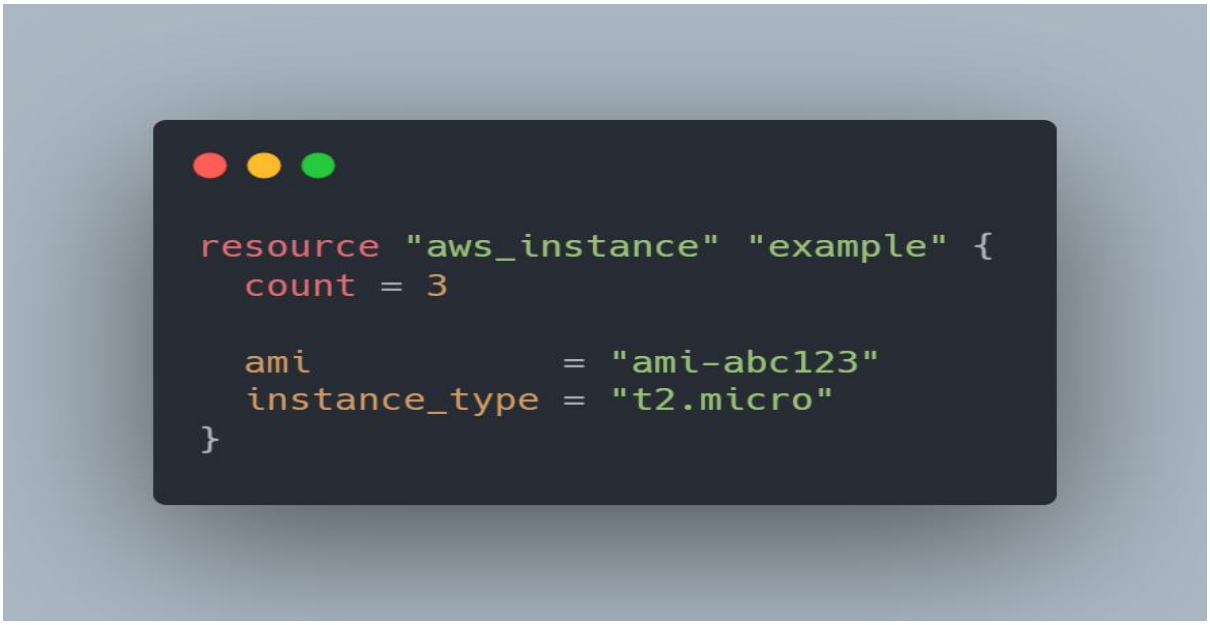
- **depends_on**

The depends_on meta-argument is used to specify that a resource should be created or updated only after another resource has been created or updated.



count

The count meta-argument is used to create multiple instances of a resource.



for_each

The for_each meta-argument is used to create multiple instances of a resource based on a list of values.

Example:

```
variable "instance_types" {
  type = list(string)
  value = ["t2.micro", "t2.small",
}t2.medium"]

resource "aws_instance" "example" {
  for_each = var.instance_types

  ami           = "ami-abc123"
  instance_type = each.value
}
```

First Terraform Project

- Create a basic EC2 instance on AWS
- Provider block
- Resource block
- Output block

Step 1: Create a new directory for your project and navigate to it in your terminal

Step 2: Initialize a new Terraform working directory

terraform init

Step 3: Create a new file called main.tf and add the following code to it

```
# Provider block
provider "aws" {
  region = "us-west-2"
}

# Resource block
resource "aws_instance" "example" {
  ami           = "ami-abc123"
  instance_type = "t2.micro"
}

# Output block
output "instance_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the EC2 instance"
}
```

Step 4: Initialize the Terraform working directory again

terraform init

Step 5: Run the terraform plan command to see the execution plan

terraform plan

Step 6: Run the terraform apply command to create the EC2 instance

terraform apply

Step 7: Run the terraform destroy command to delete the EC2 instance

terraform destroy

Modules

Modules are reusable blocks of Terraform code that can be used to create and manage infrastructure resources. They are similar to functions in programming languages, but for Terraform.

Why Use Modules?

Modules are useful for several reasons:

- **Reusability**: Modules can be reused across multiple Terraform configurations, reducing code duplication and making it easier to manage infrastructure resources.
- **Organization**: Modules can be organized into separate files or directories, making it easier to manage complex Terraform configurations.
- **Flexibility**: Modules can be used to create and manage a wide range of infrastructure resources, from simple resources like EC2 instances to complex resources like AWS Lambda functions.
- **Scalability**: Modules can be used to create and manage large-scale infrastructure deployments, making it easier to manage complex infrastructure environments.

Creating and Using Custom Modules

To create a custom module, follow these steps:

1. **Create a new directory** for the module.
2. **Create a main.tf file** in the module directory.
3. **Add the module code** to the main.tf file.
4. **Use the module** in a Terraform configuration by calling the module using the module keyword.

To use the module in a Terraform configuration, call the module using the module keyword:

Calling Public Modules from the Terraform Registry

To call a public module from the Terraform Registry, use the terraform registry command to find the module, and then use the module keyword to call the module.

Here is an example of calling a public module from the Terraform Registry:

🔄 Terraform Backend

🔹 What is a Backend?

A **backend in Terraform** determines how and where the **state file** (terraform.tfstate) is stored.

By default, Terraform uses the **local backend**, which stores the state file on your local machine. But this is not safe or scalable in teams or production.

🧠 Why is Backend Important?

Terraform state stores the actual mapping of your infrastructure — if it's lost or corrupted, Terraform loses track of everything it deployed. Also, in a team:

- Everyone must use the same state
- Only one person should modify the state at a time (locking)

✅ Use Cases for Backends:

- Remote storage of state (e.g., AWS S3, Terraform Cloud)
- Locking with DynamoDB (prevents concurrent changes)
- Team collaboration and automation in CI/CD

🚀 Example: Remote Backend with AWS S3 and DynamoDB

Directory structure:

main.tf

backend.tf

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "prod/vpc/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-lock"
    encrypt        = true
  }
}
```

Commands

terraform init # Initializes and configures backend

terraform plan

terraform apply

🌍 Terraform Workspace

🔹 What is a Workspace?

Workspaces are a way to manage **multiple state files** within a single Terraform configuration.

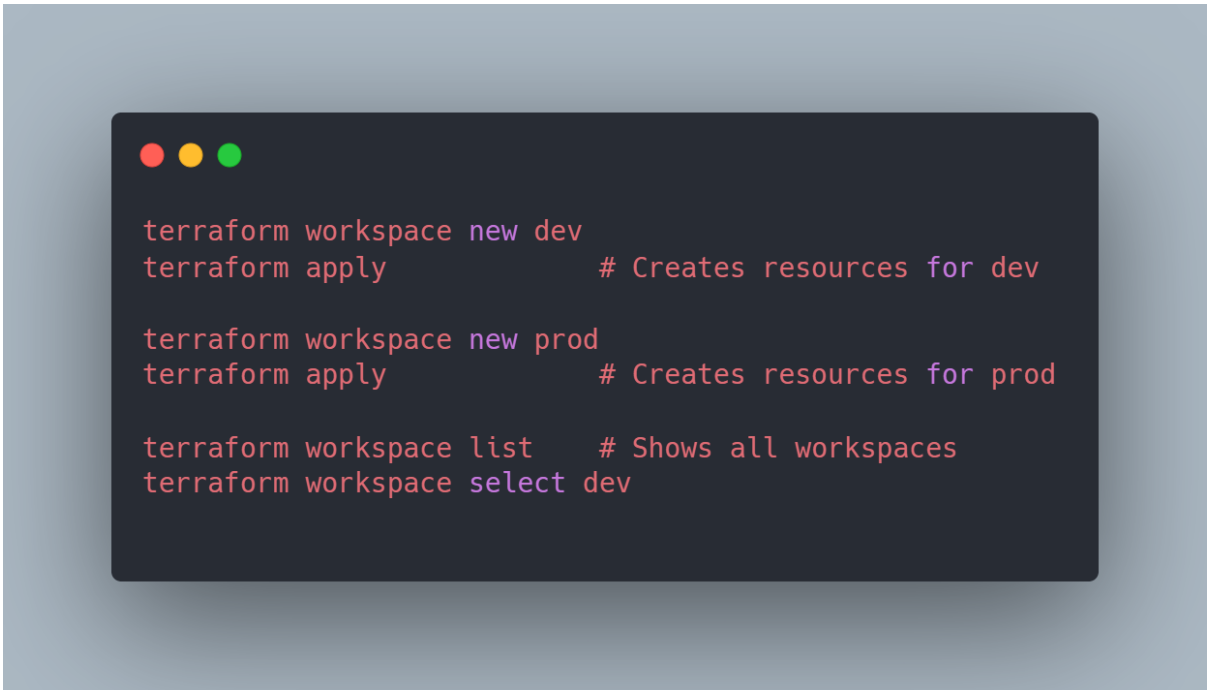
Each workspace has its own .tfstate file, letting you use the same codebase for **dev**, **staging**, and **production** without changing code.

🔄 Why Use Workspaces?

Imagine you have the same infrastructure code for dev, test, and prod environments. Rather than copy-pasting code or using branches, you use workspaces:

- default (created by default)
- dev
- staging
- prod

Each workspace keeps its **own isolated state**, so they don't interfere.



Recommended Docs:

https://registry.terraform.io/

Recommended Youtube Playlist:

https://youtu.be/S9mohJI_R34

https://youtu.be/fgp-t5SqQmM?list=PLdpzxOOAlwvI0O4PeKVV1-yJoX2AqIWuf

https://youtu.be/4JYtAf4M88Y

https://youtu.be/5FJE9HPS85c?list=PL6XT0grm_TfgdaAjTmLb4QedMCCMQHISm