Teraform

Teraform:
Terraform is an open-source **Infrastructure as Code (IaC)** tool that enables you to define, provision, and manage infrastructure resources (like servers, networks, and storage) in a declarative way using configuration files.

Why teraform:
Terraform stands out because it is **cloud-agnostic**, supports multi-cloud environments, and provides a simple, consistent, and scalable way to manage infrastructure with a declarative approach.

Teraform better the python:
While Python supports all clouds via SDKs (like Boto3 for AWS), Terraform is preferred for **infrastructure automation** because of its **declarative syntax**, built-in state management, and modularity, making it easier to track, scale, and manage infrastructure consistently across multi-cloud environments

Crossplane:
Crossplane is an open-source **Infrastructure as Code (IaC)** tool that allows you to manage **multi-cloud** and **hybrid-cloud** resources using **Kubernetes-style APIs**. It enables developers to provision, manage, and compose cloud infrastructure using declarative YAML configurations, just like Kubernetes manages containerized applications.

HashiCorp:
HashiCorp is a company that provides a suite of **open-source tools** focused on infrastructure automation, security, and management, often used in DevOps and cloud-native environments. Some of their most popular tools include

GitHub Codespaces:
GitHub Codespaces is an online development environment that runs in the cloud, allowing you to code directly in your browser or via Visual Studio Code. It is fully integrated with **GitHub** and can be customized with the necessary tools for your project, so you don't need to set up anything on your local machine.

Terraform Lifecycle Phases:

Initialize Terraform: Initialize the Terraform working directory. This command installs the required provider plugins and prepares the environment.

```
terraform init
```

The terraform fmt command is used to **format** your Terraform configuration files (.tf files) according to the standard style conventions defined by Terraform. This helps keep your code consistent and readable.

```
terraform fmt
```

Check the Terraform Plan: It's always a good practice to check what changes Terraform will make before applying them. This will show you the resources it plans to create, modify, or delete.

```
terraform plan
```

Apply the Configuration: Apply the Terraform configuration to create or update the infrastructure based on the `.tf` files. This command will prompt for confirmation before making any changes.

```
terraform apply
```

To skip the confirmation prompt and apply changes automatically, use the `-auto-approve` flag:

```
terraform apply -auto-approve
```

Destroy the Resources (Optional): If you want to destroy the infrastructure created by Terraform, use the `destroy` command:

```
terraform destroy
```

Like `apply`, you can skip the confirmation prompt with `-auto-approve`:

```
terraform destroy -auto-approve
```

Additional Commands:

Validate the Configuration: To check if your Terraform files are valid:

```
terraform validate
```

- **Show the Current State**: To view the current state of your infrastructure:

```
terraform show
```

provider

In Terraform, a **provider** is a plugin that enables Terraform to interact with different cloud services or infrastructure platforms. Providers are responsible for managing the lifecycle of resources, such as creating, reading, updating, and deleting resources on that platform.

Types of Providers:

- Providers can be for cloud services (AWS, Azure, Google Cloud), on-premises solutions (VMware), or other services like databases, DNS, etc.
- Some common providers are:
  - **AWS**: `terraform-provider-aws`
  - **Azure**: `terraform-provider-azurerm`
  - **Google Cloud**: `terraform-provider-google`
  - **Kubernetes**: `terraform-provider-kubernetes`

Initialization:

- Providers need to be configured in the Terraform configuration file and initialized before use.
- The `terraform init` command downloads the necessary provider plugins.

Using Multiple Providers

You can configure multiple providers in a single Terraform configuration file by specifying each provider and aliasing them if needed. This allows you to manage resources across different cloud providers, or even different accounts or regions within the same provider.

Example: AWS and Azure Providers:

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

# Azure Provider Configuration
provider "azurerm" {
  features {}
}

# AWS Resource Example - EC2 Instance
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

```
# Azure Resource Example - Virtual Network
resource "azurerm_virtual_network" "example" {
  name                = "example-vnet"
  location            = "East US"
  resource_group_name = "example-resources"
  address_space       = ["10.0.0.0/16"]
}
```

Using Multiple Regions in the Same Provider

You can use multiple regions within the same provider (for example, AWS) by defining **alias** for the providers. This allows you to manage resources in different regions.

Example: Multiple Regions in AWS

Here's an example where AWS resources are created in two different regions, using provider aliases:

```
        # AWS Provider Configuration for US West region (us-west-1)
provider "aws" {
  region = "us-west-1"
  alias  = "west"
}

# AWS Provider Configuration for US East region (us-east-1)
provider "aws" {
  region = "us-east-1"
  alias  = "east"
}

# EC2 Instance in US West region
resource "aws_instance" "west_instance" {
  provider = aws.west
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

# EC2 Instance in US East region
resource "aws_instance" "east_instance" {
  provider = aws.east
  ami           = "ami-87654321"
  instance_type = "t2.micro"
}
```

Using Multiple Accounts in the Same Provider

If you want to manage resources in different **AWS accounts**, you can use multiple AWS provider configurations with aliases, where each provider configuration specifies different AWS credentials.

Example: Managing AWS Resources in Different Accounts

```
        # AWS Provider Configuration for Account 1
provider "aws" {
  region  = "us-west-2"
  access_key = "account1-access-key"
  secret_key = "account1-secret-key"
  alias   = "account1"
}

# AWS Provider Configuration for Account 2
provider "aws" {
  region  = "us-east-1"
  access_key = "account2-access-key"
  secret_key = "account2-secret-key"
  alias   = "account2"
}

# EC2 Instance in Account 1
resource "aws_instance" "account1_instance" {
  provider = aws.account1
  ami          = "ami-12345678"
  instance_type = "t2.micro"
}

# EC2 Instance in Account 2
resource "aws_instance" "account2_instance" {
  provider = aws.account2
  ami          = "ami-87654321"
  instance_type = "t2.micro"
}
```

Variables

1. Input Variables

- **Definition**: Variables used to pass dynamic values into Terraform configurations. They allow you to parameterize your code.
- **Declaration**: Defined using the `variable` block.

Example:
```
variable "region" {
  default = "us-east-1"
  description = "The AWS region to deploy resources."
}

resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
}
```

Features:

- Default values can be set.
- Input is provided via CLI, `.tfvars` files, or environment variables.

---

2. Local Variables

- **Definition**: Used to simplify and re-use expressions by assigning them a name. These are only valid within the configuration file they are defined in.
- **Declaration**: Defined using the `locals` block.

Example:
```
locals {
  instance_type = "t2.micro"
  ami_id        = "ami-12345678"
}

resource "aws_instance" "example" {
  ami           = local.ami_id
  instance_type = local.instance_type
}
```

Features:

- Great for simplifying complex expressions or re-using values.
- Local variables cannot accept input from outside the configuration.

---

3. Output Variables

- **Definition**: Used to display information after Terraform applies the configuration. These variables expose specific values from your infrastructure.
- **Declaration**: Defined using the `output` block.

Example:

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

-

Features:

- Outputs can display specific information (e.g., resource IDs or IP addresses).
- Useful for passing data to other Terraform modules or external tools.

Key Differences:

Type	Purpose	Scope
Input	Accept values from the user or environment to configure resources.	Global (entire configuration)
Local	Simplify expressions or reuse values within a single configuration.	Local to a module/file
Output	Provide information about the infrastructure after execution.	Displayed after `apply`

Combining Variable Types Example:

```
# Input Variable
variable "instance_type" {
  default = "t2.micro"
}

# Local Variable
locals {
  ami_id = "ami-12345678"
}

# Resource using Input and Local Variables
resource "aws_instance" "example" {
  ami           = local.ami_id
  instance_type = var.instance_type
}

# Output Variable
output "instance_id" {
  value = aws_instance.example.id
}
```

Conditional Expressions

In Terraform, **conditions** (conditional expressions) allow you to dynamically control resource configurations or variable values based on specific criteria. They help make your code flexible and adaptable to different environments or scenarios.

Syntax for Conditional Expressions

```
<condition> ? <true_value> : <false_value>
```

- **`<condition>`**: A boolean expression that evaluates to `true` or `false`.
- **`<true_value>`**: The value returned if the condition is `true`.
- **`<false_value>`**: The value returned if the condition is `false`.

Examples of Using Conditional Expressions

1. Conditional in Variables

- Use conditions to dynamically assign variable values based on input.

```
variable "environment" {
  default = "dev"
}

resource "aws_instance" "example" {
  ami           = var.environment == "prod" ? "ami-prod" : "ami-dev"
  instance_type = "t2.micro"
}
```

- If `environment = "prod"`, the AMI will be `ami-prod`.
- Otherwise, the AMI will be `ami-dev`.

---

2. Conditional in Resource Arguments

- Dynamically enable or disable a resource or feature.

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = var.enable_large_instance ? "t2.large" : "t2.micro"
}
```

- The instance type changes based on the value of `enable_large_instance`.

---

3. Conditional with Outputs

- Use conditions to define what values are exposed in outputs.

```
output "instance_id" {
  value = var.environment == "prod" ? aws_instance.prod.id :
aws_instance.dev.id
}
```

- Outputs the instance ID based on the environment.

Using `count` for Conditional Resource Creation

Terraform doesn’t directly support enabling/disabling resources, but you can control resource creation using the `count` parameter with a conditional.

```
variable "create_instance" {
  default = true
}

resource "aws_instance" "example" {
  count         = var.create_instance ? 1 : 0
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

- If `create_instance` is `true`, one instance is created.
- If `create_instance` is `false`, no instances are created.

Using `for_each` with Conditions

When working with multiple resources, use `for_each` to conditionally filter resources.

```
variable "enabled_tags" {
  default = {
    prod = true
    dev  = false
  }
}
```

```
resource "aws_instance" "example" {
  for_each = { for k, v in var.enabled_tags : k => v if v }
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = each.key
  }
}
```

- Creates resources only for keys where the value is `true`.

Best Practices for Conditional Logic in Terraform

1. **Keep Conditions Simple**:
   - Avoid overly complex conditions that make code harder to read.
2. **Use Descriptive Variable Names**:
   - Clearly indicate what the condition represents (e.g., `is_production` instead of `env`).
3. **Test Scenarios**:
   - Ensure conditions work as expected for all possible inputs.
4. **Combine Conditions with Variables**:
   - Store condition results in variables for reuse.

Module

In Terraform, **modules** are a way to group related resources and configurations into reusable components. Modules make it easier to organize and reuse infrastructure code, especially in larger projects or when working across multiple environments.

Why Use Modules?

- **Reusability**: Define a module once and use it in multiple configurations.
- **Organization**: Break down complex infrastructure into smaller, manageable pieces.
- **Consistency**: Standardize resource creation across environments.

Module structure:
A Terraform module structure typically organizes files and directories to make the module reusable, maintainable, and easy to understand. Here's a standard structure for a Terraform module

Standard Module Directory Structure:

```
my_module/
├── main.tf         # Core resource configurations
├── variables.tf     # Input variables for the module
├── outputs.tf       # Output values from the module
├── versions.tf      # (Optional) Provider and Terraform version constraints
├── README.md        # (Optional) Documentation for the module
└── examples/        # (Optional) Example usage of the module
    └── example.tf   # Example file demonstrating how to use the module
```

Advanced: Nested Modules:

```
my_project/
├── modules/
│   ├── network/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   ├── compute/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
└── main.tf         # Root module calling network and compute modules
```

State file
The Terraform **state file** (`terraform.tfstate`) is a file that stores information about the real-world infrastructure Terraform manages. It acts as a record of all resources Terraform has created, updated, or deleted, and helps Terraform track the current state of the infrastructure to determine what changes are needed.

remote backend
A **remote backend** in Terraform is a way to store the state file (`terraform.tfstate`) in a remote, centralized location instead of locally on your computer. It allows better collaboration, security, and management of the state file for teams or complex infrastructure setups.

Example:
```
        terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "prod/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-lock-table"  # Enables state locking
  }
}
```

Common Remote Backends

- **Amazon S3** (with DynamoDB for locking)
- **Azure Blob Storage**
- **Google Cloud Storage (GCS)**
- **Terraform Cloud** or **Terraform Enterprise**
- **Consul**
- **PostgreSQL**
- **Artifactory**

State locking:
State locking is a feature in Terraform that ensures **only one process or user can modify the state file at a time**. It prevents issues like corruption, race conditions, or conflicts when multiple users or automation tools attempt to perform operations on the same infrastructure.

Backends That Support State Locking

Some remote backends natively support state locking:

- **AWS S3** (with DynamoDB)
- **Azure Blob Storage**
- **Google Cloud Storage (GCS)**
- **Terraform Cloud** or **Enterprise**
- **Consul**

For local backends, locking is not supported.

Commands to Manage State Locks

1. **Force Unlock**: If a lock is stuck, you can forcefully unlock it:
   ```
   terraform force-unlock LOCK_ID
   ```

Provisioners:
Provisioners in Terraform are used to execute scripts or commands on a resource after it is created or modified. They allow for additional configuration of infrastructure beyond what is defined in the Terraform resource.

Types of Provisioners

1. **File Provisioner**: Uploads files or directories to a target machine.
2. **Local-Exec Provisioner**: Executes a command on the machine where Terraform is running.
3. **Remote-Exec Provisioner**: Executes commands on the target resource via SSH or WinRM.

Examples

1. File Provisioner

Uploads a configuration file to a remote server.

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

  provisioner "file" {
    source      = "config.yaml"
    destination = "/etc/myapp/config.yaml"
  }
}
```

---

2. Local-Exec Provisioner

Runs a command locally after resource creation.

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

  provisioner "local-exec" {
    command = "echo Instance ${self.id} created >> log.txt"
  }
}
```

---

3. Remote-Exec Provisioner

Executes a script on a remote machine using SSH.

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

  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt update",
      "sudo apt install -y nginx",
    ]
  }
}
```

Provisioner Triggers

Provisioners run during resource creation or destruction:

1. **Creation-time**: Runs after the resource is created.
2. **Destruction-time**: Use the `when = "destroy"` argument to run commands during resource destruction.

```
provisioner "local-exec" {
  command = "echo Destroying instance ${self.id}"
  when    = "destroy"
}
```

Terraform workspace:
A **Terraform workspace** is an isolated environment within a single Terraform configuration. Workspaces allow you to manage multiple versions of infrastructure (e.g., `dev`, `staging`, `prod`) within the same codebase, each with its own state.

Workspace Commands

Create a new workspace:

```
terraform workspace new dev
```

List all workspaces:

```
terraform workspace list
```

Switch to a different workspace:

```
terraform workspace select prod
```

Show the current workspace:

```
terraform workspace show
```

Delete a workspace:

```
terraform workspace delete dev
```

Terraform lookup Function

The lookup function in Terraform is used to retrieve the value associated with a given key from a map or object. If the key does not exist, you can optionally specify a default value to return instead.

Syntax

```
lookup(map, key, default)
```

- `map`: The map or object from which the value will be looked up.
- `key`: The key whose value you want to retrieve.
- `default`: The value to return if the key is not found in the map (optional).

HashiCorp Vault

HashiCorp Vault is a tool designed to securely store and manage sensitive data such as secrets, tokens, passwords, and encryption keys. The **Terraform Vault provider** allows Terraform users to interact with HashiCorp Vault for managing secrets and configuration data in a secure way.