

## **1.what is Terraform**

Terraform is an open-source tool for building, changing, and versioning infrastructure safely and efficiently. It is an Infrastructure as Code (IAC) tool that allows you to describe your infrastructure using declarative code, rather than manual configuration. This makes it easier to manage, version control, and collaborate on infrastructure changes. Terraform can manage a wide range of resources, including virtual machines, DNS entries, databases, and more, across multiple cloud providers such as AWS, Azure, Google Cloud, and others. Terraform automates the process of provisioning and managing infrastructure, reducing the risk of manual errors and speeding up the process.

## **2.Why Terraform**

Terraform is a popular Infrastructure as Code (IAC) tool that is used to manage and provision infrastructure resources. It offers several benefits, including:

1. Multi-cloud support: Terraform supports multiple cloud providers, including AWS, Azure, Google Cloud, and more, making it easy to manage resources across multiple cloud environments.
2. Infrastructure as code: Terraform uses declarative code to manage infrastructure, making it easier to manage, version control, and collaborate on infrastructure changes.
3. Reproducibility: Terraform's state management features allow you to track changes to your infrastructure and maintain a consistent state, making it easier to replicate and test environments.
4. Automation: Terraform automates the provisioning of infrastructure resources, reducing the risk of manual errors and speeding up the process.
5. Scalability: Terraform supports scaling infrastructure resources up or down as needed, making it easier to manage large, complex infrastructure environments.

Overall, Terraform is a powerful tool for managing infrastructure that provides benefits for both operations and development teams.

## **3.Use of Terraform in Azure**

Terraform is commonly used in Azure to manage infrastructure resources in a consistent and efficient manner. Some of the main use cases for Terraform in Azure include:

1. Provisioning resources: Terraform can be used to provision and manage various resources in Azure, such as virtual machines, storage accounts, databases, and more.
2. Multi-cloud deployments: Terraform allows you to manage resources across multiple cloud providers, making it easier to deploy and manage hybrid cloud environments.
3. Infrastructure as code: Terraform allows you to define infrastructure as code, making it easier to version control, collaborate, and automate infrastructure changes.
4. Scalability: Terraform makes it easy to scale infrastructure resources up or down as needed, making it easier to manage large, complex infrastructure environments.
5. Disaster recovery: Terraform can be used to manage disaster recovery plans, ensuring that critical infrastructure can be quickly restored in the event of an outage.

Overall, Terraform is a valuable tool for managing infrastructure in Azure and provides benefits for both operations and development teams.

#### 4. Terraform workflow

The typical workflow for using Terraform includes the following steps:

1. Write Terraform configuration files: In this step, you write Terraform configuration files that describe the desired state of your infrastructure resources. The configuration files are written in HashiCorp Configuration Language (HCL).
2. Initialize Terraform: Before you can use Terraform to manage resources, you need to initialise the Terraform environment by running the **terraform init** command. This command downloads the required provider plugins and sets up the Terraform state file.
3. Plan Terraform changes: The **terraform plan** command allows you to preview the changes that Terraform will make to your infrastructure resources. This step is important because it allows you to catch any errors or issues before they occur.
4. Apply Terraform changes: The **terraform apply** command is used to apply the changes that were previewed in the previous step. Terraform will make the necessary changes to your infrastructure to bring it into the desired state.

5. **Manage Terraform state:** The Terraform state file is used to track the current state of your infrastructure resources. You can use the **terraform state** command to manage and manipulate the state file, such as importing or exporting resources.
6. **Destroy Terraform resources:** When you are finished with a Terraform-managed resource, you can use the **terraform destroy** command to remove it from your infrastructure.

This basic Terraform workflow can be repeated multiple times as you make changes to your infrastructure, allowing you to manage your infrastructure resources in a consistent and efficient manner.

### 5.What are the main components of a Terraform configuration file?

A Terraform configuration file contains various components that define and configure infrastructure resources. The main components of a Terraform configuration file are:

1. **Provider:** Specifies which provider to use to manage the resources defined in the configuration file.
2. **Variables:** Defines variables that can be used in the configuration file. Variables can be used to provide values that can change between deployments or environments.
3. **Data Sources:** Retrieves data from an external source, such as an API, and stores it in Terraform state for use within the configuration.
4. **Resources:** Defines the desired state of infrastructure resources, such as virtual machines, storage accounts, and databases.
5. **Outputs:** Displays results from the Terraform configuration, such as the public IP address of a virtual machine, after the resources have been created.
6. **Modules:** Reusable components that encapsulate a set of related resources and configurations.

Each component is written in HashiCorp Configuration Language (HCL) and must be placed in a specific section of the Terraform configuration file for Terraform to properly process it. The overall structure and organisation of the configuration file can have a significant impact on the readability, maintainability, and scalability of your Terraform infrastructure.

## 5.what is provider in Terraform

In Terraform, a provider is a plugin that implements a specific resource type and performs the necessary API calls to manage that resource type within the target infrastructure. The provider acts as an interface between Terraform and the underlying infrastructure, allowing Terraform to interact with the infrastructure to create, update, and delete resources.

Each provider supports a specific set of resource types and API calls, and can be used to manage resources within a specific infrastructure, such as AWS, Azure, Google Cloud, or a private cloud. Terraform supports many different providers, allowing it to manage resources across multiple cloud providers and on-premises infrastructure.


In a Terraform configuration file, the provider block is used to specify which provider should be used to manage the resources defined in the file. The provider block includes information such as the provider name, version, and any required authentication credentials.

## 6.How to define provider in Terraform

To define a provider in Terraform, you can use the **provider** block in your Terraform configuration file. The **provider** block contains information about the provider you want to use, including the provider name, version, and any necessary authentication credentials.

Here is an example of how to define a provider in Terraform:

java

 Copy code

```
provider "azure" {  
  version = "~> 2.0"  
  subscription_id = var.subscription_id  
  client_id = var.client_id  
  client_secret = var.client_secret  
  tenant_id = var.tenant_id  
}
```

In this example, the **provider** block is defining the Azure provider and specifying the version number. The block also includes variables for the required authentication credentials, such as the subscription ID, client ID, client secret, and tenant ID.

It is important to note that Terraform must have access to the necessary provider plugins in order to use the provider. The **terraform init** command can be used to download the necessary plugins for your Terraform environment.

## 7. Creating a Resource Group

```
terraform {  
  
  required_providers {  
  
    azurerm = {  
  
      source = "hashicorp/azurerm"  
  
      version = "3.37.0"  
  
    }  
  
  }  
  
}
```

```

provider "azurerm" {

  subscription_id = "xxxxxxxxxxxxxxxxxxxx"

  tenant_id = "xxxxxxxxxxxxxxxxxxxx"

  client_id = "xxxxxxxxxxxxxxxxxxxx"

  client_secret = "xxxxxxxxxxxxxxxxxxxx"

  features {}
}

resource "azurerm_resource_group" "Example" {

  name="ResourceGroupName"

  location="Central US"

}

```

## 8.Creating the Storage Account

```

terraform {

  required_providers {

    azurerm = {

      source = "hashicorp/azurerm"

      version = "3.37.0"

    }

  }

}

```

```
}

provider "azurerm" {

  subscription_id = "xxxxxxxxxxxxxxxxxxxx"

  tenant_id = "xxxxxxxxxxxxxxxxxxxx"

  client_id = "xxxxxxxxxxxxxxxxxxxx"

  client_secret = "xxxxxxxxxxxxxxxxxxxx"

  features {}
}

resource "azurerm_storage_account" "Example" {

  name                = "StorageAccountName"

  resource_group_name = "ResourceGroupName"

  location            = "Central US"

  account_tier        = "Standard"

  account_replication_type = "LRS"

}
```

## 9.VirtualNetwork & Subnet Creation

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.37.0"
    }
  }
}

provider "azurerm" {
  subscription_id = "xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  tenant_id = "xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  client_id = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  client_secret = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  features {}
}

resource "azurerm_resource_group" "Appgrp" {
  name="Terraform-Rg1"
  location="Central US"
}

resource "azurerm_virtual_network" "example" {
  name                = "app-network"
  location             = "Terraform-Rg1"
  resource_group_name = "Central US"
  address_space       = ["10.0.0.0/16"]

  subnet {
    name                = "subnetA"
    address_prefix      = "10.0.0.0/24"
  }

  subnet {
    name                = "subnetB"
    address_prefix      = "10.0.1.0/24"
  }
}
```



```
depends_on = [  
    azure_rm_resource_group.Appgrp  
]  
}
```