Terraform - Basics
PART-1

1. What is Infrastructure as Code (IaC)?

- **Definition**:
  - IaC is the practice of managing and provisioning computing infrastructure through machine-readable configuration files, rather than physical hardware configuration or interactive configuration tools.
- **Key Benefits**:
  - **Consistency**: Ensures infrastructure setup is repeatable without human errors.
  - **Version Control**: Infrastructure configurations can be versioned using tools like Git.
  - **Automation**: Eliminates manual steps, reducing deployment time and effort.
  - **Scalability**: Easily replicate and scale environments (e.g., Dev, QA, Production).
- **IaC Tools**:
  - Terraform, Ansible, Chef, Puppet, CloudFormation (specific to AWS).

2. Overview of Terraform

- **What is Terraform?**
  - Terraform is an open-source IaC tool by HashiCorp that allows users to define and provision infrastructure in a safe and efficient way.
  - It uses a declarative language (HCL) to describe the desired end-state of infrastructure.
- **How Terraform Works**:
  - Connects with **Providers** (e.g., AWS, Azure, Google Cloud) to manage infrastructure.
  - Executes commands like `plan`, `apply`, and `destroy` to manage resources.

3. Features and Benefits

- **Platform Agnostic**: Supports multiple cloud providers like AWS, Azure, and GCP, as well as on-premises tools like VMware.
- **Declarative Language**: Define "what" you want, and Terraform determines "how" to achieve it.
- **Execution Plan**:
  - Preview the changes Terraform will make before applying them.
- **Resource Graph**:
  - Automatically handles dependencies between resources.

- **State Management**:
  - Tracks infrastructure changes using a state file.
- **Reusable Modules**:
  - Create reusable templates for common setups.

4. Use Cases

- **Cloud Infrastructure Provisioning**:
  - Creating instances, networks, and storage in AWS, Azure, or GCP.
- **Multi-Cloud Deployments**:
  - Managing resources across multiple cloud platforms.
- **Automation of Repetitive Tasks**:
  - Automating the provisioning of servers and load balancers.
- **Disaster Recovery**:
  - Quickly recreate infrastructure from the state file.
- **Versioning Infrastructure**:
  - Track changes and roll back infrastructure to previous states.

5. Installing and Setting up Terraform

- **Prerequisites**:
  - Basic knowledge of CLI tools.
  - Internet connection for provider APIs.
- **Installation Steps**:
  - **Windows**:
    1. Download the Terraform binary from the official Terraform website.
    2. Extract and place it in a folder included in the system's PATH (e.g., `C:\Terraform`).
    3. Verify installation with `terraform --version`.
  - **macOS**:
    1. Install via Homebrew: `brew install terraform`.
    2. Verify installation with `terraform --version`.
  - **Linux**:
    1. Download the binary from the Terraform website.
    2. Extract and move it to `/usr/local/bin/`.
    3. Verify with `terraform --version`.
- **Setting Up Terraform**:
  - Create a working directory for your configurations.
  - Install any necessary plugins (automatically done during `terraform init`).

6. Core Concepts

1. **Providers**:
    - Providers are responsible for interacting with cloud APIs and services.
    - Examples:
        - `aws`: Manages AWS services.
        - `azurerm`: Manages Azure resources.
        - `google`: Manages Google Cloud resources.

    Configuration Example:

    ```
    resource "azurerm_resource_group" "example" {
         name = "example-resources"
         location = "West Europe"
    }
    ```

To Understand this Better Let’s take Our Own Telugu Film Industry As an Example

Providers are like **film producers** who fund and manage the movie. Just as producers decide which actors, locations, and technicians to hire, providers decide which cloud services to manage.

2. **Resources**:
    - Resources represent the infrastructure components you want to manage (e.g., Virtual Networks, Blobs, VMs).

    ```
    # Create a virtual network within the resource group

    resource "azurerm_virtual_network" "example" {
     name = "example-network"
     resource_group_name = azurerm_resource_group.example.name
    location = azurerm_resource_group.example.location
     address_space = ["10.0.0.0/16"]
    }
    ```

    In TFI Context, Resources are like the **actors, technicians, and crew members** involved in a movie. Each resource plays a specific role in the project.

    - Example:
        - An **Azure VM** is like the **hero** (e.g., Allu Arjun), central to the story.
        - An **Storage Account** is like the **production equipment** (e.g., cameras, props) that stores and transfers data.

        - A **database** is like the **screenplay** that stores the entire story structure.

3. **State**:
    - **Definition**: A state file keeps track of all the infrastructure Terraform manages.
    - **Local vs Remote State**:
        - Local: Saved on your machine.
        - Remote: Stored in a backend (e.g., S3, Azure Blob).
    - Use Cases:
        - Synchronize infrastructure changes.
        - Prevent drift between real-world infrastructure and configuration.

The state is like the **script or continuity notes** that track what scenes have been shot and what remains to be done. It ensures consistency in the filmmaking process.

4. Without state:
    - A scene could be re-shot unnecessarily, wasting resources.
5. With state:
    - The team knows which scenes (resources) are complete and can plan the next steps efficiently.

6. **Modules**:
    - Modules are containers for multiple resources that are used together.
    - Benefits:
        - Reuse and share infrastructure configurations.
        - Simplify management of complex setups.
    - Example:
        - A module for setting up a VPC with subnets and security groups.

Modules are like **departments in a film production**, where each department is responsible for a specific task.

- Examples:
    - **Cinematography Department**: Responsible for the camera setup and lighting (like a module managing VPC and subnets).
    - **Costume Department**: Manages all outfits and accessories (like a module setting up user permissions).
    - **Editing Team**: Processes raw footage into the final cut (like a module configuring post-production tools).
- These departments (modules) can be reused across multiple movies (projects).
    -

Terraform Learning Lab 1: Provisioning an Azure Resource Group

Objective:

Learn the basics of Terraform by creating and managing an Azure Resource Group.

Prerequisites:

- Azure account with access to the portal.
- Terraform installed on your local machine.
- Azure CLI installed and authenticated.

Step 1: Set Up the Lab Environment

- Open a terminal or command prompt.
- Authenticate Azure CLI:

```
az login
```

- Create a directory for your Terraform project:

```
mkdir terraform-azure-lab1

cd terraform-azure-lab1
```

Step 2: Write the Terraform Configuration

- Create a file named `main.tf:`

```
# Provider configuration

terraform {

required_providers {

      azurerm = {
```

```
      source = "hashicorp/azurerm"

      version = "~> 3.0"

 }

}

required_version = ">= 1.0.0"

}

provider "azurerm" {

      features {}

}

# Resource Group configuration

resource "azurerm_resource_group" "example" {

name = "example-resource-group"

location = "East US"

}
```

Step 3: Initialize and Apply Terraform

- Initialize Terraform to download the required provider plugins:

  ```
  terraform init
  ```

- Validate the configuration:

  ```
  terraform validate
  ```

- Format your files:

  ```
  terraform fmt
  ```

- Preview the resources to be created:

  ```
  terraform plan
  ```

- Apply the configuration:

  ```
  terraform apply
  ```

  Confirm with yes when prompted.

Step 4: Verify the Resource in Azure

1. Log in to the Azure Portal.
2. Navigate to "Resource Groups."
3. Verify the creation of `example-resource-group` in the "East US" region.

Step 5: Clean Up Resources

```
terraform destroy
```

Confirm with yes when prompted.