What is Terraform?

- Terraform is a robust Infrastructure as Code (IaC) tool designed to create, modify, and manage infrastructure with safety and ease.
- It's Open Source
- Uses **Declarative language**

So, Terraform is a tool for **infrastructure provisioning .** But What does this mean exactly?

> Let's Say you just started a project where you create an application and you want to set up an infrastructure from scratch where this application will run .
>
> How does your infrastructure look like ?
>
> Let's say you want to spin up several servers where you will deploy your five microservice application that make up your application as docker containers
>
> Also you're going to deploy a database container
>
> You decide to use **AWS** platform to build your whole infrastructure on

first step will be to go to AWS and prepare the setup so the applications can be deployed there , This means:

- You create your private network space
- You create an EC2 server instances
- You install docker on each one of those plus any other tools that you might need for your application
- You set up security on your servers like firewalls etc

Once the infrastructure is prepared, **You can now deploy** your docker applications or docker containers on that prepared infrastructure

In the setup process, there are **two distinct tasks**:

1. Provisioning the Infrastructure: This step involves preparing the environment where the application will be deployed (networking, servers, databases, etc.).
2. Deploying the Applications: Once the infrastructure is ready, the application is deployed on it.

These tasks can be handled by different teams or individuals. For example, a DevOps team member may provision the infrastructure, while a developer deploys the application.

Terraform plays a critical role in the infrastructure provisioning phase by automating the setup of the infrastructure. It allows DevOps teams to define infrastructure as code, ensuring consistent and repeatable deployments.

Terraform is used for the first part, where you provision the infrastructure to prepare it for the application deployment

- Creating the VPC
- Spinning up the servers
- Creating the security
- The AWS users and permissions
- Maybe installing docker , specific versions on servers etc

And Obviously , all of this needs to be done in a correct order because one task may depends on the other

Difference between Terraform and Ansible and When to use each?

They seem to be doing the same thing especially if you use the official definitions or official documentation, they “sound” like the same tools . So the question is pretty logical , What is the difference between them ? And which one should you use for your project ?

First Of All, **Terraform & Ansible** are both **Infrastructure as a Code .**

This means that they are both used to automate provisioning , configuring and managing the infrastructure

- Terraform is mainly **infrastructure provisioning tool** ; that's where its mains power lies . Terraform also has possibilities to deploy applications in other tools on that infrastructure
- Ansible on the other hand , is mainly a **configuration tool** , so once the infrastructure is provisioned and it's thereAnsible can now be used to configure it, deploy applications, Install and Update Software on that infrastructure etc

It's a common practice where DevOps engineers use the combination of these tools to cover the whole set up end to end using both for their own strengths instead of just using one tool

What is Terraform used for ?

Now, let's go back to our use case , where we created the infrastructure using Terraform , and on AWS provisioned successfully for your project , and you deployed the application on it

Now We decide that you want to add 5 more servers to the existing infrastructure to deploy more micro services , because your team developed some more features and they need to be deployed , and you also need to add some security configuration or maybe remove some stuff that you configured at the beginning

So Now we are in the **phase of managing the existing infrastructure** , adding some stuff, reconfiguring , removing etc *

By Using Terraform, You can make such adjustments to infrastructure pretty easily

And this task of managing the infrastructure is just as **important** , because once you've created the initial infrastructure for your project , you will be **continually adjusting and changing it** , and because of that, you also need an **automation tool** that will do most of the heavy lifting for you so you don't have to manually configure .

So once you are set up with Terraform to create and change or maintain your infrastructure , another useful thing or a common use case could be: **Replicating that infrastructure**

After testing your development environment, you can use **Terraform** to quickly replicate the setup for production. Terraform allows you to reuse the same code to spin up identical **production** and **staging** environments, making the process simple and ensuring consistency across all environments. This approach also keeps the development environment for testing and updates before launching into production.

Terraform Architecture

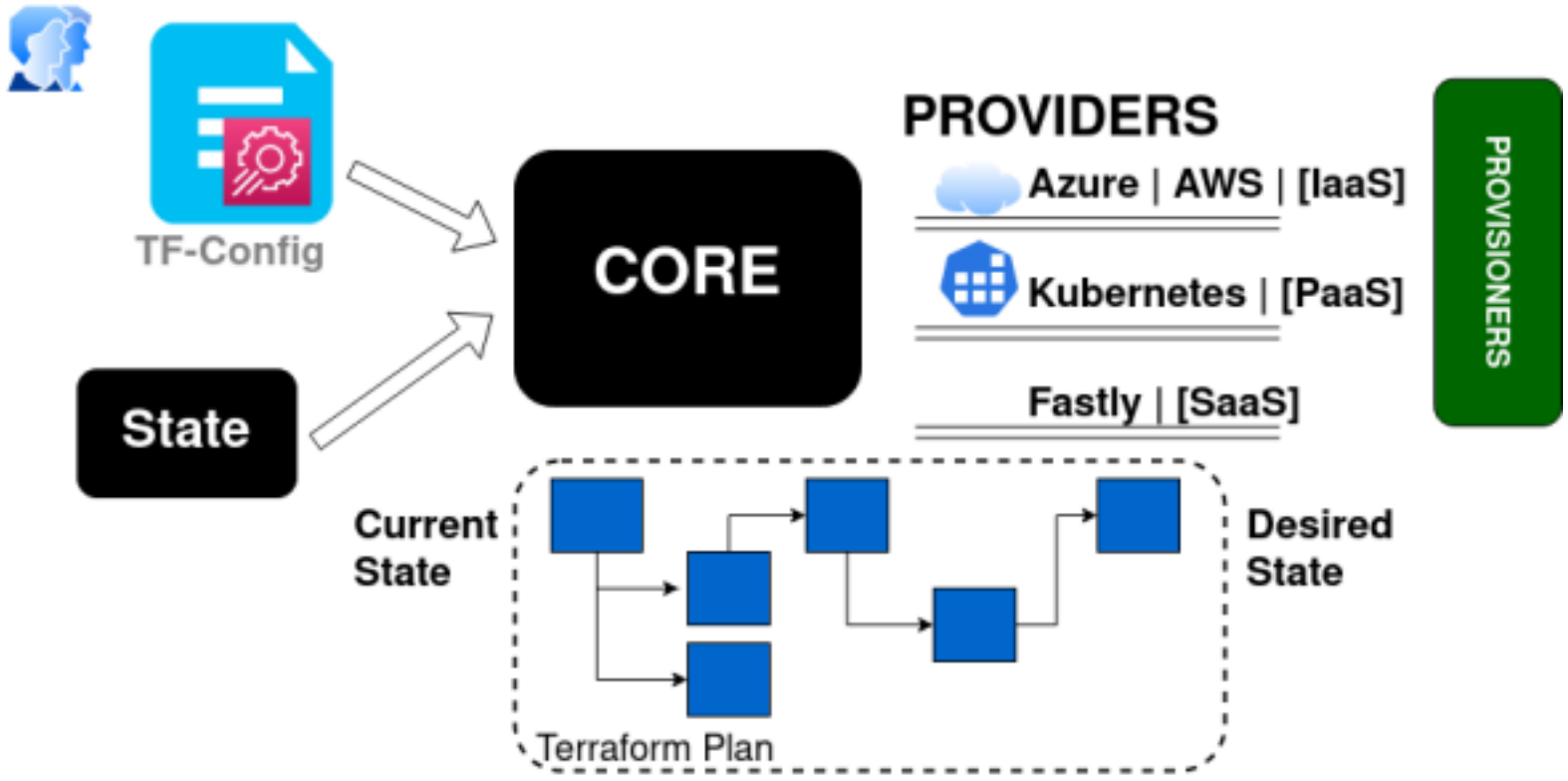
How does Terraform do all this? How does Terraform actually connect to this infrastructure provider platforms and use all these technologies to provision stuff?

For example, How does Terraform connect to AWS to create virtual space , start EC2 instances , configure networking etc ?

To perform its tasks, Terraform relies on two main components in its architecture:

1. **Terraform Core**: The engine that drives Terraform's functionality.
2. **Input Sources**: Terraform Core uses two inputs:
   a. Terraform Configuration (TF-Config): This is written by the user and defines what needs to be created or provisioned.
   b. Terraform State: Tracks the current state of the infrastructure and helps Terraform manage updates and changes

Where Terraform keeps up-to-date state of HOW the current setup of the infrastructure looks like



Terraform Core takes the two inputs—**TF_Config and State**—and determines the plan of action. It compares the current state (the existing infrastructure) with the desired state (defined in the configuration). If there are differences, Terraform figures out what changes are needed to reach the desired state, including what needs to be created, updated, or deleted, and in what order these actions should occur in the infrastructure.

Terraform State Management

Understanding the State File:

- The state file in Terraform is a JSON document that stores details about the resources and data objects deployed via Terraform
- It encompasses metadata and crucial information about each resource.
- Given that the state file can hold sensitive data, it is important to ensure it is securely encrypted and protected.

```
{
  "version": 4,
  "terraform_version": "1.0.0",
  "serial": 1,
  "lineage": "your-lineage-here",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_s3_bucket",
      "name": "example_bucket",
      "provider": "provider.aws",
      "instances": [
        {
          "attributes": {
            "acl": "private",
            "bucket": "example-bucket",
            "force_destroy": false,
            "id": "example_bucket",
            "region": "us-east-1",
            "tags": {}
          },
          "private": "bnVsbA=="
        }
      ]
    }
  ]
}
```


Local Backend vs. Remote Backend:

Storing the State File:

- Local Backend: The state file is stored within the working directory of the project
- Remote Backend: The state file is stored in a remote object store or a managed service like Terraform Cloud

Local Backend's advantages:

- Easy to set up and use
- Stores the state file alongside your code

Local Backend's disadvantages:

- Stores sensitive values in plain text
- Not suitable for collaboration
- Requires manual interaction for applying changes

Remote Backend's advantages:

- Encrypts sensitive data
- Allows collaboration among multiple developers
- Supports automation through CI/CD pipelines

Remote Backend's disadvantages:

- Increased complexity compared to the local backend

Self-managed Backend (AWS S3):

For a self-managed backend using AWS S3 in Terraform, you set up an S3 bucket to store the state file and a DynamoDB table to handle state locking. The configuration within Terraform involves specifying both resources

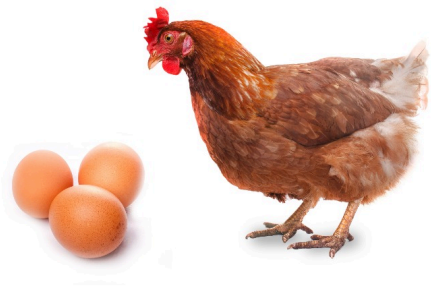
- S3 Bucket: This bucket is designated to hold the Terraform state files. The state file includes all the necessary information about the managed infrastructure, ensuring persistence and consistency of Terraform operations.

```
terraform {
  backend "s3" {
    bucket         = "devops-directive-tf-state"
    key            = "tf-infra/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-state-locking"
    encrypt        = true
  }
}
```

- DynamoDB Table: This table is used for state locking to prevent the execution of concurrent Terraform apply operations that could potentially corrupt the state file (because here we are accessing it using multiple accounts). The locking mechanism ensures that only one set of changes can be applied at a time, safeguarding the integrity of your infrastructure's configuration.

Issue: how we will provision these s3 buckets and dynamodb tables given that we want to provision everything with IaC, but we don't have these resources provisioned yet.

Chicken and eggs Problem !!



Solution: Bootstrapping Process for AWS S3 Backend:(provision these resources and then import them into our configuration so that even the remote backend resources can be managed by terraform as well)

The second component of Terraform's architecture is its **Providers**, which are responsible for interacting with specific technologies. These can include:

- Cloud Providers like AWS or Azure (IaaS platforms).
- High-level components like Kubernetes (PaaS) or even SaaS tools.

Terraform allows you to provision infrastructure at various levels. For example, you can create AWS infrastructure, deploy Kubernetes on top of it, and then configure services or components within the Kubernetes cluster—all using the appropriate Terraform providers. These providers enable seamless integration across different platforms and services.

Terraform has over 1000 providers for these different technologies

So Once the Core creates an execution plan based on the input from Config file and State , It then uses providers for specific technologies to execute the plan , to connect to those platforms , and to actually carry out those execution steps .

To also have an idea of how Terraform **Configuration File** looks like , this is an example where you see AWS provider is configured , and through the provider, you now have 2 AWS ressource like VPC

```
# Configure the AWS Provider
provider "aws" {
  version = "~> 2.0"
  region  = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

The same way , You have Kubernetes provider here configures , and through that you can create a Kubernetes namespace ressource where you pass some attributes

```
# Configure the Kubernetes Provider
provider "kubernetes" {
  config_context_auth_info = "ops"
  config_context_cluster   = "mycluster"
}

resource "kubernetes_namespace" "example" {
  metadata {
    name = "my-first-namespace"
  }
}
```

The syntax is very intuitive , basically you define the resource of a certain technology or certain provider created , and then you define its attribute and that's what Terraform will create or do for you.

Declarative VS Imperative

I mentioned in the beginning about the declarative approach that Terraform's Configuration Files are written in . And this is important to understand !

Terraform uses a **declarative approach** in its configuration files, which is important to understand. Instead of specifying the exact steps to create infrastructure (like a VPC or EC2 instances), you define the **desired end state** in your configuration file. For example, you might specify:

- 5 servers with a certain network configuration
- An AWS user with specific permissions

Terraform then takes care of executing the necessary steps to reach that end state.

In contrast, an **imperative approach** would require you to specify each step to create or modify resources to reach the desired outcome.

Initially, the difference between declarative and imperative might seem small, but it becomes crucial when updating infrastructure. For example, in a declarative approach, you can simply modify the configuration to **add or remove servers**, and Terraform will automatically figure out the required changes.

Terraform Basic Commands

Once you've created the Terraform configuration file that defines your desired infrastructure on AWS, you can use Terraform commands to take action. The process is simple and involves the following commands:

- **`terraform init:`** Initializes your project , download the necessary providers and store them in the .terraform directory , The .terraform.lock.hcl file contains information about the installed dependencies and providers
- **`terraform refresh`**: Updates the state to reflect the latest infrastructure state on AWS.
- **`terraform plan`**: Generates an execution plan by comparing the current state with your desired configuration. This gives you a **preview** of what changes will be made (e.g., creating, updating, or deleting resources) without actually making any changes.
- **`terraform apply`**: Executes the plan and applies the changes to your infrastructure, provisioning the resources defined in your configuration. It first refreshes the state, creates a plan, and then applies it in one step.
- **`terraform destroy`**: Removes all resources associated with the Terraform configuration , Use it with caution, as it permanently deletes resources . Typically used to clean up resources at the end of a project or example

With these commands, Terraform automates infrastructure provisioning and updates efficiently while keeping everything consistent and in sync with your configuration.