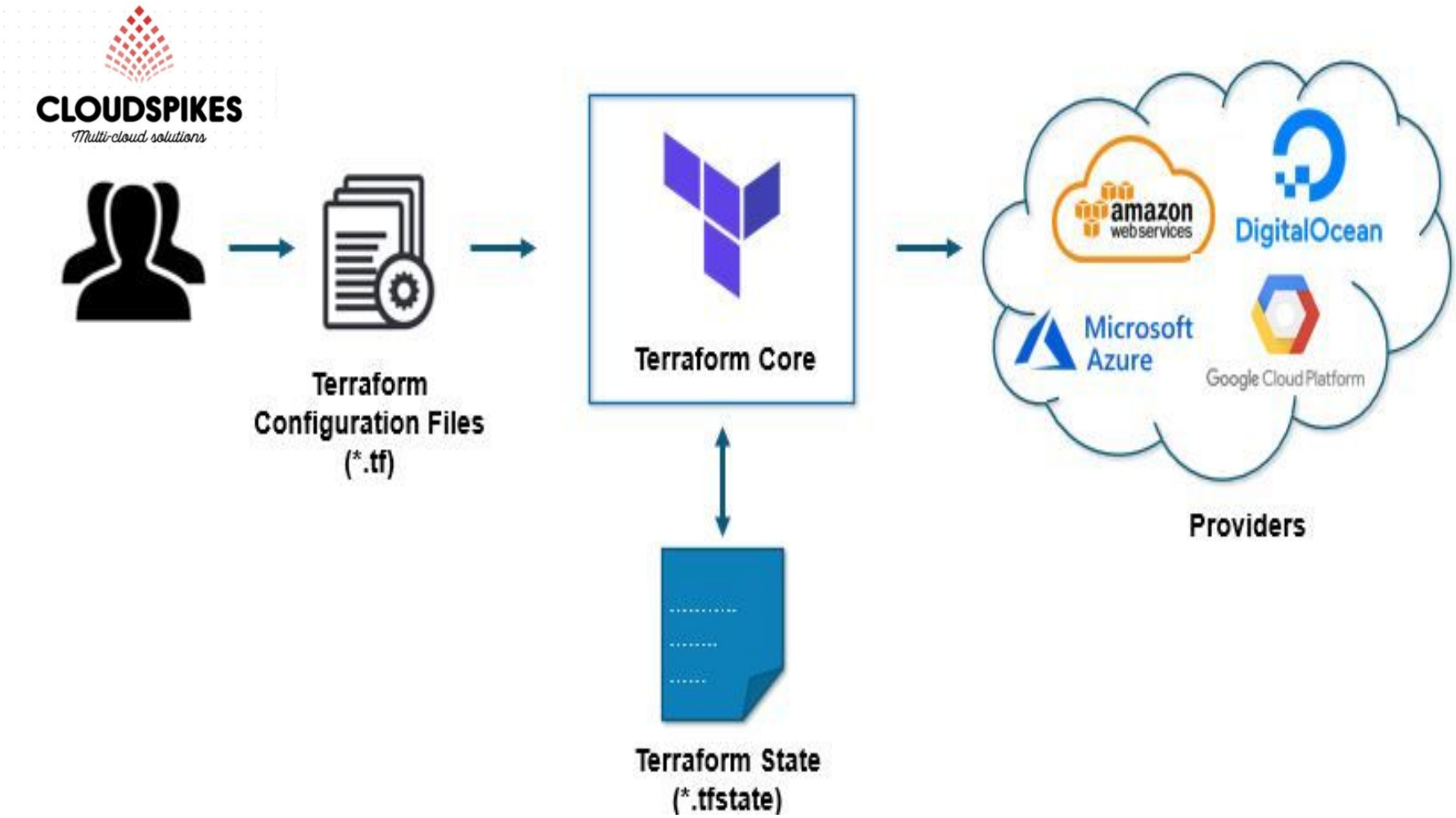
AWS Cloud DevOps Training

Prepared by Dhruv Rana - CloudSpikes

Terraform Architecture



Terraform Architecture

Terraform architecture mainly consists of the following components:

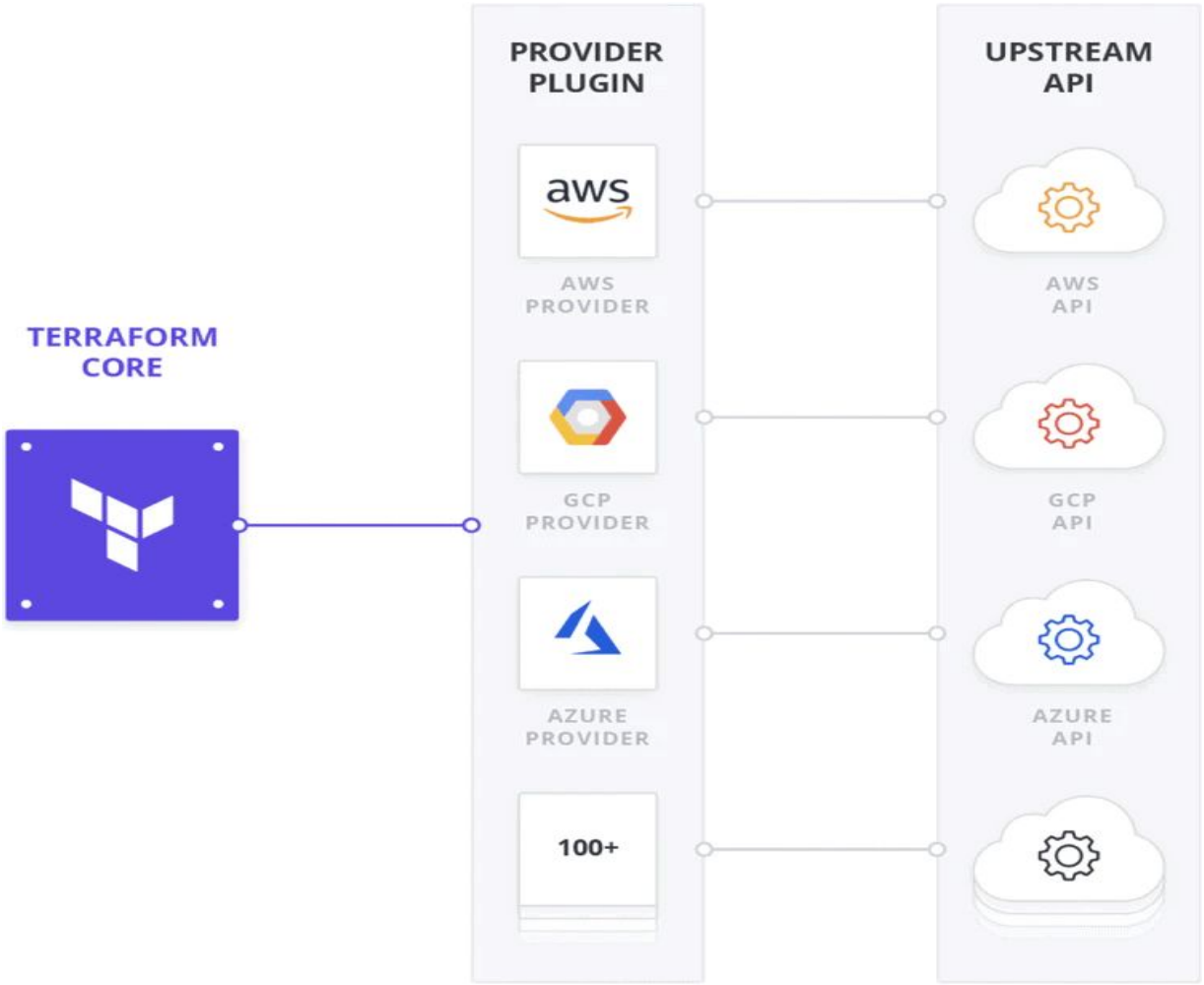
- Terraform Core
- Providers
- State file

Terraform Core

Terraform's core (also known as Terraform CLI) is built on a statically-compiled binary that's developed using the Go programming language.

This binary is what generates the command line tool (CLI) known as "terraform," which serves as the primary interface for Terraform users. It is open source and can be accessed on the Terraform GitHub repository.

Terraform Providers



Terraform Architecture

Providers

Terraform providers are modules that enable Terraform to communicate with a diverse range of services and resources, including but not limited to cloud providers, databases, and DNS services.

Each provider is responsible for defining the resources that Terraform can manage within a particular service and translating Terraform configurations into API calls that are specific to that service.

Providers are available for numerous services and resources, including those developed by major cloud providers like AWS, Azure, and Google Cloud, as well as community-supported providers for various services. By utilizing providers, Terraform users can maintain their infrastructure in a consistent and reproducible manner, regardless of the underlying service or provider.

Terraform Architecture

Provider Block

A provider block specifies the cloud provider and the API credentials required to connect to the provider's services. It includes the provider name, version, access key, and secret key.

For example, if you are using Azure as your service provider, it would look as follows:

```
provider "azurerm" {
  features {}
  subscription_id = "00000000-0000-0000-0000-000000000000"
  tenant_id = "11111111-1111-1111-1111-111111111111"
}
```

Terraform Architecture

State file

The Terraform state file is an essential aspect of Terraform's functionality. It is a JSON file that stores information about the resources that Terraform manages, as well as their current state and dependencies.

Terraform utilizes the state file to determine the changes that need to be made to the infrastructure when a new configuration is applied. It also ensures that resources are not unnecessarily recreated across multiple runs of Terraform.

The state file can be kept locally on the machine running Terraform or remotely using a remote backend like Azure Storage Account or Amazon S3, or HashiCorp Consul. It is crucial to safeguard the state file and maintain frequent backups since it contains sensitive information about the infrastructure being managed.

Terraform Structure

Declaring resources is very easy in Terraform. Terraform files always end with the extension `.tf`.

The basic Terraform structure contains the following elements.

Terraform Architecture

Terraform Block

A Terraform block specifies the required providers that terraform needs in order to execute the script. This block also contains the source block that specifies from where terraform should download the provider and also the required version.

Below is an example:

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.0.0"
    }
  }
}
```

Terraform Architecture

Resource Block

A resource block represents a particular resource in the cloud provider's services. It includes the resource type, name, and configuration details. This is the main block that specifies the type of resource we are trying to deploy.

Below is an example for creating a resource group in Azure.

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

Terraform Architecture

Data Block

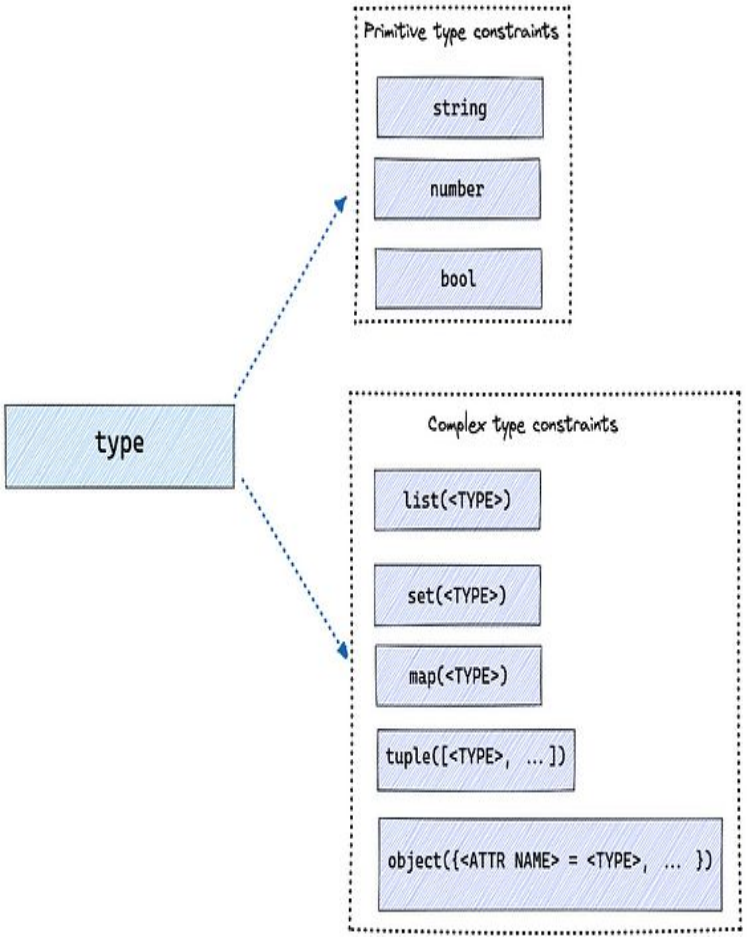
A data block is used to fetch data from the provider's services, which can be used in resource blocks. It includes the data type and configuration details.

This is used in scenarios where the resource is already deployed, and you would like to fetch the details of that resource.

The code snippet below helps you to fetch details of an existing resource group that is already deployed.

```
data "azurerm_resource_group" "example" {
  name = "existing"
}
```

Terraform Variable Types



validation

condition

an expression that uses the value of the variable to return true if the value is valid, false if invalid

error_message

a sentencea that Terraform will include when it produces a validation error

Terraform Architecture

Variable Block

A variable block is used to define input variables that are used in the Terraform configuration. It includes the variable name, type, and default value.

Following is an example of a variable block in Terraform.

```
variable "instance_type" {
  description = " The type of instance to start. Updates
  default     = "t2.micro"
  type        = string
}
```

Terraform Architecture

Output Block

An output block is used to define output values that are generated by the Terraform configuration. It includes the output name and value.

```
output "resource_group_id" {
  value = azurerm_resource_group.rg.id
}
```

Terraform Architecture

Provisioners

Terraform provisioners are a feature that allows Terraform to execute scripts or commands on newly created resources or instances. These scripts can be used for various purposes, such as setting up and configuring the infrastructure, installing software, running tests, and performing any other necessary actions.

Provisioners are executed after a resource has been created and can also be triggered when a resource is destroyed. Terraform comes with several built-in provisioners, including the file provisioner for copying files to a resource and the remote-exec provisioner for running commands on a remote machine. It is also possible to create custom provisioners for more advanced use cases.

This is mainly used as a last resort unless you are unable to achieve your desired outcome with existing resource blocks in Terraform.

Terraform Workflow

Terraform workflow is the set of steps and actions that a user follows to manage infrastructure using Terraform.

Following are the general steps that are involved in the lifecycle of a resource creation:

1. **Define**: Author your Infrastructure as Code in a Terraform configuration file with all the required blocks.
2. **Initialize**: Initialize the Terraform working directory and download any necessary plugins. This is usually done using `terraform init` command.
3. **Review**: Review the Terraform execution plan to see what changes will be made to the infrastructure. This is done using the `terraform plan` command. Running this command will show the actions that Terraform is going to perform when `terraform apply` command is run.
4. **Apply**: Apply the changes to create or modify the infrastructure. Once you are comfortable with the changes that are shown in the output of `terraform plan` command, you can apply those changes using `terraform apply` command.
5. **Inspect**: You can Inspect the state of the infrastructure using the Terraform state file.

Terraform Workflow

In case you need to apply any new configuration, make the necessary changes in the config and follow the above steps to apply the configuration.

These steps are often repeated as part of a Continuous Integration/Continuous Delivery (CI/CD) pipeline or as part of ongoing infrastructure management. Terraform's declarative approach to infrastructure as code allows for a consistent and reproducible workflow, where infrastructure changes can be tracked, reviewed, and audited over time.

Any questions or queries?

- Reach out to me on Insta, LinkedIn, Email or WhatsApp:

  - Insta: @dhruv_rana_29_10
  - LinkedIn: @dhruv-r-b033949b
  - Email: support@cloudspikes.ca
  - WhatsApp: +1-647-376-7753