

DevOps
Cloud
Computing

Caltech

Center for Technology &
Management Education

Post Graduate Program in DevOps

DevOps
Cloud
Computing



Caltech

**Center for Technology &
Management Education**

Configuration Management with Ansible and Terraform



Terraform

A Day in the Life of a DevOps Engineer

You are working in an organization as a DevOps Engineer. The organization is looking for an automation tool that will help them create, update, and control the versioning of cloud infrastructure.

The goal is to generate and execute plans based on the desired end state and reprovision infrastructure based on configuration changes.

The organization mainly deals with hybrid cloud and multi-cloud deployments.

The organization is looking for the following key requirements:

- You need to automate the orchestration of a large number of resources.
- You have developer resources to support the development of Terraform code.



A Day in the Life of a DevOps Engineer

- You need to scale up and down the infrastructure according to variable online workloads.
- You need to deploy a large system that involves a complex topology.
- You need to apply repeated and clearly defined procedures on cloud resources.
- You need to perform orchestration on a heterogeneous system that involves multi-cloud and hybrid cloud platforms.

To achieve all the above, along with some additional features, you will be learning a few concepts in this lesson that will help find a solution for the given scenario.



Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Install and configure Terraform
- 🕒 Understand how to build, edit, and version infrastructure in a secure and efficient manner
- 🕒 Demonstrate Terraform language
- 🕒 Understand Configuration files



Introduction to Terraform

What Is Terraform?

Terraform is an infrastructure as code (IaC) tool that lets a user build, edit, and version infrastructure in a secure and efficient manner.



Key Features of Terraform

The key features of Terraform are:



Infrastructure as Code



Resource Graph



Execution Plans



Change Automation

Terraform Concepts

The following are the key Terraform concepts:

Variables

Apply

Provider

Plan

Module

Output Values

State

Data Source

Resources

Terraform Lifecycle

The Terraform lifecycle consists of – **init**, **plan**, **apply**, and **destroy**.

Init

terraform init

Plan

terraform plan

Apply

terraform apply

Destroy

terraform destroy

Terraform Lifecycle

Init
terraform init

Initializes the working directory

Plan
terraform plan

Creates an execution plan to reach a desired state of the infrastructure

Apply
terraform apply

Makes the changes in the infrastructure as defined in the plan

Destroy
terraform destroy

Deletes all the old infrastructure resources

Terraform Workflow

The core Terraform workflow has three steps:

Write

Author infrastructure as code

Plan

Preview changes before applying

Apply

Provision reproducible infrastructure

How Terraform Works?

Terraform is designed on a plugin-based architecture, which allows developers to extend it by developing new plugins or building updated versions of the current ones.

Terraform architecture is comprised of two key components:

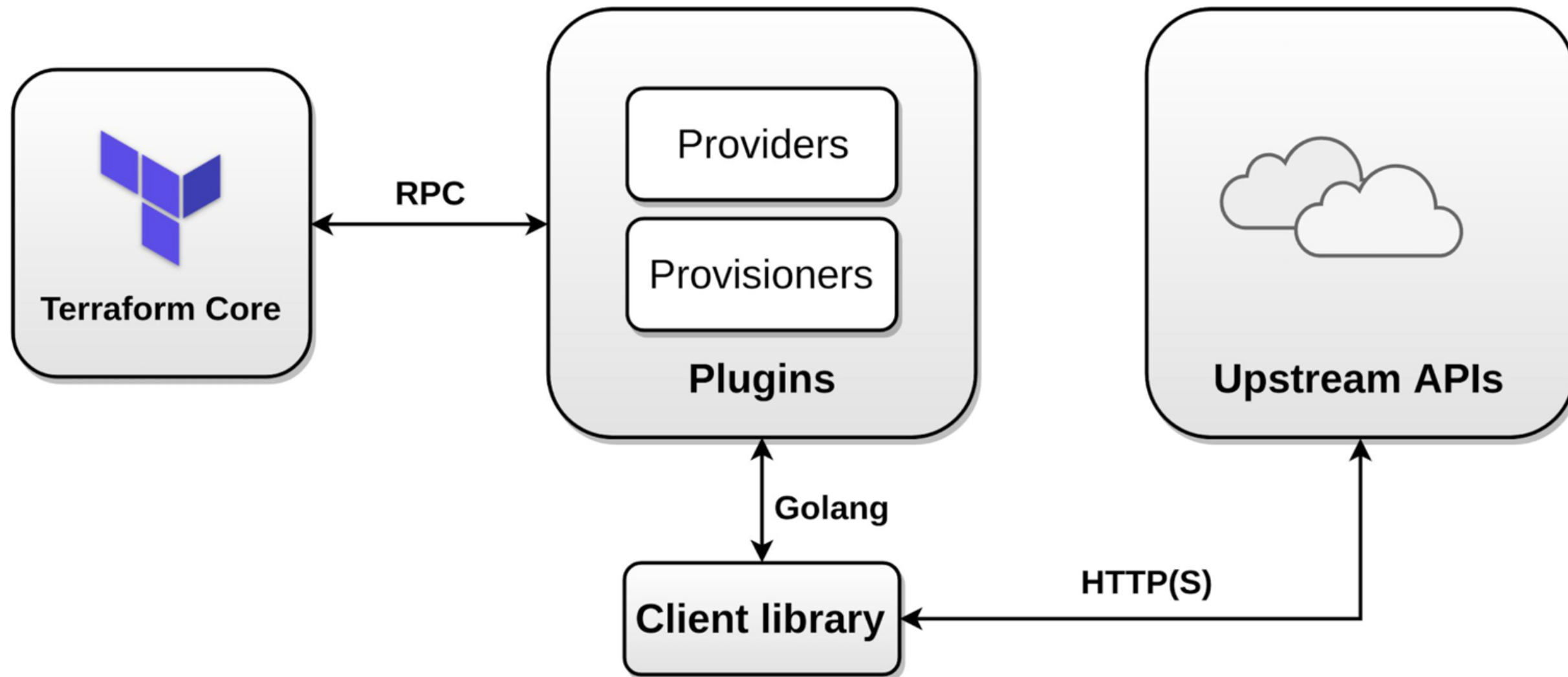
Terraform Core

Terraform Plugins

Terraform Core communicates with Terraform Plugins using remote procedure calls (RPC).

Terraform Architecture

The below diagram represents the architecture of Terraform:

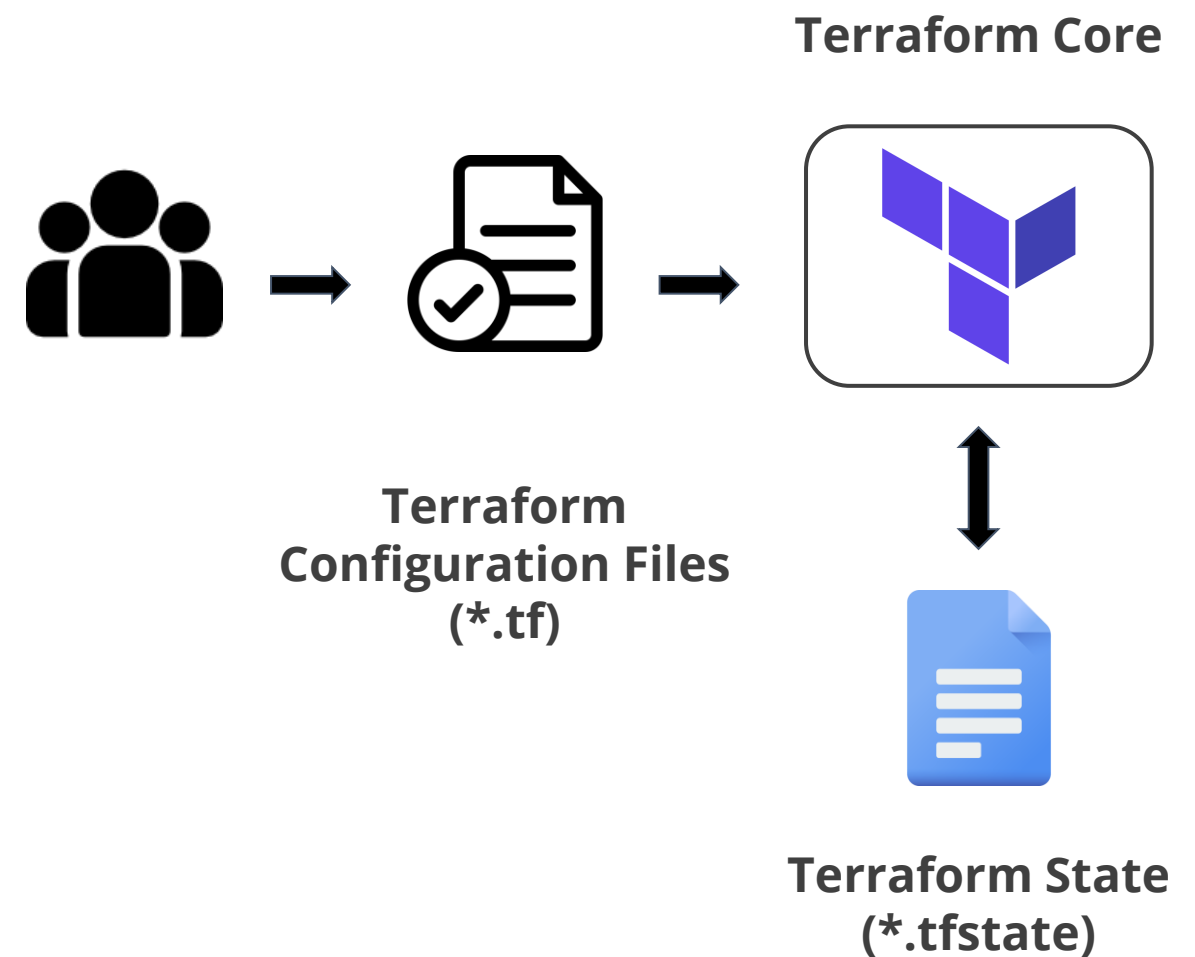


Terraform Core

Terraform Core is a statically-compiled binary written in the Go programming language.

Terraform core uses two input sources to do its job:

- Terraform Configuration Files
- Terraform State



Terraform Core

The primary responsibilities of Terraform Core are:

Infrastructure as code: reading and interpolating configuration files and modules

Resource state management

Construction of the Resource Graph

Plan execution

Communication with plugins over RPC

Terraform Plugins

Terraform Plugins are Go executable binaries that use an RPC interface to communicate with Terraform Core. Each plugin exposes an implementation for a specific service or provisioner, such as **AWS** or **bash**.

Providers

Provisioners

Plugins

All Providers and Provisioners used in Terraform configurations are plugins.

Terraform Plugins

The primary responsibilities of Providers Plugins are:

Initialization of any included libraries used to make API calls

Authentication with the Infrastructure Provider

Define resources that map to specific services

The primary responsibility of Provisioner Plugins is:

Executing commands or scripts on the designated resource after creation, or on destruction

Installing and Configuring Terraform

Assisted Practice

Setting Up and Configure Terraform

Duration: 10 Min.

Problem Statement:

You've been given the task of installing and configuring Terraform in your system, which is an infrastructure as code (IaC) tool that allows you to safely and efficiently build, change, and version infrastructure.

Assisted Practice: Guidelines

Steps to be followed:

1. Downloading the appropriate package
2. Adding the binary file into the bin directory



Benefits of Terraform

Advantage of Terraform for Managing Resource

Below are the top five reasons to use Terraform:

1

Improved multi-cloud infrastructure deployment

2

Automated infrastructure management

3

Infrastructure as code

4

Reduced development costs

5

Reduced provisioning time

Terraform Cloud Provider

Providers

Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs.

- Each **Provider** provides a collection of resource types or data sources, or both, that Terraform can manage.
- The majority of service providers set up a specific infrastructure platform (either cloud or self-hosted).

The **Terraform Registry** is the central repository for publicly available Terraform Providers, and it hosts Providers for the majority of the infrastructure platforms.

Providers

Providers are delivered independently from Terraform, and has its own release cycle and version number.

A provider is responsible for understanding API interactions and exposing resources.

It is an executable plugin that contains code necessary to interact with the API of the service.

Terraform configurations must specify which provider is required for Terraform to install and use.

Cloud Providers

Terraform is compatible with a variety of popular providers, including major cloud providers.

A few major cloud providers are:



AWS



Azure



GCP



Kubernetes



Oracle Cloud



Alibaba Cloud

HashiCorp Configuration Language (HCL)

Terraform Language

Terraform's primary user interface is the Terraform language. A configuration specified in the Terraform language is always at the center of the Terraform workflow in every edition.

- The Terraform language's principal function is to declare resources, which are infrastructure objects.
- The purpose of all other language features is to make resource definition more flexible and convenient.

A Terraform configuration is a document written in the Terraform language that instructs Terraform how to manage a set of infrastructure. Multiple files and directories can make up a configuration.

Terraform Language

Blocks

Blocks serve as containers for other content and usually describe the configuration of an object, such as a resource.

Arguments

Arguments assign a value to a name.

Expressions

Expressions represent a value, either directly or by referencing and combining other values.

Terraform is a declarative language that describes an intended goal rather than the processes to reach that goal.

HashiCorp Configuration Language (HCL)

HCL (HashiCorp Configuration Language) is a unique configuration language developed by HashiCorp to work with HashiCorp products, particularly Terraform.

HCL consists of three sub-languages:

- Structural
- Expression
- Templates

When the sub-languages are combined, they generate a well-structured HCL configuration file. This structure aids in the precise and simple description of environmental configurations required by the Terraform tool.

HCL Syntax

The HCL Syntax describes the native grammar of the Terraform language.

It is based on two key syntax components: Arguments and Blocks.

Arguments

- An argument assigns a value to a particular name:

```
image_id = "abc123"
```

- The identifier before the equals sign is the argument name, and the expression after the equals sign is the argument's value.

HCL Syntax

Blocks

- A block is a container for other content:

```
resource "aws_instance" "example" {  
  ami = "abc123"  
  
  network_interface {  
    # ...  
  }  
}
```

- A block has a *type*. In this example: resource.

HCL Syntax: Identifiers

Identifiers include argument names, block type names, and the names of most Terraform-specific entities such as resources and input variables.

- Identifiers can contain letters, digits, underscores (`_`), and hyphens (`-`).
- The first character of an identifier must not be a digit to avoid ambiguity with literal numbers.



Terraform uses the Unicode identifier syntax for complete identifier rules, which has been extended to include the ASCII hyphen character.

HCL Syntax: Comments

The Terraform language supports three different syntaxes for comments:

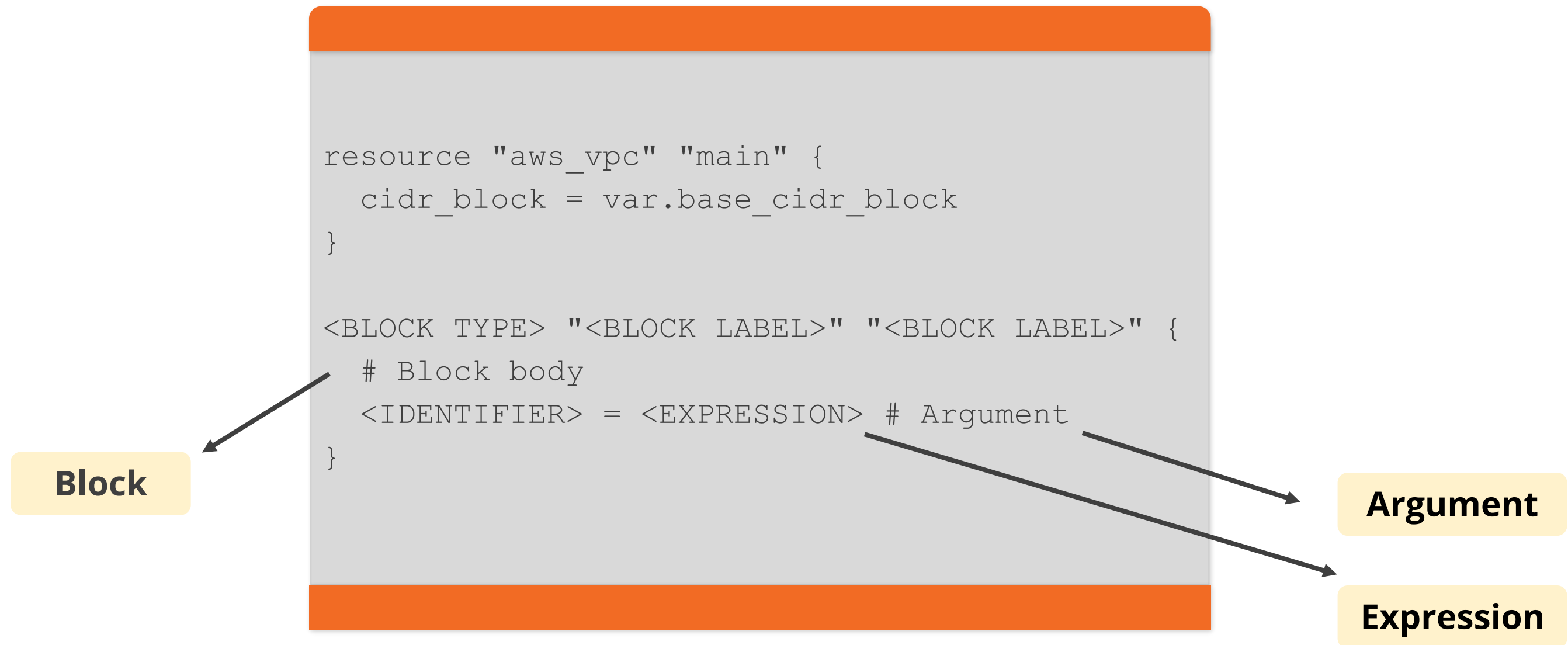
- **#** begins a single-line comment, ending at the end of the line.
- **//** can be used as an alternative to **#** to begin a single-line comment.
- **/* and */** are the start and end delimiters for a comment that might span over multiple lines.



The **# single-line** comment style is the default comment style and should be used in most cases.

Terraform Language

The syntax of the Terraform language consists of only a few basic elements: Block, Argument, and Expression.



Features of HCL

HCL is a JSON-compatible language that adds functionality to the Terraform tool to help a user get the most out of it.

The features of HCL are:

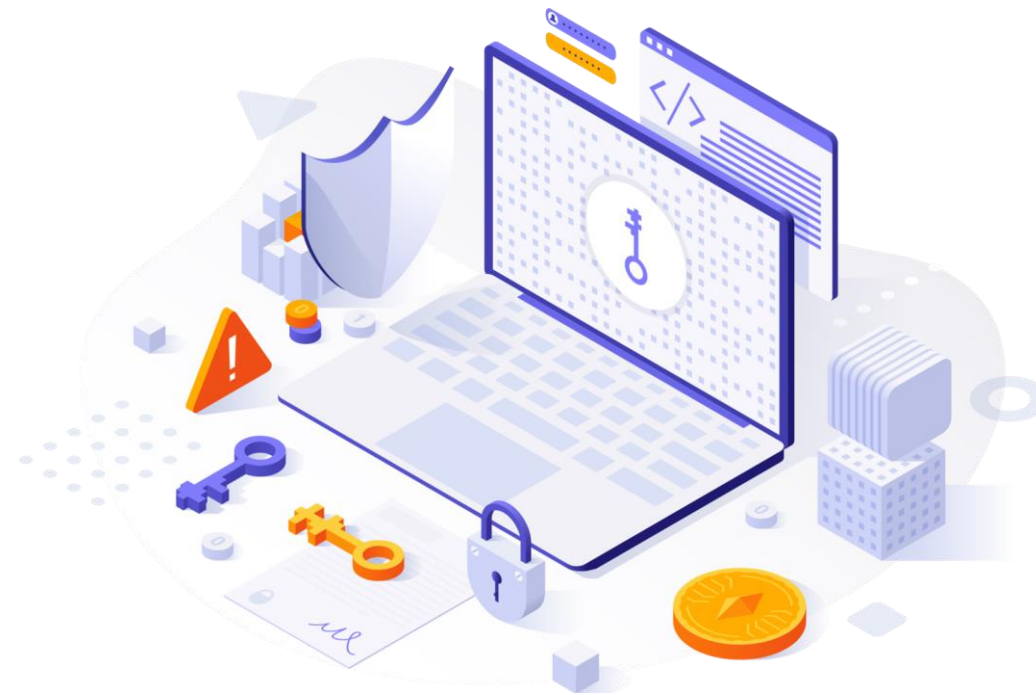
- Comments are available as a single line or multi-line.
 - Single Line: **#** **or** **//**
 - Multi-Line: **/* */** (no nesting of block comments)
- Variable assignments use the **key = value** construction where whitespace does not matter, and the value can be a primitive such as a string, number, boolean, object, or a list.

Features of HCL

- Strings can contain any UTF-8 characters and are quoted.
- Numbers can be written and parsed in several different ways:
 - Base 10 numbers are the default
 - Hexadecimal: Prefix a number with 0x
 - Octal: Prefix a number with a 0
 - Scientific numbers: Use notation such as 1e10
- Arrays and lists of objects are easy to create using [] for arrays and { **key** = **value** } for lists.

HCL Syntax: Character Encoding and Line Endings

Terraform configuration files must always be UTF-8 encoded.

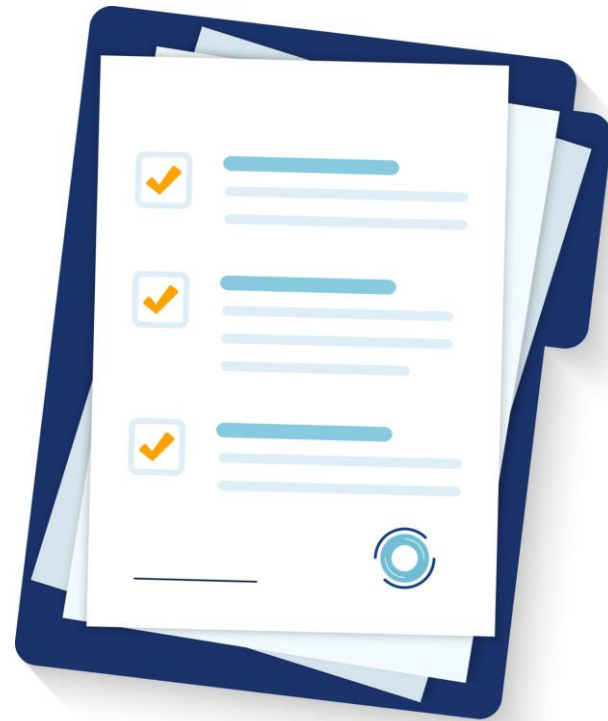


Terraform allows non-ASCII characters in identifiers, comments, and string values, even though the language's delimiters are all ASCII characters.

Terraform Files and Directories

Terraform File Extension

The Terraform language uses plain text files with the **.tf** extension to store code. A JSON-based variant of the language with a **.tf.json** file extension is also available.



Files containing Terraform code are called configuration files.

Directories and Modules

A module is a directory that contains a collection of **.tf** and/or **.tf.json** files.

- A Terraform module consists of the top-level configuration files in a directory.
- Terraform evaluates all of a module's configuration files, thereby considering the entire module as a single document.
- Module calls allow Terraform modules to explicitly include other modules into the configuration.

The Root Module

Terraform always runs in the context of a single ***root module***.

A Terraform configuration consists of:

- Root module
- Tree of child modules



The root module in Terraform CLI is the working directory where Terraform is executed.

Assisted Practice

Creating First Terraform Script

Duration: 10 Min.

Problem Statement:

You've been given the task of writing your first Terraform script to define and provision data center architecture using declarative configuration.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Terraform script to launch an AWS EC2 instance



Assisted Practice

Validating Terraform Configuration File

Duration: 10 Min.

Problem Statement:

You have been asked to validate your first Terraform script.

Assisted Practice: Guidelines

Steps to be followed:

1. Validating the Terraform script

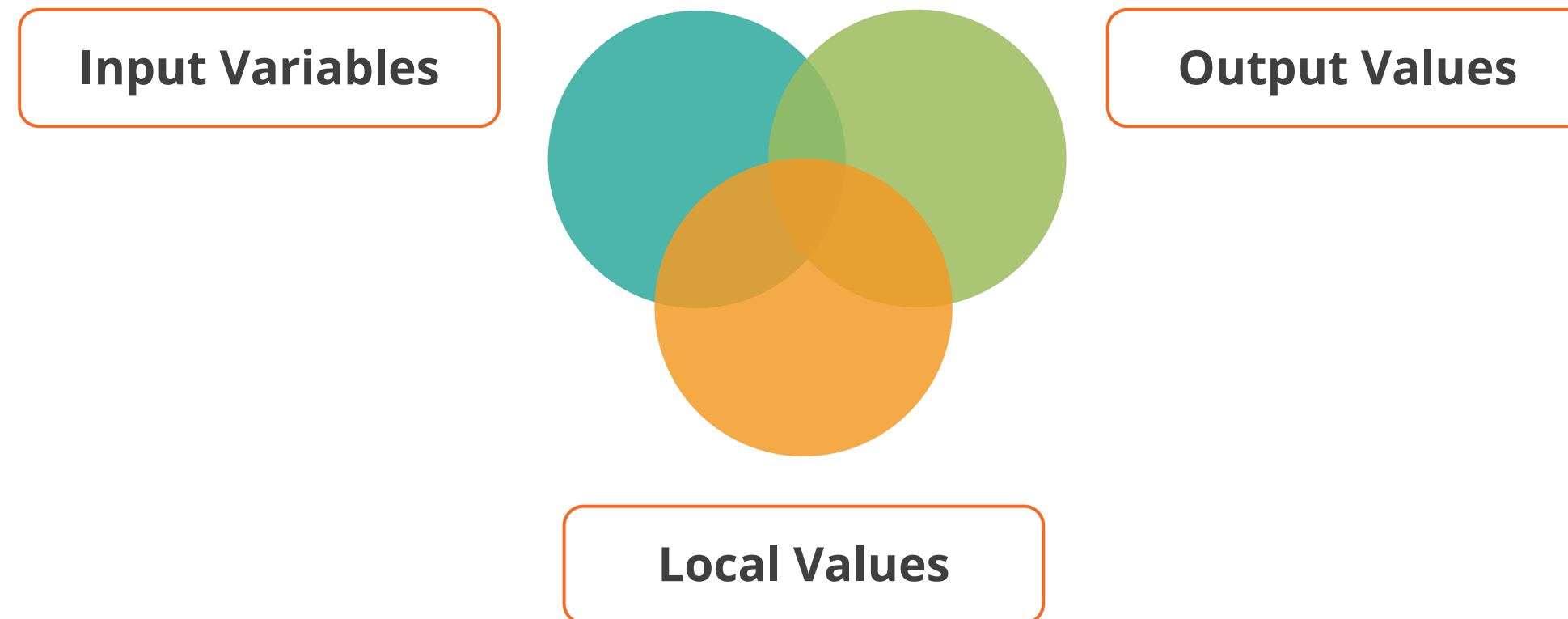


Variables and Outputs

Variables and Outputs

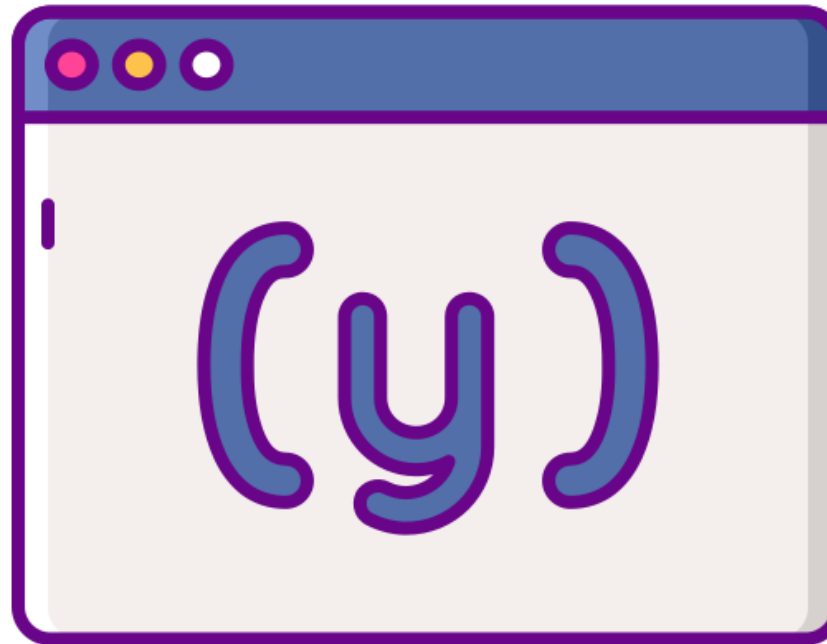
There are several types of blocks in the Terraform language for requesting or publishing named values.

The different types of blocks are:



Input Variables

Input variables are used as parameters in Terraform modules, allowing them to be altered without modifying the module's source code, and sharing modules across configurations.



Input variables are like function arguments.

Declaring an Input Variable

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type      = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

- A **variable** block must be used to declare each input variable that a module accepts.
- The label after the variable keyword is the name for the variable.
- The name of a variable can be any valid identifier except the following: source, version, providers, count, for_each, lifecycle, depends_on, locals.

Input Variable: Arguments

Terraform CLI defines the following optional arguments for variable declarations:

default

A default value makes the variable optional.

type

This argument specifies the value types accepted for the variable.

description

This specifies the input variable documentation.

validation

A block to define validation rules, usually in addition to type constraints.

sensitive

Limits Terraform UI output when the variable is used in the configuration.

Output Values

Output values are like the return values of a Terraform module, and have several uses:

- A parent module can access a subset of a child module's resource attributes using outputs.
- After running ***terraform apply***, a root module can use outputs to print specific values in the CLI output.
- Root module outputs can be accessed by other configurations using a `terraform_remote_state` data source when using a remote state.

Note

When the meaning is clear from context, output values are frequently referred to as "outputs".

Declaring an Output Value

```
output "instance_ip_addr" {  
    value =  
    aws_instance.server.private_ip  
}
```

- Each output value exported by a module must be declared using an `output` block.
- The label immediately after the output keyword is the name, which must be a valid identifier.
- The value argument takes an expression whose result is to be returned to the user.

Note

When Terraform implements a plan, the outputs are rendered. The outputs will not be rendered if a user runs ***terraform plan***.

Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as **module.<MODULE NAME>.<OUTPUT NAME>**.

Example:

If a child module named ***web_server*** declared an output named ***instance_ip_addr***, a user could access that value as ***module.web_server.instance_ip_addr***.

Output Values: Optional Arguments

Output blocks can optionally include the following arguments:

description

A user can briefly describe the purpose of each value using the optional description argument.

sensitive

The optional sensitive argument can be used to mark output as containing sensitive content.

depends_on

This argument can be used to create additional explicit dependencies or rare cases.

Types and Values

Types and Values

A value is the result of an expression. All values have a type that determines where they can be used and what transformations they can experience.

The Terraform language uses the following types for its values:



Note

A **Literal expression** is an expression that represents a constant value directly. Each of the above-mentioned value types has a literal expression syntax in Terraform.

Type Conversion

The most common application of expressions is to set values for resource and child module parameters. The argument has an expected type in certain circumstances, and the specified expression must return a value of that type.

- Terraform automatically converts values from one type to another when possible in order to produce the desired type.
- If this isn't possible, Terraform will throw an error about a type mismatch, and the user will need to replace the configuration with a more appropriate expression.

Type Conversion

Terraform transforms numeric and bool values to strings automatically, when needed. It also converts strings to numbers or bools if the string has a valid number or bool value.

- **true** converts to "**true**", and vice-versa
- **false** converts to "**false**", and vice-versa
- **15** converts to "**15**", and vice-versa

Terraform Registry

Terraform Registry

The Terraform Registry is an interactive site for finding a wide range of Terraform integrations (providers) and configuration packages (modules).



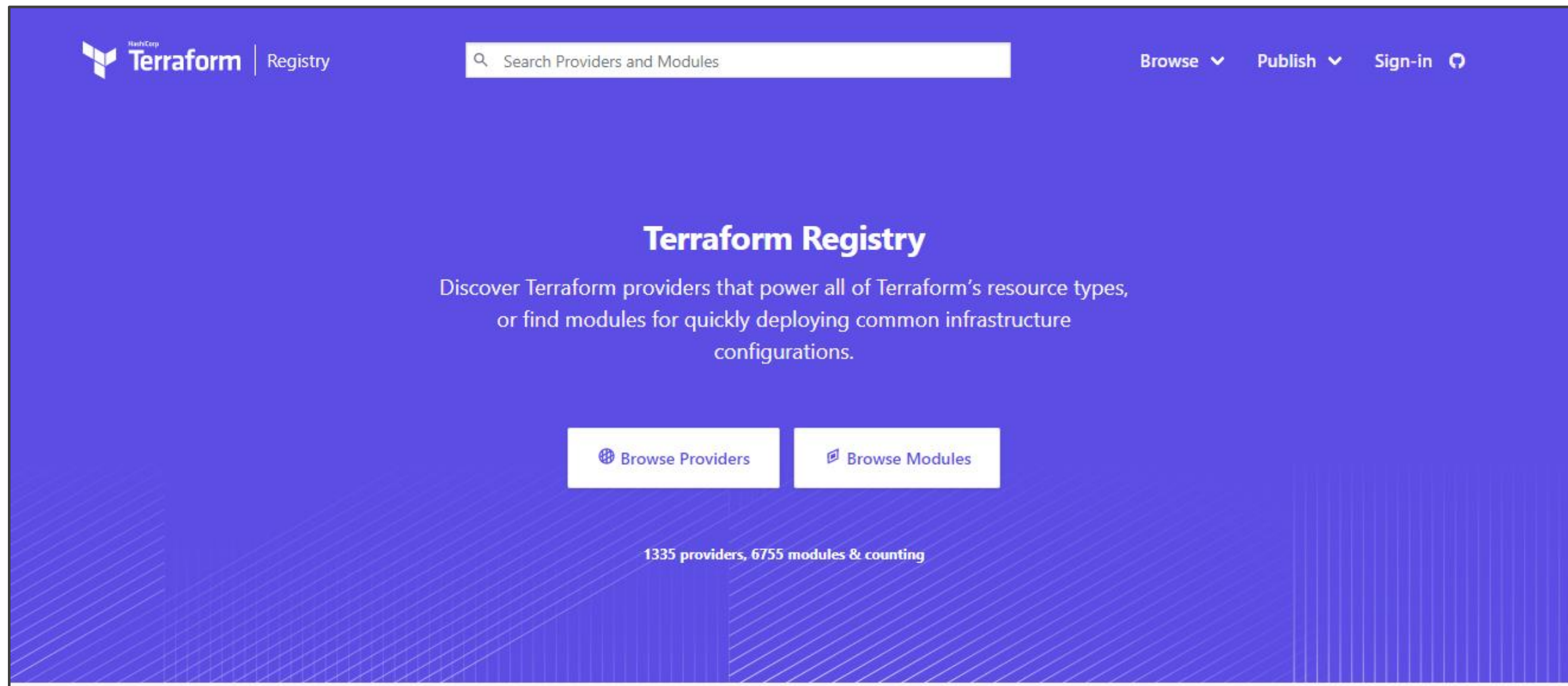
The Registry includes HashiCorp-developed solutions, as well as those from third-party partners and the Terraform community.



The Registry's purpose is to provide plugins for managing any infrastructure API, pre-made modules for fast configuring common infrastructure components, and samples of good Terraform code.

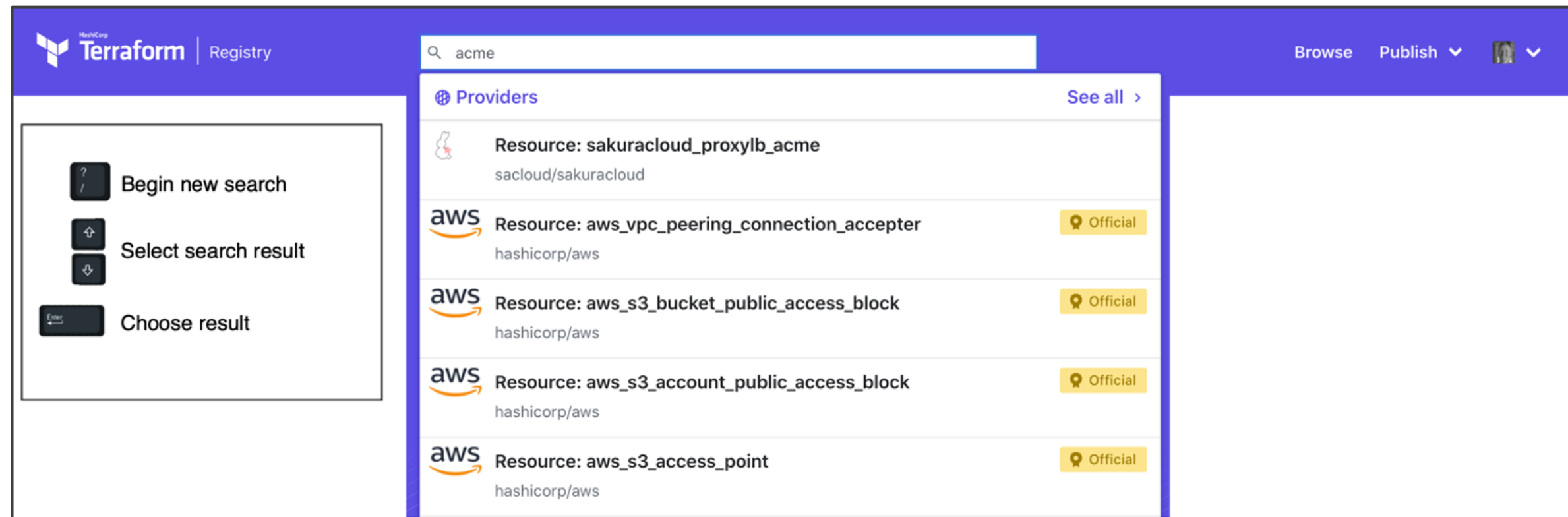
Terraform Registry

The Terraform Registry is integrated directly into Terraform so a user can directly specify providers and modules.



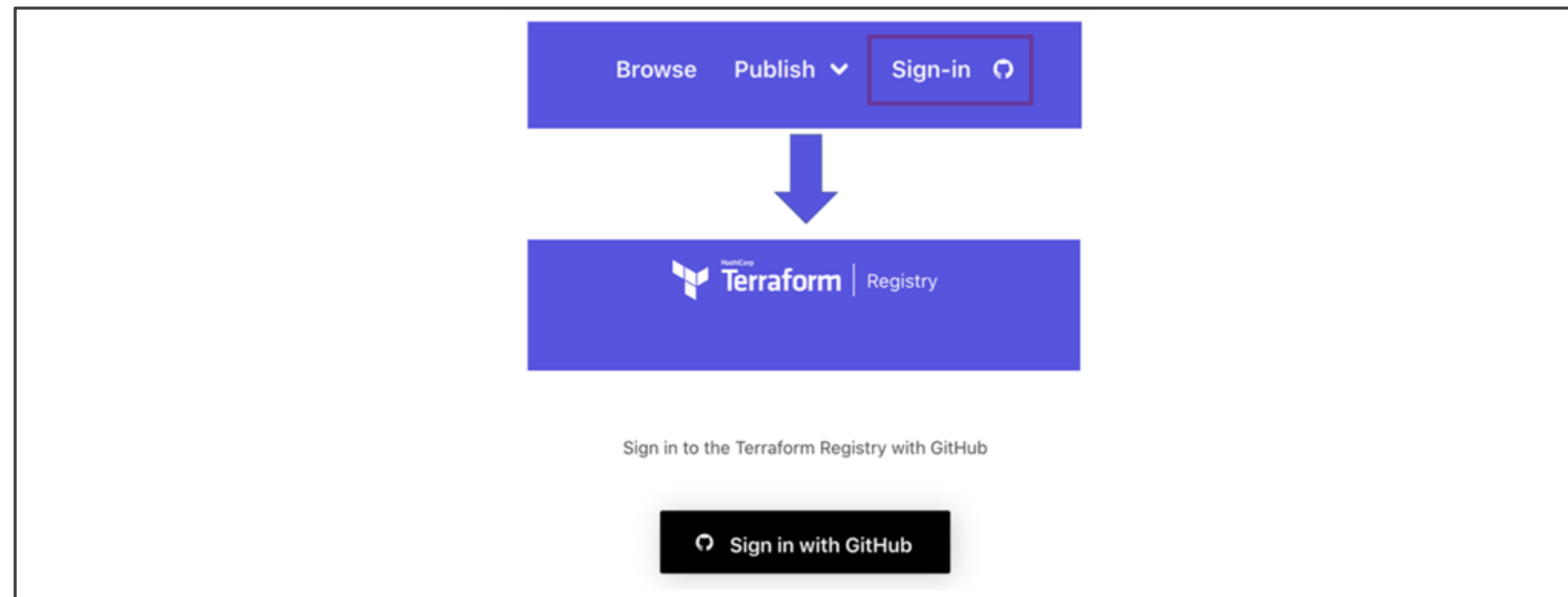
Navigating the Registry

The registry has several different categories for both modules and providers to help with navigating a large number of available options.




Terraform Registry: User Account



Anyone with a GitHub account who wants to publish a provider or module can create an account and sign in to the Terraform Registry.



Terraform Registry: Getting Help




A user can generate issues and contributions for a specific provider or module by using the "Report an issue" link on GitHub.

**aws**

 Official by:  HashiCorp

Public Cloud

Lifecycle management of AWS resources, including EC2, Lambda, EKS, ECS, VPC, S3, RDS, DynamoDB, and more. This provider is maintained internally by the HashiCorp AWS Provider team.

VERSION	 PUBLISHED	 INSTALLS	 SOURCE CODE
2.70.0	20 days ago	6,695,042	terraform-providers/terraform-provider-aws

HELPFUL LINKS

[Using Providers](#)

[Learn Terraform](#)

[Report an issue](#)

Key Takeaways

- 🕒 Terraform is an infrastructure as code (IaC) tool that allows users to securely build, change, and version infrastructure.
- 🕒 Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs.
- 🕒 The Terraform language's principal function is to declare resources, which are infrastructure objects.
- 🕒 The Terraform language uses plain text files with the .tf extension to store code.



Building and Testing a Terraform Module

Duration: 25 Min.

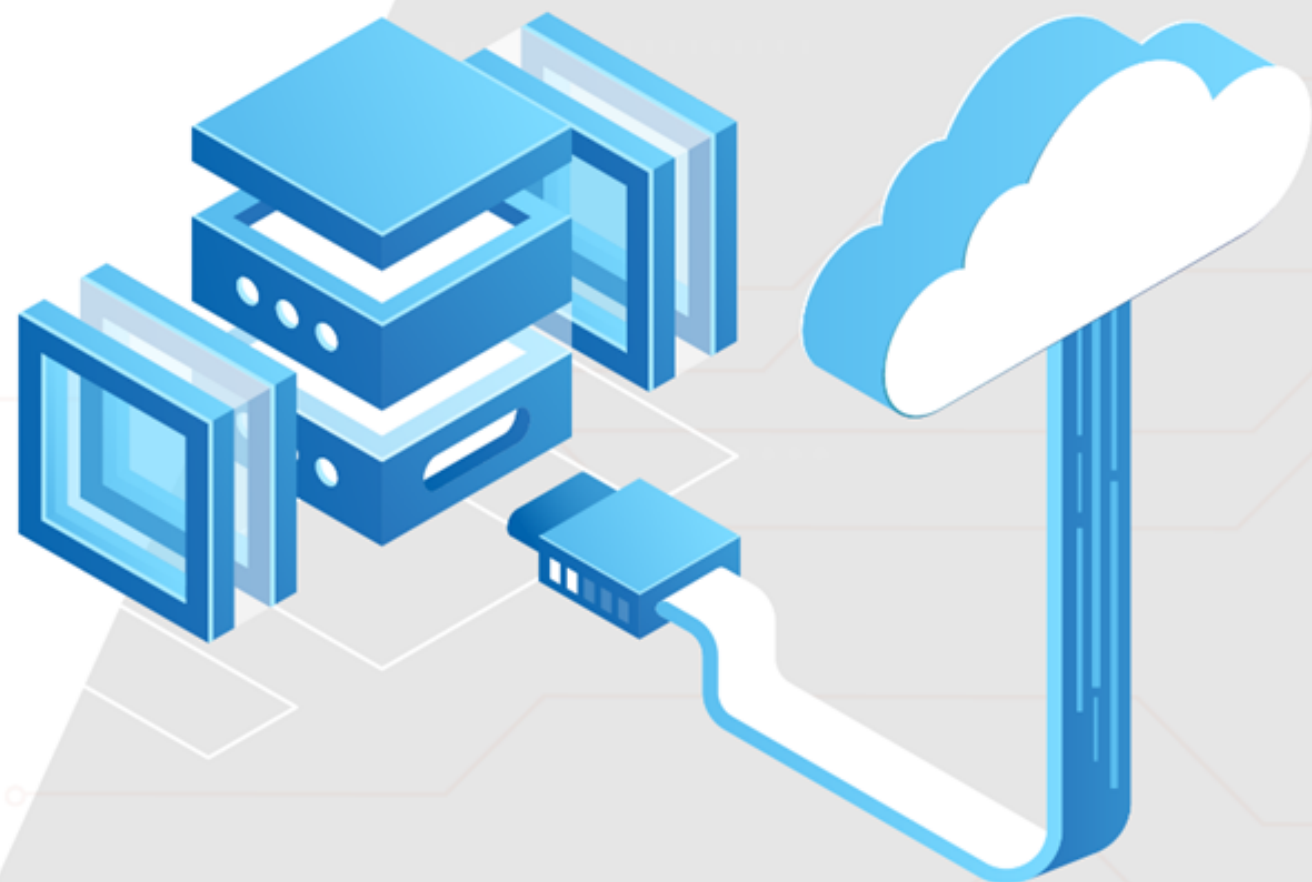


Project agenda: To build and test a Terraform Module

Description: A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. In this project, you have been asked to build and test a Terraform Module.

Perform the following:

- Configuring the AWS CLI from the terminal
- Creating the directory structure for the Terraform project
- Writing your Terraform VPC Module code
- Writing your main Terraform project code
- Deploying your code and test out your Module



Thank you