A Weekend Project: Terraform 

What even is it?

In the **next few minutes** you will have a solid grasps on the basics and an example project you can follow along with.

By the way this is a preview from my new ebook: TechOneTwenty
So if you enjoy this kind of broken down, jargon free learning, check it out.

Easy to learn. Hard to master - That sums up Terraform pretty well.

What is Terraform??

I love Terraform, it's in huge demand right now and if you want to work in the cloud it's a tool you need to learn. Terraform is just one of a number of IaC tools, there's others like: AWS CloudFormation or Pulumi but Terraform, at the movement is the most popular. Designed to help Cloud Engineers and ultimately businesses set up and manage their cloud infrastructure across various platforms like AWS, Microsoft Azure, Google Cloud, and others, all through a simple, consistent format.
Imagine it as a blueprint that allows you to describe what kind of infrastructure you want - such as servers, databases, and networks - using a straightforward, readable language.

Once you've outlined what you need, IaC automates the process of building and updating your setup, ensuring everything is configured exactly as specified, without the need to manually set up each component. This saves time but also reduces the chances of errors and differences in environments, making the management of IT resources more efficient and predictable - In a nutshell it's version control for your infrastructure, giving every engineer a single source of truth for what the cloud estate looks like.

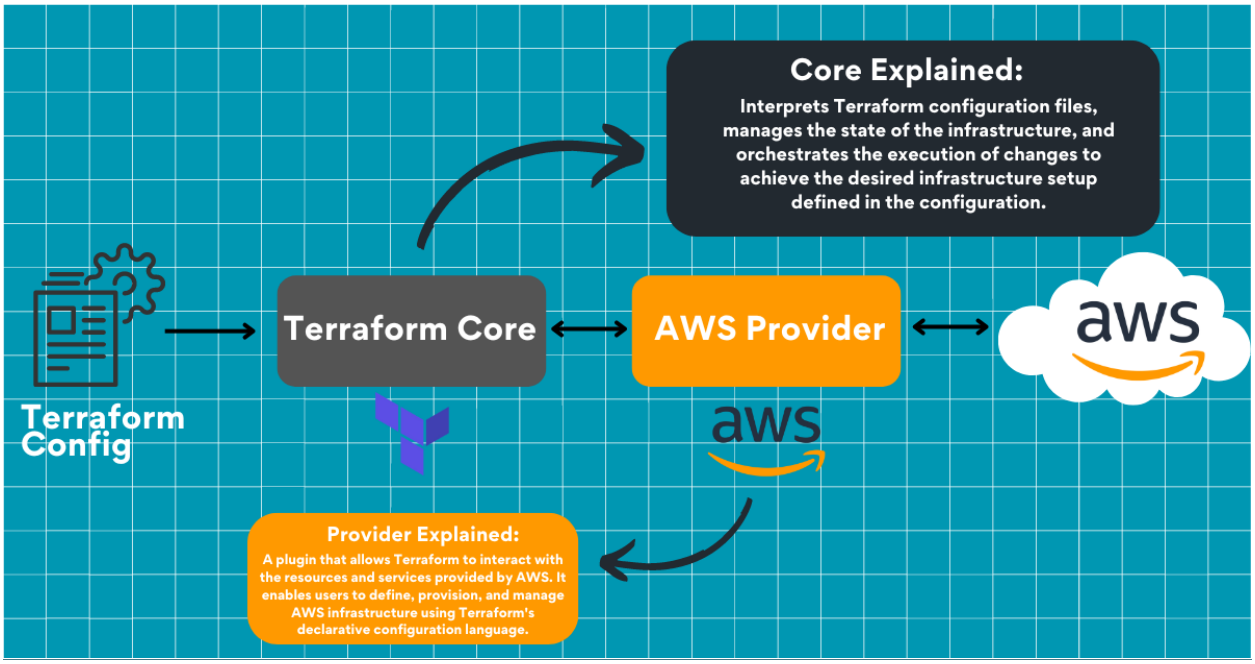

Image: TechOneTwenty Ebook - WJPearce

let's take a look at some terms you will see pop up when learning Terraform:

Configuration Files:

Written in a clear and straightforward language called HashiCorp Configuration Language (HCL), allow you to define resources such as virtual machines, network interfaces, or cloud storage in a structured way. Think of these files as a blueprint that Terraform follows to build your infrastructure. By using these configuration files, you can ensure that your infrastructure is set up consistently and can be easily replicated or adjusted as needed - We will take a closer look at these in a moment in the example.

Providers:

In Terraform, providers are like bridges that connect Terraform to various cloud platforms and services, such as Amazon Web Services, Google Cloud Platform, or Microsoft Azure. Providers are plugins that you use in your Terraform configuration to specify which cloud services you want to interact with. They understand the API of these services and translate your Terraform configurations into actual actions on those cloud platforms—like creating a new database or setting up a network We will take a quick looks at providers for AWS in the example section.

Resources and Modules:

Try not to worry about these until you've wrapped your head around the fundamentals properly, having said that it is essential you learn what resources and modules are. In Terraform, "resources" are the basic building blocks that describe one piece of your infrastructure, such as a virtual machine, a network, or a database. Think of them as individual ingredients in a recipe. You define each resource in your configuration files to tell Terraform exactly what you want to create or manage. "Modules," on the other hand, are like pre-packaged recipes—collections of resources and configurations that can be reused and shared. Modules allow you to organize your infrastructure into manageable, reusable components. For example, you might have a module for setting up a web server, including not just the server itself but also the necessary networking and security settings. This makes managing complex infrastructures more straightforward and promotes best practices by enabling reuse of configurations.

To Sum Up:

Terraform is an open-source tool used for building, changing, and versioning infrastructure efficiently across various cloud providers. **If there's one tool I recommend learning on your Cloud Journey it's this one.**

If you want to keep reading and to the project, below is some basic commands and a simple deployment project with the HCL you'll need.

Getting Started:

We first need to install Terraform.

Installing Terraform on different operating systems involves downloading the Terraform binary and making it available in your system's PATH. Here are the steps for macOS, Linux, and Windows: Get Started Docs

Basic Commands:

These are the five basic commands you will use over and over in when using Terraform. We can also pass in variables at run time here but that's a little more advance than this book will cover, we really want to get comfortable with the basic.

terraform init
Initialises a Terraform working directory by preparing the necessary files and directories. This command is used to install and configure the Terraform providers specified in the configuration, setting up the backend for state storage, and performing other necessary setup steps.

terraform plan
Creates an execution plan, which Terraform uses to determine the actions it needs to take to change the infrastructure to match the configuration. This command is used for previewing changes without applying them, helping to understand what Terraform will do before making any modifications to real resources.

terraform apply
If your super confident in your config you can actually skip the plan command and skil to stragiht to the apply. This, as you might guess applies the changes required to reach the desired state of the configuration, or the specified plan. This command is used to create, update, or delete resources according to the Terraform configuration and the plan generated by terraform plan.

terraform destroy
Destroys all the resources defined in the Terraform configuration. This command is used to tear down the infrastructure managed by Terraform, removing all the managed resources safely and efficiently.

terraform validate
Validates the syntax of the Terraform configuration files in the directory. It checks for any errors (such as syntax errors, invalid resource names, etc.) in the configuration. This command is useful for verifying configurations before applying them, ensuring they are syntactically correct and internally consistent.

Example Project:

You can follow along here: setting this example up will allow you to deploy the following

Defines an AWS provider and sets the region to "us-east-1".

- Creates a Virtual Private Cloud (VPC) with a CIDR block of "10.0.0.0/16".
- Adds a subnet to the VPC in the "us-east-1a" availability zone with a CIDR block of "10.0.1.0/24".
- Sets up a security group within the VPC that allows incoming SSH (port 22) and HTTP (port 80) traffic from any IP address and allows all outgoing traffic.
- Launches an EC2 instance within the defined subnet and security group, using a specified AMI (Amazon Machine Image).

providers.tf

This file will define your provider, in this case, AWS, and specify the region.

```
provider "aws" {
  region = var.aws_region
}
```

variables.tf

This file will declare variables used across your Terraform configuration.

```
variable "aws_region" {
  description = "The AWS region to create resources in."
  default     = "us-east-1"
}

variable "instance_type" {
  description = "The instance type of the EC2 instance."
  default     = "t2.micro"
}

variable "ami_id" {
  description = "The AMI ID to use for the EC2 instance."
}
```

main.tf

This file contains the core resources: VPC, subnet, security group, and EC2 instance.

```
resource "aws_vpc" "example_vpc" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true

  tags = {
    Name = "example-vpc"
  }
}

resource "aws_subnet" "example_subnet" {
  vpc_id     = aws_vpc.example_vpc.id
  cidr_block = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"

  tags = {
    Name = "example-subnet"
  }
}

resource "aws_security_group" "example_sg" {
  name        = "example-security-group"
  description = "Allow SSH and HTTP"
  vpc_id      = aws_vpc.example_vpc.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "example-sg"
  }
}
```

```
resource "aws_instance" "example_instance" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.example_subnet.id
  vpc_security_group_ids = [aws_security_group.example_sg.id]

  tags = {
    Name = "example-instance"
  }
}
```

outputs.tf

This file will define outputs that you might find useful, such as the public IP of the EC2 instance.

```
output "vpc_id" {
  description = "The ID of the VPC."
  value       = aws_vpc.example_vpc.id
}

output "instance_public_ip" {
  description = "The public IP of the EC2 instance."
  value       = aws_instance.example_instance.public_ip
}
```

versions.tf

This file specifies the required Terraform version and provider versions.

```
terraform {
  required_version = ">= 0.12"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```