How To Use Terraform like a Pro.

- Rajesh Bhojwani
IOD Expert

With cloud deployment, everything you do to build an application becomes code, whether business logic, configuration, infrastructure, or some other component. Infrastructure as Code (IaC) enables you to provision your servers, security, and configuration files with version control, speed, reliability, and scalability. The beauty of IaC is that it can be shared and re-used to build similar environments.

Terraform is one of the most popular IaC tools due to its platform-agnostic behavior. It can be applied to any cloud platform (AWS, GCP, etc.) and can provision any desired resource on almost any provider (GitHub, Docker, etc.). It uses HashiCorp Configuration Language (HCL) to automate this whole process.

In this post, the first in a two-part series about Terraform, I'll discuss how to use the various aspects of the Terraform configuration as effectively as possible. I'll also review some of Terraform's most advanced features, as well as several third-party tools that you can use with it.

How Terraform Works

Terraform uses a plugin-based architecture that enables developers to write new plugins or modify existing ones, according to their requirements. It has two major parts: Terraform Core and Terraform Plugins.

Terraform Core

Terraform Core is a CLI that is written in Go language. Its primary purpose is reading and interpolating configuration files and modules. It also manages state by enabling the state file to be stored either locally or remotely (such as with Amazon S3, Azure Blob Storage, and others). Terraform Core generates an execution plan when you execute the command terraform plan. It then shows the blueprint of what will be applied by Terraform and helps review it before provisioning.

Terraform also builds a resource dependency graph based on the configurations and parallelizes the creation and modification of these resources by traversing each node. Additionally, it communicates with all the plugins over RPC.

Terraform Plugins

Terraform Plugins are executed as a separate process, in which each provisioner provides an implementation of a specific service through the plugin. They are dynamically discovered by Terraform Core using Terraform's discovery process during terraform init.

Terraform Best Practices

Now that I've reviewed how Terraform works, let's get into how to best utilize it for IaC.

Modules

Modules are a major building block for Terraform configuration, and, these days, I simply can't imagine a Terraform script without them. Modules allow you to group together multiple resources in one container and enable the reuse of configuration. They can be called multiple times and can be shared with other configurations as well.

Here is an example of modules in AWS

```
--Module
   --- LoadBalancer
        --- main.tf
        --- variable.tf
  --- EKS
      --- main.tf
      --- output.tf
      --- variable.tf
 --- SecurityGroup
     --- main.tf
     --- output.tf
     --- variable.tf
```

Structuring a Terraform Configuration

Structuring a Terraform configuration is very important in order to maintain readability. As you add more and more resources, it can become difficult to understand the configuration flow, so it's best practice to segregate the different aspects of the configuration in different files.

Here is one of example

```
--- tf/
        --- modules/
          --- applications/
                -- backend/
                    --- env/
    --- dev.tfvars
    --- qa.tfvars
    --- prod.tfvars
         --- main.tf
    --- frontend/
          --- env/
                --- dev.tfvars
    --- qa.tfvars
    --- prod.tfvars
           --- network/
               ---  main.tf
               ---  dns.tf
               ---  output.tf
               --- variables.tf
```

State Persistence

By default, Terraform creates state files locally. This makes it difficult to share with others who might be working on the same Terraform configuration and to run Terraform script by other members. For this reason, it's always best to store the state file in a centralized location, such as Amazon S3 or Azure Blob Storage. Some teams use GIT and enable version control, but this is not good practice, since the file is not encrypted and may expose sensitive data, such as AWS credentials.

Also, you should enable locking to this state file, which ensures that only one process is running on a state file and prevents data loss and data corruption of the state file. Backend resources are responsible for state locking, and they need to support locking for the storage they use. For example, Amazon S3 uses Amazon DynamoDB for consistency checks. With a single DynamoDB table, you can lock multiple remote state files.

Variable Files

Variables are the best way to manage the input data to configuration, especially when you're managing multiple environments. There are several options in Terraform for passing the values:

- Variable.tf
- Var command-line
- Variable definition files (- .tfvars)
- Environment variables

Credentials Management

Never store credentials directly in a .tf file. This is a plain text file, and whoever has access to version control can easily gain access to the credentials. I recommend using environment variables as a first step to storing secrets.

```
$ export AWS_ACCESS_KEY_ID=”accesskey”
$ export AWS_SECRET_ACCESS_KEY=”secretkey”
$ export AWS_DEFAULT_REGION=”ap-southeast2-"
$ terraform plan
```

Tagging Resources

Terraform doesn't support traditional if/else conditions, but it does allow ternary operation for conditions.

```
condition ? caseTrue : caseFalse
prod = "${var.environment == "PROD" ? "east" : ""}"
```

For more complex use cases, such as multiple-option conditions, I recommend using map and lookup options. For example, the requirement may be that if your region is us-east1-, you need to scale the application to three instances, while if it is us-west1-, you should scale to two instances

```
/* In variables.tf */
variable "region_mapping" {
  description = "mapping for scaling
applications"
  default = {
    "us-east3" = "1-",
    "us-west2" = "1-"
  }
}
```

/* Define a lookup to get the instance count from the deployment region. */

```
resource "app" "app" {
  region = "${lookup(var.region_mapping, var.region)}"
}
```

In Terraform >=0.12, you can now loop through the existing maps and list, but you cannot generate them. Using a for expression in square brackets[ ] produces a list.

```
cidr_blocks = [
      for num in var.subnets:
      cidrsubnet(data.aws_vpc.config.cidr_block, 8, num)
    ]
```

Using a for expression in braces {} produces a map.

```
instance_ids = {
    for instance in aws_instance.config:
    instance.id => instance.private_ip
  }
```

Terraform also supports for_each for creating dynamic nested blocks. This is better than using count arguments on resources.

Terraform Tools

Now that I've reviewed most of Terraform's advanced features, I'll discuss some of the third-party tools that enhance its capabilities even further.

Terraformer

Terraform script is very good at creating new IaC, but don't forget about the existing infrastructure created manually over the last 4–3 decades. Terraform's import feature pulls the existing infrastructure. However, this just generates the current state; you still have to manually create the Terraform files.

That's where Terraformer, a CLI tool built in Go language, comes into play. Terraformer imports the already created infrastructure as HashiCorp TF files and includes the .tfstate file. It also provides

options for remote state sharing and exporting to specified bucket locations. It has a Terraform-like execution plan feature which allows you to see the blueprint of the configuration it's going to pull before it executes on the existing infrastructure.

Terragrunt

Terragrunt was launched to solve some issues Terraform had in 2016. Its main purpose was to provide the locking feature for Terraform state and configure Terraform state as code. However, Terraform adopted these features in a later release, and Terragrunt began to focus on new challenges—namely, Terragrunt keeps backend configuration and CLI arguments DRY.

Backend Configuration DRY

When defining backend configuration in Terraform, you need to define key, bucket, and other parameters. However, Terraform doesn't accept the variables or expressions in these parameters. So, if there are multiple modules, you'll have to copy/paste in each module manually, which is prone to error.You can use Terragrunt to keep your backend configuration DRY by defining it once in a root folder and inheriting that configuration in all modules.

Let's take a simple example. Here is the folder layout:

```
stage
├── backend-app
│   └── main.tf
└── postgresql
    └── main.tf
```

You can create a terragrunt.hcl file, defining the backend configuration once at root level, and then add one terragrunt.hcl file at each module to inherit it.

Root level file:

```
# stage/terragrunt.hcl
remote_state {
  backend = "s3"
  config = {
    bucket = "example-terraform-state"

    key = "${path_relative_to_include()}/terraform.tfstate"
    region         = "ap-southeast1-"
    encrypt        = true
    dynamodb_table = "example-lock-table"
  }
}
```

Module level file:

```
# stage/postgresql/terragrunt.hcl
include {
  path = find_in_parent_folders()
}
```

Final folder layout:

```
stage
├── terragrunt.hcl
├── backend-app
│   ├── main.tf
│   └── terragrunt.hcl
└── postgresql
     ├── main.tf
     └── terragrunt.hcl
```

Terraform CLI Arguments DRY

When applying a Terraform configuration, you need to provide a common variable using the -var-file option

```
$ terraform apply \
    -var-file=../../base.tfvars \
    -var-file=../region.tfvars
```

Remember these arguments every time you apply. Terragrunt provides the option to configure all these arguments in a terragrunt.hcl file, like this:

```
# terragrunt.hcl
terraform {
  extra_arguments "common_vars" {
    commands = ["plan", "apply"]

    arguments = [
      "-var-file=../../base.tfvars",
      "-var-file=../region.tfvars"
    ]
  }
}
```

Terragrunt offers many other features, such as immutable versioned Terraform modules and executing Terraform commands on multiple modules at once.

Testing Strategies

Terraform is a code, and has to be tested before you use it in production. Each team uses a different testing strategy, depending on DevOps maturity and knowledge of IaC.

Based on my experience as a DevOps engineer who has built and run Terraform modules in production for years, I recommend the following:

- Either use TFLint in your laptop or a CI/CD pipeline to validate both the structure and content of the Terraform configuration.
- Use GOSS, a YML-based open-source tool that can assert the test results (i.e., verifying if the SSH port 22 is closed or not).
- Do unit testing using RSpec-based tools, such as Serverspec and Inpec. You can also use a TDD approach, but I personally feel this slows down the development work.
- For integration testing, use kitchen-framework, which DevOps engineers used with Chef in the past. Terratest, which can test anything that has API, is another option.

Let's explore some use cases to show you how to get the most out of Terraform, simplifying your DevOps environment.

Multi-Tier Applications

Multi-tier architecture is the most common pattern for building systems. In this architecture, you generally use a two or three tier structure. In a two-tier structure, the first tier has a cluster of web servers, and the second tier is a pool of different databases used by the first tier's servers. With more complicated systems requiring API servers, caching, middleware, event buses, and so on, you can add the third tier.

With Terraform, each tier can be segregated as a collection of resources, and you can create dependencies between them using Terraform configuration. This ensures that your databases and middleware are ready before you provision your API and web servers. Terraform's advantage is that it brings scalability and resilience into the system, as each tier can be scaled automatically using configuration.

Platform as a Service (PaaS) Setup

PaaS is a great choice if you don't want to invest too much in building skills in infrastructure. Platforms like Cloud Foundry and Red Hat OpenShift are widely used and are being deployed on AWS, GCP, Azure, and other cloud platforms.

With Terraform, the platforms are enabled to scale based on demand. These platforms need regular patching, upgrades, re-configurations, and extension support, and can be enabled using Terraform configuration.

Multi-Cloud Deployment

Due to compliance requirements and/or the need to avoid vendor lock-in, many organizations have started implementing multi-cloud deployment, which helps increase availability, fault tolerance, and system resiliency.

To support Infrastructure as Code (IaC), each cloud vendor provides its own configuration tools. However, these tools are cloud specific. That's where Terraform comes into play—you can use it to support multi-cloud deployments. It also provides support to multiple cloud providers and simplifies the orchestration between each provider's resources.

Multi-Repo Environment Setup

For simple small projects, one Terraform main configuration file in a single directory is a good place to start. However, it will become a monolith over time as resources increase. Also, you'll need to have multiple environments to support deploying applications.

Terraform, however, offers several options, such as directories and workspaces to modularize your configuration so that you can manage it smoothly. You can also separate the directories for each environment, which ensures that you only touch the intended infrastructure. For example, making changes to the Dev environment won't impact the QA or Prod environments. However, this option duplicates the Terraform code and is useful only if your deployment requirements are different for each environment.

If you want to reuse the Terraform code with different environment parameters, workspace-separated environments are a better option. In this case, you will have a separate state file for each environment.

Multi-Cloud Setup with Terraform

Now that I've reviewed a few Terraform use cases, I'll explore some of them in greater detail to show you how they can be implemented. First, I'll dive deep into the multi-cloud setup configuration using Terraform.

Let's take a simple example: an httpd server being installed in AWS and Azure on a CentOS 8. Here, the same httpd server is getting deployed in multiple clouds using Terraform.

Step 1: Create a Common Variable Configuration

To start, create a common configuration file named common-variables.tf.
This file has all the variables shared among other modules. The configuration looks like this:

```
#Environment
variable "application_env" {
  type = string
  description = "Application environment
like Dev, QA or Prod"
  default = "dev"
}

#Application name
variable "application_name" {
  type = string
  description = "Name of the Application"
  default = "multiclouddemo"
}
```

Step 2: Terraform Configuration for Httpd Server on AWS

Now, create a Terraform file that has configuration for httpd server on a CentOS EC2 instance.
Define a variable file for AWS authentication, AZ, VPC, and CIDR.

```
#variables.tf
#for brevity, not putting authentication
related variables.

#AWS Region
variable "region" {
  type = string
  description = "AWS Region for the VPC"
  default = "ap-southeast1-"
}
#AWS Availability Zone
variable "az" {
  type = string
  description = "AWS AZ"
  default = "ap-southeast1-a"
}
#VPC CIDR
variable "vpc_cidr" {
  type = string
  description = "VPC CIDR"
  default = "16/10.2.0.0"
}
#Subnet CIDR
variable "subnet_cidr" {
  type = string
  description = "Subnet CIDR"
  default = "24/10.2.1.0"
}
```

Next, create a shell script that installs the httpd server.

```
#! /bin/bash
sudo apt-get update
sudo apt-get install -y apache2
sudo systemctl start apache2
sudo systemctl enable apache2
echo "<h1>Deployment on AWS</h1>" | sudo tee
/var/www/html/index.html
```

Then create all the resources in the main Terraform file.

```
#for brevity, not putting each and every
parameter name. Only keeping the ones that
are relevant for the article.

#main.tf
#Initialize the AWS Provider
provider "aws" {
 ---
}
#VPC definition
resource "aws_vpc" "aws-vpc" {
  ----
}
#subnet definition
resource "aws_subnet" "aws-subnet" {
  ---
}
#Define the internet gateway
#Define the route table to the internet
#Assign the public route table to the subnet
#Define the security group for HTTP web
server

#Centos 8 AMI
data "aws_ami" "centos_8" {
  most_recent = true
  owners = ["02342412312"]
  filter {
    name = "name"
    values = ["centos/images/hvm-ssd/centos-
-8.03amd-64
      server-*"]
  }
  filter {
    name = "virtualization-type"
    values = ["hvm"]
  }
}
```

```
#Define Elastic IP for web server
resource "aws_eip" "aws-web-eip" {
  ----
}
# EC2 Instances
resource "aws_instance" "aws-web-server" {
  ami = data.aws_ami.centos_8.id
  instance_type = "t3.micro"
  subnet_id = aws_subnet.aws-subnet.id
  vpc_security_group_ids = [aws_security_
group.aws-web-sg.id]
  associate_public_ip_address = true
  source_dest_check = false
  key_name = var.aws_key_pair
  user_data = file("aws-data.sh")
  tags = {
    Name = "${var.application_name}-${var.
application_env}-web-server"
    Env = var.application_env
  }
}
#Define Elastic IP
```

Step 3: Terraform Configuration for Httpd Server on Azure

Similar to what I just showed you for AWS, you now need to define variables for Azure authentication and resources:

```
#Azure authentication variables

#Location Resource Group
variable "rg_location" {
  type = string
  description = "Location of Resource Group"
  default = "South East"
}
#Virtual Network CIDR
variable "vnet_cidr" {
  type = string
  description = "Vnet CIDR"
  default = "16/10.3.0.0"
}
#Subnet CIDR
variable "subnet_cidr" {
  type = string
  description = "Subnet CIDR"
  default = "24/10.4.1.0"
}
# Define centos linux User related variables
```

Next, create a shell script, similar to the AWS one, which installs the httpd server on Azure with a different message:

```
#! /bin/bash
sudo apt-get update
sudo apt-get install -y apache2
sudo systemctl start apache2
sudo systemctl enable apache2
echo "<h1>Deployment on Azure</h1>" | sudo
tee /var/www/html/index.html
```

Then create all the Azure resources in the main Terraform file:

```
#main.tf

#for brevity, not putting each and every
parameter names.Only keeping the one is
relevant for the article.
#Configure the Azure Provider
provider "azurerm" {
  --
}
#Define Resource Group
resource "azurerm_resource_group" "azure-
resource_grp" {
  --
}
#Define a virtual network
resource "azurerm_virtual_network" "azure-
vnet" {
  --
}
#Define a subnet
resource "azurerm_subnet" "azure-subnet" {
  ---
}
#Create Security Group to access Web Server
resource "azurerm_network_security_group"
"azure-web-nsg" {
  ---
}
#Associate the Web NSG with the subnet
resource "azurerm_subnet_network_security_
group_association" "azure-web-nsg-
association" {
  ---
```

```
}
#Get a Static Public IP
resource "azurerm_public_ip" "azure-web-ip"
{
  ---
}
#Create Network Card for Web Server VM
resource "azurerm_network_interface" "azure-
web-nic" {
  ---
}
#Create web server vm
resource "azurerm_virtual_machine" "azure-
web-vm" {
  name = "${var.application_name}-${var.
application_env}-web-vm"
  location = azurerm_resource_group.azure-
resource_grp.location
  resource_group_name = azurerm_resource_
group.azure-resource_grp.name
  network_interface_ids = [azurerm_network_
interface.azure-web-
    nic.id]

  storage_image_reference {
   ---
  }
  tags = {
    environment = var.application_env
  }
}
#Output
output "azure-web-server-external-ip" {
  value = azurerm_public_ip.azure-web-ip.
ip_address
}
```

Now the Terraform configuration is ready for both AWS and Azure. You can run the following commands to create the multi-cloud application using Terraform:

```
$ terraform init
$ terraform apply
```

There is one additional configuration for distributing the traffic to both AWS and Azure using the same URL. For that, you can use Amazon Route 53 or Cloudflare.

Multi-Repo Environment Application

Earlier, I briefly discussed how you can use directories and workspaces to support multi- repository applications. I'll now explore how to implement workspaces to reuse the same Terraform configuration for multiple environments.

Terraform configurations generally have a default workspace. You can check this by running the following command:

```
$ terraform workspace list
   * default
```

Note: * means the current workspace.

Step 1: Create Variables

Start with defining a file named `variables.tf`:

```
variable "aws_region" {
  description = "AWS region where our web
application will be deployed."
}

variable "env_prefix" {
  description = "Environment like dev, qa or
prod"
}
```

Step 2: Define Main Configuration

Next, define a main.tf configuration defining all the resources required for a small web application:

```
#main.tf
provider "aws" {
  region = var.region
}


resource "random_country" "countryname" {
  length    = 20
  separator = "-"
}

resource "aws_s3_bucket" "bucket" {
  bucket = "${var.env_prefix}-${random_
country.countryname.id}"
  acl    = "public-read"
```

```
  policy = <<EOF
  {
      ---
  }
  EOF

  website {
    index_document = "welcome.html"
    error_document = "error.html"

  }
  force_destroy = true
}

resource "aws_s3_bucket_object" "countryapp"
{
  acl          = "public-read"
  key          = "welcome.html"
  bucket       = aws_s3_bucket.bucket.id
  content      = file("${path.module}/assets/
welcome.html")
  content_type = "text/html"

}
```

Step 3: Define Variables for Command Line Interface (CLI)

Now, define the dev.tfvars file:

```
region = "ap-southeast1-"
prefix = "prod"
```

These files can be kept in different repositories in order to isolate them. Which repository a file is kept in depends on the roles of the users who will be allowed to access them.

Step 4: Define Output File

The output file will be the same for both of the environments:

```
output "website_endpoint" {
  value = "http://${aws_s3_bucket.bucket.
website_endpoint}/index.html"
}
```

Step 5: Create Workspaces

Next, create two workspaces: one for dev and one for prod.

```
$ terraform workspace new dev
```

Once you create the dev workspace, it will become your current workspace.
Now, initialize the directory and then apply the dev.tfvars file using the flag `-var-file:`

```
$ terraform init
$ terraform apply -var-file=dev.tfvars
```

The output of this configuration will execute in the dev workspace, and the web application will launch in the browser.
You can create the prod workspace similarly by applying `prod.tfvars:`

```
$ terraform workspace new prod
$ terraform init
$ terraform apply -var-file=prod.tfvars
```

Now you should be able to run the web application in a prod environment as well.
Also, your folder structure will have three repositories. The first repository will have two workspaces: dev and prod. This ensures that the state is maintained in accordance with the flag `-var-file.`
You'll also notice that there is a separate state file for each environment/workspace.
Here is the structure of the three repositories:

```
├── README.md
├── assets
│   └── index.html
├── main.tf
├── outputs.tf
├── terraform.tfstate.d
│   ├── dev
│   │   └── terraform.tfstate
│   ├── prod
│   │   └── terraform.tfstate
└── variables.tf


├── README.md
├── dev.tfvars


├── README.md
├── prod.tfvars
```

Summary

In this article, I reviewed several use cases that show how Terraform can help DevOps processes run smoothly and enable you to maintain the infrastructure with versioning, reuse of the code, automated scalability, and much more. Just remember that I gave some of the most well known examples, but since Terraform allows the extension of its features, there are many other use cases that show how Terraform can enable IaC.