Introducción a Infraestructura como Código (IaC)

Miguel Barajas

Version 1.0, 2021

Contenido

Dedicación

A Valeria, Roberto y Carlota. Mi mundo

Prefacio

Por qué escribí este libro

Una gran parte de mi carrera profesional, la he dedicado a la automatización y orquestación. Es un tema que me apasiona, sin embargo, la automatización "tradicional" cada vez se ha convertido en un *break and fix*. Es estática y requiere mucho esfuerzo mantener los flujos de trabajo, integraciones y *scripts* para que la automatización se siga dando en un mundo tan dinamico como el que vivimos en las tecnologías de la información. Desde el primer momento que descubrí los manejadores de configuración, quedé enamorado. En los tiempos de ***CFENGINE*** hacíamos cosas maravillosas que nos permitían enforcarnos realmente a la arquitectura y estrategía en vez de estar configurando pequeñas aplicaciones distribuidas una por una. Podíamos replicar cambios en miles de equipos en un momento. Luego tuve oportunidad de trabajar con ***Chef*** y ***Puppet***, los cuales son excelentes manejadores de configuración, pero no fue hasta que ***Ansible*** llegó con una forma clara y estandarizada de escribir las configuraciones, que el tema realmente empezó a despegar. Las configuraciones de *Ansible* están echas en ***YAML*** [1] Esto permite que sean claramente *leibles* por un humano.

El tema con *Ansible* es que fue concebido como un manejador de configuraciones para Sistemas Operativos y aplicaciones, pero faltaba una parte: **La Infraestructura**. No teníamos forma de tratar la configuración de la infraestructura de la misma manera que tratábamos la del sistema operativo o la aplicación, es ahí donde *Terraform* brilla, es una herramienta orientada justo a esto último. Lo cual facilita meter a la infraestructura en el proceso de ***CI/CD*** [2].

Por qué debería de leer este libro

Convenciones usadas en este libro

Recursos en linea

Como contactarme

COPIA ESTE LIBRO

Sí, copia este libro, usalo para cualquier fin, auméntalo, regalalo. Este libro está bajo la licencia CC

[1] Yet Another Markup Lenguage

[2] Continous Integration / Continous Delivery

Capítulo 1. Introducción.

Para entender la infraestructura como código (IaC por sus siglas en inglés) primero debemos entender qué es infraestructura y qué es código. Infraestructura en el contexto de informática, es el hardware, físico, virtual o lógico que soporta una aplicación o plataforma. Del mismo modo, el código son las instrucciones que permiten crear una aplicación o software. Luego entonces, la infraestructura como código, es la descripción del estado deseado de la configuración de esta infraestructura en un documento, normalmente escrito en texto plano. Al estar escrito en texto plano, esta descripción puede tratarse de la misma manera que se trata un código de software, donde podemos versionarlo, probarlo, compartirlo, etc.

Para hacer esto realidad, debemos contar con un intermediario o intérprete que comprenda esta descripción y la convierta en una configuración que sea entendida por la infraestructura. Esto nos permite automatizar la infraestructura de tal manera que esa configuración la tratemos como software, esto nos da varias ventajas que estaremos discutiendo durante la presente obra, algunas de ellas son:

- Automatización declarativa en vez de imperativa
- Versionamiento de la configuración
- Replicación sencilla
- **ESCRIBIR MAS**

Automatización de Infraestructura

La automatización de la infraestructura no es un tema nuevo, desde hace décadas, está práctica existe, dado a la necesidad de hacer más eficiente el despliegue de esta infraestructura, esto se hace mediante ***scripts*** o desarrollos que permiten que esta infraestructura se despliegue automáticamente, el problema con este acercamiento, es que no solo le tenemos que planear qué se va a desplegar, si no tambien cómo se desplegará, por lo que es una automatización **imperativa**, esto nos acarrea un problema dado que muchas veces la infraestructura no puede ser cambiada una vez desplegada, puesto que en la automatización debemos tener en cuenta todas las excepciones que pudieran ocurrir durante el ciclo de vida de la infraestructura. Por ende, la automatización imperativa funciona muy bien solo para el despliegue de nueva infraestructura, pero no para mantener el ciclo de vida completa de ella. Es decir, despliegue, cambios y decomización.

Más allá de la automatización, existe el concepto de ***orquestación*** en el que normalmente existe un software llamado **orquestador** (***orchestrator*** en inglés) el cual cuenta con los conectores necesarios para automatizar el despliegue de las diferentes capas de la infraestructura: Almacenamiento, Red, Computo, Sistema Operativo, etc).

Esto permite "orquestrar" el despliegue de la pila completa de la infraestructura. Pero de la misma manera que se hace al automatizar de manera imperativa, capa por capa, la orquestación imperativa no permite manejar el ciclo de vida completo de la infraestructura.

Manejadores de configuración

Los manejadores de configuración, han existido, de igual manera, desde hace décadas, son piezas de software que nos permiten mantener la configuración de un sistema operativo o aplicación de manera declarativa. Es decir, en vez de indicar el qué y el cómo, se describe el qué, es decir se declara el estado deseado de la configuración del Sistema Operativo o aplicación y es el manejador de configuración quien hace la conciliación entre el estado declarado y el estado actual. Es decir, que solo modifica las partes de la configuración que no concuerdan con el estado deseado. Como se manifestó, los Manejadores de configuración (***Configuration Manager*** en inglés) fueron concebidos para mantener la configuración del Sistema Operativo y/o la aplicación. Con el avance de las tecnologías de virtualización y del concepto de ***Software Defined***, el despliegue y configuración de la infraestructura, también puede ser declarada y manejada como si de un sistema operativo o una aplicación se tratara.

El crecimiento de la popularidad de IaC

Sin duda, la adopción del ***Cloud Computing*** ha permitido que la Infraestructura como código se haya popularizado, la disponibilización de la interfaces de programabilidad (APIs), así como el cobro por tiempo utilizado de la infraestructura, ha hecho que IaC sea muy popular para este tipo de modelos de consumo. Sin embargo, también e influenciado los fabricantes de infraestructura física, y estos han empezado a disponibilizar *APIs* Así como también han separado el plano de control del plano de datos, que es el principio del *hardware* definido por *software*. Empresas como ***Hashicorp*** han creado soluciones de Infraestructura como código que estaremos usando en esta obra, que lleva por nombre ***Terraform***. Terraform es una herramienta de código abierto para infraestructura como código que provee un flujo de trabajo por linea de comandos consistente para manejar cientos de servicios de nube. Terraform codifica las *APIs* de las nubes en archivos de configuración declarativa. [3]

[3] https://www.terraform.io/

Capítulo 2. Caso de estudio.

Através de este libro, estaremos creando los bloques necesarios para mantener la operación de una aplicación mantenida en la nube, la cual cuenta con varios elementos de infraestructura que definiremos como código. La idea de este caso de estudio, será que podamos desplegar esta aplicación, modificarla, escalarla, decomisarla, cambiarla de una región a otra, etc.

El casó de estudio será de la empresa imaginaría de repostería "Galletería Carlota" la cual mantiene su sistema de tienda en linea en la ***Amazon Web Service*** o ***AWS*** como le llamaremos de aquí en adelante. La aplicación tienen varias capas o niveles como se muestra en la siguiente figura, esta será la arquitectura propuesta para la aplicación. El cual cuenta con lo siguiente:

- Tres (3) Balanceador de Carga nativo de *AWS* para la capas de presentación, aplicación y base de datos.
- Una capa de presentación *web* que permite ser escalada verticalmente
- Las imágenes de los productos a vender serán almacenadas en un *S3 Bucket*
- Una capa de Aplicación que será donde se procese la logíca de negocio la cual tambien debe ser verticalmente escalable
- Una capa de Base de datos, la cual es nativa a la nube y permite ser escalada verticalmente, en este caso usaremos *CoackroachDB*

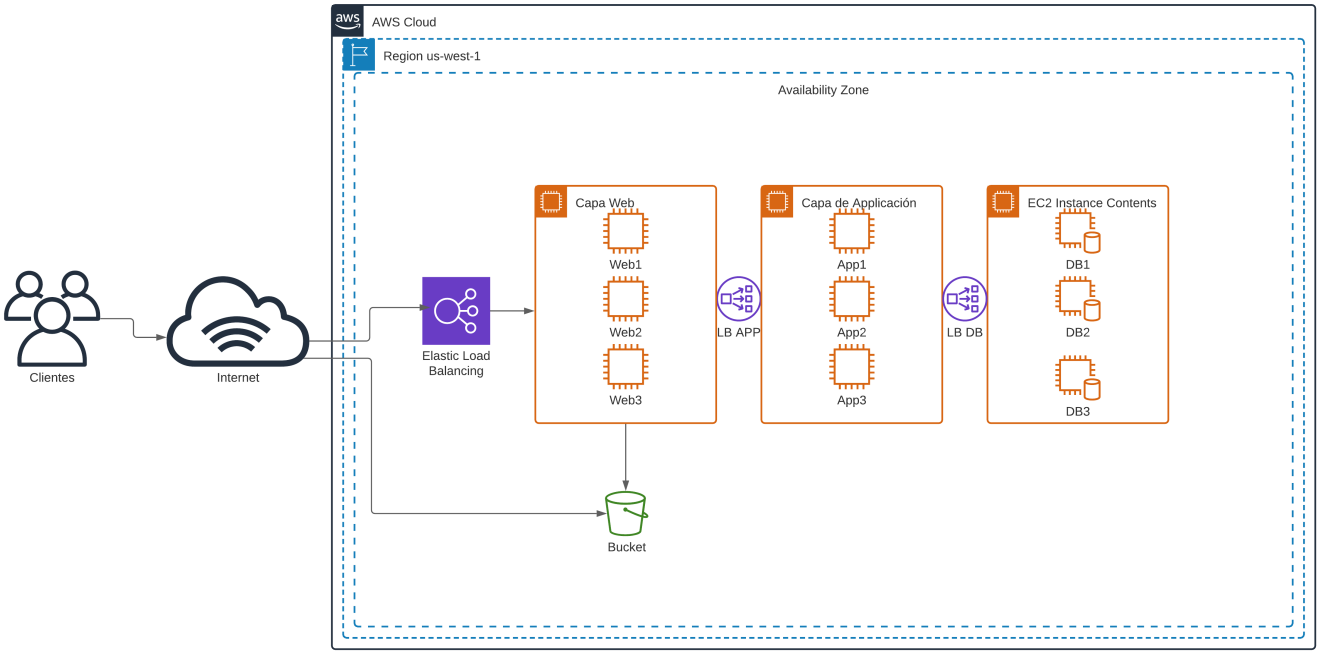

Imagen 1. Arquitectura de la Aplicación en AWS

Capítulo 3. Introducción a Terraform

Como ya se platicó en el capítulo uno, durante el desarrollo de esta obra estaremos utilizando la versión abierta y gratuita de ***Hashicorp Terraform***, la cual ha tomado mucha popularidad entre intenieros que hacen ***DEVOPS*** y ***CI/CD***, temas que trataremos más adelante. Y no es casualidad, *Terraform* ofrece un entorno bien documentado y con cientos de integraciones que permiten a las organizaciones tratar su infraestructura como código al escribir archivos con la declaración del estado deseado de las configuraciones. *Terraform* usa su propio lenguage llamado ***Hashicorp Configuration Language (HLC)*** qué permite una descripción consisa de los recursos usando bloques de código, argumentos y expresiones.

De la misma manera, *Terraform* permite correr las verificaciones necesarias antes de aplicar las configuraciones deseadas para asegurarse que los cambios a realizar son factibles y el operador tiene oportunidad de entender qué cambios sucederán y como afectará el ambiente desplegado. Una vez hecho esto el operador puede proceder a aplicar los cambios, para poder llegar al estado deseado solo modificando el delta entre el estado actual y el deseado.

Esta introducción, casi en su totalidad es una traducción simple de la documentación oficial de Hashicorp [intro_terraform]

Código Ejemplo escrito en HCL para AWS

```
variable "ami_id" {
  type        = string
  description = "AMI ID to use"
  default     = "ami-09d95fab7fff3776c"
}

variable "instance_type" {
  type        = string
  description = "Instance type to use"
  default     = "t3.micro"
}

variable "availability_zone" {
  type        = string
  description = "Availability Zone to use"
  default     = "us-east-1a"
}
```

Características

Algunas de las características más importantes de *Terraform* son:

- **Archivos de Configuración Declarativa:** Definición de Infraestructura como código para manejar el ciclo de vida completo, creación de nuevos recursos, manejo de los existentes así como la decomisación cuando estos ya no son necesarios.

- **Modulos instalables:** Intalación automática de modulos de la comunidad o terceros desde el registro de *Hashicorp* por medio de `terraform init`.
- **Planeación y predicción de cambios:** *Terraform* permite a los operadores hacer cambios a la infraestructura de manera segura y predectible.
- *Graficación de dependencias
- **Manejo del estado:** Mapeo de la configuración de los recursos del mundo real, mantenimiento de los *metadatos* y mejoramiento del *performance* en infraestructuras grandes
- **Registro de más de 500 proveedores:** El operador puede escoger de una serie de proveedores para las diferentes plataformas de nube disponibles en el mercado.

Casos de uso

Algunos de los casos de uso que se pueden cubrir con terraform son: [4]

- Despliegue de aplicaciones en *Heruku*.
- Aplicaciones Multi Capa. [5]
- *Clusteres* de Autoservicio.
- Demostraciones de *Software*.
- Ambientes deshechables.
- Despliegues multi nube.

Instalación de *Terraform*

En esta sección estaremos discutiendo la instalación del binario de *Terraform* en nuestro equipo personal, donde crearemos un ambiente de desarrollo para escribir, probar y desplegar nuestra infraestructura. Existen varios métodos para la instalación de *Terraform* a continuación discutiremos los más importantes.

Instalación Manual

Para la instalación manual de *Terraform*, necesitamos encontrar el paquete apropiado [6] para nuestro sistema operativo y bajarlo, este será un archivo *zip*.

Una vez que se haya bajado *Terraform*, procederemos a expander el archivo *zip*. Encontraremos que es un solo binario llamado `terraform`. Todos los demás archivos que pudiera contener el archivo *zip* pueden ser borrados de manera segura sin que esto afecte el funcionamiento de *Terraform*.

Finalmente, nos aseguraremos de que `terraform` se encuentre disponible en nuestro `PATH`. Esto se realiza de manera diferente, dependiendo el sistema operativo.

Mac o Linux

Obtenemos la lista de rutas que están disponibles en la variable de entorno `PATH`

```
echo $PATH
```

Movemos el binario te *Terraform* a uno de las rutas listadas. Este comando asume que el binario se encuentra en el archivo de descargas y que `PATH` contiene `/usr/local/bin`, personaliza en caso de las rutas en tu sistema operativo sean diferentes.

```
mv ~/Downloads/terraform /usr/local/bin/
```

Windows

En la siguiente dirección de internet, podemos encontrar las instrucciones exactas para modificar el `PATH` en *Windows* através de la interfaz gráfica: https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows

Instalación con *Homebrew* en *MacOS*

Homebrew es un manejador de paquetes de fuente abierta para el sistema operativo *MacOS*. Instala la *formula* oficial de *Terraform* desde la terminal.

Primero, installamos el *tap* de *HashiCorp*, un repositorio para todos los paquetes de *Homebrew* de la compañia:

```
brew tap hashicorp/tap
```

Ahora, Instalamos *Terraform* con `hashicorp/tap/terraform`

```
brew install hashicorp/tap/terraform
```

Para actualizar a la última versión, ejecutamos:

```
brew upgrade hashicorp/tap/terraform
```

Instalación con *Chocolatey* en *Windows*

Chocolatey es un manejador de paquetes de codigo abierto para *Windows*. Installamos el paquete de *Terraform* desde la linea de comandos.

```
choco install terraform
```

Instalación en Linux

Ubuntu/Debian

Agregamos la llave *GPG* de *HashiCorp*.

```
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
```

Agregamos los repositorios oficiales de *HashiCorp* para *Linux*.

```
sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com
$(lsb_release -cs) main"
```

Actualización e instalación.

```
sudo apt-get update && sudo apt-get install terraform
```

CentOS/RHEL

Instalamos `yum-config-manager` para manejar repositorios.

```
sudo yum install -y yum-utils
```

Usamos `yum-config-manager` para agregar el repositorio oficial de *HachiCorp* para *Linux*

```
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
```

Instalamos

```
sudo yum -y install terraform
```

Fedora

Instalamos `dnf config-manager` para manejar repositorios.

```
sudo dnf install -y dnf-plugins-core
```

Usamos `dnf config-manager` para agregar el repositorio oficial de *HachiCorp* para *Linux*

```
sudo dnf config-manager --add-repo
https://rpm.releases.hashicorp.com/fedora/hashicorp.repo
```

Instalamos

```
sudo dnf -y install terraform
```

Verificación de la instalación.

Para verificar la instalación abrimos una nueva terminal y ejecutamos `terraform -help`

```
terraform -help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
...
```

Cualquier sub comando despues de `terraform -help` permite aprender más sobre el mismo.

```
terraform -help plan
```

Con esto finalizamos la instalación de *Terraform* y estamos listos para empezar a construir infraestructura como código.

Contrucción de Infraestructura

Con *Terraform* installado, estamos listos para crear nuestra primera infraestructura.

Vamos a aprovisionar una ***Amazon Machine Image (AMI)*** en ***AWS*** esto lo haremos de esta manera, dado de las *AMIs* son muy populares.

Pre requisitos

Para continuar, necesitamos:

- Una cuenta de *AWS*
- La intefaz de Linea de comando de *AWS*
- Las credenciales de *AWS* configuradas localmente

Esto está descrito en el Apéndice Creación y configuración de una cuenta de Amazon Web Services

Escribir configuraciones

El set de archivos que es usado para describir la infraestructura en *Terraform* es conocido como *Terraform Configuration*. Vamos a escribir nuestra primera configuración ahora para lanzar una instancia de *EC2 de AWS*.

Cada configuración debe estar en su propio directorio. Crearemos un directorio para esta nueva

configuración.

```
mkdir iac-libro-aws-instancia
```

Nos cambiamos al directorio recién creado

```
cd iac-libro-aws-instancia
```

Pegamos la configuración que está a continuación dentro de un archivo que tenga por nombre `ejemplo.tf` y lo guardamos. *Terraform* carga todos los archivos en el directorio actual que tengan la extención `.tf`

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 2.70"
    }
  }
}

provider "aws" {
  profile = "default"
  region  = "us-west-2"
}

resource "aws_instance" "ejemplo" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
}
```

Esta es una configuración completa de *Terraform* y está lista para ser aplicada. En las siguientes secciones, veremos como funciona cada uno de estos bloques en más detalle.

Bloques de *Terraform*

El bloque `terraform {}` es requerido para que *Terraform* conosca qué proveedor tiene que descargar desde el registro de *Terraform*. En la condiguración anterior, el proveedor `aws` está definido como `hashicorp/aws` que es una abreviación de `redistry.terraform.io/hashicorp/aws`.

También podemos asignar una versión definida en el bloque `required_providers`. El argumento `version` es opcional, pero recomendado.

Proveedores

El bloque `provider` configura el nombre del proveedor, en nuestro caso `aws`, que es responsable de crear y manejar los recursos. Un proveedor es un *plugin* que *Terraform* usa para traducir las

interacciones con las *API* del servicio a administrar. Un proveedor es el responsable por entender tales interacciones y exponer los recursos. Por el hecho de que *Terraform* puede interactuar con cualquier *API*, podemos representar casi cualquier tipo de infraestructura como recursos en *Terraform*.

El atributo `profile` en nuestro bloque de proveedor se refiere, en este caso, a las credenciales de *AWS* almacenadas en la el archivo de configuración de *AWS*, que se creo cuando se hizo la configuración del *CLI* de *AWS*. La mejor práctica recomendada, es que no se use credenciales directamente en los archivos `.tf`.

Pueden existir multiples bloques de proveedor en caso de ser necesario. Podemos, incluso, usar multiples proveedores juntos. Por ejemplo podemos pasar el identificador de una intancia de *AWS* para monitorear tal recurso con *DataDog*.

Recursos

El bloque de `resources` define una pieza de la infraestructura. Un recurso puede er un componente físico o virtual como una instancia de *EC2* o puede ser un recurso lógico como una *IP elástica*.

El bloque de recursos tiene dos entradas antes del bloque como tal: el tipo de recurso y el nombre del recurso. en este ejemplo, el tipo de recurso es `aws_instance` y el nombre es `ejemplo`. El prefijo en el tipo de recurso se mapea al proveedor. En nuestro caso `aws_instance` automaticamente le dice a *Terraform* que este serña manejado por el proveedor *aws*.

Los argumentos del recurso están dentro del bloque del recurso. Los artumentos pueden ser cosas como el tamaño de la máquina, el nombre de la imagen del disco, o identificadores del *VPC* donde se desplegará la instancia. Para entender los argumentos opcionales y requeridos de cada tipo de recurso, es necesario consultar los documentos de referencia de *HashiCorp* [7]. En el caso de la instancia de *EC2* que crearemos espeficicamos un *AMI* para *Ubuntu* y el tamaño requerido será `t2.micro` que califica como *free tier* en nuestra cuenta de *AWS*.

Inicializando el directorio

Cuando creamos una nueva configuración (o hacemos *checkout* de una configuración existente desde el control de versiones) necesitamos inicializar el directorio con `terrarom init`.

Terraform usa una arquitectura basada en *plugins* que soporta cientos de proveedores de servicios de infraestructura. Inicializar un directorio de configuracion, descarga e instala los proveedores usados en la configuración qué, en este caso, es el proveedor `aws`. Los comandos subsecuentes, usarán los datos y configuraciones locales hechos durante la inicialización.

Iniciemos el directorio.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 2.70"...
- Installing hashicorp/aws v2.70.0...
- Installed hashicorp/aws v2.70.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Terraform descarga el proveedor `aws` y lo instala en un subdirectorio oculto en el directorio actual. La salida muestra la versión del plugin que fue instalada.

Formato y validación de la configuración

Una mejor práctica es usar un formato consistente en los archivos y módilos escritos por diferentes equipos. El comando `terraform fmt` automáticamente actualiza la configuración en el directorio actual para una mejor lectura y consistencia.

Formatearemos nuestra condiguración. *Terraform* retornará los nombres de los archivos formateados. En este caso, nuestro archivo de configuración esta de por sí formateado de manera correcta, es por eso que *Terraform* no retornará ningún nombre de archivo.

```
$ terraform fmt
```

Para validar que la configuración es sintáctimanete válida usamos el comando `terraform validate`. Validaremos nuestra configuración, si nuestra configuración es válira *Terraform* retornará un mensaje éxitoso.

```
$ terraform validate
Success! The configuration is valid.
```

Creación de Infraestructura

En el mismo directorio donde se encuentra `ejemplo.tf` corremos `terraform apply`. Debes ver una salida similar a la mostrada a continuación.

```
terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.ejemplo will be created
  + resource "aws_instance" "ejemplo" {
      + ami                          = "ami-830c94e3"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)

## ... Salida Omitida ...

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

Si tu configuración falla al aplicarse, quizá debas personalizar la región usada o remover el VPC por default.

Esta salida nos muestra el plan de ejecución, describiendo qué acciones tomará *Terraform* para hacer los cambios en la infraestructura real para cumplir con el estado declarado en la configuración.

El formato de la salida es similar al formato de *diff* generado por herramientas como *Git*. La salida tiene un `+` junto a `aws_instance.ejemplo`, esto significa que *Terraform* creará este recurso. Debajo de esto, muestra los atributos que seran configurados. Cuando el valor desplegado es `known after`

`apply`, significa que ese valor no será conocido hasta que el recurso no sea creado.

Terraform quedará en pausa y esperará la aprobación de los cambios antes de proceder. Si algo del plan parece incorrecto o peligroso, podemos abortar de manera segura y ningun cambio será realizado en nuestra infraestructura.

Si el plan es aceptable, entonces escribimos `yes` en la entrada que nos pide confirmar para proceder. Ejecutar este plan tomará algunos minutos, dado que *Terraform* espera a que la Instancia *EC2* este disponible.

```
  Enter a value: yes

aws_instance.ejemplo: Creating...
aws_instance.ejemplo: Still creating... [10s elapsed]
aws_instance.ejemplo: Still creating... [20s elapsed]
aws_instance.ejemplo: Creation complete after 34s [id=i-0f57d1b36088f27ae]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

¡Hemos creado infraestructura hecha con Terraform! En este momento, podemos visitar la consola de *EC2* para confirmar que se haya creado la instancia. Para esto, hay que estar seguro que tengamos seleccionada la región que fue puesta en nuestra configuración del proveedor.

Inspección del estado.

Cuando aplicamos la condiguración, *Terraform* escribió datos en un rachivo llamado `terraform.tfstate`. Este archivo ahora contiene los identificadores y propiedades de los recursos que *Terraform* creo de tal manera que ahora en adelante puede administrar y destruir los recursos.

Debemos guardar este archivo de manera segura y distribuirlo solo a miembros confiables del equipo quien necesitan manejar nuestra infraestructura. En producción, la mejor práctica es que se almacene este estado de manera remota.

Inspeccionemos el estado actual usando el comando `terraform show`.

```
$ terraform show
# aws_instance.ejemplo:
resource "aws_instance" "ejemplo" {
    ami                          = "ami-830c94e3"
    arn                          = "arn:aws:ec2:us-west-2:561656980159:instance/i-
0f57d1b36088f27ae"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2c"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    hibernation                  = false
```

```
    id                          = "i-0f57d1b36088f27ae"
    instance_state              = "running"
    instance_type               = "t2.micro"
    ipv6_address_count          = 0
    ipv6_addresses              = []
    monitoring                  = false
    primary_network_interface_id = "eni-0b899ad3349b26177"
    private_dns                 = "ip-172-31-7-180.us-west-2.compute.internal"
    private_ip                  = "172.31.7.180"
    public_dns                  = "ec2-52-40-175-176.us-west-2.compute.amazonaws.com"
    public_ip                   = "52.40.175.176"
    security_groups             = [
        "default",
    ]
    source_dest_check           = true
    subnet_id                   = "subnet-31855d6c"
    tenancy                     = "default"
    volume_tags                 = {}
    vpc_security_group_ids      = [
        "sg-0edc8a5a",
    ]

    credit_specification {
        cpu_credits = "standard"
    }

    metadata_options {
        http_endpoint               = "enabled"
        http_put_response_hop_limit = 1
        http_tokens                 = "optional"
    }

    root_block_device {
        delete_on_termination = true
        device_name           = "/dev/sda1"
        encrypted             = false
        iops                  = 0
        volume_id             = "vol-0ae0e70f9b4874d59"
        volume_size           = 8
        volume_type           = "standard"
    }
}
```

Cuando *Terraform* creo esta instancia de *EC2*, tambien obtuvo mucha información sobre la misma. Estos valores pueden ser referenciados para configurar otros recursos o salidas, esto lo discutiremos más adelante.

Manejando el estado manualmente

Terraform tiene un comando llamado `terraform state` para el manejo avanzado del estado. Por

ejemplo, si tenemos un archivo de estado muy grande, podemos listar los recursos que están en el estado, los cuales podemos obtenerlos al ejecutar el subcomando `list`.

```
$ terraform state list
aws_instance.ejemplo
```

Cambios a la Infraestructura

Previamente, creamos nuestra primera infraestructura con *Terraform*: Una instancia simple de *EC2*. En esta sección, haremos modificaciones a este recurso y veremos como *Terraform* maneja estos cambios.

La infraestructura está en constante evolución, y *Terraform* fue hecho para ayudar a manejar y hacer estos cambios. Cuando escribimos cambios en las configuraciones de *Terraform*, este creará un plan de ejecución que solo modifica lo que sea necesario para alcanzar el estado deseado.

Al usar *Terraform* para hacer cambios en la infraestructura, podemos controlar las versiones no solo de la configuración, pero tambien del estado de tal manera que podamos ver cómo la infraestructura evoluciona durante el tiempo.

Pre requisitos

Esta sección asume que se realizaron los pasos previos en esta misma obra, si no, debemos crear un directorio llamado `iac-libro-aws-instancia` y pegamos el siguiente código en un archivo llamado `ejemplo.tf`.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 2.70"
    }
  }
}

provider "aws" {
  profile = "default"
  region  = "us-west-2"
}

resource "aws_instance" "ejemplo" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
}
```

Configuración

Ahora actualizaremos el `ami` de nuestra instancia. Cambiemos el recurso `aws_instance.ejemplo` debajo del bloque de proveedor en `ejemplo.tf` reemplazando el identificador del *AMI* por uno nuevo.

El siguiente código, está en formato *diff* para darnos en contexto qué debemos cambiar en la configuración

```
resource "aws_instance" "ejemplo" {
- ami           = "ami-830c94e3"
+ ami           = "ami-08d70e59c07c61a3a"
  instance_type = "t2.micro"
}
```

Esto hará el cambio del *AMI* a Ubuntu 16.04. Las configuraciones de *Terraform* están diseñadas para modificarse directamente y sabrá qué tendrá que destruir la antigua instancia y crear una nueva.

Aplicación de los cambios.

Despues de cambiar la configuración, corremos `terraform apply` de nuevo para ver cómo *Terraform* aplicará los cambios a los recursos existentes.

```
$ terraform apply
aws_instance.ejemplo: Refreshing state... [id=i-0f57d1b36088f27ae]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # aws_instance.ejemplo must be replaced
-/+ resource "aws_instance" "ejemplo" {
      ~ ami                          = "ami-830c94e3" -> "ami-08d70e59c07c61a3a" #
forces replacement
      ~ arn                          = "arn:aws:ec2:us-west-2:561656980159:instance/i-
0f57d1b36088f27ae" -> (known after apply)
      ~ associate_public_ip_address  = true -> (known after apply)
      ~ availability_zone            = "us-west-2c" -> (known after apply)
      ~ cpu_core_count               = 1 -> (known after apply)
      ~ cpu_threads_per_core         = 1 -> (known after apply)
      - disable_api_termination      = false -> null
      - ebs_optimized                = false -> null
      - hibernation                  = false -> null
      + host_id                      = (known after apply)
      ~ id                           = "i-0f57d1b36088f27ae" -> (known after apply)
      ~ instance_state               = "running" -> (known after apply)
```

```
      ~ ipv6_address_count           = 0 -> (known after apply)
      ~ ipv6_addresses               = [] -> (known after apply)
      + key_name                     = (known after apply)
      - monitoring                   = false -> null
      + network_interface_id         = (known after apply)
      + outpost_arn                  = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      ~ primary_network_interface_id = "eni-0b899ad3349b26177" -> (known after apply)
      ~ private_dns                  = "ip-172-31-7-180.us-west-2.compute.internal" ->
(known after apply)
      ~ private_ip                   = "172.31.7.180" -> (known after apply)
      ~ public_dns                   = "ec2-52-40-175-176.us-west-
2.compute.amazonaws.com" -> (known after apply)
      ~ public_ip                    = "52.40.175.176" -> (known after apply)
      ~ security_groups              = [
          - "default",
        ] -> (known after apply)
      ~ subnet_id                    = "subnet-31855d6c" -> (known after apply)
      - tags                         = {} -> null
      ~ tenancy                      = "default" -> (known after apply)
      ~ volume_tags                  = {} -> (known after apply)
      ~ vpc_security_group_ids       = [
          - "sg-0edc8a5a",
        ] -> (known after apply)
        # (3 unchanged attributes hidden)

      - credit_specification {
          - cpu_credits = "standard" -> null
        }

      + ebs_block_device {
          + delete_on_termination = (known after apply)
          + device_name           = (known after apply)
          + encrypted             = (known after apply)
          + iops                  = (known after apply)
          + kms_key_id            = (known after apply)
          + snapshot_id           = (known after apply)
          + volume_id             = (known after apply)
          + volume_size           = (known after apply)
          + volume_type           = (known after apply)
        }

      + ephemeral_block_device {
          + device_name  = (known after apply)
          + no_device    = (known after apply)
          + virtual_name = (known after apply)
        }

      ~ metadata_options {
          ~ http_endpoint               = "enabled" -> (known after apply)
```

```
          ~ http_put_response_hop_limit = 1 -> (known after apply)
          ~ http_tokens                 = "optional" -> (known after apply)
        }

      + network_interface {
          + delete_on_termination = (known after apply)
          + device_index          = (known after apply)
          + network_interface_id  = (known after apply)
        }

      ~ root_block_device {
          ~ delete_on_termination = true -> (known after apply)
          ~ device_name           = "/dev/sda1" -> (known after apply)
          ~ encrypted             = false -> (known after apply)
          ~ iops                  = 0 -> (known after apply)
          + kms_key_id            = (known after apply)
          ~ volume_id             = "vol-0ae0e70f9b4874d59" -> (known after apply)
          ~ volume_size           = 8 -> (known after apply)
          ~ volume_type           = "standard" -> (known after apply)
        }
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

[4] https://www.terraform.io/intro/use-cases.html

[5] Este es el caso de uso específico que estaremos cubriendo en esta obra

[6] https://www.terraform.io/downloads.html

[7] https://www.terraform.io/docs/providers/index.html

Capítulo 4. Introducción a Packer

Packer es una herramienta de creación de imágenes de código abierto, escrita en Go. Packer también forma parte del stack de ***Hashicorp***. Este nos permite crear imágenes de máquinas idénticas, para múltiples plataformas de destino desde una única fuente de configuración lo que facilita la gestión de la configuración y el aprovisionamiento de la infraestructura. Packer es relativamente rápido de aprender y fácil de automatizar. Cuando se usa en combinación con herramientas de administración de configuración, se pueden crear imágenes complejas y totalmente funcionales con software preinstalado y preconfigurado.

Una imagen de máquina es una unidad estática que contiene un sistema operativo preconfigurado y un software instalado. Podemos usar imágenes para clonar o crear nuevos hosts. Las imágenes ayudan a acelerar el proceso de construcción y despliegue de nueva infraestructura. Estas existen en muchos formatos, específicos para diversas plataformas y entornos de implementación.

Packer es compatible con Linux, Windows y Mac OS X. También es compatible con una amplia variedad de formatos de imagen cmo Amazon EC2, CloudStack, DigitalOcean, Docker, Google Compute Engine, Microsoft Azure, QEMU, VirtualBox, VMware y más. También cuenta con integraciones para otras herramientas, como ***Ansible*** y ***Vagrant***.

Código Ejemplo de plantilla de Packer para crear una imagen para AWS

```
{
    "variables": {
        "aws_access_key": "{{env `AWS_ACCESS_KEY_ID`}}",
        "aws_secret_key": "{{env `AWS_SECRET_ACCESS_KEY`}}",
        "region":         "us-east-1"
    },
    "builders": [{
        "access_key": "{{user `aws_access_key`}}",
        "ami_name": "packer-aws-image-{{timestamp}}",
        "instance_type": "t3.micro",
        "region": "ap-south-1",
        "secret_key": "{{user `aws_secret_key`}}",
        "source_ami_filter": {
            "filters": {
            "virtualization-type": "hvm",
            "name": "ubuntu/images/*ubuntu-xenial-20.04-amd64-server-*",
            "root-device-type": "ebs"
         },
         "owners": ["1239720102247"],
         "most_recent": true
        },
        "ssh_username": "ubuntu",
        "type": "amazon-ebs"
    }
    ],
    "provisioners": [{
        "type": "shell",
        "script": "../scripts/setup.sh"
    }]
}
```

Características

Algunas de las características más importantes de *Packer* son:

Algunos

- **Rápido despliegue de infraestructura:** Las imágenes de Packer permiten lanzar máquinas completamente configuradas y aprovisionadas en segundos. Esto beneficia también a los ambientes de desarrollo, ya que se pueden crear Boxes de Vagrant con Packer, aprovisionadas de la misma manera que en producción, lo que mantiene la paridad.
- **Detección de problemas anticipada**. Packer instala y configura el software de una máquina en el momento en que se crea la imagen. Si hay errores en este proceso, se detectarán a tiempo en las fases de construcción de un flujo o *pipeline* de CI/CD en lugar de descubrirlos cuando inicie el entorno.
- **Basado en constructores para múltiples formatos de imagen:** Los constructores o *builders*

leen la configuración y la usan para ejecutar y generar una imagen de máquina virtual. Son constructores: VirtualBox, VMware y Amazon EC2 y más. [8]. Los constructores son el equivalente a los proveedores de Terraform.

- **Portabilidad de entornos:**. Ya que Packer puede crear imágenes idénticas para múltiples plataformas,es posible ejecutar el entorno de producción en AWS,el entorno de QA en una nube privada como OpenStack y los entornos de desarrolloen Vagrant.
- **Soporte para multiples aprovisionadores:** Los *provisioners* son componentes de Packer que instalan y configuran software dentro de una máquina antes de que esta se convierta en una imagen estática. Realizan el trabajo principal de hacer que la imagen contenga software útil. Los aprovisionadores de ejemplo incluyen scripts de shell, Ansible, etc. [9]

Casos de uso

Construir imágenes es tedioso. Es normalmente un proceso manual por lo tanto es propenso a errores. Packer puede automatizar la creación de imágenes e integrarse bien con herramientas de gestión de condiguraciones como Ansible. Packer permite crear pipelines para construir e implementar imágenes, lo que a su vez nos permite producir imágenes consistentes y repetibles.

Estos son casos de uso aplicados en la industria: [10]

Consistencia ambiental

¿Tienes una infraestructura compleja, con numerosos entornos que abarcan desarrollo, pruebas, staging y producción? Packer es ideal para estandarizar esos entornos. Como admite numerosas plataformas de destino, puede crear imágenes estándar para todo tipo de plataformas.

Por ejemplo, un equipo de seguridad puede usar Packer para crear imágenes que luego se comparten con otros grupos para proporcionar el “hardening” de base que para imponer estándares entre equipos.

Entrega continua

Packer se integra bien con las herramientas de infraestructura existentes. Se puede agregar en un pipeline de implementación. Es decir, parte del pipeline será crear la imagen. Un ejemplo de esto es la fase Bake de Spinnaker [11].

Packer puede crear imagenes de Amazon Machine Images (AMI), después Terraform puede usar esas AMI cuando se crean hosts y servicios y podemos ejecutar Ansible (periódicamente) para proporcionar la configuración final y mantener nuestros hosts configurados correctamente.

Esto significa que si necesitamos un nuevo host o tenemos que reemplazar un host que no funciona correctamente, el proceso es rápido y consistente. Nuestra infraestructura se vuelve desechable, reemplazable y repetible. Es decir, manejamos un enfoque de infraestructura inmutable.

Los beneficios de una infraestructura inmutable incluyen más consistencia y confiabilidad en la infraestructura y un proceso de implementación más simple y predecible. Mitiga o previene por completo los problemas que son comunes en las infraestructuras mutables, como las diferencias de configuración.

Instalación de *Packer*

TODO

Verificación de la instalación.

TODO

Contrucción de Imágenes

Con *Packer* installado, estamos listos para crear nuestra primer imagen.

Packer usa una plantilla en formato JSON para definir una imagen. Hay tres secciones principales en el archivo: constructores, aprovisionadores y postprocesamiento.

Los constructores es lo que determina qué tipo de imagen vamos crear. Aquí es donde le decimos a Packer que queremos una imagen para AWS en formato AMI o una para Virtualbox en formato OVF.

No estamos limitados a un solo constructor. Si necesitamos una imagen idéntica para usarla en AWS y Vagrant, definimos varios constructores.

Los aprovisionadores son la siguiente sección de un archivo JSON de Packer. Una vez instalado el sistema operativo, se invoca a los aprovisionadores para configurar el sistema.

Hay una gran cantidad de opciones disponibles, desde scripts de shell básicos hasta el uso de playbooks de Ansible.Lo importante para mantener un enfoque DevOps es usar los mismos scripts que usamos en un servidor de producción pero aplicados a un entorno de desarrollo local con Vagrant. De esta manera el entorno de Desarrollo y Producción mantendrán paridad.

Por último, están los postprocesadores. Estos son opcionales. Por ejemplo, son necesarios para crear boxes de Vagrant. Estas se generan tomando una imagen genérica en OVF para Virtualbox y empaquetándola como una imagen de Vagrant. Otras opciones comúnmente usadas en los postprocesadores son la compresión de la imagen.

Pre requisitos

Para continuar, necesitamos:

- Una cuenta de *AWS*
- La intefaz de Linea de comando de *AWS*
- Las credenciales de *AWS* configuradas localmente

Esto está descrito en el Apéndice Creación y configuración de una cuenta de Amazon Web Services

Escribir plantillas de Packer

TODO

[8] https://www.packer.io/docs/builders

[9] https://www.packer.io/docs/provisioners

[10] https://www.packer.io/intro/use-cases

[11] https://spinnaker.io/setup/bakery/#packer-templates

Apéndice A: Creación y configuración de una cuenta de *Amazon Web Services*

TODO

Glosario

AWS

: Amazon Web Services

Bibliografía

- [intro_terraform] HashiCorp. (2020). Introduction to Infrastructure as Code with Terraform. 2021, de Hashicorp Sitio web: https://learn.hashicorp.com/tutorials/terraform/infrastructure-as-code?in=terraform/aws-get-started