Terraform

Es un software que usamos para desplegar infraestructura en un proveedor de nube a partir de plantillas de código declarativo. Se interpone entre nosotros y la API de nuestro proveedor de nube utilizando Terraform provider.

IMPORTANTE TENER EN CUENTA ANTES:

SIEMPRE es una buena práctica crear un usuario específico con los permisos necesarios para Terraform (y para cualquier otra cosa que vayamos a hacer en la nube). NUNCA usar la cuenta root si no es estrictamente necesario.

Para darle los permisos lo añadiremos a grupos (como por ejemplo, AdministratorAccess).

Asegúrate de copiar tu Access Key y tu Secret Key siguiendo las instrucciones ya que no se puede volver a ver.

AWS advertirá por correo, automáticamente por seguridad sobre cualquier documento subido a Internet que los contenga la Access Key y la Secret Key. Por ejemplo, si accidentalmente se filtra al subirlo a un repositorio público de GitHub.

Creación de alias en Terraform:

alias tf="terraform"

hacemos que tf signifique los mismo que terraform. De esta forma podríamos ahorrarnos tiempo

Primeros pasos:

Añadir al main.tf la versión de Terraform que usaremos (en este caso la que es para AWS):

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

Luego hay que definir el proveedor:

```
provider "aws" {
  region       = "..."
```

```
  access_key = "..."
  secret_key = "..."
}
```

¿Cómo crear recursos?

```
resource "aws_instance" "example" {
  ami           = "ami_id"
  instance_type = "t2.micro"
}
```

Un recurso en Terraform es lo que se va a crear en nuestro proveedor de la nube. En este caso estaremos creando una instancia EC2 en AWS, tipo t2.micro, que se llame “example”, a partir de una AMI que podemos conseguir de AWS.

Utilización

terraform init

Se encarga de descargar toda la información (plugins) que requiere para contactar con el proveedor (mediante APIs).

Inicializa el backend. En el backend se guarda el estado de nuestra infraestructura. También descarga los módulos que usemos.

terraform fmt

Sirve para que Terraform formatee los archivos que tenemos a un formato correcto y estándar en caso de que nos hayamos puesto muchos espacios, tabulaciones, etc…

Es bueno tener en cuenta que el formato no afectaría al funcionamiento del archivo, porque éste funcionaria de todas formas, sino más bien a su legibilidad. Así que este comando solo hará que sea más legible para quien lo abra.

terraform plan

Debemos usar este comando antes de apply. Lo que hace es mostrarnos qué planea hacer terraform con nuestros artefactos. Es decir, los recursos que añadirá, si algunos cambiarán y/o los que destruirá. Así tendremos una previsualización y podremos ver si hemos configurado todo correctamente antes de desplegar los cambios.

También nos avisa de si hay algún error en el código y donde está.

terraform apply

Sirve para aplicar los cambios. Nos muestra lo que creará. Pide confirmación, escribimos yes o no. Si decimos yes lo ejecuta, si decimos no, no. Podemos saltar esta confirmación usando:

terraform apply -auto-approve

terraform destroy

Sirve para borrar todo lo que ha creado anteriormente en el despliegue. Nos muestra todo lo que hará y también pide confirmación. Podemos saltarla usándolo con:

terraform destroy -auto-approve

Es una buena práctica usarlo si estamos experimentando para que no cobren, o no cobren más de lo necesario por un servicio que no estamos usando.

Variables y Outputs:

Uso de variables en Terraform:

Nos puede servir para no tener que hardcodear datos sensibles, sino que se los pasamos como variables. Veamos los tipos:

- **Introducirlas manualmente** cuando las pida. Si tenemos un trozo de código donde incluyamos variables no especificadas en otra parte. Ejemplo:

  ```
  provider "aws" {
    region       = "..."
    access_key = var.access_key
    secret_key = var.secret_key
  }
  ```

  Entonces nos preguntará los valores de esas variables al ejecutarlo, y lo introduciremos manualmente.

- **Default:**

  Por ejemplo podemos crear variables en un archivo variables.tf y asignarles el valor default de lo que sea.

  Ejemplo:

  ```
  Variable "access_key" {
    Description = "access_key"
    Type        = string
    Sensitive   = true    # indica que estos datos son sensibles y que no se
                          muestren por consola
  ```

Default=”1”

}

- **Env vars TF_VAR_name:**

  Si iniciamos variables de entorno que inicien con el nombre de TF_VAR_nombedelavariable, se tomará el nombre de esta variable y su valor.

- **Terraform.tfvars/terraform.tfvars.json:**

  Son los archivos que podemos usar para definir y pasarle las variables a Terraform.

  En el ejemplo anterios meteríamos

variable “access_key” {

description = “access_key”

type = string

sensitive = true

Default=”1”

}

Y también:

Variable “secret_key” {

Description = “secret_key”

Type = string

Sensitive = true (sensitive indica que estos datos son sensibles y que no se muestren por consola)

Default=”1”

}

En el main.tf, y luego sólo tenemos que añadir los valores de la variable en el terraform.tfvars:

Access_key = “tuaccesskey”

Secret_key = “tusecretkey”

***.tfvars no cargado -var-file=:**

Es para crear un archivo con un nombre cualquiera en el que vayamos a meter variables. Estas no se cargaran automáticamente. Por lo cual cuando hagamos **terraform apply** le tendremos que pasar como atributo -var-file=(nombre_archivo).tfvars para que las localice:

terraform apply -var-file=(nombre_archivo).tfvars

***.auto.tfvars or *.auto.tfvars.json:**

Estos archivos se cargan automáticamente sin necesidad del atributo -var-file.

-var and –var-file:

Es la forma que tiene mayor prioridad a la hora de pasarle variables a Terraform.

- **-var=(valor)**
  Esta forma pasa un valor directamente a una variable.
- **-var-file=(nombre_archivo)**
  Esta forma pasa un archivo con una o más variables declaradas.

Por ejemplo: **terraform apply -var=”Nombre_Instancia”**

Locals:

Es un constructor que nos ayuda a definir variables locales. Solo estarán disponibles para nuestro archivo local. Dentro de esa variable se pueden declarar, por ejemplo, los nombres de los tags que queramos definir. Aquí estamos creando una instancia EC2 a la que le ponemos ese tag:

```
locals {
    extra_tag = “extra_tag”
}


resource “aws_instance” “example” {
    ami             = “...”
    instance_type = “t2.micro”
    subnet_id       = “...”
    tags = {
        extraTag = local.extra_tag
    }
}
```

Outputs:

Los Outputs nos permiten mostrar por pantalla y utilizar en otros archivos salidas de comando como variables.

Podemos crear un archivo llamado outputs.tf. Siguiendo el ejemplo anterior, creamos un objeto de infraestructura que es aws_instance y se llama “example”. Si queremos que nos imprima la IP de la instancia por consola sólo tendríamos que escribir en el outputs.tf lo siguiente:

```
Output “instance_ip_addr” {
    Value = aws_instance.example.private_ip
}
```

Funciones:

Terraform también tiene funciones integradas que nos ayudarán a optimizar nuestro código, algunas de ellas son:

Funciones integradas (Buit-in):

- Numeric: abs, floor, log...
- String.
- Chomp: Borrar los caracteres de nueva línea.
- Format: Concatena dos strings. Por ejemplo, **format(“Hello, %s!”, “Ander”)** se convierte en Hello Ander!

Collecion funtions:

- Alltrue: Verifica que los valores de una colección sean Verdadero.
- Anytrue: Verifica que sólo uno sea Verdadero.
- Coalesce: Devuelve el primer objeto Null.
- Compact: Elimina las líneas vacías de una lista.
- Distinct: Desduplica.
- Element: Obtiene los elementos por index.
- Index: Devuelve el index de un elemento.
- Merge: Junta elementos.

Se pueden ver todas en la documentacion oficial.

Bucles:

En Terraform contamos con For_each: , bucle típico “para cada”, que se repetirá por cada valor.

Por ejemplo si queremos crear varias instancias colocamos un bucle for_each dentro del recurso. Aquí un uso:

1. Indicamos en el tfvars esto:

   Service_names = [“service1”, “service2”, “service3”]

2. Entonces, agregamos el bucle a nuestro main.tf

   for_each = var.service_names

Asi que creará el servicio 3 veces, ya que hay 3 valores en el array.

Usando la palabra each así como ${} concatenamos el bucle de arriba con esto. Por ejemplo, si en las etiquetas quisieramos poner “EC2-nombreservicio” sería así:

Name = “EC2-${each.key}”

Tambien podemos crear un diccionario a la hora de generar un output usando:

Value = { for service, i in aws_insance.example : service => i.private_ip }

Asi nos muestra los servicios y su ip.

Módulos:

Son agrupaciones de varias llamadas a la API del proveedor de nube usando Terraform. Se utilizan sobre todo para cosas que lo requieran como creación de Security Groups o VPCs. Este es uno para los security group:

https://registry.terrafrm.io/modules/terraform-aws-modules/security-group/aws/latest

Entonces los descargo/copio en un archivo.tf

Implementación en el código de las EC2:

```
resource “aws_instance” “example” {
  for_each = var.service_names
  ami                  = “...”
  instance_type        = “t2.micro”
  subnet_id            = “module.vpc.public_subnets[0]”
  vpc_security_group_ids = [module.terraform-sg.security_group_id]
  tags = {
    extraTag = local.extra_tag
```

```
    }
}
```

Solamente dándole algunos inputs dentro de su archivo.tf nos creará la VPC, lo que nos facilita el trabajo. También se pueden referenciar dentro de otros módulos. En el del security group referencio el de la VPC:

vpc_id= modue.vpc.vpc_id

Para usar los módulos, recuerda que debemos que descargarlos. Esto lo hará Terraform automáticamente con el primer comando, **terraform init**.

Si hacemos algún cambio en los recursos en el proveedor, Terraform verá que hay una diferencia entre sus plantillas y la estructura, así que, si hacemos plan, nos ayudará a resolverlo.

Estructura de archivos:

Existen varias maneras de estructurar nuestros archivos, aquí dos de ellas.

Crear una carpeta para cada uno de los entornos:

Usamos diferentes backends. Esto nos da seguridad y una reducción de error humano. Se requieren más pasos para aprovisionar la infraestructura y tenemos mayor repetición de código.

Manejar Workspaces:

Usamos un sólo backend. Reduce la duplicación de código, es sensible al error humano. Tenemos el estado de guardado en un mismo backend y el código no muestra el estado de la infraestructura.

Para crear un nuevo workspace:

terraform workspace new (nombre_workspace)

Para listar todos los Workspaces:

terraform workspace list

Para mostrar el actual:

terraform workspace show

Para cambiar a otro Workspace:

terraform workspace select (nombre_workspace)

Para borrar un Workspace:

terraform workspace delete (nombre_workspace)