

Procedimiento

Índice

1. [Instalación](#)
2. [Providers](#)
 - 2.1. [AWS Provider](#)
3. [Configuración Simple](#)
 - 3.1. [Creación de un recurso](#)
 - 3.2. [Fichero de estado](#)
 - 3.3. [Varibales](#)
 - 3.4. [Outputs y Recurso Instancias](#)
 - 3.5. [Templates](#)
 - 3.6. [Múltiples Recursos](#)
 - 3.7. [Interpolación](#)
 - 3.8. [Elastic Load Balance](#)
 - 3.9. [Security Groups](#)
 - 3.10. [Relational Data Base](#)
4. [Ejercicio Final](#)
5. [Conclusiones](#)
6. [Bibliografía](#)

1. Instalación

La instalación de Terraform un muy sencilla, para poder instalarlo tenemos que descargar un archivo comprimido que contiene el binario de **Terraform** de la la página de [Terraform](#), ahí encontraremos las diferentes opciones para los diferentes sistemas operativos y distribuciones disponibles como MacOS, FreeBSD, Linux, OpenBSD, Solaris, Windows, también encontramos opciones a la arquitectura, ya sea 32-bits, 64-bits, Arm o Arm64.

```
vagrant@ubuntu-xenial:~$ wget https://releases.hashicorp.com/terraform/0.15.3/terraform_0.15.3_linux_amd64.zip
--2021-05-07 07:37:43-- https://releases.hashicorp.com/terraform/0.15.3/terraform_0.15.3_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.133.183, 2a04:4e42:1f::439
Connecting to releases.hashicorp.com (releases.hashicorp.com)|151.101.133.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 32743141 (31M) [application/zip]
Saving to: 'terraform_0.15.3_linux_amd64.zip'

terraform_0.15.3_linux_amd64.zip 100%[=====>] 31.23M 6.66MB/s in 4.4s

2021-05-07 07:37:48 (7.12 MB/s) - 'terraform_0.15.3_linux_amd64.zip' saved [32743141/32743141]

vagrant@ubuntu-xenial:~$ ls
terraform_0.15.3_linux_amd64.zip
```

Para este caso en concreto descargamos el comprimido .zip para una máquina de distribución Linux de 64-bits.

Lo descomprimos y lo movemos al directorio /usr/bin/ para poder ejecutarlo desde cualquier lugar:

```
vagrant@ubuntu-xenial:~$ unzip terraform_0.15.3_linux_amd64.zip
Archive:  terraform_0.15.3_linux_amd64.zip
  inflating: terraform
vagrant@ubuntu-xenial:~$ sudo mv terraform /usr/bin/.
vagrant@ubuntu-xenial:~$ terraform -v
Terraform v0.15.3
on linux_amd64
```

y ya estaría instalado y listo para usarse.

2. Providers

Los providers son **"plugins"** que permiten a **Terraform** interactuar con sistemas remotos, la primera configuración de **Terraform** tiene que ser la declaración de que providers vamos a usar, así él puede instalarlos y usar los tipos de recursos (**resource types**) y fuente de datos (**data sources**).

- **resource types:**

Son los elementos más importantes al momento de configurar la infraestructura, ya que estos elementos describen uno o mas objetos como redes virtuales, instancias, etc.

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

- **data sources:**

Son fuentes de datos que nos proporciona el provider que no están incluidas en Terraform, con

estas fuentes de datos podemos buscar y crear recursos referenciando a esas fuentes de datos.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name     = "app-server"
    Tested   = "true"
  }
}
```

2.1. AWS Provider

Para hacer la integración de **Terraform** con **AWS** lo primero que tenemos que hacer es crear una cuenta **IAM** en **AWS** por tal que **Terraform** pueda crear recursos.

También tendremos que instalar el AWS CLI (command line interface) por tal de poder gestionar nuestras credenciales que tenemos en **AWS** y que nos permitirán gestionar los recursos desde **Terraform**.

- **Instalación AWS CLI:**

La instalación es muy sencilla y está descrita paso a paso y con diferentes alternativas en la [página de AWS](#), en este caso instalaremos la versión de **AWS CLI para Linux**

- Descargamos el archivo comprimido que contiene, entre otras cosas, un script para la instalación de **AWS CLI**:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
```

- Los descomprimos:

```
unzip awscliv2.zip
```

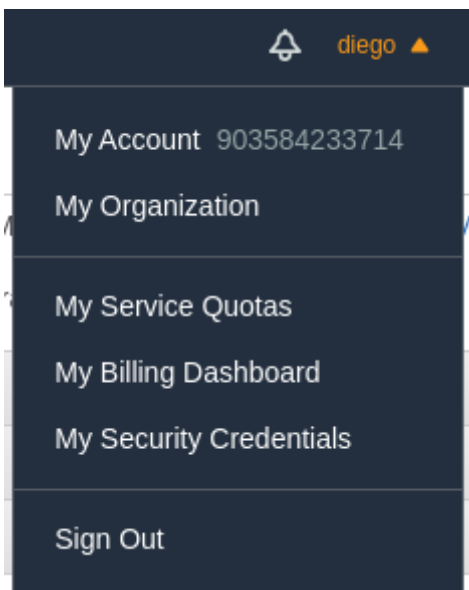
- Y ejecutamos el script de instalación:

```
sudo ./aws/install
```

Y listo ya tenemos el CLI de **AWS** instalado, lo podemos comprobar haciendo `aws --version` :

```
[didi@localhost projecte-edt]$ aws --version
aws-cli/2.2.3 Python/3.8.8 Linux/5.9.16-200.fc33.x86_64 exe/x86_64.fedora.33 prompt/off
```

Una vez instalado el CLI de **AWS** necesitamos conseguir las credenciales y el ID-Account de **AWS**, los cuales se encuentran en:



▼ Access keys (access key ID and secret access key)

Use access keys to make programmatic calls to AWS from the AWS CLI, Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time.

For your protection, you should never share your secret keys with anyone. As a best practice, we recommend frequent key rotation.
If you lose or forget your secret key, you cannot retrieve it. Instead, create a new access key and make the old key inactive. [Learn more](#)

Created	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Apr 29th 2021	AKIA5EYORHDZKP03UJWJ	2021-04-30 10:19 UTC+0200	eu-west-3	ec2	Active	Make Inactive Delete

[Create New Access Key](#)

Al momento de crear tanto las credenciales lo recomendable es guardarlas en un lugar seguro.

Para poder usar el CLI de **AWS** como autenticador de AWS tenemos ejecutar la orde
aws configure , seguidamente nos pedirá crear insertar tanto la Access Key ID y la Secret Key,
luego nos pedirá insertar la region y el output (estos dos últimos no son obligatorios).

Una vez hecho este paso las credenciales se guardarán en el fichero ~/.aws/credentials con un **profile** que es con las cuales se identificarán estas claves en el sistema operativo.

```
[isx2031424@i01 ~]$ cat .aws/credentials
[terraform]
aws_access_key_id = AKIA5EYORHDZKP03UJWJ
aws_secret_access_key = [REDACTED]
```

Podemos comprobar que tenemos conectividad haciendo por ejemplo

```
aws ec2 describe-instances --profile terraform
```

```
[didi@localhost projecte-edt]$ aws ec2 describe-instances --profile terraform
{
  "Reservations": []
}
```

En este caso podemos ver que nos responde pero no tenemos ninguna instancia creada aún.

Ahora bien, ya hemos hecho la configuración para que, desde la terminal, podamos acceder a **AWS**, pero aún queda hacer la integración para que **Terraform** pueda acceder y realizar cambios en **AWS**,

para ello tendremos que crear un fichero `.tf`, que es la extensión de ficheros de configuración que utiliza **Terraform**, y en él poner lo siguiente:

```
terraform {
  required_version = ">=0.15.1"
}

provider "aws" {
  region = "eu-west-3"
  allowed_account_ids = [ "903584233714" ]
  profile = "terraform"
}
```

- **required_version:** Es la versión mínima que queremos usar de **Terraform**.
- **region:** Es la región en la cual se encuentra el host de **AWS**.
- **allowed_account_ids:** Es el ID de la o las cuentas con las que nos conectaremos para gestionar los recursos de **AWS**.
- **profile:** Es el perfil (el que hemos creado anteriormente con **AWS CLI**) que contiene que las credenciales para poder acceder a **AWS** y gestionar los recursos.

y hacemos `terraform init` en el directori donde se encuentra este fichero `.tf`:

```
[didi@localhost punto1]$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.39.0...
- Installed hashicorp/aws v3.39.0 (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.
```

Esto lo que hará es instalar todos los plugins necesarios para poder usar los recursos que proporciona el proveedor de **AWS**.

3. Configuración Simple

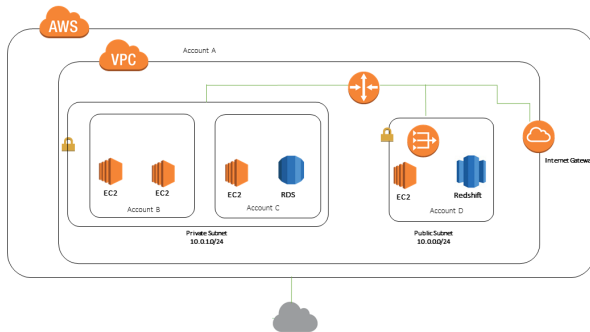
En este punto trataremos la creación de una plantilla simple de **Terraform**.

3.1. Creación de un recurso

Para este punto vamos a crear un recurso básico y simple, que es un **vpc** de **AWS**, que viene a ser una red privada donde se pueden lanzar distintos recursos de **AWS**.

Una Amazon Virtual Private Cloud (**VPC**) es un servicio que permite lanzar recursos de AWS en una red virtual aislada que nosotros definamos, en este servicio podemos crear subnets, configurar la tabla de enrutamiento y crear y configurar puertas de enlace (**gateways**)

Con la **VPC** podemos crear tanto subnet públicas (aquí podríamos poner un servidor en el que se tiene acceso desde el exterior) como subnets privadas (aquí se podría poner un base de datos por ejemplo), también podemos añadir capas de seguridad como **Security Groups** y **listas de acceso por IP**.



Para indicarle a **Terraform** que queremos crear un **vpc** tenemos que poner lo siguiente:

```
resource "aws_vpc" "vpc" {
  cidr_block = "10.0.0.0/24"
  #las instancias tienen un DNS privado
  enable_dns_hostnames = true
  enable_dns_support = true
  tags = {
    "Name" = "edt"
  }
}
```

En el cual indicamos el nombre del recurso que queremos, el nombre que le pondremos a este recurso, la ip de la red que tendrá, le asignamos un DNS privado (opcional) y le ponemos el tag de "edt".

Ahora lo que quedaría es aplicar el cambio, pero antes **Terraform** ofrece un comando que nos deja visualizar los cambios que se realizarán sin aplicarlos, así podemos comprobar que de verdad

Terraform hará lo que queremos que haga, este comando es: `terraform plan` :

```
[didi@localhost punto1]$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_vpc.vpc will be created
+ resource "aws_vpc" "vpc" {
  + arn                               = (known after apply)
  + assign_generated_ipv6_cidr_block = false
  + cidr_block                        = "10.0.0.0/24"
  + default_network_acl_id           = (known after apply)
  + default_route_table_id           = (known after apply)
  + default_security_group_id        = (known after apply)
  + dhcp_options_id                  = (known after apply)
  + enable_classiclink                = (known after apply)
  + enable_classiclink_dns_support   = (known after apply)
  + enable_dns_hostnames              = true
  + enable_dns_support                = true
  + id                               = (known after apply)
  + instance_tenancy                  = "default"
  + ipv6_association_id               = (known after apply)
  + ipv6_cidr_block                   = (known after apply)
  + main_route_table_id               = (known after apply)
  + owner_id                         = (known after apply)
  + tags                             = {
    + "Name" = "edt"
  }
  + tags_all                          = {
    + "Name" = "edt"
  }
}

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

Luego que hemos comprobado que los cambios que hará **Terraform** son los que queremos hacer, solo quedaría hacer un `terraform apply` para aplicar estos cambios.

```
[didi@localhost punto1]$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_vpc.vpc will be created
+ resource "aws_vpc" "vpc" {
  + arn                               = (known after apply)
  + assign_generated_ipv6_cidr_block = false
  + cidr_block                        = "10.0.0.0/24"
  + default_network_acl_id           = (known after apply)
  + default_route_table_id           = (known after apply)
  + default_security_group_id        = (known after apply)
  + dhcp_options_id                  = (known after apply)
  + enable_classiclink                = (known after apply)
  + enable_classiclink_dns_support   = (known after apply)
  + enable_dns_hostnames              = true
  + enable_dns_support                = true
  + id                               = (known after apply)
  + instance_tenancy                  = "default"
  + ipv6_association_id               = (known after apply)
  + ipv6_cidr_block                   = (known after apply)
  + main_route_table_id               = (known after apply)
  + owner_id                         = (known after apply)
  + tags                             = {
    + "Name" = "edt"
  }
  + tags_all                          = {
    + "Name" = "edt"
  }
}

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: yes

aws_vpc.vpc: Creating...
aws_vpc.vpc: Still creating... [10s elapsed]
aws_vpc.vpc: Creation complete after 13s [id=vpc-0cfd0b6237d78db6d]

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

Vemos que, **Terraform**, con la frase del final en color verde nos confirma que ha realizado los cambios.

También podemos ver que en algún momento nos ha pedido la confirmación para hacer los cambios, si queremos evitar esto tenemos que poner el argumento `--auto-approve` seguidamente de `terraform apply`.

<input type="checkbox"/>	edt	vpc-0cfd0b6237d78db6d	Available	10.0.0.0/24	-	dopt-c619d5ae	rtb-0a98ae081642efd12
--------------------------	-----	-----------------------	-----------	-------------	---	---------------	-----------------------

Y si nos dirigimos a la página de **AWS** vemos que nuestro **vpc** se ha creado correctamente, con los tags y la ip indicados.

3.2. Fichero de estado

Al momento de hacer el `terraform apply` en el punto anterior, es decir cuando creamos el **vpc**, **Terraform** crea un fichero de nombre **terraform.tfstate** el cual contiene el estado de la infraestructura que tiene **AWS** y de lo que se ha creado en él.

Si accedemos al fichero **terraform.tfstate** podremos ver que está declarado el recurso que hemos creado con sus respectivas características:


```

"resources": [
  {
    "mode": "managed",
    "type": "aws_vpc",
    "name": "vpc",
    "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
    "instances": [
      {
        "schema_version": 1,
        "attributes": {
          "arn": "arn:aws:ec2:eu-west-3:903584233714:vpc/vpc-0cfd0b6237d78db6d",
          "assign_generated_ipv6_cidr_block": false,
          "cidr_block": "10.0.0.0/24",
          "default_network_acl_id": "acl-055d48260db4531df",
          "default_route_table_id": "rtb-0a98ae081642efd12",
          "default_security_group_id": "sg-0b26e7b02de970ed6",
          "dhcp_options_id": "dopt-c619d5ae",
          "enable_classiclink": null,
          "enable_classiclink_dns_support": null,
          "enable_dns_hostnames": true,
          "enable_dns_support": true,
          "id": "vpc-0cfd0b6237d78db6d",
          "instance_tenancy": "default",
          "ipv6_association_id": "",
          "ipv6_cidr_block": "",
          "main_route_table_id": "rtb-0a98ae081642efd12",
          "owner_id": "903584233714",
          "tags": {
            "Name": "edt"
          },
          "tags_all": {
            "Name": "edt"
          }
        }
      }
    ]
  }
]

```

En caso de que hagamos algún cambio y se altere el fichero **terraform.tfstate**, **Terraform** crea un fichero **terraform.tfstate.backup** contiene el antiguo fichero **terraform.tfstate** por si lo necesitamos recuperar.

Este fichero es muy importante ya que, aparte de lo mencionado anteriormente, **Terraform** lo usa como punto de vista del mundo exterior, es decir de la infraestructura actual que tiene **AWS** en este caso y, por lo tanto, en caso de que este fichero sea borrado **Terraform** no sabría que hay en **AWS** y lo volvería a crear todo.

Esto también implica que si hacemos algún cambio directamente en **AWS**, cambiar el tan Name de la **vpc** por ejemplo, como el fichero **terraform.tfstate** no se ha actualizado **Terraform** seguirá pensando que la **vpc** tiene que tener el nombre que está especificado en el fichero .tf:

Cambiamos directamente en **AWS** el tag de la **vpc** de "edt" a "hola", pero sin modificar el fichero **terraform.tfstate**.

Hacemos un `terraform plan` a ver que detecta **Terraform**:

```
[didi@localhost punto2]$ terraform plan
aws_vpc.vpc: Refreshing state... [id=vpc-0cfd0b6237d78db6d]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
~ update in-place

Terraform will perform the following actions:

# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
  id           = "vpc-0cfd0b6237d78db6d"
  ~ tags       = {
    ~ "Name" = "diego" -> "edt"
  }
  ~ tags_all   = {
    ~ "Name" = "diego" -> "edt"
  }
}

[didi@localhost punto2]$ terraform plan
aws_vpc.vpc: Refreshing state... [id=vpc-0cfd0b6237d78db6d]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
~ update in-place

Terraform will perform the following actions:

# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
  id           = "vpc-0cfd0b6237d78db6d"
  ~ tags       = {
    ~ "Name" = "hola" -> "edt"
  }
  ~ tags_all   = {
    ~ "Name" = "hola" -> "edt"
  }
  # (12 unchanged attributes hidden)
}

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

Vemos que **Terraform** detecta que en su definición está puesto que el tag debe ser "edt" y en **AWS** tiene el tag "hola" por lo tanto él lo cambiará.

```
[didi@localhost punto2]$ terraform apply
aws_vpc.vpc: Refreshing state... [id=vpc-0cfd0b6237d78db6d]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
~ update in-place

Terraform will perform the following actions:

# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
  id           = "vpc-0cfd0b6237d78db6d"
  ~ tags       = {
    ~ "Name" = "hola" -> "edt"
  }
  ~ tags_all   = {
    ~ "Name" = "hola" -> "edt"
  }
  # (12 unchanged attributes hidden)
}

Plan: 0 to add, 1 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: yes

aws_vpc.vpc: Modifying... [id=vpc-0cfd0b6237d78db6d]
aws_vpc.vpc: Still modifying... [id=vpc-0cfd0b6237d78db6d, 10s elapsed]
aws_vpc.vpc: Modifications complete after 13s [id=vpc-0cfd0b6237d78db6d]

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

y si actualizamos la página de **AWS** vemos que se aplican los cambios:

También es importante fijarnos los cambios que se pueden hacer `in-place` , es decir que se pueden aplicar en caliente y lo que fuerzan a hacer un `destroy` y un `add`, esto lo sabremos haciendo `terraform plan`

3.3. Variables

Con el uso de variables tendremos la posibilidad de crear plantillas que sean modificables ya que los valores se encontrarán en un único fichero `.tf` y por lo tanto podremos acceder inmediatamente.

Para declarar el recurso de variables se hace de la siguiente manera:

variables.tf:

```
variable "cidr" {
  type = string
  default = "10.0.0.0/24"
}
```

vpc.tf:

```
resource "aws_vpc" "vpc" {
  cidr_block = "${var.cidr}"
  #las instancias tienen un DNS privado
  enable_dns_hostnames = false
  enable_dns_support = false
  tags = {
    "Name" = "edt"
  }
}
```

También es de buena práctica mantener el directorio de trabajo ordenado, y aprovechando que **Terraform** nos permite tener más de un fichero `.tf` separaremos los diferentes recursos que usemos en ficheros diferentes.

3.4. Outputs y Recurso Instancia

Para este apartado crearemos una instancia de tipo **ec2**, para ello necesitamos declarar un nuevo recurso de **AWS** llamado: `aws_instance` :

```
resource "aws_instance" "servidor-web" {
  ami = "${var.ami-id}"
  instance_type = "${var.instance-type}"
}
```

El valor de la ami que se usará y el tipo de instancia están declarados en el fichero [variables.tf](#)

```
variable "ami-id" {
  type = string
  default = "ami-0f7cd40eac2214b37"
}

variable "instance-type" {
  type = string
  default = "t2.micro"
}
```

Una vez tenemos la configuración de la instancia que queremos desplegar usaremos otro recurso llamado **output** que al momento de crearse la instancia nos saldrá por la terminal (stdout) la información que hayamos indicado al recurso **output**.

Este recurso, en determinados casos es muy útil, por ejemplo al desplegar la instancia queremos saber qué ip pública tendrá para luego poder acceder a ella y con el recurso **output** nos ahorramos tener que dirigirnos a la página web de **AWS** y buscar cuál es la dirección ip pública de la instancia creada.

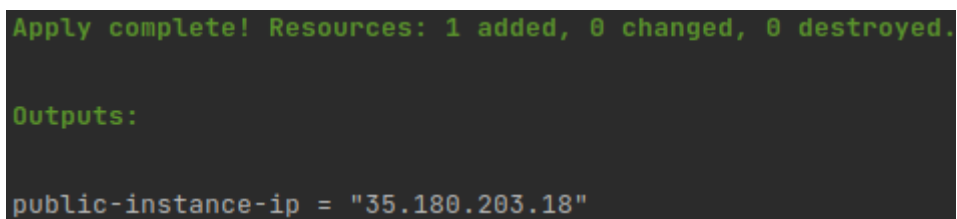
Siguiendo el criterio de orden crearemos otro fichero de nombre ["outputs.tf"](#) y ahí declararemos todos los output que queramos.

Para declarar el recurso **output** se hace de la siguiente forma:

```
output "public-instance-ip" {
  value = "${aws_instance.servidor-web.public_ip}"
}
```

Podemos ver que en **value** le indicamos el recurso de instancia, el nombre de la instancia que hayamos puesto y qué característica de la instancia queremos que muestre.

Ahora queda hacer un `terraform apply` y ver el resultado del output:



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

Outputs:

public-instance-ip = "35.180.203.18"
```

✓	-	i-00313b6a337a2a65d	Running	🔍	t2.micro	-	No alarms	+	eu-west-3a	ec2-35-180-203-18.eu-...	35.180.203.18
---	---	---------------------	---------	---	----------	---	-----------	---	------------	--------------------------	---------------

3.5. Templates

El recurso **templates** es muy útil ya que, entre otras utilidades, nos permite hacer scripts con variables que se les puede pasar para luego ejecutarlo como **user_data**, **user-data** viene a ser la utilidad que tiene **AWS** para ejecutar scripts.

Esta utilidad junto con este recurso vienen muy bien al momento de hacer el despliegue de servicios.

Para declarar el recurso **templates** tenemos que poner lo siguiente:

```
data "template_file" "install" {
  template = "${file("templates/install.tpl")}"
  vars = {
    webserver = "apache2"
  }
}
```

En verdad este recurso pertenece a otro proveedor, por lo tanto tenemos que ejecutar `terraform init` para que pueda instalar los plugins necesarios para el funcionamiento de este recurso.

El nombre del recurso será install y usará un fichero install.tpl que a su vez se encuentra en el directorio templates.

install.tpl

```
#!/bin/bash
apt-get -y install $(webserver)
service $(webserver) start
```

El fichero "install.tpl" consiste en un script que hace la instalación de apache2 en una instancia con ami de Ubutu.

Pero para que esto se pueda aplicar a la instancia que estamos desplegando, tenemos hacer uso de la utilidad **user_data** que proporcias el recurso de instancias de **AWS**:

```
resource "aws_instance" "servidor-web" {
  ami = "${var.ami-id}"
  instance_type = "${var.instance-type}"
  user_data = "${data.template_file.install.rendered}"
}
```

Le indicamos que queremos usar el *"render"* del script ya que este sustituye las variables por su valor en si, en este caso "apache2" por \$(webserver), esto se puede ver en el fichero **terraform.tfstate**:

```
"provider": "provider[\"registry.terraform.io/hashicorp/template\"]",
"instances": [
  {
    "schema_version": 0,
    "attributes": {
      "filename": null,
      "id": "e50497bdc150950c69bffffd0c760ab1719b0e5f3cba2a03a01e5456ffe3ed3bb",
      "rendered": "#!/bin/bash\napt-get -y install apache2\nservice apache2 start",
      "template": "#!/bin/bash\napt-get -y install ${webserver}\nservice ${webserver} start",
      "vars": {
        "webserver": "apache2"
      }
    }
  }
]
```


Una vez hecha la configuración, hacemos un `terraform plan` y podremos ver que este cambio requiere hacer un destroy y un add de la instancia.


```
Terraform will perform the following actions:

# aws_instance.servidor-web must be replaced
```

Ahora queda hacer un `terraform apply --auto-approve` para poder aplicar los cambios.

Edit user data
Info

Instance ID
 I-0218b4811d532a38a

Current user data
 Current user data

User data currently associated with this instance

```
#!/bin/bash
apt-get -y install apache2
service apache2 start
```

Podemos ver que en **AWS** se ha aplicado el script en `user_data`.

3.6. Múltiples recursos

Terraform tiene dos opciones muy útiles para lanzar varios recursos a la vez sin tener que poner la misma estructura repetitivamente, estas opciones son:

- **count:**

```
resource "aws_instance" "webserver" {
  ami = "ami-0f7cd40eac2214b37"
  instance_type = "t2.micro"
  count = 2
  tags {
    Name = "webserver"
  }
}
```

Como en el valor de **count** le hemos puesto 2, **Terraform** nos creará 2 instancias ec2, webserver[0] y webserver[1].

Ahora si hacemos un terraform apply podremos comprobar que las dos instancias se han creado correctamente:

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Y si vamos a la página web de **AWS** podremos ver en el **dashboard** que las dos instancias están creadas.

<input type="checkbox"/>	webserver	i-0e6357bba73f7eb92	 Running		t2.micro	-	No alarms		eu-west-3b	ec2-35-180-27-203.eu-...	35.180.27.203
<input type="checkbox"/>	webserver	i-0ec54df5f81bd05dd	 Running		t2.micro	-	No alarms		eu-west-3b	ec2-15-237-124-23.eu-...	15.237.124.23

- **autoscaling:**

Este método es un poco mas complidao pero a cambio nos da mucha más flexibilidad de opciones al momento de crear las instancias:

```

resource "aws_launch_configuration" "web-server" {
  name_prefix = "web-server-"
  image_id = "ami-0f7cd40eac2214b37"
  instance_type = "t2.micro"
  key_name = "terraform-key"
  security_groups = ["${aws_security_group.web-sg.id}"]
  user_data = "${file("templates/install.tpl")}"
  lifecycle {
    create_before_destroy = true
  }
}

resource "aws_autoscaling_group" "as-web" {
  name = "${aws_launch_configuration.web-server.name}"
  launch_configuration = "${aws_launch_configuration.web-server.name}"
  max_size = 1
  min_size = 1
  load_balancers = ["${aws_elb.elb-web.id}"]
  vpc_zone_identifier = ["${aws_subnet.publica.id}"]
  wait_for_elb_capacity = 1
  tag {
    key = "Name"
    propagate_at_launch = true
    value = "web-server"
  }
  lifecycle {
    create_before_destroy = true
  }
}

```

Como podemos ver primero creamos el recurso de ***aws_launch_configuration*** que es donde vamos a especificar todas las características que van a tener las instancias que vamos a crear posteriormente con el **autoscaling** (esto es opcional pero es de buena práctica ya que así lo tenemos mas ordenado y su vez inteligible).

Una vez creado y configurado el recurso **aws_launch_configuration** pasamos a crear y configurar el recurso que, de hecho, desplegará las instancias, este recurso es **aws_autoscaling_group**:

- **name:**
En este elemento pondremos el nombre que tendrá el recurso **autoscaling** en **AWS**
- **launch_configuration:**
Aquí le especificamos que cual es el **launch configuration** con las características de las instancias que lanzaremos.
- **max_size:**
Aqui le decimos el máximo de instancias que queremos lanzar
- **min_size:**
Aquí le decimos el mínimo de instancias que queremos lanzar

Estas dos últimas opciones son obligatorias ya que así el autoscaling sabe cuantas instancias tiene que lanzar

- **tag:**
 - **key:**
Aquí le indicamos que tipo de tag le pondremos, en este caso en nombre.
 - **propagate_at_launch:**
Esta opción es para indicarle que este nombre se aplique a todas las intancias desplegadas,
 - **value:**
Y en esta opción ponemos del valor del nombre que queremos que todas instancias tengas al momento de ser lanzadas.
- **lifecycle:**
 - **create_before_destroy:**
Esta opción es para indicarle que, en caso de que haya otro recurso **autoscaling**, primero cree el nuevo y despues destruya el anterior.

Hacemos `terraform apply` y comprobamos que los recursos e instancia se han creado correctamente:

<input checked="" type="checkbox"/>	web-server	i-025ea7dc1a4c93f13	Running	t2.micro	2/2 checks passed	No alarms	+	eu-west-3a	ec2-13-36-166-3.eu-we...	13.36.166.3
<input type="checkbox"/>	web-server-202...	ami-l0f7cd40eac2...	t2.micro	-	Sun May 09 2021 13:05:55 GMT+0200 (Central European Summer Time)					
<input type="checkbox"/>	web-server-2021050911055547020	web-server-202105091105554702...	1	-	1	1	1	1	1	eu-west-3a

3.7. Interpolación:

Este tipo de interpolación es muy útil ya que dependiendo de la región donde nos encontremos cambiarán los IDs de las amis de **AWS**, por lo tanto creamos esta utilidad para que sea mucho más fácil usar la ami correcta dependiendo de la región donde nos encontramos.

Para usar este método necesitamos crear una variable de tipo map que se rige por **key = value**, es decir que dependiendo de la region se le asignará una ami diferenete:

```

variable "aws_amis" {
  type = map
  default = {
    "us-east-1" = "ami-09e67e426f25ce0d7" #Virginia
    "us-east-2" = "ami-00399ec92321828f5" #Ohio
    "us-west-1" = "ami-0d382e80be7ffdae5" #California
    "us-west-2" = "ami-03d5c68bab01f3496" #Oregon
    "ap-northeast-3" = "ami-0001d1dd884af8872" #Osaka
    "ap-northeast-2" = "ami-04876f29fd3a5e8ba" #Seoul
    "ap-southeast-1" = "ami-0d058fe428540cd89" #Singapore
    "ap-southeast-2" = "ami-0567f647e75c7bc05" #Sydney
    "ap-northeast-1" = "ami-0df99b3a8349462c6" #Tokyo
    "ca-central-1" = "ami-0801628222e2e96d6" #Central
    "eu-central-1" = "ami-05f7491af5eef733a" #Frankfurt
    "eu-west-1" = "ami-0a8e758f5e873d1c1" #Ireland
    "eu-west-2" = "ami-0194c3e07668a7e36" #London
    "eu-west-3" = "ami-0f7cd40eac2214b37" #Paris
    "eu-north-1" = "ami-0ff338189efb7ed37" #Stockholm
    "sa-east-1" = "ami-054a31f1b3bf90920" #São Paulo
  }
}

```

Las amis son de una máquina ubuntu

Por lo tanto, una vez tenemos esta variable creada, podremos hacer uso de esta:

Por ejemplo al momento de especificar que ami usar en el **launch_configuration** del punto anterior:

```

resource "aws_launch_configuration" "web-server" {
  name_prefix = "web-server-"
  image_id = "${lookup(var.aws_amis, var.region)}"
  instance_type = "${var.instance_type}"
  key_name = "terraform-key"
  security_groups = ["${aws_security_group.web-sg.id}"]
  user_data = "${file("templates/install.tpl")}"
  lifecycle {
    create_before_destroy = true
  }
}

```

Hacemos uso de la función **lookup** que lo que hace es recibir la lista de amis que hemos creado anteriormente y la región donde estamos usando **AWS**.

En este caso la región está guardada en una variable llamada "region":

```
variable "region" {  
    type = string  
    default = "eu-west-3"  
}
```

3.8. Elastic Load Balancer:

Primero que nada, ¿qué es un Elastic Load Balancer(elb)?

Un Load Balancer acepta tráfico de entrada de peticiones provenientes de clientes o routers hacia unos "targets" registrados en **AWS**, una instancia EC2 por ejemplo, y una vez que el "target" está funcionando correctamente y puede recibir peticiones, redirige esta petición a dicho "target".

También nos proporciona un **domain name** por tal de poder acceder poniendo este en vez de la ip pública del servicio, en caso de que sea un servidor web por ejemplo.

La declaración de este recurso es la siguiente:

```
name = "edt-web"  
cross_zone_load_balancing = true  
subnets = ["${aws_subnet.publica.id}"]  
  
listener {  
    instance_port = 80  
    instance_protocol = "http"  
    lb_port = 80  
    lb_protocol = "http"  
}  
security_groups = ["${aws_security_group.elb-sg.id}"]  
}
```

- **name:**

En esta opción le especificamos el nombre que tendrá el elb en **AWS**.

- **cross_zone_load_balancing:**

Esta opción es para especificarle que actúe en más de una *"availability zone"*.

- **subnets:**

Para indicarle en que sub redes actúe el balanceador, esta opción es obligatoria si se está usando una VPC para poner todos los recursos.

- **listener:**

Tiene que poder *"escuchar"* por al menos un puerto, en este caso el puerto 80.

Load balancer: **edt-web**

[Description](#)
[Instances](#)
[Health check](#)
[Listeners](#)
[Monitoring](#)
[Tags](#)
[Migration](#)

Basic Configuration

Name	edt-web	Creation time	May 9, 2021 at 1:06:02 PM UTC+2
* DNS name	edt-web-1493575994.eu-west-3.elb.amazonaws.com (A Record)	Hosted zone	Z3Q77PNBQS71R4
Type	Classic (Migrate Now)	Status	1 of 1 instances in service
Scheme	internet-facing	VPC	vpc-07470621585198481
Availability Zones	subnet-0b379a4af6d2fcd17 - eu-west-3a		

Port Configuration

Port Configuration	80 (HTTP) forwarding to 80 (HTTP) Stickiness: Disabled
---------------------------	---

[Edit stickiness](#)

3.9. Security Groups

Este recurso sirve para añadir un security group en **AWS**, un security group es un grupo de reglas de entrada y salida en la cuales se especifica que puerto o rango de puertos quedan abiertos para determinadas ips.

La declaración de este recurso es la siguiente:

```
resource "aws_security_group" "elb-sg" {
  name = "elb-sg"
  vpc_id = "${aws_vpc.vpc.id}"
  ingress {
    from_port = 80
    protocol = "tcp"
    to_port = 80
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port = 0
    protocol = "-1"
    to_port = 0
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

En este ejemplo le estamos especificando un security group al balanceador del apartado anterior.

Estamos permitiendo el tráfico de entrada al puerto 80 (from_port y to_port) del protocolo "tcp" (protocol) desde cualquier origen (cidr_blocks).

<input checked="" type="checkbox"/>	-	sg-0e9da374ad3b332b7	elb-sg	vpc-07470621585198481 ...	Managed by Terraform	903584233714	1 Permission entry
-------------------------------------	---	--------------------------------------	--------	---	----------------------	--------------	--------------------

Reglas de entrada:

Details	Inbound rules	Outbound rules	Tags
Inbound rules (1)			
Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0

Reglas de salida:

Details	Inbound rules	Outbound rules	Tags
Outbound rules (1)			
Type	Protocol	Port range	Destination
All traffic	All	All	0.0.0.0/0

3.10. Relational Data Base:

Este recurso crea una instancia de base datos dentro de **AWS** que es llamada **RDB**.

La **RDB** es un tipo de instancia que proporciona un entorno aislado en la nube de **AWS**, este tipo de instancia también necesita tener al menos 2 subnet por si en caso falla una, empieza a actuar otra que se encuentre en otra subnet.

La declaración de este recurso es de la siguiente forma:

```
resource "aws_db_subnet_group" "subn-groups" {
  subnet_ids = ["${aws_subnet.privada1.id}", "${aws_subnet.privada2.id}"]
}

resource "aws_db_instance" "mydb" {
  instance_class = "db.t2.micro"
  identifier = "mydb"
  username = "${var.rds_username}"
  password = "${var.rds_passwd}"
  engine = "postgres"
  allocated_storage = 10
  storage_type = "gp2"
  multi_az = false
  db_subnet_group_name = "${aws_db_subnet_group.subn-groups.name}"
  vpc_security_group_ids = ["${aws_security_group.rds-sg.id}"]
  publicly_accessible = true
  skip_final_snapshot = true
}
```

Como se ha comentado antes, tenemos que crear un grupo de subnets para los RDBs.

El nombre del recurso que proporciona el RDB es `aws_db_instance` en **Terraform**.

- **instance class:**

Es donde especificamos el tipo de instancia que tendrá la RDB.

- **identifier:**

El nombre que tendrá la RDB.

- **username:**

El user con el que se creará la RDB.

- **password:**

El password que tendrá el user.

- **engine:**

Que engine de base de datos relacional va a tener la RDB, en este caso **postgresql**.

- **allocated_storage:**

El espacio de disco que tendrá la RDB.

- **storage_type**

El tipo de dispositivo de almacenaje que tendrá el disco.

- **multi_az:**

Es para indicarle si la RDB estará esparcida por diferentes availability zones.

- **db_subnet_group:**

El grupo de subntes en las que estarán cada una de las RDBs.

- **vpc_security_group_ids:**





El security group que tendrá la RDB, para poder controlar las reglas de entrada y salida que tendrá la RDB.

- **publicly_accessible:**

Esta opción es para indicar que esta RDB dipondrá de acceso público

- **skip_final_snapshot:**

Esta opción es para que al momento de elimarse esta RDB se cree una snapshot.

 mydb	Instance	PostgreSQL	eu-west-3b	db.t2.micro	 Available	 2.30%	 0 Connections
--	----------	------------	------------	-------------	---	---	---

4. Ejercicio final:

Para este ejercicio final tendremos:

- Una instancia tipo EC2 en la cual habrá instalado un servidor web con el servicio de apache y php, también contará con security group que hará referencia al security group de de un balanceador

- Habrá una instancia de RDB de postgres que también contará con su security group permitiendo el tráfico del puerto 5432.
- Y, por último, todos estos recursos y servicios estarán dentro de una **VPC**

5. Conclusiones

Con Terraform me he podido profundizar más en el mundo del despliegue y automatización de infraestructuras mediante código.

Una de las buenas cosas de **Terraform** es que, a pesar de ser relativamente nuevo en el mercado, consta de mucha documentación tanto la que proporciona la misma empresa que se encarga del desarrollo [Hashicorp](https://github.com/hashicorp/terraform) como la que hay por parte externa.

6. Bibliografía

<https://github.com/hashicorp/terraform>

<https://www.terraform.io/docs/index.html>

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

<https://openwebinars.net/>

<https://aws.amazon.com/blogs/apn/terraform-beyond-the-basics-with-aws/>

<https://www.toptal.com/devops/terraform-aws-cloud-iac>

<https://aws.amazon.com/vpc/?vpc-blogs.sort-by=item.additionalFields.createdDate&vpc-blogs.sort-order=desc>