

Terraform: de la consola al código

Describí tu infraestructura en un archivo, versionála con Git, y creála con un solo comando.

EL PROBLEMA

En la sesión 2 creaste dos EC2 desde la **consola de AWS**: hiciste click en "Launch Instance", elegiste AMI, configuraste Security Groups, copiaste IPs... Funcionó, pero:

SIN TERRAFORM

Clicks en la consola, uno por uno. Si necesitás recrear el ambiente, tenés que **recordar** cada paso. No hay historial, no hay versión, y si te equivocás no sabés qué cambió.

CON TERRAFORM

Un archivo `.tf` describe todo. Lo guardás en Git. Si necesitás lo mismo de nuevo: `terraform apply` y listo. Siempre igual, sin errores humanos.

QUÉ ES TERRAFORM

CONCEPTO CLAVE

Terraform es una herramienta de **Infrastructure as Code (IaC)**. Escribís un archivo de texto que describe qué recursos querés (EC2, S3, VPC...), y Terraform los crea, modifica o destruye en AWS por vos.

ANALOGÍA

Terraform es como un **docker-compose pero para infraestructura de AWS**. Así como `docker-compose.yml` describe tus contenedores y `docker compose up` los crea, un archivo `.tf` describe tus recursos y `terraform apply` los crea.

Es **declarativo**: no le decís los pasos ("creá esto, después aquello"), le decís el resultado que querés ("quiero una EC2 con esta AMI en esta subnet") y Terraform se encarga de llegar ahí.

CÓMO FUNCIONA: LOS 4 COMANDOS

1

**init**

2

**plan**

3

**apply**

4

**destroy**



EL LENGUAJE: HCL

Terraform usa su propio lenguaje llamado **HCL** (HashiCorp Configuration Language). Es simple — parece un JSON más legible:

```
# Esto crea un bucket S3 en AWS
resource "aws_s3_bucket" "mi_bucket" {
  bucket = "mi-primer-bucket-terraform-2026"
}
```

La estructura siempre es:

```
resource "tipo_de_recurso" "nombre_local" {
  propiedad = "valor"
}
```

resource	Palabra clave que indica que vas a crear un recurso
"aws_s3_bucket"	El tipo de recurso de AWS (siempre empieza con <code>aws_</code>)
"mi_bucket"	El nombre local — así lo referís dentro de Terraform
bucket = "..."	Las propiedades del recurso (dependen del tipo)

EL PROVIDER: CONECTARSE A AWS

Antes de crear recursos, Terraform necesita saber **con qué nube trabajar**. Eso se configura con el **provider**:

```
provider "aws" {
  region = "us-east-1"
}
```

Terraform usa tus credenciales de **AWS CLI** (las que configuraste con `aws configure`). No tenés que poner access keys en el código.

NUNCA PONGAS CREDENCIALES EN ARCHIVOS .TF

Las access keys van en `~/.aws/credentials` (donde las dejó `aws configure`). Si las ponés en un `.tf` y pushés a GitHub, cualquiera puede usarlas.

EJEMPLO COMPLETO: CREAR UN BUCKET S3

Un archivo `main.tf` completo y funcional:

```
provider "aws" {  
  region = "us-east-1"  
}  
  
resource "aws_s3_bucket" "mi_bucket" {  
  bucket = "mi-primer-bucket-terraform-2026"  
}
```

Y los comandos para ejecutarlo:

```
$ terraform init  
Initializing provider plugins...  
Terraform has been successfully initialized!  
  
$ terraform plan  
Plan: 1 to add, 0 to change, 0 to destroy.  
  
$ terraform apply  
Do you want to perform these actions? (yes)  
Apply complete! Resources: 1 added.
```

Con esas 7 líneas de código y 3 comandos, tenés un bucket S3 real en AWS.

VARIABLES: NO HARDCODEAR

En vez de escribir valores directamente, usás **variables** para que el código sea reutilizable:

```
# variables.tf  
variable "region" {  
  default = "us-east-1"  
}  
  
variable "nombre" {  
  description = "Nombre de la alumna"  
}
```

```
# main.tf - usar las variables  
provider "aws" {  
  region = var.region
```

```
resource "aws_s3_bucket" "mi_bucket" {
  bucket = "bucket-${var.nombre}-2026"
}
```

Cuando hacés `terraform apply`, te pregunta el valor de `nombre` (las variables sin default).
O se lo pasás directo:

```
$ terraform apply -var="nombre=mica"
```

EL STATE: CÓMO TERRAFORM SABE QUÉ EXISTE

Después de hacer `apply`, Terraform guarda un archivo `terraform.tfstate`. Es un JSON con la lista de todo lo que creó.

PARA QUÉ SIRVE

La próxima vez que hagas `terraform plan`, Terraform compara tu `.tf` con el state. Si cambiaste algo, te dice exactamente qué va a modificar. Si no cambió nada, te dice "no changes".

NO TOCAR EL .TFSTATE A MANO

Es un archivo que maneja Terraform automáticamente. Si lo borrás, Terraform "olvida" lo que creó y los recursos quedan huérfanos en AWS (siguen existiendo pero Terraform ya no los controla).

EJEMPLO: CREAR UNA EC2

Esto es lo que hiciste con clicks en la sesión 2, pero en código:

```
resource "aws_instance" "backend" {
  ami      = "ami-0c02fb55956c7d316" # Amazon Linux 2
  instance_type = "t2.micro" # free tier

  tags = {
    Name = "backend-sesion4"
  }
}
```

Cada recurso de AWS tiene su tipo en Terraform. Los más comunes:

<code>aws_instance</code>	EC2 — una máquina virtual
<code>aws_s3_bucket</code>	S3 — almacenamiento de archivos

<code>aws_security_group</code>	Security Group — firewall de red
<code>aws_vpc</code>	VPC — red privada virtual
<code>aws_subnet</code>	Subnet — subdivisión de la VPC
<code>aws_key_pair</code>	Key Pair — llave SSH para conectarte a EC2

TODOS LOS COMANDOS

Flujo principal	
<code>terraform init</code>	Inicializa el proyecto — descarga el provider de AWS
<code>terraform plan</code>	Muestra qué va a crear/cambiar/destruir (sin tocar nada)
<code>terraform apply</code>	Crea o modifica los recursos en AWS
<code>terraform destroy</code>	Elimina TODO lo que creó Terraform

Útiles	
<code>terraform fmt</code>	Formatea tu código (indentación, espacios)
<code>terraform validate</code>	Verifica que la sintaxis esté bien
<code>terraform show</code>	Muestra el estado actual de los recursos
<code>terraform state list</code>	Lista todos los recursos que maneja
<code>terraform output</code>	Muestra las salidas (IPs, URLs, etc.)

OUTPUTS: VER LAS IPS Y DATOS

Después de crear una EC2, quieres saber su IP. Para eso usás `output` :

```
output "ip_publica_backend"
  value  aws_instance.backend.public_ip
```

Después del `apply` , Terraform te muestra:

```
ip_publica_backend = "54.210.123.45"
```

ESTRUCTURA DE UN PROYECTO TERRAFORM

Por convención, los archivos se organizan así:

<code>main.tf</code>	Los recursos principales (EC2, S3, etc.)
<code>variables.tf</code>	Las variables de entrada
<code>outputs.tf</code>	Los valores que querés ver al final (IPs, URLs)
<code>terraform.tfstate</code>	Estado actual — NO editar, NO subir a Git
<code>.terraform/</code>	Carpeta con el provider descargado

.GITIGNORE

Agregá `terraform.tfstate`, `.terraform/` y `*.tfstate.backup` al `.gitignore`. El state puede contener datos sensibles y no debería estar en Git.

EN LA VIDA REAL

ESCENARIO: NECESITÁS UN AMBIENTE DE TESTING IGUAL AL DE PRODUCCIÓN

SIN TERRAFORM

Entrás a la consola de AWS, creás cada recurso a mano. Te olvidás un Security Group. No coincide con producción. Te lleva 40 minutos. Cuando lo querés borrar, te olvidás una EC2 corriendo y te llega la factura.

CON TERRAFORM

`terraform apply` — ambiente idéntico en 2 minutos. Cuando terminás: `terraform destroy` — borra TODO, sin dejar nada corriendo.

CONSOLA VS TERRAFORM

CONSOLA DE AWS

- Hay que recordar cada paso
- No es repetible fácilmente
- No hay historial de cambios
- Fácil olvidarse de borrar algo

TERRAFORM

- Todo queda documentado en código
- Repetible: `apply` y listo
- Versionado con Git
- `destroy` borra todo

LA EVOLUCIÓN DEL CURSO



Sesión 2: hiciste todo a mano. Sesión 3: empaquetaste tu app en contenedores. Ahora en la sesión 4: automatizas la infraestructura donde corren esos contenedores.