Introducción a Terraform y Salt Project

Unidad 03. Variables, outputs y estado en Terraform

Autor: Sergi García

Actualizado Noviembre 2025

Licencia

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

ÍNDICE

Unidad 03 - Variables, outputs y estado en Terraform

1. 🎯 Introducción a la parametrización en Terraform

La **parametrización** en Terraform consiste en **separar los valores fijos del código** de infraestructura, permitiendo definirlos como **variables** que pueden cambiar según el entorno o el uso.
Esto vuelve la infraestructura más **reutilizable, flexible y escalable**.

🧩 1.1 ¿Por qué parametrizar la infraestructura?

Sin parametrización, una configuración puede volverse rígida y difícil de mantener.
Al parametrizar, conseguimos:

Beneficio	Descripción
Reutilización	Usar el mismo código para distintos entornos (dev, test, prod).
Escalabilidad	Modificar parámetros (región, tamaño de instancia, etc.) sin cambiar el código base.
Facilidad de mantenimiento	Actualizar valores sin tocar los archivos .tf.
Automatización	Integrar variables desde pipelines o scripts externos.
Consistencia	Definir reglas y defaults centralizados para toda la infraestructura.

💡 **Ejemplo:** Vamos a ver un ejemplo donde estamos creando variables para realizar un despliegue de un contenedor Docker con Terraform.

Si defines el nombre de la imagen como variable, con un valor por defecto:

```
variable "imagen" {
  description = "Nombre de la imagen Docker a desplegar"
  default     = "nginx:latest"
}
```

Puedes usar esa variable dentro del recurso Docker:

```
provider "docker" {}

resource "docker_container" "web" {
  name  = "webserver"
  image = var.imagen
  ports {
    internal = 80
    external = 8080
  }
}
```

De esta forma, puedes desplegar el mismo contenedor con una imagen diferente sin modificar el archivo .tf, simplemente cambiando el valor de la variable desde la línea de comandos:

```
terraform apply -var="imagen=httpd:latest"
```

Terraform descargará la nueva imagen (en este caso httpd:latest) y lanzará el contenedor con esa configuración, manteniendo el código genérico y reutilizable.

🔁 1.2 Reutilización y flexibilidad de configuraciones

Terraform permite escribir infraestructura genérica, adaptable a diferentes casos mediante variables y archivos .tfvars.

💡 **Ejemplo práctico:**

Supongamos que queremos definir un contenedor Docker, pero el nombre de la **imagen** y el **puerto** pueden cambiar según el entorno.

```
variable "imagen" {
  description = "Imagen Docker a desplegar"
  type        = string
  default     = "nginx:latest"
}

variable "puerto_externo" {
  description = "Puerto externo para exponer el contenedor"
  type        = number
  default     = 8080
}

provider "docker" {}
resource "docker_container" "web" {
  name  = "webserver"
  image = var.imagen

  ports {
    internal = 80
    external = var.puerto_externo
  }
}
```

Con esta configuración, puedes reutilizar la misma infraestructura, cambiando solo los valores de las variables, sin modificar el código base.

Ejemplo:

```
terraform apply -var="imagen=httpd:latest"
```

o bien:

```
terraform apply -var="puerto_externo=9090"
```

Esto te permite usar el mismo código Terraform para desplegar distintos contenedores (Nginx, Apache, etc.) o diferentes puertos, sin duplicar archivos ni tocar la configuración principal.

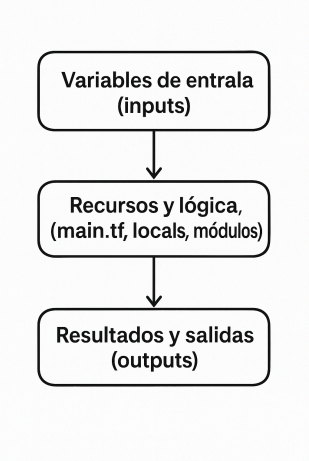
⚔️ 1.3 Comparación entre hardcodeo vs uso de variables

Enfoque	Ejemplo	Problemas / Ventajas
Hardcodeado	image = "nginx:latest"	❌ Difícil de mantener, no reutilizable, propenso a errores.
Con variables	image = var.imagen	✅ Flexible, reutilizable, fácil de automatizar.

El uso de variables es esencial para **infraestructura como código profesional**, evitando la repetición y los valores fijos.

🔄 1.4 Flujo general de paso de datos: inputs → lógica → outputs

Terraform sigue un flujo claro para el manejo de datos:



2. ⚙️ Variables en Terraform

Las variables son los parámetros de entrada que permiten personalizar el comportamiento de la configuración.
Se declaran en archivos .tf, se asignan valores mediante .tfvars, CLI o entorno, y se consumen dentro del código con var.<nombre>.

🔍 2.1 ¿Qué son las variables en Terraform?

Las variables son **contenedores de datos reutilizables** que permiten parametrizar configuraciones sin modificarlas directamente.
Cada variable tiene:

- Un **nombre.**
- Un **tipo.**
- (Opcionalmente) un **valor por defecto.**
- Una **descripción.**

Ejemplo simple eligiendo regiones para un despliegue en AWS (no visto en el curso):

```
variable "region" {
  description = "Región donde desplegar la infraestructura"
  type        = string
  default     = "us-east-1"
}
```

Y se usa así:

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

🧱 2.2 Estructura básica de una variable

```
variable "<nombre>" {
  description = "Descripción de lo que representa la variable"
  type        = <tipo de dato>
  default     = <valor por defecto>     # opcional
}
```

💡 **Ejemplo práctico:**

```
variable "instance_count" {
  description = "Número de instancias a crear"
  type        = number
  default     = 2
}
```

🧬 2.3 Tipos de datos soportados

Terraform soporta varios **tipos de datos** para sus variables, que pueden usarse también al definir configuraciones de **contenedores Docker**:

⚙️ 2.3.1 string

El contenido simplemente es texto plano.

```
variable "imagen" {
  type    = string
  default = "nginx:latest"
}
```

💡 Uso:

```
image = var.imagen
```

Permite cambiar fácilmente la imagen del contenedor (por ejemplo, nginx:latest, httpd:latest, mysql:8.0, etc.) sin modificar el código.

⚙️ 2.3.2 number

Valores numéricos (enteros o decimales).

```
variable "puerto_externo" {
  type    = number
  default = 8080
}
```

💡 Uso:

```
ports {
  internal = 80
  external = var.puerto_externo
}
```

Permite definir dinámicamente el puerto externo donde se expone el contenedor.

⚙️ 2.3.3 bool

Valores booleanos (true o false).

```
variable "enable_logs" {
  type    = bool
  default = true
}
```

💡 Uso:

```
resource "local_file" "logs" {
  count    = var.enable_logs ? 1 : 0
  filename = "logs_enabled.txt"
  content  = "Los logs del contenedor están habilitados."
}
```

Permite activar o desactivar características opcionales, como registros o monitorización.

⚙️ 2.3.4 list

Lista ordenada de elementos.

```
variable "puertos" {
  type    = list(number)
  default = [8080, 8081, 8082]
}
```

💡 Uso:

```
ports {
  internal = 80
  external = var.puertos[0]
}
```

Permite definir una lista de puertos para exponer varios servicios o contenedores.

⚙️ 2.3.5 map

Mapa de claves y valores.

```
variable "labels" {
  type = map(string)
  default = {
    "environment" = "dev"
    "project"     = "DockerInfra"
  }
}
```

💡 Uso:

```
labels = var.labels
```

Define etiquetas o metadatos para los contenedores, útiles para identificar su entorno o propósito.

⚙️ 2.3.6 object

Estructura compuesta con varios tipos de datos.

```
variable "container_config" {
  type = object({
    nombre = string
    puerto = number
    imagen = string
  })

  default = {
    nombre = "web01"
    puerto = 8080
    imagen = "nginx:latest"
  }
}
```

💡 Uso:

```
resource "docker_container" "web" {
  name  = var.container_config.nombre
  image = var.container_config.imagen

  ports {
    internal = 80
    external = var.container_config.puerto
  }
}
```

Permite agrupar toda la configuración de un contenedor en una sola variable estructurada.

⚙️ 2.3.7 any (dinámico)

Acepta **cualquier tipo de dato**. Ideal para configuraciones flexibles o genéricas, aunque **pierde validación estricta**.

```
variable "config_dinamica" {
  type = any
}
```

💡 Uso:

```
locals {
  configuracion = var.config_dinamica != null ? var.config_dinamica : "valor_por_defecto"
}
```

💬 **Ejemplo:** puede servir para recibir estructuras o lstas generadas dinámicamente desde un script o pipeline externo.

📄 2.4 Archivos .tfvars y convenciones

Los archivos .tfvars permiten definir valores externos para las variables, de modo que puedas reutilizar la misma configuración Terraform con diferentes entornos o parámetros (por ejemplo: desarrollo, pruebas o producción).

Por convención:

- Se llaman terraform.tfvars o <entorno>.tfvars
- Terraform carga automáticamente el archivo si se llama `terraform.tfvars`

💡 **Ejemplo (Docker):**

Supongamos que tienes las siguientes variables definidas en tu proyecto:

```
variable "imagen" {
  description = "Imagen Docker a desplegar"
  default     = "nginx:latest"
}
variable "" {
  description = "Puerto donde se expone el contenedor"
  type        = number
  default     = 8080
}
```

Puedes crear un archivo **terraform.tfvars** para sobreescribir los valores por defecto:

```
imagen         = "httpd:latest"
puerto_externo = 9090
```

💻 **Ejecución:**

```
terraform apply
```

Terraform detectará automáticamente el archivo terraform.tfvars y aplicará esos valores, desplegando un contenedor Apache en lugar de Nginx, y exponiéndolo en el puerto 9090.

🔄 2.5 Formas de pasar variables

Terraform permite asignar valores de varias maneras (ordenadas por prioridad):

Método	Ejemplo	Prioridad
1. Línea de comandos	terraform apply -var="imagen=httpd:latest"	🔝 Alta
2. Variables de entorno	export TF_VAR_imagen="nginx:stable"	Alta
3. Archivo .tfvars	terraform apply -var-file="produccion.tfvars"	Media
4. Valor por defecto	Definido en variables.tf	Baja

2.5.1 Interactivamente

Si una variable no tiene valor, Terraform la pedirá al usuario:

```
var.imagen
Imagen Docker a desplegar
Enter a value: nginx:latest
```

2.5.2 Archivo .tfvars

Puedes definir los valores en un archivo externo, por ejemplo:

docker.tfvars

```
imagen         = "httpd:latest"
puerto_externo = 9090
```

Y ejecutar:

```
terraform apply -var-file="docker.tfvars"
```

Terraform cargará esos valores automáticamente y desplegará el contenedor Apache en el puerto 9090.

2.5.3 Variables de entorno

También puedes pasar valores a Terraform mediante variables de entorno, anteponiendo **TF_VAR_** al nombre de la variable.

```
export TF_VAR_imagen="nginx:stable"
export TF_VAR_puerto_externo=8080
terraform apply
```

Terraform leerá esos valores y los aplicará durante el despliegue.

2.5.4 Línea de comandos

Por último, puedes pasar las variables directamente con la opción -var:

```
terraform apply -var="imagen=alpine:latest" -var="puerto_externo=8081"
```

💡 Esta opción tiene **la prioridad más alta**, ideal para sobrescribir valores puntualmente sin modificar archivos.

🧩 2.6 Validaciones de variables (validation blocks)

Terraform permite **validar el valor de una variable antes de usarla**, evitando errores y manteniendo consistencia en la configuración de contenedores Docker.Esto resulta especialmente útil para limitar qué imágenes, puertos o entornos se pueden usar en un despliegue.

💡 **Ejemplo:**

```
variable "imagen" {
  type        = string
  description = "Imagen Docker permitida para el despliegue"

  validation {
    condition     = contains(["nginx:latest", "httpd:latest", "mysql:8.0"], var.imagen)
    error_message = "La imagen debe ser nginx:latest, httpd:latest o mysql:8.0."
  }
}

variable "puerto_externo" {
  type        = number
  description = "Puerto válido de exposición del contenedor"

  validation {
    condition     = var.puerto_externo >= 1024 && var.puerto_externo <= 65535
    error_message = "El puerto externo debe estar entre 1024 y 65535."
  }
}
```

💻 **Ejemplo de uso:**

```
terraform apply -var="imagen=nginx:latest" -var="puerto_externo=8080"
```

✔️ Terraform aceptará estos valores válidos y procederá al despliegue.

❌ Pero si el usuario introduce una imagen o puerto no permitido, mostrará un error

3. ⚙️ LOCALS EN TERRAFORM

💡 3.1 ¿Qué son los locals y cómo se diferencian de las variables?

Los **locals** son **variables locales internas** a tu configuración de Terraform. Sirven para **definir valores calculados o constantes** que se utilizan dentro del mismo módulo, sin depender de entradas externas.

Concepto	Variables (variable)	Locals (locals)
Propósito	Recibir valores de entrada (desde usuario o entorno)	Definir valores calculados o reutilizables dentro del código
Origen de datos	Externos	Internos
Modificables por el usuario	✅ Sí	❌ No
Ejemplo de uso	region = var.region	server_name = "app-${var.env}"

💬 **Ejemplo básico**

```
variable "env" {
  type    = string
  default = "dev"
}

locals {
  instance_name = "web-${var.env}"
}

resource "local_file" "example" {
  filename = "${path.module}/${local.instance_name}.txt"
  content  = "Instancia para entorno ${var.env}"
}
```

📄 **Resultado:**
Terraform crea un archivo llamado web-dev.txt con el contenido "Instancia para entorno dev".

🔁 3.2 Uso de locals para evitar repetición de código

Uno de los usos principales de los locals es centralizar lógica repetida.

💡 **Ejemplo sin locals (código repetido):**

```
provider "docker" {}

variable "project_name" {
  default = "miapp"
}

variable "env" {
  default = "dev"
}

resource "docker_container" "web" {
  name  = "${var.project_name}-${var.env}-web"
  image = "nginx:latest"
}

resource "docker_container" "db" {
```

```
  name  = "${var.project_name}-${var.env}-db"
  image = "mysql:8.0"
}
```

💡 **Ejemplo con locals (mejor práctica):**

```
locals {
  name_prefix = "${var.project_name}-${var.env}"
}

resource "docker_container" "web" {
  name  = "${local.name_prefix}-web"
  image = "nginx:latest"
}

resource "docker_container" "db" {
  name  = "${local.name_prefix}-db"
  image = "mysql:8.0"
}
```

➡️ Así, si cambias “var.project_name” o “var.env”, todo el código se actualiza automáticamente sin repetir cadenas.

🧠 3.3 Buenas prácticas al usar locals

Práctica	Descripción
🏷️ **Usa nombres descriptivos**	Nombres claros como resource_prefix o formatted_tags facilitan la lectura.
📚 **Agrupa lógicamente**	Puedes declarar varios locals {} en distintos archivos si mejora la organización.
🧮 **Evita cálculos complejos dentro de recursos**	Define toda la lógica en locals y usa local.<nombre> dentro de los recursos.
🧩 **Usa locals para mantener consistencia**	Por ejemplo, definir un formato de nombre común o etiquetas compartidas.
🚫 **No abuses**	Si algo debe venir del usuario o entorno, sigue usando variable, no locals.

🧩 3.4 Ejemplos de casos comunes

Veamos algunos casos reales donde locals hacen una gran diferencia 👇

📛 3.4.1. Estandarización de nombres de recursos

```
variable "env" {
  type    = string
  default = "prod"
}
```

```
variable "project" {
  type    = string
  default = "analytics"
}

locals {
  resource_prefix = "${var.project}-${var.env}"
}

resource "local_file" "log" {
  filename = "${path.module}/${local.resource_prefix}-log.txt"
  content  = "Archivo generado para ${local.resource_prefix}"
}
```

📄 **Resultado:** Archivo → analytics-prod-log.txt

📂 3.4.2. Cálculo de rutas dinámicas

```
variable "base_path" {
  type    = string
  default = "/tmp"
}

locals {
  log_path  = "${var.base_path}/logs"
  data_path = "${var.base_path}/data"
}

resource "local_file" "config" {
  filename = "${local.log_path}/config.txt"
  content  = "Ruta de datos: ${local.data_path}"
}
```

➡️ Permite mantener consistencia entre carpetas y evita escribir rutas manualmente varias veces.

🏷️ 3.4.3. Centralización de etiquetas (tags)

En Docker, es buena práctica asignar **labels** (etiquetas) a los contenedores para identificarlos fácilmente. Con **locals**, puedes centralizar esa información y aplicarla a todos tus recursos.

```
variable "env" {
  default = "dev"
}
variable "project" {
  default = "webapp"
}

locals {
  common_labels = {
    "environment" = var.env
    "project"     = var.project
    "managed_by"  = "Terraform"
  }
}
```

```
provider "docker" {}

resource "docker_container" "web" {
  name  = "${var.project}-${var.env}-web"
  image = "nginx:latest"
  labels = local.common_labels
}

resource "docker_container" "db" {
  name  = "${var.project}-${var.env}-db"
  image = "mysql:8.0"
  labels = local.common_labels
}
```

✅ **Resultado:**
Todos los contenedores compartirán etiquetas coherentes:

```
"environment" = "dev"
"project"     = "webapp"
"managed_by"  = "Terraform"
```

🔤 3.4.4. Condiciones y cálculos dinámicos

Los **locals** también pueden contener **expresiones condicionales o cálculos** que ajusten la configuración de los contenedores según el entorno o las variables definidas.

💡 **Ejemplo:**

```
variable "env" {
  description = "Entorno de despliegue"
  default     = "prod"
}

locals {
  imagen = var.env == "prod" ? "nginx:stable" : "nginx:alpine"
  puerto = var.env == "prod" ? 80 : 8080
}

provider "docker" {}

resource "docker_container" "web" {
  name  = "web-${var.env}"
  image = local.imagen

  ports {
    internal = 80
    external = local.puerto
  }
}
```

📈 **Resultado:**

- Si env = "prod", se usa la imagen nginx:stable y se expone el contenedor en el puerto 80.
- Si env = "dev", se usa nginx:alpine y se expone en el puerto 8080.

Esto permite adaptar automáticamente el despliegue según el entorno, sin duplicar código.

🧩 3.5.5. Uso combinado de locals + variables

Puedes combinar **variables** (valores externos) y **locals** (valores calculados) para generar nombres o configuraciones completas de forma dinámica.

💡 **Ejemplo:**

```
variable "env" {
  description = "Entorno de ejecución"
  default     = "test"
}

variable "imagen" {
  description = "Imagen base del contenedor"
  default     = "nginx:latest"
}

variable "puerto" {
  description = "Puerto externo del contenedor"
  default     = 8080
}

locals {
  container_name = "docker-${var.env}-${replace(var.imagen, ":", "-")}"
}

provider "docker" {}

resource "docker_container" "web" {
  name  = local.container_name
  image = var.imagen

  ports {
    internal = 80
    external = var.puerto
  }
}

output "nombre_contenedor" {
  value = local.container_name
}
```

💬 **Output esperado:**

```
nombre_contenedor = "docker-test-nginx-latest"
```

🧩 3.6. Resumen del uso de locals

Concepto	Descripción	Ejemplo
locals	Variables internas calculadas o reutilizables	`locals { prefix = "${var.project}-${var.env}" }`
Uso principal	Evitar repetición de lógica y valores	Formateo de nombres, rutas, etiquetas
Acceso	local.<nombre>	local.prefix
Ventaja	Código más limpio, menos redundancia, más coherencia	✅

✔️ Los **locals** son una herramienta para mejorar la calidad del código Terraform.
✔️ Te ayudan a mantener la **coherencia**, la **claridad** y la **modularidad** en tus configuraciones.
✔️ En proyectos grandes, son esenciales para definir **naming conventions**, **paths**, y **tags** comunes.

4. ⚙️ Outputs en Terraform

📣 4.1 ¿Qué son los outputs y para qué sirven?

Los outputs son valores de salida que Terraform muestra después de aplicar (terraform apply) o planificar (terraform plan). Representan los resultados importantes de la infraestructura, como:

- IPs públicas
- Nombres de recursos
- IDs de instancias
- Contraseñas generadas
- URLs de endpoints

📘 En otras palabras:

Los outputs son la forma de *exponer información útil* del despliegue, tanto para humanos como para otros módulos o scripts.

💡 **Ejemplo básico**

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}

output "instance_id" {
  description = "ID de la instancia creada"
  value       = aws_instance.web.id
}
```

Al ejecutar:

```
terraform apply
```

Terraform mostrará:

```
instance_id = "i-0abc123456def789"
```

🧱 4.2 Estructura de un bloque output

```
output "<nombre>" {
  description = "Descripción del valor"
  value       = <expresión o variable>
  sensitive   = <true|false> # opcional
}
```

Atributo	Descripción
nombre	Identificador único del output
description	(Opcional) Explica qué representa
value	Valor o referencia al recurso que se mostrará
sensitive	(Opcional) Oculta valores sensibles (ej. contraseñas)

💡 **Ejemplo:**

```
output "bucket_name" {
  description = "Nombre del bucket S3"
  value       = aws_s3_bucket.main.bucket
}
```

🧾 4.3 Mostrar resultados en la CLI

Los outputs se pueden visualizar de varias formas:

- 🔹 **Automáticamente tras un terraform apply**

Terraform los muestra al final de la ejecución.

- 🔹 **Manualmente con terraform output**

Muestra todos los outputs del estado actual.

- 🔹 **Mostrar un output específico**

terraform output <nombre>

Por ejemplo:

```
terraform output bucket_name
```

🔹 **Mostrar en formato JSON (para integraciones con scripts o CI/CD)**

```
terraform output -json
```

💡 Esto permite exportar outputs a otras herramientas o pipelines.

🧭 4.4 Exportar outputs a scripts externos

Puedes usar “terraform output -json” para pasar información a otros sistemas.

Ejemplo en Bash:

```
VPC_ID=$(terraform output -raw vpc_id)
echo "La VPC creada es: $VPC_ID"
```

Esto facilita automatizaciones en CI/CD pipelines o scripts de despliegue.

🔐 4.5 Sensibilidad de datos: sensitive = true

Por seguridad, Terraform permite **ocultar información sensible** (contraseñas, tokens, claves, etc.) en los outputs.

💡 **Ejemplo:**

```
resource "random_password" "db_password" {
  length  = 12
  special = true
}

output "database_password" {
  value     = random_password.db_password.result
  sensitive = true
}
```

Al ejecutar “terraform apply”, Terraform mostrará:

```
database_password = <sensitive>
```

Si luego ejecutas “terraform output”, se mostrará como:

```
database_password = <sensitive>
```

⚠️ **Para verla explícitamente (si es necesario):**

```
terraform output -raw database_password
```

En resumen:

✔️ Los *outputs* son útiles para exponer información útil, reutilizar módulos y automatizar flujos.
✔️ Son el puente entre **lo que Terraform crea** y **lo que otros sistemas necesitan saber**.
✔️ Usar correctamente outputs con “sensitive” y “terraform output -json” mejora la seguridad y la integración profesional.

5. 💾 El archivo de estado (terraform.tfstate)

🧠 5.1 ¿Qué es el estado en Terraform?

El **estado** es la **memoria interna de Terraform**. Guarda un **registro detallado de todos los recursos creados, sus atributos, dependencias y relaciones**. Cuando ejecutas:

```
terraform apply
```

Terraform realiza las siguientes acciones:

1. Lee tus archivos .tf.
2. Compara lo que *deseas tener* con lo que *ya existe* (según el estado).
3. Ejecuta los cambios necesarios para igualar ambos mundos.

💬 **En resumen:** El estado es lo que Terraform ***cree que existe*** en tu infraestructura.

📁 5.2 ¿Para qué sirve el archivo terraform.tfstate?

Terraform almacena el estado en un archivo JSON llamado terraform.tfstate (por defecto en tu directorio actual).

Este archivo contiene:

- Los **IDs reales** de los recursos creados.
- Los **valores actuales** (como IPs, nombres, etc.).
- Las **dependencias** entre recursos.
- Los **valores de variables y outputs**.
- Información del **backend** y **providers**.

📄 **Ejemplo de contenido simplificado:**

```
{
  "version": 4,
  "resources": [
    {
      "type": "aws_instance",
      "name": "web",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "attributes": {
            "id": "i-0a1b2c3d4e5f6g7h8",
            "ami": "ami-123456",
            "instance_type": "t2.micro",
            "tags": { "Name": "web-dev" }
          }
        }
      ]
    }
  ]
}
```

Terraform necesita este archivo para poder **destruir**, **actualizar** o **importar** recursos correctamente. Si lo pierdes, Terraform ***no sabrá qué existe realmente en tu entorno.***

🔍 5.3 Contenido del estado: recursos, variables y outputs

El archivo terraform.tfstate guarda tres tipos principales de información:

Tipo	Descripción	Ejemplo
Recursos	Todos los recursos creados (aws_instance, local_file, etc.)	ID, nombre, atributos
Variables	Valores que Terraform usó durante el apply	"region": "us-east-1"
Outputs	Resultados definidos en outputs.tf	"ip_publica": "54.221.123.45"

🧩 Esto permite a Terraform saber **qué cambiar** sin recrear todo desde cero.

🧰 5.4 Comando terraform show y lectura del estado

Terraform te permite **inspeccionar el estado actual** con varios comandos útiles:

📜 Ver el estado formateado

```
terraform show
```

Muestra en texto los recursos y atributos actuales conocidos por Terraform.

📄 Ver el estado en formato JSON

```
terraform show -json > estado.json
```

Guarda el estado en formato JSON para integraciones con otros sistemas.

🔎 Ver un recurso específico

Puedes buscar dentro del archivo terraform.tfstate directamente:

```
cat terraform.tfstate | grep "aws_instance"
```

(Útil para debugging rápido)

💡 Ejemplo de salida:

```
# aws_instance.web:
resource "aws_instance" "web" {
    id               = "i-0a1b2c3d4e5f6g7h8"
    ami              = "ami-123456"
    instance_type    = "t2.micro"
    availability_zone = "us-east-1a"
}
```

🌍 5.5 Estado local vs remoto

Terraform puede guardar el estado **localmente** o en un **backend remoto**. Algunas diferencias 👇

Tipo de Estado	Descripción	Ubicación	Uso recomendado
Local	Por defecto, se guarda en terraform.tfstate	En tu máquina	Ideal para pruebas, desarrollo o proyectos personales
Remoto	Se almacena en la nube (S3, GCS, Terraform Cloud, etc.)	En un backend configurado	Recomendado para equipos o producción

💾 Ejemplo de estado local (por defecto)

Archivo: terraform.tfstate en el directorio del proyecto.

- Solo el usuario local tiene acceso.
- Fácil de usar, pero **riesgoso en equipos** (pueden sobrescribirse cambios).
- Sin bloqueo concurrente (dos usuarios aplicando al mismo tiempo → caos 😱).

☁️ Ejemplo de estado remoto (mejor práctica)

```
terraform {
  backend "s3" {
    bucket         = "terraform-estado-global"
    key            = "infra/dev/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-lock"
  }
}
```

✅ Ventajas:

- Centraliza el estado.
- Permite trabajo en equipo.
- Bloquea el estado (usando DynamoDB o similar).
- Respalda automáticamente.

📘 Otras opciones comunes:

- **Terraform Cloud / Enterprise**
- **Google Cloud Storage (GCS)**
- **Azure Blob Storage**
- **Consul backend**

⚠️ 5.6 Riesgos y mejores prácticas en el manejo del estado

El “terraform.tfstate” es extremadamente sensible. Contiene datos reales de tu infraestructura e incluso credenciales o tokens si los proveedores los exponen. Veamos los principales riesgos y cómo mitigarlos 👇

🧨 5.6.1 Versionado del estado

❌ No se recomienda guardar terraform.tfstate en **Git** si es local.
✅ En cambio, usa **backends remotos** o **.gitignore** para excluirlo:

```
# .gitignore
terraform.tfstate
terraform.tfstate.backup
.terraform/
```

🧩 5.6.2 Respaldo del estado

Terraform crea automáticamente una copia:

```
terraform.tfstate.backup
```

Sin embargo, para mayor seguridad:

- Guarda backups en un bucket o sistema versionado.
- Usa **versioning en el backend remoto**.
- Automatiza copias periódicas con scripts o CI/CD.

🚷 5.6.3 No compartir el estado sin control

El estado **no debe compartirse manualmente** entre miembros del equipo (por ejemplo, enviando el archivo por correo o Slack). Esto puede causar:

- Pérdida de sincronización.
- Duplicación de recursos.
- Inconsistencias entre entornos.

✅ Solución:

- Usa un **backend remoto** con **state locking**.
- Controla el acceso con IAM, ACLs o políticas de permisos.

💡 Consejo: intenta inspeccionar sin modificar.

Puedes leer el estado sin riesgo con:

```
terraform state list
```

➡️ Muestra los recursos registrados en el estado.

Y para ver un recurso específico:

```
terraform state show <resource_name>
```

Ejemplo:

```
terraform state show aws_instance.web
```

Concepto	Descripción	Recomendación
Estado	Registro de la infraestructura real conocida por Terraform	No manipular manualmente
Archivo	terraform.tfstate (JSON)	Protegerlo y no versionarlo
Comando útil	terraform show, terraform state list	Para inspección y debugging
Estado local	Guardado en el disco	Ideal para pruebas
Estado remoto	Guardado en S3, GCS, Terraform Cloud, etc.	Ideal para trabajo en equipo
Seguridad	Puede contener datos sensibles	Encriptar, proteger y versionar

✔️ El terraform.tfstate es el corazón del funcionamiento interno de Terraform.
✔️ Debes tratarlo como un activo crítico: protegerlo, respaldarlo y nunca editarlo manualmente.
✔️ Migrar a un backend remoto con bloqueo es una práctica estándar en entornos profesionales.
✔️ Dominar la gestión del estado evita errores costosos, duplicación de recursos o pérdida de control.

6. 🧩 Providers útiles: random, template y templatefile

🌱 6.1 Provider random

El provider random de HashiCorp permite generar valores aleatorios o únicos dentro de Terraform. Esto es útil para crear identificadores, contraseñas, nombres únicos o etiquetas dinámicas, evitando conflictos o duplicaciones.

⚙️ 6.1.1 Recursos disponibles

Recurso	Descripción	Ejemplo de uso
random_pet	Genera nombres aleatorios combinando adjetivos y animales (🐱🦅🐍)	happy-tiger
random_string	Crea cadenas aleatorias de longitud personalizada	"Xb91zY4p"
random_integer	Genera números aleatorios dentro de un rango	42
random_password	Genera contraseñas seguras	"xA$!7zTfPq"

💡 Ejemplo completo con varios recursos

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

provider "random" {}

# Nombre aleatorio tipo mascota
resource "random_pet" "server_name" {
  length = 2
}

# Cadena aleatoria
resource "random_string" "id" {
  length  = 8
  special = false
  upper   = false
}

# Entero aleatorio
resource "random_integer" "port" {
  min = 8000
  max = 9000
}

# Contraseña segura
resource "random_password" "db_password" {
  length  = 12
  special = true
}

output "nombre_servidor" {
  value = random_pet.server_name.id
}

output "id_unico" {
  value = random_string.id.result
}

output "puerto" {
  value = random_integer.port.result
}

output "password" {
  value       = random_password.db_password.result
  sensitive   = true
}
```

🧪 **Ejecuta:**

```
terraform init
terraform apply
```

📜 **Salida esperada:**

```
nombre_servidor = "curious-owl"
id_unico        = "abcde123"
puerto          = 8542
password        = <sensitive>
```

🔁 **Regeneración de valores aleatorios**

Por defecto, los valores aleatorios **se mantienen estables** (almacenados en el estado).
Si deseas regenerarlos manualmente:

```
terraform taint random_string.id
terraform apply
```

Esto marca el recurso como “cambiado” y se regenerará en el siguiente apply.

🎨 6.2 Provider template (obsoleto en nuevas versiones)

Antes de Terraform 0.12, el provider template se usaba para procesar archivos con variables y contenido dinámico. Actualmente, está en desuso, reemplazado por la función templatefile() integrada en Terraform Core.

Aun así, es importante conocerlo por razones históricas y compatibilidad.

📜 6.2.1 Uso histórico

Ejemplo antiguo usando el provider “template”:

```
provider "template" {}

data "template_file" "config" {
  template = "Bienvenido, ${nombre}!"
  vars = {
    nombre = "Carlos"
  }
}

output "mensaje" {
  value = data.template_file.config.rendered
}
```

Salida:

```
mensaje = "Bienvenido, Carlos!"
```

🔔 **Pero hoy en día, esto debe hacerse con templatefile().**

🧰 6.2.2 Reemplazo moderno con templatefile()

📁 Estructura del proyecto:

template-demo/
├── main.tf
├── mensaje.tmpl

mensaje.tmpl

```
Hola, ${nombre}!
Bienvenido al entorno ${entorno}.
Tu identificador único es: ${id}.
```

main.tf

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

provider "random" {}

resource "random_string" "id" {
  length = 6
  special = false
}

variable "nombre" {
  default = "Carlos"
}

variable "entorno" {
  default = "producción"
}

locals {
  mensaje_final = templatefile("${path.module}/mensaje.tmpl", {
    nombre   = var.nombre
    entorno  = var.entorno
    id       = random_string.id.result
  })
}

output "mensaje_renderizado" {
  value = local.mensaje_final
}
```

🧪 Ejecución:

```
terraform init
terraform apply
```

📜 Resultado:

```
mensaje_renderizado = <<EOT
Hola, Carlos!
Bienvenido al entorno producción.
Tu identificador único es: Abc123
EOT
```

💡 templatefile() permite:

- Generar archivos de configuración dinámicos.
- Plantillas para scripts o configuraciones cloud-init.
- Contenido parametrizable sin código duplicado.

🧱 6.3 Funciones auxiliares útiles

Terraform tiene funciones integradas para manipular texto, listas y archivos que complementan random y templatefile().

Función	Descripción	Ejemplo
join(separator, list)	Une elementos de una lista con un separador	join("-", ["web", "01", "dev"]) → "web-01-dev"
format(format_string, values...)	Crea texto formateado tipo printf	format("srv-%03d", 7) → "srv-007"
file(path)	Lee el contenido de un archivo	file("${path.module}/config.json")
templatefile(path, vars)	Rellena variables dentro de una plantilla	templatefile("user.tmpl", { name = "Ana" })

💡 Ejemplo combinado de funciones

```
locals {
  nombre_base = "app"
  numero      = 3
  region      = "us-east-1"

  nombre_formateado = format("%s-%02d-%s", local.nombre_base, local.numero, local.region)
  nombres_concatenados = join(",", ["dev", "qa", "prod"])
}

output "formateado" {
  value = local.nombre_formateado
}
```

```
output "concatenados" {
  value = local.nombres_concatenados
}
```

📜 **Resultado:**

```
formateado   = "app-03-us-east-1"
concatenados = "dev,qa,prod"
```

En resumen:

Concepto	Descripción	Ejemplo
Provider random	Genera valores aleatorios controlados por Terraform	random_pet, random_string, random_integer
Provider template	(Obsoleto) Antes se usaba para plantillas dinámicas	Reemplazado por templatefile()
templatefile()	Inserta variables en archivos externos (.tmpl)	templatefile("archivo.tmpl", { var = "valor" })
Funciones auxiliares	join, format, file, templatefile	Manipulación avanzada de texto y listas

✔️ El provider random es ideal para crear valores únicos y reproducibles (controlados por estado).
✔️ La función templatefile() reemplaza al antiguo provider template, ofreciendo más poder y simplicidad.
✔️ Combinarlos con funciones nativas (join, format, etc.) te permite construir configuraciones dinámicas, limpias y reutilizables.
✔️ Son herramientas perfectas para automatizar, generar contenido y eliminar código repetido en tus proyectos Terraform.

7. 📚 WEBGRAFÍA

- **Terraform Language: Variables**
  - 🔗https://developer.hashicorp.com/terraform/language/values/variables
  - Referencia oficial de HashiCorp sobre las variables en Terraform: tipos de datos, archivos .tfvars, validaciones y buenas prácticas de parametrización.
- **Terraform Language: Locals**
  - 🔗https://developer.hashicorp.com/terraform/language/values/locals
  - Documentación sobre el uso de valores locales (locals) para definir expresiones reutilizables dentro de un módulo y evitar duplicación de código.
- **Terraform Language: Outputs**
  - 🔗 https://developer.hashicorp.com/terraform/language/values/outputs
  - Explicación detallada sobre los bloques output, cómo mostrar resultados tras apply, exportarlos entre módulos y marcarlos como sensibles.
- **Terraform State: Conceptos y gestión del estado**
  - 🔗 https://developer.hashicorp.com/terraform/language/state
  - Guía técnica sobre el archivo terraform.tfstate, su propósito, estructura, comandos de inspección y configuración de backends locales o remotos.