Introducción a Terraform y Salt Project

Unidad 04. Bloques avanzados, bucles y dependencias 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 04. Bloques avanzados, bucles y dependencias en Terraform

1. 🌍 Introducción al control dinámico de recursos

1.1 ¿Por qué necesitamos lógica avanzada en Terraform?

Terraform permite describir infraestructura como código de manera declarativa, lo cual es muy poderoso. Sin embargo, al crecer un proyecto, surgen nuevas necesidades:

- Crear **recursos de forma repetitiva** (como 10 instancias, 5 buckets, etc.)
- Activar o desactivar recursos según una condición (var.enabled)
- Evitar duplicación (repetir bloques con distinto nombre, puerto, ID, etc.)
- Controlar **el orden de creación** y dependencias explícitas

👉 Para resolver esto, Terraform incorpora herramientas como:

- count
- for_each
- Expresiones condicionales (condición ? A : B)
- depends_on
- Funciones integradas (ej: length, join, lookup, etc.)

Estas herramientas permiten aplicar *lógica dinámica* a recursos estáticos, sin perder la filosofía declarativa.

1.2 Casos comunes de lógica avanzada

Escenario	Solución Terraform
Crear múltiples recursos con nombres diferentes	for_each con una lista o mapa
Crear un recurso **solo si** una variable es verdadera	count = var.enabled ? 1 : 0
Usar diferentes parámetros según el entorno	Expresiones condicionales
Generar recursos a partir de listas configurables	Bucles for, for_each, count
Reutilizar lógica	locals o módulos
Crear recursos en orden	depends_on explícito

1.3 Beneficios del control dinámico

- ✅ **Escalabilidad**: puedes crear cientos de recursos sin duplicar código
- ✅ **Mantenibilidad**: más fácil hacer cambios cuando los datos están parametrizados
- ✅ **Flexibilidad**: puedes condicionar la creación según flags (ej: var.enable_logs)
- ✅ **Reutilización**: lógica común en módulos y locals
- ✅ **Menos errores humanos**: la lógica lo maneja, no tú

💡 **Ejemplo**: Imagina que quieres crear 5 buckets de S3. ¿Escribirías 5 bloques resource manuales? ¡No! Con for_each, defines una lista de nombres y Terraform hace el trabajo.

1.4 Cómo se ve en la práctica

👉 Para resolver esto, Terraform incorpora herramientas como:

- count
- for_each
- Expresiones condicionales (condición ? A : B)
- depends_on
- Funciones integradas (ej: length, join, lookup, etc.)

Estas herramientas permiten aplicar *lógica dinámica* a recursos estáticos, sin perder la filosofía declarativa.

1.2 Casos comunes de lógica avanzada

Escenario	Solución Terraform
Crear múltiples recursos con nombres diferentes	for_each con una lista o mapa
Crear un recurso **solo si** una variable es verdadera	count = var.enabled ? 1 : 0
Usar diferentes parámetros según el entorno	Expresiones condicionales
Generar recursos a partir de listas configurables	Bucles for, for_each, count
Reutilizar lógica	locals o módulos
Crear recursos en orden	depends_on explícito

1.3 Beneficios del control dinámico

- ✅ **Escalabilidad**: puedes crear cientos de recursos sin duplicar código
- ✅ **Mantenibilidad**: más fácil hacer cambios cuando los datos están parametrizados
- ✅ **Flexibilidad**: puedes condicionar la creación según flags (ej: var.enable_logs)
- ✅ **Reutilización**: lógica común en módulos y locals
- ✅ **Menos errores humanos**: la lógica lo maneja, no tú

💡 **Ejemplo**: Imagina que quieres crear 5 buckets de S3. ¿Escribirías 5 bloques resource manuales? ¡No! Con for_each, defines una lista de nombres y Terraform hace el trabajo.

1.4 Cómo se ve en la práctica

💡 **Ejemplo simple con count condicional:**

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

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

resource "docker_container" "web" {
  count = var.enable_web ? 1 : 0

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

  ports {
    internal = 80
    external = 8080
  }
}
```

➡️ Si enable_web = true, Terraform crea el contenedor.
Si enable_web = false, no se crea nada.

💡 **Ejemplo con for_each y lista:**

```
variable "containers" {
  type    = list(string)
  default = ["frontend", "backend", "database"]
}

resource "docker_container" "multi" {
  for_each = toset(var.containers)

  name  = "${each.value}-container"
  image = each.value == "database" ? "mysql:8.0" : "nginx:latest"
}
```

➡ Terraform creará tres contenedores:

- frontend-container (nginx)
- backend-container (nginx)
- database-container (MySQL)

Todo controlado desde una simple lista.

1.5 Relación con otras herramientas

Terraform no tiene estructuras de control tradicionales como if, while, for como en lenguajes imperativos, pero sí tiene:

- Evaluaciones condicionales
- Iteraciones declarativas (for, for_each)
- Uso de expresiones (ternarios, funciones)

Eso lo hace diferente pero extremadamente potente.

2. 🧩 Control de recursos con count

2.1 ¿Qué es count y cómo funciona?

"count" es una metaargumento de Terraform que permite crear múltiples instancias de un mismo recurso de forma repetitiva. Su valor debe ser un número entero (>= 0), y Terraform creará esa cantidad de copias del recurso.

> 📌 Se utiliza especialmente cuando quieres repetir **el mismo recurso** varias veces sin necesidad de nombres distintos.

⚙️ 2.2 Sintaxis básica con recursos simples (Docker)

```
resource "docker_container" "web" {
  count = 3
  name  = "web-${count.index}"
  image = "nginx:latest"

  ports {
    internal = 80
    external = 8080 + count.index
  }
}
```

💡 Este código crea **3 contenedores Docker**:

- docker_container.web[0]
- docker_container.web[1]
- docker_container.web[2]

Cada uno escucha en un puerto diferente: **8080**, **8081** y **8082**, respectivamente.

⚠️ Para acceder a cada uno:

docker_container.web[0].id
docker_container.web[1].name

🧩 2.3 Casos comunes: habilitar o deshabilitar recursos

Puedes usar una variable para decidir **si se crea un contenedor o no**, lo que resulta muy útil en entornos donde ciertos servicios son opcionales (por ejemplo, un contenedor de monitorización).

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

resource "docker_container" "logs" {
  count = var.enable_logs ? 1 : 0

  name  = "log-service"
  image = "fluentd:latest"
}
```

💡 Si enable_logs = true, Terraform creará el contenedor log-service.
Si enable_logs = false, no creará nada.

➡️ Ideal para **recursos opcionales** en tu infraestructura (como monitorización, backups, etc.).

📄 2.4 Limitaciones: índice (count.index) y uso con condicionales

Cuando usas count, puedes acceder al índice de cada recurso mediante count.index.
Esto te permite diferenciar recursos homogéneos.

```
resource "docker_container" "demo" {
  count = 3

  name  = "demo-${count.index}"
  image = "nginx:latest"

  ports {
    internal = 80
    external = 8000 + count.index
  }
}
```

💬 **Resultado:**

- demo-0 ⟶ puerto 8000
- demo-1 ⟶ puerto 8001
- demo-2 ⟶ puerto 8002

⚠️ **Limitaciones:**

- Todos los recursos creados con count son idénticos, salvo por el valor de count.index.
- No puedes usar count junto con for_each.
- No es ideal si cada recurso necesita **parámetros únicos** (por ejemplo, imágenes o volúmenes distintos).

🧠 2.5 Buenas prácticas y errores comunes

✔️ **Buenas prácticas:**

- Usa count cuando los contenedores sean **homogéneos** (misma imagen y configuración).
  - Perfecto para activar o desactivar recursos con condicionales:
    count = var.enable_web ? 1 : 0
- Ideal para **entornos de prueba o demos**, donde replicas el mismo servicio varias veces.

❌ **Errores comunes:**

- Acceder a índices inexistentes cuando count = 0:

```
docker_container.web[0].id # ❌ Falla si no hay instancias creadas
```

- Usar count con recursos que requieren datos únicos (nombres o configuraciones distintas).
  - 👉 En esos casos, usa for_each con un mapa o lista, que te permite dar claves y valores personalizados.

3. 🔁 Control de recursos con for_each

3.1 Diferencias entre count y for_each

Característica	count	for_each
Tipo de entrada	Número	Lista, mapa o conjunto (set)
Acceso a valores	count.index	each.key / each.value
Nombres únicos	No	Sí
Ideal para...	Recursos repetitivos	Recursos con datos únicos por instancia

3.2 Estructura básica y sintaxis

Con lista:

```
variable "users" {
  default = ["alice", "bob", "carol"]
}

resource "local_file" "user_file" {
  for_each = toset(var.users)
  filename = "${each.key}.txt"
  content  = "Bienvenida, ${each.key}"
}
```

Con mapa:

```
variable "user_emails" {
  default = {
    alice = "alice@mail.com"
    bob   = "bob@mail.com"
  }
}

resource "local_file" "user_email_file" {
  for_each = var.user_emails
  filename = "${each.key}.txt"
  content  = "Email: ${each.value}"
}
```

3.3 Uso con listas (list(string)) y mapas (map(any))

- Si usas lista, debes convertirla con toset() para que for_each lo acepte.
- Si usas mapa, puedes acceder fácilmente a clave y valor (each.key, each.value)

🌟 for_each es perfecto cuando cada recurso tiene algo único: nombre, configuración, puerto, etc.

3.4 Acceder a propiedades: each.key y each.value

Ejemplo con mapa más complejo:

```
variable "contenedores" {
  default = {
    nginx1 = 8081
    nginx2 = 8082
  }
}

resource "docker_container" "nginx" {
  for_each = var.contenedores
  name     = each.key
  image    = "nginx:alpine"
  ports {
    internal = 80
    external = each.value
  }
}
```

Esto creará 2 contenedores con nombres distintos y puertos diferentes.

3.5 Comparación práctica: ¿cuándo usar count vs for_each?

Quiero...	¿Qué usar?
Crear 5 recursos iguales	count
Crear 3 recursos con nombres únicos	for_each
Activar o desactivar un recurso con una flag	count
Iterar sobre un mapa de configuraciones	for_each
Necesito usar each.key o each.value	for_each
Repetir recurso según número fijo	count

4. 🔗 Gestión de dependencias con depends_on

4.1 ¿Qué son las dependencias en Terraform?

Terraform construye un **grafo de dependencias** internamente para decidir **qué recursos crear primero** y en qué orden.

- Las dependencias pueden ser:
  - **Implícitas**: cuando un recurso usa el output de otro.
  - Explícitas: cuando le decimos directamente con depends_on.

⚙️ 4.2 Dependencias implícitas

Terraform **detecta automáticamente** el orden en que deben crearse los recursos cuando uno depende de los atributos del otro.
No es necesario indicar nada manualmente: la dependencia se deduce de las referencias.

```
resource "docker_image" "nginx" {
  name = "nginx:latest"
}

resource "docker_container" "web" {
  name  = "webserver"
  image = docker_image.nginx.latest
}
```

👉 Terraform sabe que debe descargar primero la imagen (docker_image.nginx) antes de crear el contenedor (docker_container.web),
porque el segundo recurso usa el valor del primero (docker_image.nginx.latest).

🧩 4.3 ¿Cuándo y por qué usar depends_on explícitamente?

A veces no existe una referencia directa entre los recursos, pero queremos forzar el orden de creación.
En esos casos, se utiliza el atributo depends_on.

```
resource "null_resource" "esperar_red" {
  depends_on = [docker_network.red_app]
}
```

✔️ **Útil en casos como:**

- Crear **contenedores solo después de una red Docker** específica.
- Asegurar que un volumen o archivo exista antes de montar.
- Ejecutar scripts (null_resource) que dependan de que un contenedor esté listo.

📦 4.4 Ejemplo real donde depends_on es obligatorio

```
resource "docker_network" "red_local" {
  name = "infra_net"
}

resource "docker_container" "nginx" {
  name  = "web1"
  image = "nginx:alpine"

  depends_on = [docker_network.red_local] # 👈 importante

  networks_advanced {
    name = docker_network.red_local.name
  }
}
```

➡️ **Explicación:**

- Sin depends_on, Terraform podría intentar crear el contenedor antes de que exista la red, provocando un error.
- Con depends_on, se garantiza que la red infra_net esté lista antes de conectar el contenedor.

🧠 4.5 Buenas prácticas

- ✔️ **Usa depends_on solo cuando sea necesario:** Terraform ya resuelve la mayoría de dependencias automáticamente.
- ⚠️ **Evita sobreusarlo o agregar dependencias falsas**: Añadir depends_on entre recursos que no dependen realmente entre sí puede ralentizar el despliegue y hacerlo menos legible.
- ❌ **No uses depends_on "por si acaso":** Solo cuando un recurso no hace referencia directa a otro, pero su creación sí depende de él.

5. 🧱 Condicionales en Terraform

5.1 Operadores ternarios

Terraform no tiene if/else clásico, pero sí operadores ternarios:

```
var.condición ? valor_si_verdadero : valor_si_falso
```

💡 Ejemplo básico

```
resource "docker_container" "optional" {
  count = var.deploy_container ? 1 : 0

  name  = "web-${var.env}"
  image = var.env == "prod" ? "nginx:stable" : "nginx:alpine"

  ports {
    internal = 80
    external = var.env == "prod" ? 80 : 8080
  }
}
```

💬 **Explicación:**

- Si var.deploy_container = true, Terraform crea el contenedor.
- Si var.env = "prod", usa la imagen estable y el puerto 80.
- En otro caso, usa la versión liviana y el puerto 8080.

⚙️ 5.2 Aplicación en count, for_each y atributos

Puedes usar expresiones ternarias en distintos lugares:

Uso	Descripción	Ejemplo
count	Para crear (1) o no crear (0) un recurso	count = var.enable_web ? 1 : 0

for_each	Para filtrar o elegir qué lista usar	for_each = var.env == "prod" ? toset(["frontend", "backend"]) : toset(["test"])
Atributos	Para definir valores dentro del recurso	image = var.is_production ? "nginx:stable" : "nginx:alpine"

💡 **Ejemplo completo:**

```
resource "docker_container" "example" {
  count    = var.create_container ? 1 : 0
  name     = var.is_production ? "web-prod" : "web-dev"
  image    = var.is_production ? "nginx:stable" : "nginx:alpine"
  hostname = "srv-${var.is_production ? "prod" : "dev"}"
}
```

🔘 5.3 Uso combinado con variables booleanas (var.enabled)

Una variable booleana puede funcionar como **interruptor general** para crear o no crear recursos.

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

variable "env" {
  default = "dev"
}

resource "docker_container" "monitor" {
  count = var.enable_monitoring ? 1 : 0

  name  = "monitor-${var.env}"
  image = "grafana/grafana:latest"
}
```

💬 Si enable_monitoring = true, se crea el contenedor Grafana.
Si enable_monitoring = false, no se crea ningún contenedor.

🧩 5.4 Casos reales de uso

Escenario	Ejemplo Docker
Crear contenedor de logs solo si es entorno producción	count = var.env == "prod" ? 1 : 0
Elegir entre dos imágenes	image = var.use_light_image ? "nginx:alpine" : "nginx:latest"
Activar backups solo si están habilitados	count = var.enable_backups ? 1 : 0
Cambiar puerto según entorno	external = var.env == "dev" ? 8080 : 80

⚠️ 5.5 Limitaciones de expresiones condicionales

Terraform no soporta estructuras de control tradicionales como if o else if. En su lugar, todo se resuelve con expresiones ternarias que se evalúan completamente.

⚠️ Ten en cuenta:

- No existe ejecución diferida; **ambas ramas** se validan al evaluar el plan.
- No intentes acceder a variables o atributos **nulos o vacíos** en una rama no válida.
- Evita condicionales anidados excesivos: pueden hacer el código menos legible.

💬 Ejemplo incorrecto:

```
image = var.env == "prod" ? var.image_prod : var.image_dev
# ❌ Falla si var.image_prod no está definido y var.env != "prod"
```

✅ Solución:
Define valores por defecto o valida las variables antes:

```
variable "image_prod" {
  default = "nginx:stable"
}
variable "image_dev" {
  default = "nginx:alpine"
}
```

6. 🧰 Funciones integradas de Terraform

6.1 Introducción a las funciones nativas

Terraform incluye muchas funciones integradas para:

- Manipular listas, strings, mapas.
- Hacer cálculos.
- Transformar datos.
- Convertir estructuras.

👉 Las funciones se usan dentro de bloques, `locals`, outputs, condiciones, etc.

Algunas de las funciones comunes más utilizadas son:

length()
Cuenta el número de elementos en una lista o mapa.

```
length(var.contenedores)       # → 3
```

concat()
Une varias listas en una sola.

```
concat(["web", "db"], ["monitor"])   # → ["web", "db", "monitor"]
```

join()
Convierte una lista en texto, separando por un delimitador.

```
join("-", ["nginx", "prod", "01"])   # → "nginx-prod-01"
```

split()
Divide un texto en lista según un separador.

```
split("-", "nginx-prod-01")         # → ["nginx", "prod", "01"]
```

lower() / upper()
Convierte texto a minúsculas o mayúsculas.

```
lower("DOCKER") → "docker"
upper("nginx") → "NGINX"
```

lookup()
Busca una clave en un mapa con valor por defecto.

```
lookup(var.imagenes, "web", "nginx:latest")
# Devuelve la imagen "web" o "nginx:latest" si no existe
```

element()
Devuelve un elemento de una lista por su índice (útil con count.index).

```
element(var.puertos, count.index)
# var.puertos = [8080, 8081, 8082]
# → 8080 en la primera iteración
```

contains()
Comprueba si un valor está dentro de una lista.

```
contains(["dev", "test", "prod"], "prod")  # → true
```

format()
Crea texto con formato tipo printf.

```
format("web-%02d", count.index)   # → "web-01"
```

file()
Lee el contenido de un archivo.

```
file("${path.module}/docker.conf")
```

replace()
Reemplaza texto dentro de una cadena.

```
replace("nginx:latest", ":", "-")   # → "nginx-latest"
```

6.2 Combinación de funciones con locals y outputs

```
locals {
  nombres_formateados = join(", ", var.nombres)
}

output "nombres" {
  value = upper(local.nombres_formateados)
}
```

6.3 Ejemplo práctico: formatear nombres de archivos

```
variable "usuarios" {
  default = ["ana", "luis", "maria"]
}
```

```
resource "local_file" "user_files" {
  for_each = toset(var.usuarios)
  filename = "saludo-${upper(each.key)}.txt"
  content  = "Hola, ${title(each.key)}!"
}
```

Resultado:

- saludo-ANA.txt → "Hola, Ana!"
- saludo-LUIS.txt → "Hola, Luis!"

7. 🧱 Módulos en Terraform

7.1 ¿Qué es un módulo en Terraform?

Un **módulo** es una agrupación de recursos de Terraform reutilizables, que se pueden **invocar desde otros proyectos o módulos**.1

🧠 Piensa en los módulos como **funciones** en la programación tradicional:

- Tienen **entradas** (variables)
- Ejecutan una **lógica interna** (infraestructura)
- Producen **salidas** (outputs)

7.2 Estructura de un módulo

Mínimamente, un módulo tiene los siguientes archivos:

```
modules/
└── files/
    ├── main.tf        # Recursos principales
    ├── variables.tf   # Variables de entrada
    └── outputs.tf     # Valores de salida
```

💡 Opcionalmente puedes usar locals.tf, providers.tf, README.md, etc.

7.3 Ventajas de los módulos

✔️ **Reutilización:** escribe una vez, usa en muchos entornos (dev, prod, QA...)
✔️ **Organización:** separa lógica por componente (base de datos, red, servidores, etc.)
✔️ **Escalabilidad:** tu infraestructura crece con estructura clara
✔️ **Consistencia:** reduces errores por copiar/pegar
✔️ **Compartibilidad:** puedes subir módulos al Terraform Registry

7.4 Crear y usar un módulo local (ej: modules/files)

➤ **Paso 1: Estructura del módulo files**

```
# modules/files/variables.tf
variable "filename" {
  type = string
}
```

```
variable "content" {
  type = string
}

# modules/files/main.tf
resource "local_file" "this" {
  filename = var.filename
  content  = var.content
}

# modules/files/outputs.tf
output "file_path" {
  value = local_file.this.filename
}
```

➤ **Paso 2: Uso del módulo en el proyecto principal**

```
module "file1" {
  source   = "./modules/files"
  filename = "saludo.txt"
  content  = "Hola desde el módulo 👋"
}

output "archivo_creado" {
  value = module.file1.file_path
}
```

7.5 Pasar variables y recibir outputs desde módulos

Las variables que definimos en variables.tf del módulo se pueden sobreescribir al invocar el módulo.

📥 **Inputs** → Se pasan con claves nombre = valor
📤 **Outputs** → Se accede con module.<nombre>.output

```
output "contenido" {
  value = module.file1.file_path
}
```

7.6 Módulos públicos: Terraform Registry

El **Terraform Registry** es una plataforma con **miles de módulos reutilizables** creados por la comunidad y por HashiCorp. Permiten **ahorrar tiempo**, mantener **buenas prácticas** y **evitar duplicar código**. 🔗 **Web oficial:** https://registry.terraform.io/

📦 **Ejemplos de módulos populares**

🐳 **Docker**

- kreuzwerker/docker-provider → Configuración de imágenes, contenedores y redes Docker.
- terraform-docker-compose → Genera configuraciones basadas en Docker Compose.
- terraform-docker-swarm → Despliega y gestiona clústeres Docker Swarm.

☁️ **AWS**

- terraform-aws-modules/vpc/aws → Crea redes VPC completas.
- terraform-aws-modules/s3-bucket/aws → Buckets S3 con políticas y versionado.
- terraform-aws-modules/ec2-instance/aws → Despliegue rápido de instancias EC2.

🧱 **Azure**

- Azure/network/azurerm → Redes virtuales y subredes.
- Azure/compute/azurerm → Máquinas virtuales configurables.

🧩 **Google Cloud**

- terraform-google-modules/vpc/google → Creación de redes y subredes GCP.
- terraform-google-modules/gke/google → Despliegue de clústeres Kubernetes.

8. 📚 Webgrafía

- **Terraform Registry**
  - 🔗 https://registry.terraform.io/
  - Portal oficial para buscar módulos públicos, revisar versiones, ejemplos de uso y explorar la comunidad de módulos de Terraform.
- **Use a module — HashiCorp Developer**
  - 🔗 https://developer.hashicorp.com/terraform/registry/modules/use
  - Guía oficial sobre cómo usar módulos desde el registro (configuración del source, versioning, etc.).
- **Learn Terraform Modules (HashiCorp Education)**
  - 🔗 https://github.com/hashicorp-education/learn-terraform-modules
  - Repositorio oficial de HashiCorp Education con ejemplos prácticos sobre cómo estructurar, crear y reutilizar módulos en Terraform. Incluye código comentado, ejemplos de main.tf, variables.tf y outputs.tf, así como módulos para servicios de AWS y demostraciones que acompañan al curso Advanced Modules de HashiCorp Learn.