Introducción a Terraform y Salt Project

# Unidad 01. Introducción y fundamentos de 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 01. Introducción y fundamentos de Terraform

## 1. 🧩 Infraestructura como Código (IaC)

### 1.1 Definición de Infraestructura como Código

La **Infraestructura como Código (Infrastructure as Code, IaC)** es una práctica que consiste en **definir, aprovisionar y gestionar infraestructuras de TI mediante archivos de configuración legibles por máquina**, en lugar de hacerlo de forma manual a través de interfaces gráficas o comandos individuales.

Esto significa que los **servidores, redes, balanceadores, bases de datos y demás recursos** se describen en código, lo que permite:

- Automatizar su creación y modificación.
- Versionar la infraestructura igual que el código de una aplicación.
- Garantizar entornos reproducibles y consistentes.

💡 **Ejemplo conceptual:**
Un archivo IaC puede definir un servidor en AWS con una red y una base de datos asociada. Al ejecutar este archivo, se crea la infraestructura automáticamente.

### 1.2 Ventajas de las IaC

La **Infraestructura como Código (IaC)** ofrece numerosas ventajas frente a la gestión manual. En primer lugar, **la automatización** reduce el trabajo repetitivo y minimiza los errores humanos al definir los entornos mediante scripts. Gracias a la **reproducibilidad**, es posible crear entornos idénticos para producción, pruebas o desarrollo, garantizando coherencia en el ciclo de vida del software. La **escalabilidad** permite desplegar de forma rápida y eficiente múltiples recursos o servidores según la demanda. Además, el **control de versiones** posibilita gestionar la infraestructura con herramientas como Git, facilitando la colaboración y el seguimiento de cambios. En cuanto a la **auditoría y trazabilidad**, todo ajuste o modificación queda documentado en el código, mejorando la transparencia y el control. Finalmente, los **despliegues consistentes** evitan la llamada “deriva de configuración”, asegurando que todos los entornos mantengan la misma configuración y comportamiento.

### 1.3 Enfoque declarativo vs enfoque imperativo

Existen dos **enfoques principales en la Infraestructura como Código (IaC)**: el imperativo y el declarativo. El **enfoque imperativo** se basa en especificar los pasos exactos para alcanzar el estado deseado de la infraestructura; por ejemplo, indicar “crea una máquina virtual, luego instala nginx y finalmente abre el puerto 80”. Este método ofrece un **control detallado y paso a paso**, lo que lo hace útil para tareas de scripting o automatizaciones concretas. En cambio, el **enfoque declarativo** se centra en describir el **estado final deseado**, sin preocuparse por los pasos intermedios; por ejemplo, “quiero una máquina virtual con nginx y el puerto 80 abierto”. Este enfoque resulta **más mantenible, predecible y menos propenso a errores**, ya que el sistema se encarga de alcanzar el estado definido de manera automática.

### 1.4 Herramientas populares de IaC en el ecosistema actual

| Categoría | Herramienta | Descripción |
|---|---|---|
| **Multi-Cloud / Generalista** | **Terraform** | Herramienta declarativa creada por HashiCorp para múltiples proveedores. |
| **Configuración** | **Ansible**, **Chef**, **Puppet, Salt Project** | Más orientadas a la configuración de servidores y software. |
| **Cloud específica** | **AWS CloudFormation**, **Azure ARM Templates**, **Google Deployment Manager** | Enfocadas en un proveedor concreto. |
| **Lenguaje de programación** | **Pulumi** | Permite usar lenguajes como TypeScript o Python para IaC. |

Terraform se posiciona como una herramienta de **aprovisionamiento y gestión declarativa de infraestructura**. Se sitúa en la capa donde se **crean y administran recursos** (instancias, redes, bases de datos, etc.), integrándose fácilmente con herramientas de configuración como Ansible.

- 🌱 Terraform = “Define la infraestructura que necesitas”.
- 🧂 Salt Project = “Configura esa infraestructura una vez creada”.

## 2. ☁️ ¿Qué es Terraform?

### 2.1 Descripción general de Terraform

**Terraform** es una herramienta de **Infraestructura como Código (IaC)** desarrollada por **HashiCorp**, que permite definir, implementar y gestionar infraestructuras completas mediante archivos de configuración en formato **HCL (HashiCorp Configuration Language)**.

Permite gestionar infraestructura en múltiples entornos:

- Nubes públicas (AWS, Azure, GCP)
- Plataformas privadas (VMware, OpenStack)
- Servicios SaaS (GitHub, Cloudflare, etc.)
- Infraestructura local (archivos, redes, contenedores)

💡 La comunidad open-source sigue muy activa, y la mayoría de proveedores de nube ofrecen soporte oficial.

### 2.2 Características principales

✔️ Lenguaje declarativo (HCL) — Describe el estado deseado de los recursos.

✔️ Gestión de dependencias — Calcula el orden correcto de creación y destrucción.

✔️ Multi-cloud y extensible — Funciona con cientos de proveedores.

✔️ Planificación antes de aplicar — Permite previsualizar cambios (terraform plan).

✔️ Gestión del estado — Mantiene un archivo terraform.tfstate para rastrear los recursos existentes.

✔️ Modularidad — Posibilidad de crear módulos reutilizables.

✔️ Automatización e integración CI/CD — Se integra con pipelines y herramientas de control de versiones.

### 2.3 Casos de uso comunes

| Caso de uso | Ejemplo |
|---|---|
| **Infraestructura en la nube** | Crear VPCs, instancias EC2, bases de datos, balanceadores. |
| **Entornos locales** | Gestionar archivos, redes, y máquinas virtuales locales (VirtualBox, VMware). |
| **Testing y entornos temporales** | Crear entornos desechables para pruebas automáticas. |
| **SaaS y servicios externos** | Gestionar repositorios GitHub, DNS en Cloudflare o usuarios en Okta. |

## 3. ⚙️ Instalación de Terraform

Terraform es una herramienta de línea de comandos (CLI), distribuida como un binario único, que se instala fácilmente en cualquier sistema operativo.
A continuación veremos los **requisitos**, los **métodos de instalación** por sistema operativo, y cómo **verificar** que la instalación sea correcta.

### 🧩 3.1 Requisitos previos

Antes de instalar Terraform, asegúrate de cumplir con los siguientes requisitos:

| Requisito | Descripción |
|---|---|
| **Sistema Operativo** | Linux, macOS o Windows (compatible con x86 y ARM).En algunos casos, es necesario tener activada la virtualización por hardware (VT-x o AMD-V) en la BIOS/UEFI. |
| **Terminal / Consola** | Acceso a una shell como Bash, PowerShell o CMD. |
| **Conexión a Internet** | Necesaria para descargar el binario y los providers. |
| **Privilegios de instalación** | Permisos de administrador o sudo. |
| **Herramientas opcionales** | Git (para clonar módulos), un editor como VS Code con extensión Terraform. |

## 🐧 3.2 Métodos de instalación

En la página oficial de Terraform, se indican los distintos tipos de instalaciones https://developer.hashicorp.com/terraform/install

Ahí observamos Terraform puede instalarse de varias maneras. Los métodos más comunes son:

1. Descargando el binario desde el sitio oficial de HashiCorp.
2. Usando un gestor de paquetes (apt, brew, chocolatey).
3. Usando herramientas automatizadas (como asdf o tfenv).

### 3.2.1 Instalación en Linux

**Método A — Usando apt (Debian/Ubuntu):**

```
wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com $(grep -oP '(?<=UBUNTU_CODENAME=).*'
/etc/os-release || lsb_release -cs) main" | sudo tee
/etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform
```

**Método B — Descarga manual:**

Descargar la última versión

```
wget https://releases.hashicorp.com/terraform/1.13.3/terraform_1.13.3_linux_amd64.zip
```

Descomprimir el binario

```
unzip terraform_1.13.3_linux_amd64.zip
```

Moverlo al PATH

```
sudo mv terraform /usr/local/bin/
```

### 3.2.2 Instalación en macOS

**Método A — Usando Homebrew (recomendado):**

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

**Método B — Descarga manual:**

Descargar la última versión

```
curl -o terraform.zip
https://releases.hashicorp.com/terraform/1.13.3/terraform_1.13.3_darwin_amd64.zip
```

Descomprime el fichero zip

```
unzip terraform.zip
```

Moverlo al PATH

```
sudo mv terraform /usr/local/bin/
```

### 3.2.3 Instalación en Windows

**Método A — Usando Chocolatey (recomendado):**

```
choco install terraform
```

**Método B — Manual:**

1. Descarga el archivo ZIP desde
   👉 https://developer.hashicorp.com/terraform/downloads
2. Extrae el binario terraform.exe.
3. Añade su carpeta al PATH del sistema:
   *Panel de Control → Sistema → Configuración avanzada → Variables de entorno → Path → Editar.*

## ✅ 3.3 Verificación de la instalación

Una vez instalado Terraform, abre tu terminal y ejecuta:

```
terraform version
```

También puedes ejecutar:

```
terraform
```

para ver la lista completa de comandos disponibles (plan, apply, init, destroy, etc.).

💡 Si el comando no se reconoce, revisa que la ruta al binario esté en tu variable de entorno PATH.

## 4. 🚀 Primer contacto con el CLI de Terraform

Vamos a realizar un **HelloWorldo** para validar que Terraform funciona correctamente.

**Crea una carpeta para tu primer proyecto:**

```
mkdir terraform-hello
cd terraform-hello
```

📄 **Crea un archivo main.tf:**
Desde Linux, puedes crearlo por ejemplo con la orden “nano main.tf”

El contenido del fichero será:
📄 **main.tf**

```
terraform {
  required_version = ">= 1.0.0"
}

# Usando un provider local
provider "local" {}
```

```
resource "local_file" "hello" {
  filename = "${path.module}/hello.txt"
  content  = "¡Hola mundo Terraform!"
}
```

**Inicializa el proyecto:**

Abre una terminal y sitúate en el directorio raíz del proyecto (donde se encuentra el archivo main.tf). Una vez allí puedes ejecutar el comando:

```
terraform init
```

👉 **Terraform descargará el provider "local".**

**Planifica los cambios:**

```
terraform plan
```

Con este comando un resumen con los recursos que se crearán. Este comando no hace cambios, solo te informa de los cambios que se realizarán, de posibles errores, etc.

**Aplica los cambios:**

Con este comando, aplicaremos los cambios.

```
terraform apply
```

Escribe "yes" cuando se te solicite. En este ejemplo, Terraform creará el archivo hello.txt.

**Verifica el resultado:**

```
cat hello.txt
```

Deberías ver:

```
¡Hola mundo Terraform!
```

**Destruye los recursos:**

Una vez hayas finalizado, con este comando se destruirán los recursos creados por Terraform.

```
terraform destroy
```

🎉 ¡Listo! Has hecho tu primera ejecución completa con Terraform, utilizando el ciclo:

**init → plan → apply → destroy.**

## 5. 🏗️ Estructura de un proyecto Terraform

Terraform organiza la infraestructura en **archivos de configuración** (con extensión `.tf`) que definen recursos, variables, outputs y otros elementos del entorno.
Una buena estructura es clave para la **mantenibilidad, escalabilidad y colaboración** entre equipos.

### 📂 5.1 Archivos y carpetas típicos

A medida que el proyecto crece, es recomendable seguir una estructura clara y modular.
Un proyecto básico suele tener:

**terraform-project/**
**├── main.tf**
**├── variables.tf**
**├── outputs.tf**
**├── terraform.tfvars**

## ⚙️ 5.2 ¿Qué es main.tf, variables.tf, outputs.tf, terraform.tfstate?

Veamos el propósito de cada uno:

### 🧱 main.tf

Es el **núcleo del proyecto**, donde se definen los **recursos principales** (instancias, redes, bases de datos, etc.).

Ejemplo:

```
resource "local_file" "hello" {
  filename = "${path.module}/hello.txt"
  content  = "Hola mundo Terraform!"
}
```

### ⚙️ variables.tf

Aquí se declaran las **variables de entrada**, que permiten parametrizar tu infraestructura.
Esto facilita reutilizar el mismo código en distintos entornos (dev, staging, prod).

**Ejemplo:**

```
variable "filename" {
  description = "Ruta del archivo a crear"
  type        = string
  default     = "hello.txt"
}

variable "message" {
  description = "Contenido del archivo"
  type        = string
}
```

### 📤 outputs.tf

Define las **salidas o valores de salida** del proyecto, útiles para mostrar información relevante al final de una ejecución o compartirla con otros módulos.

**Ejemplo:**

```
output "file_path" {
  description = "Ruta del archivo generado"
  value       = local_file.hello.filename
}
```

### 🗒️ terraform.tfstate

Este archivo es **crítico**: contiene el **estado actual de la infraestructura** gestionada por Terraform. Guarda identificadores, atributos y dependencias de todos los recursos.

### ⚠️ Nunca lo edites manualmente.

- Se genera automáticamente al ejecutar “terraform apply”.
- Sirve para comparar el estado real con el declarado en código.
- Puede almacenarse localmente o en un backend remoto (S3, GCS, Terraform Cloud).

Ejemplo (extracto simplificado):

```
{
  "version": 4,
  "resources": [
    {
      "type": "local_file",
      "name": "hello",
      "instances": [
        {
          "attributes": {
            "filename": "hello.txt",
            "content": "Hola mundo Terraform!"
          }
        }
      ]
    }
  ]
}
```

## 🧩 5.3 Separación de responsabilidades en archivos

Separar la configuración en varios archivos `.tf` **no afecta** el orden de ejecución: Terraform combina todos los archivos del mismo directorio antes de aplicar los cambios.

Ejemplo de distribución recomendada:

| Archivo | Función |
|---|---|
| **provider.tf** | Define los proveedores (AWS, Azure, GCP, local, etc.) |
| **main.tf** | Define los recursos principales. |
| **variables.tf** | Declara las variables de entrada. |
| **outputs.tf** | Muestra información relevante al final de la ejecución. |
| **terraform.tfvars** | Asigna valores concretos a las variables declaradas. |

### 🧠 5.4 Buenas prácticas desde el inicio

✔️ **1. Mantén una estructura modular:**
Divide tu infraestructura en módulos reutilizables (por servicio o componente).

✔️ **2. Usa control de versiones (Git):**
No subas el archivo terraform.tfstate ni .terraform/ al repositorio.
Agrega un **.gitignore:**

```
.terraform/
terraform.tfstate
terraform.tfstate.backup
```

✔️ **3. Define versiones explícitas:**
Bloquear versiones de Terraform y providers evita comportamientos inesperados.

✔️ **4. Usa nombres descriptivos:**
Los nombres de variables y outputs deben reflejar su propósito real.

✔️ **5. Mantén el estado remoto:**
Para equipos, guarda el terraform.tfstate en un backend remoto como S3, Azure Blob o Terraform Cloud.

✔️ **6. Documenta:**
Para proyectos de cierta envergadura, es una buena prácticar Incluir un README.md explicando cómo ejecutar el proyecto, las variables requeridas y las dependencias.

## 6. 🔁 El ciclo de vida de Terraform

Terraform sigue un ciclo bien definido que garantiza **consistencia, seguridad y control de cambios** sobre la infraestructura. Los comandos principales de este ciclo son:

**init → plan → apply → destroy**

Cada etapa tiene un propósito específico dentro del flujo de trabajo de IaC.

| Etapa | Descripción |
|---|---|
| **terraform init** | Inicializa el proyecto y descarga los providers necesarios. |
| **terraform plan** | Compara el estado actual con el deseado y muestra los cambios que se aplicarán. |
| **terraform apply** | Ejecuta los cambios propuestos para alcanzar el estado deseado. |
| **terraform destroy** | Elimina todos los recursos gestionados por Terraform. |

💡 **Idea clave:** Terraform nunca aplica cambios directamente sin que puedas **ver o aprobar previamente** lo que hará.

## ⚙️ 6.1 ¿Qué hace cada comando?

### 🧩 terraform init

- Prepara el entorno para su uso.
- Descarga los **providers** (por ejemplo, AWS, Azure, local, etc.).
- Crea la carpeta ".terraform/" donde guarda dependencias y plugins.
- Debe ejecutarse **solo una vez** al comenzar un proyecto o si se cambian providers.

**Ejemplo:**

```
terraform init
```

### 🔍 terraform plan

- Analiza los archivos .tf y el estado actual (terraform.tfstate).
- Muestra los cambios que se harán sin aplicarlos todavía.
- Permite revisar y aprobar antes de ejecutar.

**Ejemplo:**

```
terraform plan
```

💡 **Consejo:** Guarda el plan en un archivo para revisión:

```
terraform plan -out=plan.tfplan
```

Y luego puedes aplicarlo directamente:

```
terraform apply plan.tfplan
```

### 🚀 terraform apply

- Aplica los cambios planificados.
- Terraform crea, modifica o elimina los recursos necesarios para alcanzar el **estado deseado**.
- Antes de ejecutar, pide confirmación (a menos que uses -auto-approve).

**Ejemplo:**

```
terraform apply
```

💡 Terraform actualiza el archivo **terraform.tfstate** automáticamente al finalizar.

### 💣 terraform destroy

- Elimina **todos los recursos definidos** en el estado actual.
- Muy útil para **limpiar entornos de prueba o desarrollo**.
- Solicita confirmación antes de borrar.

**Ejemplo:**

```
terraform destroy
```

### 🎛️ 6.2 Control de cambios: planificación vs ejecución

Terraform separa la planificación (plan) de la ejecución (apply) para garantizar seguridad y previsibilidad.

| Concepto | Planificación | Ejecución |
|---|---|---|
| Propósito | Analizar los cambios requeridos | Aplicar los cambios |
| Modifica recursos | ❌ No | ✅ Sí |
| Modifica el estado (tfstate) | ❌ No | ✅ Sí |
| Ideal para | Revisar PRs, auditorías | Despliegue controlado |

💡 **Ejemplo práctico:**

- En entornos CI/CD, se suele ejecutar “terraform plan” en las Pull Requests (solo lectura) y “terraform apply” en el pipeline de despliegue (solo tras aprobación).

### 🗒️ 6.4 Gestión del estado: terraform.tfstate

El archivo de estado (terraform.tfstate) es la pieza central del ciclo de vida de Terraform.
Contiene información detallada sobre todos los recursos que Terraform gestiona.

📘 **Funciones del estado:**

- Guarda los **IDs reales** de los recursos en la nube.
- Permite comparar el **estado actual** con el **deseado**.
- Permite a Terraform aplicar **solo los cambios necesarios**.
- Facilita la **detección de drift** (deriva de configuración).

📂 **Ubicación:**

Por defecto, se guarda localmente en el directorio del proyecto, pero puede almacenarse remotamente:

**Ejemplo de backend remoto en S3:**

```
terraform {
  backend "s3" {
    bucket         = "mi-bucket-terraform"
    key            = "infraestructura/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
  }
}
```

✅ Ventajas del backend remoto:

- Permite colaboración entre equipos.
- Evita pérdida del estado local.
- Habilita bloqueo de estado (state locking) para evitar conflictos concurrentes.

### 🔒 6.5 Consideraciones de seguridad y backup del estado

El archivo **terraform.tfstate** contiene información sensible, como:

- IDs de recursos.
- Nombres de usuarios.
- Contraseñas, tokens o claves (dependiendo del provider).

**Buenas prácticas:**

✔️ 1. No lo subas al repositorio.
Agrega en .**gitignore**:

```
terraform.tfstate
terraform.tfstate.backup
.terraform/
```

✔️ **2. Usa backends remotos seguros.**
Ejemplo: AWS S3 + DynamoDB, GCS + locking, Terraform Cloud.

✔️ **3. Cifra el estado.**
Utiliza el parámetro encrypt = true o herramientas externas (KMS, Vault).

✔️ **4. Realiza backups automáticos.**
Terraform genera un ".backup" automáticamente, pero puedes programar copias adicionales.

✔️ **5. Controla accesos.**
Solo los usuarios autorizados deben poder leer o escribir el tfstate.

## 6. ☁️ Introducción a los Providers en Terraform

Los **providers (proveedores)** son **plugins** que permiten a Terraform comunicarse con diferentes plataformas o servicios.
Cada provider ofrece un conjunto de **recursos** y **datos** que se pueden usar para crear, leer, actualizar y eliminar infraestructura.

### 🔍 6.1 ¿Qué es un Provider?

Un **provider** es el **componente que traduce las configuraciones declaradas en Terraform (HCL)** a llamadas reales a APIs. Por ejemplo:

- El provider **AWS** crea instancias EC2 o buckets S3 mediante la API de AWS.
- El provider **Azure** aprovisiona redes virtuales o máquinas.
- El provider **local** crea archivos o ejecuta acciones locales.
- El provider **random** genera contraseñas o strings aleatorios.

💡 Terraform describe *qué* quieres; el *provider* se encarga de *cómo* hacerlo en la plataforma correspondiente.

### ⚙️ 6.2 Cómo declarar un provider

Los providers se declaran dentro del bloque terraform o en archivos como provider.tf.

Ejemplo general:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
provider "aws" {
  region = "us-east-1"
}
```

💡 **Notas:**

- source: indica la ubicación del provider (normalmente en el registro oficial de HashiCorp).
- version: bloquea la versión del provider (buena práctica para evitar incompatibilidades).
- Los providers se descargan automáticamente con terraform init.

### 🌐 6.3 El ecosistema de providers oficiales y de terceros

Terraform tiene un **ecosistema enorme de providers**, disponibles en el Terraform Registry.

| Tipo | Ejemplos | Descripción |
|---|---|---|
| **Oficiales (HashiCorp)** | aws, azurerm, google, kubernetes | Mantenidos directamente por HashiCorp. |
| **Comunitarios (Terceros)** | cloudflare, github, datadog, okta, docker, vagrant | Mantenidos por la comunidad o empresas externas. |
| **Locales / utilitarios** | local, null, random | No interactúan con una nube, pero son útiles para pruebas o lógica local. |

Ejemplo de varios providers combinados:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    github = {
      source = "integrations/github"
    }
  }
}
```

## 🧰 6.4 Providers locales

Terraform incluye varios providers "utilitarios" que no dependen de la nube. Son ideales para **probar conceptos**, generar datos, o realizar tareas locales.

### 6.4.1 Provider local

Permite interactuar con archivos y recursos locales en tu máquina.

**Ejemplo: crear un archivo de texto local**

```
provider "local" {}

resource "local_file" "hello" {
  filename = "${path.module}/saludo.txt"
  content  = "¡Hola desde el provider local!"
}
```

➡️ Crea el archivo `saludo.txt` en tu directorio actual.

### 6.4.2 Provider null

Permite ejecutar comandos o definir recursos "vacíos" que sirven como conectores o triggers.
Muy útil para automatización o integración con scripts externos.

**Ejemplo: ejecutar un comando después de crear un recurso**

```
provider "null" {}

resource "null_resource" "post_create" {
  provisioner "local-exec" {
    command = "echo 'Infraestructura desplegada con éxito!'"
  }
}
```

➡️ Ejecuta el comando en tu terminal cuando se aplique la infraestructura.

💡 null_resource también puede usar triggers para ejecutarse solo cuando cambian variables:

```
resource "null_resource" "reload" {
  triggers = {
    version = var.app_version
  }

  provisioner "local-exec" {
    command = "echo 'Nueva versión: ${var.app_version}'"
  }
}
```

### 6.4.3 Provider random

Permite generar valores aleatorios, como contraseñas, IDs o nombres de recursos únicos.

**Ejemplo: generar una contraseña aleatoria**

```
provider "random" {}
resource "random_password" "user_pass" {
  length  = 16
  special = true
}
```

Puedes mostrar el resultado con un output:

```
output "password_generada" {
  value = random_password.user_pass.result
}
```

💡 Cada vez que se destruye el recurso, la contraseña cambia (a menos que uses keepers para mantener estabilidad entre ejecuciones).

**🔒 Buenas prácticas con Providers**

✔️ Usa versiones específicas (version = "~> X.Y") para garantizar consistencia.
✔️ Mantén los providers en archivos separados (provider.tf).
✔️ Si trabajas en equipo, define un backend remoto antes de usar providers que afecten la nube.
✔️ Usa terraform providers para listar todos los providers utilizados en el proyecto.
✔️ Evita credenciales planas en el código — usa variables de entorno o vault para la autenticación.

## 7. 📚 WEBGRAFÍA

### Documentación oficial

- **HashiCorp Terraform Documentation** 🔗https://developer.hashicorp.com/terraform/docs
  - Guía oficial completa sobre instalación, comandos, lenguaje HCL, módulos, providers y mejores prácticas.
- **Terraform Registry** 🔗 https://registry.terraform.io/
  - Repositorio oficial de *providers* y *módulos* certificados y comunitarios.
- **Terraform CLI Commands Reference** 🔗 https://developer.hashicorp.com/terraform/cli
  - Referencia oficial de comandos: init, plan, apply, destroy, etc.
- **Terraform Installation Guide** 🔗 https://developer.hashicorp.com/terraform/install
  - Instrucciones detalladas para la instalación en Linux, macOS y Windows.