Terraform Intro

O **Terraform**, desenvolvido pela **HashiCorp**, é uma ferramenta de **Infraestrutura como Código (IaC – Infrastructure as Code)** que permite definir, provisionar e gerenciar infraestrutura de forma automatizada e consistente, utilizando arquivos de configuração declarativos.

Ele é uma ferramenta de **declaração e orquestração de infraestrutura**.
Ou seja, ele **não sabe** como criar uma máquina virtual, um container, uma rede, ou um volume — ele **precisa de um intermediário** que saiba **como interagir com o ambiente real**. Esse intermediário é o que chamamos de *Provider*

Com ele, é possível criar recursos em diversos provedores, como **AWS**, **Azure**, **Google** Cloud, **VMware**, entre outros.

Basic Arch

Um projeto Terraform é composto por arquivos de configuração com a extensão `.tf`.
Os principais são:

- **main.tf:** define os recursos principais da infraestrutura.
- **variables.tf:** declara variáveis que podem ser reutilizadas em várias partes do código.
- **outputs.tf:** define saídas (outputs) exibidas após o provisionamento.
- **provider.tf:** especifica o provedor de nuvem e credenciais de autenticação.

Basic Commands

- **terraform init:** inicializa o diretório e baixa os plugins necessários.
- **terraform plan:** exibe um plano das ações que o Terraform executará.
- **terraform apply:** aplica o plano e cria/modifica os recursos.
- **terraform destroy:** remove todos os recursos definidos no estado atual.
- **terraform validate:** valida a sintaxe e a coerência das configurações.
- **terraform fmt:** formata o código conforme o padrão oficial.

State Management

O Terraform mantém um arquivo chamado **terraform.tfstate**, que armazena o estado atual da infraestrutura.
Esse arquivo é crítico, pois o Terraform o utiliza para determinar o que precisa ser criado, alterado ou destruído.
Recomenda-se armazená-lo de forma segura (por exemplo, em um backend remoto como S3, Azure Blob ou Terraform Cloud).

Good practices

- Utilizar **módulos** para reutilizar código.
- Versionar o código em **Git**.
- Proteger o arquivo de estado com controle de acesso.
- Usar variáveis e workspaces para gerenciar diferentes ambientes (ex: desenvolvimento, homologação, produção).
- Adotar o comando `terraform plan` antes de cada `apply` para evitar mudanças inesperadas.

Providers

Cada provider:

- Implementa os **recursos (resources)** e **dados (data sources)** que o Terraform pode manipular.
- Define como aplicar as ações de **criação, modificação e destruição**.

Em termos práticos:

- Se você quer criar **containers**, usa o provider **Docker**.
- Se quer criar **VMs no Proxmox**, usa o provider **Proxmox**.
- Se quer criar **instâncias EC2 na AWS**, usa o provider **AWS**.
- Se quer gerenciar **usuários em LDAP**, usa o provider **LDAP**.

Sem provider, o Terraform **não tem "motor" para agir** — ele só entende o "plano" (as instruções), mas não tem quem as execute.

Conceptual operating architecture

- **Você escreve os arquivos .tf**, declarando os recursos desejados.
- **Terraform lê o código** e identifica quais providers são necessários.

- **terraform init** baixa esses providers (como plugins).
- **terraform apply** usa o provider para aplicar as ações no ambiente.
- O **estado** (`terraform.tfstate`) é atualizado com o que foi criado.

Provider Docker example

- Utilizado para gerenciar containers em hosts locais ou clusters on-premises.
- Muito útil em pipelines CI/CD locais, onde é necessário subir e destruir containers dinamicamente para testes, validação ou empacotamento de aplicações.
- Exemplo de uso real:
  - Automatizar a criação de ambientes de teste efêmeros para cada *commit* de código.
  - Garantir reprodutibilidade na infraestrutura de containers.

Benefício: evita scripts shell manuais e mantém a orquestração de containers versionada como código.

Provider Proxmox / VMware / Libvirt

- Altamente utilizados em ambientes corporativos que usam virtualização local.
- Permitem criar, destruir e modificar VMs no datacenter interno da empresa, da mesma forma que se faz na nuvem.
  - Integram-se facilmente com pipelines de automação e gerenciamento de configuração (Ansible, Puppet, etc.).

Benefício: unificação da gestão — o mesmo Terraform que gerencia AWS, por exemplo, também pode gerenciar Proxmox ou VMware no mesmo fluxo de código.

Provider Local

- Permite criar **arquivos, diretórios e templates** no sistema de arquivos local.
- Frequentemente usado para **gerar arquivos de configuração dinamicamente**, como um `docker-compose.yml`, um `nginx.conf`, ou um `inventory` do Ansible.

Benefício: facilita a integração entre infraestrutura e configuração de software.

Provider Null

- Executa comandos locais ou scripts personalizados após o provisionamento.
- Exemplo: instalar pacotes, configurar serviços, ou validar ambientes.

Benefício: adiciona flexibilidade sem depender de provisionadores externos complexos.

Provider External

- Permite chamar scripts ou APIs externas e usar os resultados dentro do Terraform.
- Exemplo: consultar um serviço interno de inventário de IPs ou nomes de host e usar essa informação para criar novos recursos.

Benefício: integração com sistemas corporativos internos (CMDBs, automações legadas, etc.).

- Você gerencia **infraestrutura própria** (servidores físicos ou VMs locais).
- Deseja **evitar dependência de nuvem** (por custo, política interna ou segurança).
- Busca **aprendizado estruturado** em Terraform antes de aplicar na nuvem.
- Precisa **automatizar pipelines de containers, testes ou empacotamento**.

Usar providers locais pode trazer vantagens estratégicas significativas:

1. **Padronização e versionamento da infraestrutura on-premises**
   - Toda a configuração de VMs, containers e redes locais é descrita em código, auditável e reprodutível.
2. **Automação completa de datacenters locais**

- É possível orquestrar ambientes inteiros sem cliques manuais no hypervisor ou painel.

3. **Ambientes híbridos consistentes**

- Empresas que usam nuvem e datacenter próprio podem manter o mesmo fluxo Terraform para ambos.

4. **Melhor governança e rastreabilidade**

- Cada mudança de infraestrutura é registrada como *commit* no Git, garantindo rastreabilidade e rollback.

5. **Integração DevOps local**

- Pode ser integrado a Jenkins, GitLab CI, ou GitHub Actions para pipelines internos, sem depender de nuvem.

Docker Provider example

O **provider Docker** permite que o Terraform **crie, gerencie e destrua containers, imagens, volumes e redes Docker**.
Ele atua sobre o **Docker Engine local** (por meio do socket `/var/run/docker.sock`) ou sobre um **daemon remoto** configurado via API.

Ele transforma ações que normalmente seriam manuais (como `docker run`, `docker network create`, `docker rm`, etc.) em **declarações de infraestrutura versionáveis e reprodutíveis**.

Mesmo fora da nuvem, o uso do provider Docker tem aplicações práticas e maduras:

- **Ambientes de desenvolvimento padronizados:**
  Garante que todos os desenvolvedores usem o mesmo ambiente de container.
- **Pipelines de CI/CD locais:**
  Cria containers temporários para testes, builds ou validação de código.
- **Ambientes efêmeros de teste:**
  Subir e destruir containers automaticamente para rodar testes de integração.
- **Infraestrutura de laboratório interno:**
  Criar e versionar topologias simples (por exemplo, Nginx + PostgreSQL + Aplicação Python).

Conceptual operating architecture

Um projeto simples pode conter:

- `provider.tf` - define qual provider será usado (no caso, Docker).
- `main.tf` - descreve o que será criado (containers, imagens, redes, volumes).
- `outputs.tf` - mostra resultados úteis após o provisionamento.

`provider.tf

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

```
}

provider "docker" {
  host = "unix:///var/run/docker.sock"
}
```

Esse pequeno trecho informa ao Terraform:

- Que ele usará o provider oficial `hashicorp/docker`;
- Que o Docker Engine está sendo acessado localmente via socket padrão do Linux.

Depois de declarar o provider, podemos adicionar um **recurso básico** que cria um container. Por exemplo, um **Nginx** local exposto na porta 8080:

```
resource "docker_container" "meu_nginx" {
  name  = "meu_nginx"
  image = "nginx:latest"
  ports {
    internal = 80
    external = 8080
  }
}
```

Equivalente ao comando docker ->
` docker run -d -p 8080:80 --name meu_nginx nginx:latest

Terraform vai gerenciar esse container — se você mudar algo (como a porta ou imagem), ele atualizará automaticamente o container na próxima execução de `terraform apply`.

Process

- `terraform init`
  → Baixa o provider Docker e inicializa o diretório do projeto.
- `terraform plan`
  → Mostra o que será criado (sem executar ainda).
- `terraform apply`
  → Cria o container conforme as definições.
- `terraform destroy`
  → Remove o container e limpa o estado.

O arquivo `terraform.tfstate` é criado automaticamente e guarda o **estado atual da infraestrutura** — no caso, o container criado.

- **Reprodutibilidade total:** o mesmo código cria o mesmo container em qualquer máquina.
- **Rastreabilidade:** toda modificação é registrada no controle de versão (ex.: Git).
- **Integração fácil:** pode ser usado dentro de pipelines Jenkins, GitLab CI, ou GitHub Actions.
- **Automação local completa:** sem necessidade de nuvem, e sem dependência de scripts `bash`.

Limitações

- O provider Docker **não substitui totalmente** o Docker Compose (embora possa replicar grande parte de suas funcionalidades).
- Não possui nativamente mecanismos de **escalonamento ou orquestração distribuída** (para isso, usaria-se Kubernetes).
- Requer que o **Docker Engine** esteja rodando localmente (ou acessível via API remota).

O **provider Docker** é ideal para:

- Aprender a mentalidade do Terraform de forma segura e controlada;
- Automatizar pipelines e testes locais;
- Padronizar ambientes internos;
- Operar em contextos **on-premises** e sem dependência de nuvem.

Pense no Terraform não como "mais uma ferramenta de DevOps", mas como uma **linguagem de controle da infraestrutura**.
Ele permite que você:

- Padronize ambientes;
- Audite mudanças;
- Reproduza infraestruturas inteiras com um único comando;
- Integre segurança, rede e automação em um fluxo único.