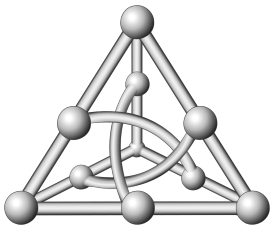
LIVRO DE APOIO

Gabriel Paes Duarte Baltazar e Kaê de Oliveira Budke

Guia teórico-prático das tecnologias mais utilizadas no mercado.

Área de Concentração: Computação Distribuída

Orientador: Prof. Brivaldo Alves da Silva Jr



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dezembro, 2025

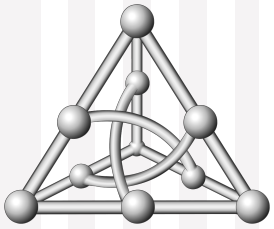
LIVRO DE APOIO

Gabriel Paes Duarte Baltazar e Kaê de Oliveira Budke

Guia teórico-prático das tecnologias mais utilizadas no mercado.

Área de Concentração: Computação Distribuída

Orientador: Prof. Brivaldo Alves da Silva Jr



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dezembro, 2025

Sumário

Capítulo 1

Linux Containêres - LXC

Neste capítulo, iniciamos nossa exploração prática das tecnologias de contêineres em nível de sistema operacional, começando pelo LXC (Linux Containers). O LXC oferece um método leve de virtualização, permitindo que múltiplos sistemas Linux isolados rodem em um único host, compartilhando o mesmo kernel.

O primeiro passo para utilizar o LXC é a sua instalação. Em sistemas baseados em Debian, como o utilizado nestes laboratórios, o processo de instalação é direto através do gerenciador de pacotes `apt`, assumindo privilégios de superusuário:

```
$ su -
$ apt-get install lxc
```

Com o LXC instalado, nosso próximo passo é provisionar um contêiner. Para isso, utilizamos o pacote `lxc-templates`, que contém os scripts necessários para criar "imagens" base de diversas distribuições Linux. Em seguida, usamos o comando `lxc-create` para instanciar nosso primeiro contêiner, que chamaremos de `teste`, baseado no template do `debian`.

```
$ apt install lxc-templates -y
$ lxc-create -n teste -t debian
```

Este comando inicia um processo que, por baixo dos panos, utiliza a ferramenta `debootstrap` para baixar os pacotes base do Debian e montar o sistema de arquivos raiz do contêiner.

O ciclo de vida básico de interação com o contêiner é simples. Primeiro, podemos listar os contêineres existentes com `lxc-ls` para confirmar que o `teste` foi criado:

```
$ lxc-ls
teste
```

Em seguida, iniciamos o contêiner com `lxc-start`:

```
$ lxc-start -n teste
```

Finalmente, para acessar o shell do contêiner, usamos `lxc-attach`. Este comando nos "anexa" ao namespace do contêiner, nos dando um terminal interativo dentro dele:

```
$ lxc-attach -n teste
root@teste:$
```

Note que o prompt do terminal muda para `root@teste`, indicando que estamos logados como superusuário dentro do ambiente isolado do contêiner `teste`. Para sair do contêiner e retornar ao host, basta usar o comando `exit` ou o atalho `CTRL+D`.

1.1 Analisando a Rede do Contêiner

Uma das mágicas do LXC acontece na camada de rede. Ao iniciar um contêiner, o LXC configura automaticamente a conectividade. Do ponto de vista do host, é possível pingar o IP do contêiner. Internamente, o LXC cria uma interface de rede virtual do tipo **vETH** (Virtual Ethernet Pair).

Uma verificação das interfaces de rede no host com `ip a` revela essa nova arquitetura. Notamos duas novas entidades principais:

- `lxcbr0`: Uma interface de bridge Linux, que atua como um switch virtual para onde todos os contêineres serão conectados.
- `veth...`: Uma interface par-a-par que conecta o namespace de rede do contêiner à bridge `lxcbr0` no host.

O exemplo de saída abaixo ilustra essa configuração:

```
$ ip a
...
4: lxcbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue state UP group default qlen 1000
                            <===== rede para os containers LXC
    link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.1/24 brd 10.0.3.255 scope global lxcbr0
...
6: vethONrrIZ@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
   qdisc noqueue master lxcbr0 state UP group default qlen 1000
      <===== interface compartilhada com o container
    link/ether fe:e1:9c:59:d3:8d brd ff:ff:ff:ff:ff:ff
       link-netnsid 0
...
```

1.2 Inspeção e Monitoramento de Contêineres

Para inspecionar o estado de um contêiner específico, utilizamos o comando `lxc-info`. Ele fornece um resumo vital, incluindo o estado (RUNNING), o PID do processo principal no host, o IP alocado e a interface `veth` correspondente.

```
$ lxc-info -n teste

Name:           teste
State:          RUNNING
PID:            26205
IP:             10.0.3.19
Link:           vethONrrIZ
 TX bytes:      1.73 KiB
 RX bytes:      2.20 KiB
 Total bytes:   3.93 KiB
```

Para um monitoramento contínuo dos recursos (CPU, Memória, I/O) de todos os contêineres ativos, o LXC fornece um utilitário análogo ao `top` tradicional, chamado `lxc-top`:

```
$ lxc-top
Container                     CPU           CPU           CPU
                                    BlkIO         Mem
Name                         Used           Sys          User
                       Total(Read/Write)       Used
teste                        0.00          0.00          0.00
   3087018381.88 GiB(...)    0.00
TOTAL 1 of 1                 0.00          0.00          0.00
   3087018381.88 GiB(...)    0.00
```

1.3 Parando Contêineres

O ciclo de vida do contêiner se completa com o comando `lxc-stop`. Vamos parar nosso contêiner `teste` e, em seguida, tentar nos conectar a ele novamente:

```
$ lxc-stop -n teste
$ lxc-attach -n teste
lxc-attach: teste: ../src/lxc/attach.c: get_attach_context: 406
   Connection refused - Failed to get init pid
lxc-attach: teste: ../src/lxc/attach.c: lxc_attach: 1470
   Connection refused - Failed to get attach context
```

A falha no `lxc-attach` é esperada. O erro "`Failed to get init pid`" nos informa que o processo principal (PID 1) do contêiner não existe mais, portanto, não

há a o que se anexar. Isso sublinha a natureza do LXC como um gerenciador de processos isolados, e não uma máquina virtual completa.

1.4 Verificando Configurações do Kernel

O funcionamento do LXC depende intrinsecamente de recursos modernos do kernel Linux, como Namespaces e Cgroups. O utilitário `lxc-checkconfig` é uma ferramenta de diagnóstico crucial que varre a configuração do kernel atual e informa se os módulos e recursos necessários estão habilitados.

```
$ lxc-checkconfig
LXC version 6.0.4
...
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
...
--- Control groups ---
Cgroups: enabled
Cgroup namespace: enabled
...
--- Misc ---
Veth pair device: enabled, loaded
...
```

1.5 Arquivos de Configuração do Contêiner

Para um controle mais granular, podemos inspecionar e editar os arquivos de configuração do contêiner. O arquivo de configuração principal para o nosso contêiner `teste` reside em `/var/lib/lxc/teste/config`.

Este arquivo define aspectos cruciais, como o tipo de rede, o caminho para o sistema de arquivos raiz (`rootfs`), e quais perfis de configuração adicionais devem ser incluídos (como `debian.common.conf`).

```
$ cat /var/lib/lxc/teste/config
...
lxc.net.0.type = veth
lxc.net.0.hwaddr = 00:16:3e:d6:57:df
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
...
lxc.rootfs.path = dir:/var/lib/lxc/teste/rootfs
```

```
# Common configuration
lxc.include = /usr/share/lxc/config/debian.common.conf

# Container specific configuration
lxc.uts.name = teste
lxc.arch = amd64
...
```

As configurações de DNS, por sua vez, são gerenciadas de forma tradicional *dentro* do contêiner, no arquivo `/etc/resolv.conf`.

1.6 Integração com Systemd

Para ambientes de produção ou para garantir que os contêineres subam com o host, é vital que eles sejam gerenciados como serviços. O LXC integra-se nativamente ao `systemd` através do serviço `lxc@.service`.

Podemos iniciar nosso contêiner `teste` usando o `systemctl`:

```
$ systemctl start lxc@teste
$ systemctl status lxc@teste
• lxc@teste.service - LXC Container: teste
     Loaded: loaded (/lib/systemd/system/lxc@.service;
        disabled; preset: enabled)
     Active: active (running) since Tue 2024-08-27 10:06:13
        -04; 1s ago
...
```

Para que o contêiner inicialize junto com o boot do sistema, basta habilitar o serviço:

```
$ systemctl enable lxc@teste
Created symlink
   '/etc/systemd/system/multi-user.target.wants/lxc@teste.
service' → '/usr/lib/systemd/system/lxc@.service'.
```

E para desativar essa inicialização automática:

```
$ systemctl disable lxc@teste
Removed '/etc/systemd/system/multi-user.target.wants/lxc@teste.
service'.
```

1.7 Disponibilidade de Templates

Uma dúvida comum é se o LXC, rodando em um host Debian, está restrito a contêineres Debian. A resposta é não. O LXC é agnóstico em relação à distribuição, desde que um template de criação exista. O diretório `/usr/share/lxc/templates/` revela a vasta gama de opções disponíveis:

```
$ ls /usr/share/lxc/templates/
lxc-alpine     lxc-busybox  lxc-debian    lxc-fedora
   lxc-kali ...
lxc-altlinux   lxc-centos   lxc-devuan    lxc-fedora-legacy
   lxc-local  ...
lxc-archlinux  lxc-cirros   lxc-download  lxc-gentoo
   lxc-oci  ...
```

1.7.1 Download de Templates

Além dos scripts de template locais (como o do Debian que usou `debootstrap`), o LXC pode baixar imagens de contêiner pré-construídas usando o template `download`. Este método é frequentemente mais rápido.

Podemos listar todas as imagens remotas disponíveis com a flag `-list`. A lista é extensa, então vamos mostrar apenas um extrato:

```
$ lxc-create -t download -n alpha -- --list
Downloading the image index

---
DIST            RELEASE     ARCH   VARIANT  BUILD
---
almalinux       10          amd64  default  20250925_23:08
alpine          3.20        amd64  default  20250927_13:00
archlinux       current     amd64  default  20250926_19:46
busybox         1.36.1      amd64  default  20250927_06:00
centos          9-Stream    amd64  default  20250924_08:35
debian          bookworm    amd64  default  20250927_05:24
fedora          40          amd64  default  20250926_20:33
opensuse        tumbleweed  amd64  default  20250927_04:20
rockylinux      9           amd64  default  20250926_02:06
ubuntu          jammy       amd64  default  20250927_07:42
ubuntu          noble       amd64  default  20250927_07:42
... (e muitas outras) ...
```

Como exemplo, vamos baixar a imagem do Alpine Linux (versão 3.20, arquitetura amd64):

```
$ lxc-create -t download -n alpine -- -d alpine -r 3.20 -a amd64
```

```
Downloading the image index
...
Unpacking the rootfs
---
You just created an Alpinelinux 3.20 x86_64 (20240826_13:00)
   container.
```

Agora, o contêiner `alpine` está disponível para ser iniciado e utilizado como qualquer outro.

```
$ lxc-ls
alpine teste
$ lxc-start -n alpine
$ lxc-attach -n alpine
root@alpine:~#
```

1.8 Gerenciamento de Snapshots

O LXC oferece um recurso poderoso para controle de versão do sistema de arquivos: os snapshots. Um snapshot é uma "foto"do estado do contêiner em um determinado momento.

Para criar um snapshot, o contêiner precisa estar parado. Vamos verificar os snapshots do nosso contêiner `teste` e, em seguida, criar um:

```
$ lxc-snapshot -L -n teste
No snapshots

$ lxc-stop -n teste
$ lxc-snapshot -n teste
$ lxc-snapshot -L -n teste
snap0 (/var/lib/lxc/teste/snaps) 2025:09:27 16:39:56
```

Podemos criar múltiplos snapshots. Cada um é numerado sequencialmente (`snap0`, `snap1`, etc.).

Para restaurar um snapshot, usamos a flag `-r`. Uma prática recomendada é restaurar o snapshot como um *novo* contêiner, usando a flag `-N`, o que preserva o contêiner original e o próprio snapshot:

```
$ lxc-snapshot -n teste -r snap1 -N teste-snap1
$ lxc-ls
alpine      teste       teste-snap1
```

Para destruir (apagar) um snapshot específico, usamos a flag `-d`:

```
$ lxc-snapshot -n teste -d snap0
$ lxc-snapshot -L -n teste
snap1 (/var/lib/lxc/teste/snaps) 2025:09:27 16:40:32
```

E para remover completamente um contêiner (como o `teste-snap1` que criamos a partir da restauração), usamos `lxc-destroy`:

```
$ lxc-destroy teste-snap1
$ lxc-ls
alpine teste
```

1.9 Tipos de Interfaces de Rede

O LXC é extremamente flexível na configuração de rede. A diretiva `lxc.net.0.type` no arquivo de configuração define o comportamento da rede. A seguir, detalhamos os tipos mais comuns.

1.9.1 `empty`

Este é o tipo mais restritivo. O contêiner é iniciado apenas com uma interface de *loopback* (`lo`). Se nenhuma outra interface for definida, o contêiner ficará completamente isolado da rede do host e do mundo exterior.

1.9.2 `phys`

O tipo `phys` (físico) concede ao contêiner acesso direto a uma interface física existente no sistema host. A interface do host é especificada com `lxc.net.0.link`.

```
# Exemplo: Passando a interface eth0 do host para o contêiner
lxc.net.0.type = phys
lxc.net.0.flags = up
lxc.net.0.link = eth0
```

1.9.3 `veth`

Este é o tipo mais comum e o padrão usado em nossa instalação. Ele cria um *Virtual Ethernet Pair Device* (par veth) para fazer a ponte ou rotear o tráfego entre o host e o contêiner.

`bridge` mode

Este é o modo padrão do `veth`. O par veth é conectado a uma interface de bridge no host (definida por `lxc.net.0.link`), que em nosso caso é a `lxcbr0`. Todos os contêineres na mesma bridge podem se comunicar.

```
# Exemplo: Conectando o contêiner à bridge lxcbr0
lxc.net.0.type = veth
lxc.net.0.flags = up
lxc.net.0.link = lxcbr0
```

`router` mode

Neste modo, em vez de usar uma bridge, rotas estáticas são criadas entre a interface do host e a interface veth do contêiner, permitindo comunicação roteada.

1.9.4 `vlan`

O tipo `vlan` permite compartilhar uma interface do host com o contêiner, mas restringindo a comunicação a uma ID de VLAN específica (`lxc.net.0.vlan.id`).

```
# Exemplo: Conectando o contêiner à VLAN 100 na eth0
lxc.net.0.type = vlan
lxc.net.0.flags = up
lxc.net.0.link = eth0
lxc.net.0.vlan.id = 100
```

1.9.5 `macvlan`

O tipo `macvlan` permite que uma única interface física do host seja "dividida"em múltiplas interfaces virtuais, cada uma com seu próprio endereço MAC. Isso permite que o contêiner apareça na rede como um dispositivo físico separado.

`private` mode

Este é o modo padrão do `macvlan`. A interface virtual dentro do contêiner não pode se comunicar com a interface física principal no host.

`vepa` (Virtual Ethernet Port Aggregator)

Similar ao modo *private*, mas os pacotes são forçados a passar por um switch físico externo. Isso permite que diferentes contêineres `macvlan` no mesmo host se comuniquem, desde que o switch suporte *hairpin mode*.

`passthru`

Este modo oferece um alto nível de isolamento, semelhante ao `phys`, mas o contêiner recebe a interface `macvlan` em vez da interface física bruta.

1.10 Conclusão

Com isso, encerramos nosso laboratório introdutório sobre os contêineres LXC, cobrindo desde a criação e gerenciamento básico até conceitos avançados de rede e snapshots.

1.11 Atividades

Para solidificar o conhecimento, propomos os seguintes exercícios:

1. Crie 5 containers, sendo 2 Debian, 1 Ubuntu e 2 Alpine.
2. Instale o servidor SSH em um dos contêineres e acesse-o via SSH a partir do host usando um usuário comum (não-root).
3. Configure o acesso SSH para o contêiner usando autenticação baseada em chaves (par de chaves SSH) da sua máquina local.
4. Desative o login por senha no servidor SSH do contêiner, permitindo apenas a conexão via chaves.
5. Permita a conexão remota via chaves para o usuário `root` do contêiner.
6. (Avançado) Configure uma aplicação web, como o Nextcloud, em um contêiner, e seu banco de dados (ex: MySQL) em um segundo contêiner, fazendo com que o Nextcloud se conecte ao banco de dados na rede interna do LXC.

Capítulo 2

Incus: O Sucessor do LXD

Após explorarmos os fundamentos do LXC, avançamos para o **Incus**. O Incus é um projeto de código aberto, mantido pela comunidade, que surgiu como um *fork* direto do LXD (LXC Daemon) após mudanças em seu licenciamento e manutenção. Ele herda toda a poderosa API e a experiência de usuário do LXD, focando em ser um gerenciador robusto tanto para **contêineres** de sistema (como o LXC) quanto para **máquinas virtuais**. Para um estudo mais aprofundado, você pode conferir mais sobre a ferramenta na documentação.

Sua adoção tem crescido, e em novas versões de distribuições como o Debian 13 (Trixie), o Incus já é o substituto padrão. Em nosso ambiente Debian 12 (Bookworm), precisamos adicioná-lo através de um repositório externo. Utilizaremos o repositório mantido pela Zabbly. Aqui vamos utilizar o `sudo` para executar os comandos, mas você pode entrar no modo `root` com `su -` para não precisar incluir `sudo` sempre que rodar os comandos.

2.1 Adicionar Repositório Zabbly

O primeiro passo é estabelecer confiança com o repositório, baixando sua chave GPG (GNU Privacy Guard). Isso garante que os pacotes que instalarmos sejam autênticos e não tenham sido modificados.

```
$ sudo mkdir -p /etc/apt/keyrings/
$ sudo curl -fsSL https://pkgs.zabbly.com/key.asc -o
   /etc/apt/keyrings/zabbly.asc
```

Com a chave em vigor, informamos ao `apt` onde encontrar os pacotes do Incus, criando um novo arquivo de fontes em `/etc/apt/sources.list.d/`.

```
$ sudo sh -c 'cat <<EOF >
   /etc/apt/sources.list.d/zabbly-incus-lts-6.0.sources
Enabled: yes
Types: deb
```

```
URIs: https://pkgs.zabbly.com/incus/lts-6.0
Suites: $(. /etc/os-release && echo ${VERSION_CODENAME})
Components: main
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/zabbly.asc

EOF'
```

2.2 Atualizar Pacotes e Instalar Incus

Finalmente, com o repositório configurado, atualizamos o índice de pacotes do `apt` e solicitamos a instalação do Incus.

```
$ sudo apt-get update
$ sudo apt-get install incus -y
$ incus --version
```

2.3 Setup inicial e Comandos básicos

2.3.1 Inicializando a ferramenta

Antes de levantarmos contêiners é necessário realizar o `init` da ferramenta na nossa máquina. Para isto, executamos `sudo incus admin init` Neste processo, ele vai realizar algumas perguntas básicas de configuração referente ao armazenamento e rede do novo ambiente.

```
$ sudo incus admin init

Would you like to use clustering? (yes/no) [default=no]: no
Do you want to configure a new storage pool? (yes/no)
   [default=yes]:
Name of the new storage pool [default=default]: teste
Name of the storage backend to use (dir, btrfs) [default=btrfs]:
Would you like to create a new btrfs subvolume under
   /var/lib/incus? (yes/no) [default=yes]:
Would you like to create a new local network bridge? (yes/no)
   [default=yes]:
What should the new bridge be called? [default=incusbr0]:
What IPv4 address should be used? (CIDR subnet notation, 'auto'
   or 'none') [default=auto]:
What IPv6 address should be used? (CIDR subnet notation, 'auto'
   or 'none') [default=auto]: none
```

```
Would you like the server to be available over the network?
   (yes/no) [default=no]: yes
Address to bind to (not including port) [default=all]:
Port to bind to [default=8443]:
Would you like stale cached images to be updated automatically?
   (yes/no) [default=yes]:
Would you like a YAML 'init' preseed to be printed? (yes/no)
   [default=no]: yes

config:
  core.https_address: '[::]:8443'
networks:
- config:
    ipv4.address: auto
    ipv6.address: none
  description: ''
  name: incusbr0
  type: ""
  project: default

storage_pools:
- config:
    source: /var/lib/incus/storage-pools/teste
  description: ''
  name: teste
  driver: btrfs
storage_volumes: []
profiles:
- config: {}
  description: ''
  devices:
    eth0:
      name: eth0
      network: incusbr0
      type: nic
    root:
      path: /
      pool: teste
      type: disk
  name: default
  project: default
projects: []
certificates: []
cluster: null
```

2.3.2 Brincando com contêiners

Para compreender seu funcionamento básico, vamos iniciar, entrar e parar um contêiner com Ubuntu 25.04.

```
# Verificando as imagens remotas disponiveis
$ sudo incus image list images: ubuntu

$ sudo incus launch images:ubuntu/25.04 teste
Launching teste
$ sudo incus list
+------+-------+------+------+------+-----------+
| NAME | STATE | IPV4 | IPV6 | TYPE | SNAPSHOTS |
+------+-------+------+------+------+-----------+
| teste | RUNNING | 10.231.124.199 (eth0) |  | CONTAINER | 0 |
+------+-------+------+------+------+-----------+
$ sudo incus exec teste -- bash
root@teste:~# exit
$ sudo incus stop teste
$ sudo incus delete teste
```

Você também pode ver as informações mais detalhadas de uma instância com `sudo incus info <instancia>`, no nosso caso, vamos encontrar informações semelhantes à esta:

```
$ sudo incus info teste
Name: teste
Description:
Status: RUNNING
Type: container
Architecture: x86_64
PID: 4092
Created: 2025/12/01 11:47 EST
Last Used: 2025/12/01 12:08 EST
Started: 2025/12/01 12:08 EST
....
```

2.4 Gerenciamento de Snapshots

O Incus também realiza o controle de versões do ambiente via Snapshots. Vamos criar duas 'fotos' do nosso contêiner e restaurar à primeira versão.

```
$ sudo incus snapshot create teste snap0
$ sudo incus snapshot create teste snap1
$ sudo incus snapshot list teste
+-------+----------------------+------------+----------+
```

```
| NAME  |       TAKEN AT       | EXPIRES AT | STATEFUL |
+-------+----------------------+------------+----------+
| snap0 | 2025/12/01 12:05 EST |            | NO       |
+-------+----------------------+------------+----------+
| snap1 | 2025/12/01 12:06 EST |            | NO       |
+-------+----------------------+------------+----------+
$ sudo incus snapshot restore teste snap0
```

2.5 Gerenciamento de Redes

Uma outra característica do Incus é a possibilidade de criar redes e perfis para contextos específicos nos seus ambientes. Por padrão, ao executar `sudo incus network list` a gente observa que existe diversos tipos de redes já reconhecidas pela ferramenta, e, conforme configuramos no inicio a rede `incusbr0` é a padrão para qualquer novo contêiner. De forma análoga, a ferramenta já atribui um perfil padrão `default` para qualquer novo contêiner.

Devido a herança do LXC e de toda esquemática de redes do Linux, os tipos de rede do Incus são os mesmos que apresentamos no último capítulo. Conforme recomenda na documentação, a melhor alternativa é utilizar a rede via bridge – padrão – para todas as instancias, todavia, é possível criar e gerenciar novas redes.

```
# Criando uma nova rede
$ sudo incus network create redinha

# Adicionando teste à essa rede e verificando no container
$ sudo incus network attach redinha teste
$ sudo incus exec teste -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
   UNKNOWN group default qlen 1000
   ...
22: eth1@if23: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state
   DOWN group default qlen 1000
    link/ether 10:66:6a:11:66:a2 brd ff:ff:ff:ff:ff:ff
       link-netnsid 0

# Ativando a interface e entregando um IPv4 à máquina
$ sudo incus exec teste -- ip link set eth1 up && dhclient eth1

$ sudo incus exec teste -- ip a
...
22: eth1@if23: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue state UP group default qlen 1000
    link/ether 10:66:6a:11:66:a2 brd ff:ff:ff:ff:ff:ff
       link-netnsid 0
```

```
    inet 10.159.55.142/24 brd 10.159.55.255 scope global
       dynamic eth1
       valid_lft 3597sec preferred_lft 3597sec
...
```

Podemos ainda fazer duas instâncias se comunicarem. Como segunda máquina, vamos usar um Debian 12 semelhante à instância t2.micro da aws (1 vCPU, 1GiB de RAM) com a rede já determinada.

```
$ sudo incus launch images:debian/12 debinho -t aws:t2.micro -n
   redinha

# Verificando a rede dentro do container
$ sudo incus exec debinho -- ip a
...
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue state UP group default qlen 1000
    link/ether 10:66:6a:aa:92:4f brd ff:ff:ff:ff:ff:ff
       link-netnsid 0
    inet 10.159.55.111/24 metric 1024 brd 10.159.55.255 scope
       global dynamic eth0
       valid_lft 3571sec preferred_lft 3571sec
...

# Fazendo a comunicação
$ sudo incus exec debinho -- ping -c 3 teste
PING teste(teste.incus
   (fd42:5b4c:aabe:ab28:1266:6aff:fe11:66a2)) 56 data bytes
64 bytes from teste.incus
   (fd42:5b4c:aabe:ab28:1266:6aff:fe11:66a2): icmp_seq=1 ttl=64
   time=0.132 ms
64 bytes from teste.incus
   (fd42:5b4c:aabe:ab28:1266:6aff:fe11:66a2): icmp_seq=2 ttl=64
   time=0.086 ms
64 bytes from teste.incus
   (fd42:5b4c:aabe:ab28:1266:6aff:fe11:66a2): icmp_seq=3 ttl=64
   time=0.086 ms

--- teste ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.086/0.101/0.132/0.021 ms
```

2.6 Gerenciando Armazenamento

O Incus realiza seu armazenamento por meio do conceito de ***storage pool*** que é, basicamente, um espaço na qual a ferramenta pode usar para guardar *root filesystems* de

containers ou imagens do sistema. Ele pode ser baseado em diferentes tipos de backends, como:

- **dir**: apenas um diretório no sistema de arquivos do host. Simples, mas sem snapshots eficientes.
- **zfs**: sistema de arquivos ZFS. Permite snapshots instantâneos e clones eficientes.
- **btrfs**: similar ao ZFS, com snapshots e subvolumes.
- **lvm**: volumes lógicos no LVM.
- **ceph**: para armazenamento distribuído.
- **cephfs**: Ceph FS, também distribuído.
- **custom**: você pode usar drivers de storage externos.

Além disso, também é possível armazenar os arquivos do sistema de modo compartilhado com o host ou isoladamente. Isso garante flexibilidade ao arquiteto ao analisar as necessidades e riscos de determinado caso de uso. Alguns dos comandos para explorar essas funcionalidades são:

```
$ sudo incus storage list
+-------+--------+-------------+---------+---------+
| NAME  | DRIVER | DESCRIPTION | USED BY |  STATE  |
+-------+--------+-------------+---------+---------+
| teste | btrfs  |             | 5       | CREATED |
+-------+--------+-------------+---------+---------+

# Criando uma nova pool
$ sudo incus storage create piscina dir
Storage pool piscina created
$ sudo incus storage list
+---------+--------+-------------+---------+---------+
|  NAME   | DRIVER | DESCRIPTION | USED BY |  STATE  |
+---------+--------+-------------+---------+---------+
| piscina | dir    |             | 0       | CREATED |
+---------+--------+-------------+---------+---------+
| teste   | btrfs  |             | 5       | CREATED |
+---------+--------+-------------+---------+---------+

# Alternando o armazenamento do debinho para piscina
$ sudo incus stop debinho
$ sudo incus move debinho --storage piscina

# Confirme com
$ sudo incus config device show debinho
```

```
eth0:
  name: eth0
  network: redinha
  type: nic
root:
  path: /
  pool: piscina
  type: disk
```

Capítulo 3

Podman: A Arquitetura Daemonless

Continuando nossa jornada pelas tecnologias de contêineres, saímos do nível de "sistema"do LXC e entramos no mundo dos "contêineres de aplicação". O principal expoente moderno nesta área, e uma alternativa direta ao Docker, é o **Podman**.

O Podman (Pod Manager) se distingue por sua arquitetura *daemonless* (sem daemon) e sua filosofia *rootless* (sem raiz) como padrão. Diferente do Docker, que depende de um processo de fundo (o daemon) rodando como `root` para gerenciar contêineres, o Podman interage diretamente com o kernel. Isso significa que todo o ciclo de vida de um contêiner — desde o download da imagem até sua execução — pode ser gerenciado por um usuário comum, sem privilégios de superusuário. Esta abordagem representa um avanço significativo na segurança de contêineres.

3.1 Instalação

A instalação do Podman em sistemas baseados em Debian é direta, utilizando o gerenciador de pacotes `apt` com privilégios de superusuário:

```
$ su -
$ apt-get update
$ apt-get -y install podman
```

Após a instalação, podemos verificar a versão com o comando:

```
$ podman -v
```

3.2 Iniciando Nosso Primeiro Contêiner

O rito de passagem para qualquer ferramenta de contêineres é executar uma imagem de teste. A sintaxe do Podman é, por design, idêntica à do Docker, o que nos permite testar

a ferramenta com um `podman run hello-world`. No entanto, devemos usar o nome qualificado da imagem para evitar erros de resolução ("short-name resolution") comuns em algumas distribuições:

```
$ podman run docker.io/library/hello-world
```

Este comando baixa a imagem de teste (caso não esteja em cache) e a executa. Se tudo estiver configurado corretamente, você verá uma mensagem de boas-vindas do Docker. É possível que o Podman venha solicitar uma autenticação para poder puxar a imagem do hub do docker. Para o nosso caso você pode tentar realizar a autenticação conforme solicitado ou optar por puxar uma outra imagem como `podman run -it ubuntu`.

3.3 Operando Contêineres no Podman

O Podman oferece um conjunto de comandos robustos para gerenciar o ciclo de vida dos contêineres.

3.3.1 Comandos Essenciais

Para interagir com um contêiner em execução, como abrir um terminal interativo, usamos `podman exec`:

```
$ podman exec -it <ID_CONTAINER> /bin/bash
```

Também é possível utilizar o nome do contêiner ao invés do seu id. Para exibir informações detalhadas sobre a configuração de um contêiner (como IPs, volumes montados, etc.), usamos `podman inspect`:

```
$ podman inspect <ID_CONTAINER>
```

O ciclo de vida de execução é gerenciado com `pause`, `unpause`, `stop` e `start`. Para remover um contêiner que não é mais necessário, usa-se `podman rm`. A flag `-f` força a remoção de um contêiner que ainda esteja em execução.

```
$ podman pause <ID_CONTAINER>
$ podman unpause <ID_CONTAINER>
$ podman stop <ID_CONTAINER>
$ podman start <ID_CONTAINER>
$ podman rm -f <ID_CONTAINER>
```

3.3.2 Gerenciamento de Recursos

O Podman permite monitorar o consumo de recursos (CPU, memória, rede) em tempo real com `podman stats`.

Mais importante, ele permite limitar dinamicamente os recursos de um contêiner em execução usando `podman update`. Por exemplo, para limitar um contêiner a 50% de um núcleo de CPU e 128 MB de RAM:

```
# Limita o contêiner a 50% da CPU
$ podman update --cpus 0.5 <ID_CONTAINER>

# Limita o contêiner a 128 MB de memória
$ podman update --memory 128M <ID_CONTAINER>
```

3.3.3 Primeiro Containerfile com Podman

Assim como o Docker utiliza um `Dockerfile` para definir os passos de construção de uma imagem, o Podman utiliza o mesmo formato de arquivo. Por convenção, para diferenciar o contexto, a comunidade Podman frequentemente nomeia este arquivo como **Containerfile**.

Vamos criar um diretório para nosso projeto e definir um `Containerfile` que instala o utilitário `stress`:

```
mkdir meucontainer
cd meucontainer
nano Containerfile
```

O conteúdo do `Containerfile` utiliza a sintaxe padrão (note o uso da imagem base completa para evitar erros):

```
FROM docker.io/library/debian:latest

LABEL app="MeuContainer"

RUN apt-get update && apt-get install -y stress && apt-get clean

CMD stress --cpu 1 --vm-bytes 32m --vm 1
```

Para construir a imagem a partir deste arquivo, usamos `podman build`. A flag `-t` define o nome (tag) da imagem:

```
$ podman build -t meucontainer .
```

Com a imagem construída, podemos executá-la da mesma forma que executamos a imagem `hello`:

```
$ podman run -d --name meu_teste meucontainer
```

Erro Comum: Imagem vs. Contêiner

Um erro frequente é tentar gerenciar a execução usando o nome da imagem em vez do nome do contêiner.

Se você rodar o comando acima sem a flag `-name`, o Podman criará um contêiner com um nome aleatório (como `practical_bell`). Se você tentar rodar `podman stop meucontainer`, receberá um erro, pois `meucontainer` é o nome da imagem.

A Solução: Sempre use o comando `podman ps` para listar os contêineres em execução e descobrir o nome correto (na coluna `NAMES`) antes de tentar pausar ou remover uma instância.

3.4 Aprofundando em Ambientes Rootless

Na sessão anterior, introduzimos o conceito de *rootless* como a principal vantagem de segurança do Podman. Agora, vamos aprofundar tecnicamente no que isso significa, como funciona "por baixo dos panos"e como configurar corretamente o ambiente do host para suportá-lo.

3.4.1 O que é um Ambiente Rootless?

A verdadeira potência do Podman é sua capacidade nativa de operar em modo *rootless*. Para isso, basta garantir que seu usuário comum tenha as permissões corretas e possa executar os comandos sem `sudo` ou `su -`. Tradicionalmente, ferramentas de contêiner dependiam de um daemon central rodando como `root`. Isso criava um vetor de ataque significativo: se um processo malicioso conseguisse "escapar"do contêiner, ele poderia ganhar acesso ao daemon e, consequentemente, obter privilégios de superusuário no sistema host. Um ambiente *rootless* quebra esse paradigma.

Com o Podman, todo o ciclo de vida do contêiner — desde o download da imagem até a execução e o gerenciamento de rede — ocorre inteiramente dentro do espaço de privilégios do usuário que executou o comando. Nesse sentido, ao executar como um usuário comum, o Podman cria e armazena os contêineres e imagens dentro do diretório `home` daquele usuário (`~/.local/share/containers`), sem tocar nos diretórios do sistema.

3.4.2 Os Bastidores do Rootless

Para que um usuário comum possa realizar tarefas que normalmente exigiriam privilégios de `root` (como montar sistemas de arquivos e configurar redes), o Podman utiliza duas tecnologias fundamentais do kernel Linux.

User Namespaces (userns)

Os *User Namespaces* são a tecnologia central. Eles permitem que um processo tenha privilégios de "root"*dentro* de seu próprio namespace, sem ser o `root` do sistema

host.

O sistema mapeia o ID do usuário (por exemplo, `UID 1000`) no host para o `UID 0` (root) dentro do contêiner. Da mesma forma, uma faixa de UIDs "subordinados"é alocada para aquele usuário no host, que será mapeada para os UIDs de usuários comuns dentro do contêiner (ex: `UID 100000` no host se torna `UID 1` no contêiner).

Rede com `slirp4netns`

Como um usuário comum não pode criar ou gerenciar interfaces de rede no host (como a bridge `docker0`), o Podman utiliza `slirp4netns`. Esta ferramenta cria uma rede virtual no "espaço do usuário", permitindo que os contêineres acessem a rede externa através do namespace de rede do próprio usuário, de forma semelhante a como uma máquina virtual em modo "NAT"se conecta.

3.4.3 Configurando o Ambiente Host para Rootless

Embora o Podman em si possa ser instalado facilmente, para que o modo *rootless* funcione corretamente, o host precisa de algumas dependências e configurações.

Instalando Dependências Essenciais

Como superusuário, precisamos garantir que o host tenha os pacotes que fornecem as funcionalidades de rede e armazenamento para o modo rootless:

```
# Use sudo ou troque para root com 'su -'
sudo apt-get update
sudo apt-get -y install slirp4netns fuse-overlayfs
```

- `slirp4netns`: Fornece a rede para os contêineres rootless.
- `fuse-overlayfs`: Permite a criação de camadas de sistema de arquivos (overlay) sem privilégios de root.

Configurando UIDs e GIDs Subordinados

Este é o passo mais crítico. O sistema precisa saber quais faixas de User IDs (UIDs) e Group IDs (GIDs) um usuário tem permissão para usar em seus namespaces. Essas faixas são definidas nos arquivos `/etc/subuid` e `/etc/subgid`.

Podemos verificar se nosso usuário já possui essas faixas alocadas:

```
$ grep $USER /etc/subuid
$ grep $USER /etc/subgid
```

Se os comandos não retornarem nada, precisamos adicioná-los. O comando `usermod`, executado como `root`, aloca uma faixa de 65.536 UIDs e GIDs para o usuário especificado:

```
# Substitua 'seu_usuario' pelo seu nome de usuário
sudo usermod --add-subuids 100000-165535 --add-subgids
   100000-165535 seu_usuario
```

Após esta alteração, o usuário precisa fazer logout e login novamente para que as mudanças tenham efeito.

Troubleshooting: Problemas Comuns de Configuração

Em ambientes de laboratório ou instalações mínimas (como Debian netinst ou containers LXC), é comum encontrar dois obstáculos nessa etapa:

1. **Arquivo `subuid` ausente:** Às vezes, o arquivo `/etc/subgid` existe, mas o `/etc/subuid` não. Isso impede o mapeamento de usuários. *Correção:* Se o comando `usermod` falhar, você pode criar o arquivo manualmente. O formato deve ser idêntico ao do subgid: `usuario:100000:65536`.
2. **Falta do `sudo` e Dependências:** Em instalações "cruas", o comando `sudo` pode não vir instalado. *Correção:* É necessário logar como root real (via `su -`) para instalar as dependências críticas (`slirp4netns` e `fuse-overlayfs`). Sem elas, o Podman até pode rodar, mas falhará ao criar a rede ou montar o sistema de arquivos.

3.4.4 Operando em Modo Rootless na Prática

Com o ambiente configurado, podemos verificar se o Podman está operando corretamente. O comando `podman info` revelará que os caminhos de armazenamento (`graphRoot`) e execução (`runRoot`) agora apontam para o diretório `home` do usuário, e não para `/var/lib/containers`:

```
$ podman info | grep -E 'graphRoot|runRoot'
```

A saída será semelhante a:

```
  graphRoot: /home/seu_usuario/.local/share/containers/storage
  runRoot: /run/user/1000/containers
```

Isso prova que o Podman está armazenando todas as suas imagens e dados dentro do espaço do usuário.

3.4.5 Limitações do Modo Rootless: Mapeamento de Portas

Uma limitação importante do modo rootless é que usuários comuns não podem mapear serviços para portas privilegiadas do host (aquelas abaixo de 1024), pois isso é uma restrição do kernel.

Por exemplo, tentar expor um servidor web na porta 80 do host falhará:

```
# ERRO: Usuário comum não pode usar a porta 80 do host
$ podman run -d --name web -p 80:80 nginx
Error: rootlessport cannot expose privileged port 80
```

A solução é mapear para uma porta não privilegiada (acima de 1024):

```
# CORRETO: Mapeia a porta 8080 do host para a porta 80 do
   contêiner
$ podman run -d --name web -p 8080:80 nginx
```

O servidor web estará, então, acessível em `http://localhost:8080`.

3.5 Orquestração com Podman Compose

Até agora, nossos comandos `podman run` lidaram com um único contêiner por vez. No entanto, aplicações do mundo real raramente são tão simples. Uma aplicação web moderna, como um WordPress ou Nextcloud, tipicamente envolve múltiplos componentes — um servidor web (como Nginx ou Apache), a aplicação em si (em PHP) e um banco de dados (como MySQL ou PostgreSQL) — todos rodando em contêineres separados que precisam de rede, volumes e uma ordem de inicialização específica.

Gerenciar essa complexidade manualmente com múltiplos comandos `podman run` é impraticável e propenso a erros. Para resolver isso, utilizamos uma abordagem declarativa, definindo o estado desejado de nossa aplicação em um único arquivo. No ecossistema Podman, essa ferramenta é o `podman-compose`.

3.5.1 O que é `podman-compose`?

É crucial entender que o `podman-compose` difere filosoficamente do `docker-compose`. Enquanto a ferramenta do Docker (especialmente a V2) é um plugin que se comunica com a API do daemon Docker, o `podman-compose` é uma ferramenta independente, escrita em Python, que atua como um **tradutor**.

Ele foi projetado para ser compatível com a sintaxe dos arquivos `docker-compose.yml`, o que facilita a migração. Sua função principal é:

1. Ler e interpretar o arquivo `compose.yml` que define os serviços, redes e volumes.

2. Traduzir essas definições em uma série de comandos `podman` equivalentes.

Por exemplo, uma seção `service` no YAML é traduzida para um `podman run` com todos os mapeamentos de porta, volumes e variáveis de ambiente corretos. Uma seção `network` se torna um `podman network create`.

A maior vantagem desta abordagem é que ela herda todos os benefícios do Podman: opera em modo *rootless* por padrão e não depende de um daemon central.

3.5.2 Instalação do `podman-compose`

Assumindo que o ambiente *rootless* já foi configurado (conforme o capítulo anterior, com `slirp4netns` e `fuse-overlayfs`), a instalação do `podman-compose` em sistemas Debian é feita através do `apt`:

```
# Como root
apt-get install podman-compose
```

3.6 Aplicações Práticas: Nextcloud e WordPress

Vamos explorar a orquestração através de dois estudos de caso idênticos em sua estrutura: a implantação do Nextcloud e do WordPress. Ambas são aplicações que exigem dois serviços principais:

- **O serviço de banco de dados** (ex: `mariadb` ou `mysql`).
- **O serviço da aplicação** (ex: `nextcloud` ou `wordpress`).

O arquivo `compose.yml` é onde descrevemos essa relação. Vamos analisar a estrutura para o Nextcloud.

3.6.1 Estrutura de um Arquivo `compose.yml`

Primeiro, criamos um diretório para o projeto e, dentro dele, o arquivo `compose.yml`:

```
mkdir nextcloud-podman && cd nextcloud-podman
nano compose.yml
```

O conteúdo do arquivo define nossos dois serviços, `db` e `app` (usando imagens qualificadas para evitar erros):

```
services:
  db:
    image: docker.io/mariadb:10.6
```

```
    container_name: nextcloud_db
    restart: always
    command: --transaction-isolation=READ-COMMITTED
       --binlog-format=ROW
    volumes:
      - db_data:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=seu_password_super_secreto
      - MYSQL_PASSWORD=nextcloud_password
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud

  app:
    image: docker.io/nextcloud
    container_name: nextcloud_app
    restart: always
    ports:
      - "8080:80"  # Porta alta para rootless
    volumes:
      - nextcloud_data:/var/www/html
    depends_on:
      - db

volumes:
  db_data:
    name: nextcloud_db_data
  nextcloud_data:
    name: nextcloud_app_data
```

3.6.2 Analisando a Anatomia do Compose

Este arquivo é um excelente exemplo de orquestração. Vamos destacar os conceitos-chave:

- **Serviços (`services`)**: Cada bloco, `db` e `app`, é um serviço. O `podman-compose` criará um contêiner para cada um.
- **Persistência (`volumes`)**: A seção `volumes:` no final declara "volumes nomeados"gerenciados pelo Podman. Dentro de cada serviço, a linha `volumes:` (ex: `db_data:/var/lib/mysql`) mapeia esse volume nomeado para um diretório *dentro* do contêiner. Isso garante que, se o contêiner for destruído, os dados do banco de dados e os arquivos do Nextcloud persistam.
- **Rede e Descoberta**: O `podman-compose` cria automaticamente uma rede interna para este projeto. É por isso que o serviço `app` pode se conectar ao banco de dados

usando o nome `db` como host (veja a variável `WORDPRESS_DB_HOST` no exemplo do WordPress).

- **Ordem de Inicialização (`depends_on`)**: A diretiva `depends_on: - db` no serviço `app` instrui o Podman a iniciar o contêiner do banco de dados *antes* de iniciar o contêiner da aplicação.
- **Portas (`ports`)**: A linha `"8080:80"` no serviço `app` é a única que expõe algo ao mundo exterior. Ela mapeia a porta 8080 do nosso host (lembre-se, *rootless* não pode usar portas < 1024) para a porta 80, onde o servidor web do Nextcloud está escutando dentro do contêiner.

A implantação do WordPress segue um padrão idêntico, apenas substituindo as imagens e as variáveis de ambiente apropriadas.

Para iniciar a aplicação, o comando é simples e, o mais importante, executado como um usuário comum:

```
# O -d significa "detached" (em segundo plano)
podman-compose up -d
```

O `podman-compose` lerá o arquivo, criará os volumes, a rede e os contêineres na ordem correta. A aplicação estará acessível em `http://localhost:8080`.

3.7 Gerenciamento Avançado de Rede com Traefik

Embora o mapeamento de portas (como `8080:80`) funcione para uma ou duas aplicações, ele rapidamente se torna complexo. Teríamos que memorizar que `localhost:8080` é o Nextcloud, `localhost:8081` é o WordPress, `localhost:8082` é o Portainer, e assim por diante.

A solução profissional para isso é um **Reverse Proxy** (Proxy Reverso). O **Traefik** é um proxy reverso moderno, nativo para a nuvem, projetado especificamente para contêineres.

Sua principal vantagem é a **descoberta de serviço automática**. Em vez de editarmos manualmente um arquivo de configuração toda vez que subimos um novo serviço, o Traefik "assiste"à API do Podman. Quando ele vê um novo contêiner subir com *labels* específicas, ele automaticamente configura o roteamento para ele.

3.7.1 Diferenças na Configuração com Podman

Para o Traefik funcionar com o Podman rootless, duas correções são necessárias em relação à configuração padrão do Docker:

1. **O Socket da API**: O Podman expõe sua API em um socket de usuário (user socket), geralmente em `/run/user/<UID>/podman/podman.sock`.

2. **O Provedor**: Devido a incompatibilidades em versões recentes (como a v2.11), o provedor nativo `podman` pode falhar. A solução robusta é utilizar o provedor `docker` padrão, mas apontando-o para o socket do Podman.

Primeiro, habilitamos o socket da API do Podman para nosso usuário. **Atenção:** Não use sudo, pois o socket deve pertencer ao usuário:

```
# Habilita e inicia o socket para o usuário atual
systemctl --user enable --now podman.socket
```

3.7.2 Configuração do Traefik com Compose

A seguir, um arquivo `compose.yml` corrigido que implanta o Traefik e um serviço de exemplo `whoami`, resolvendo os problemas de portas e provedores relatados:

```
services:
  traefik:
    image: docker.io/traefik:v3.6
    container_name: traefik
    command:
      # Instruções para o Traefik
      - "--api.insecure=true"
      # Usamos o provider Docker compatível com a API do Podman
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      # Apontamos para o socket do usuário
      -
        "--providers.docker.endpoint=unix:///var/run/podman/podman.sock"
      - "--entrypoints.web.address=:8081"
    ports:
      - "8080:8080"      # Porta 8080 (Host) mapeia para 8080 (Traefik)
        [Rootless]
      - "8081:8081"    # Porta para o Dashboard do Traefik
    volumes:
      # Monta o socket do Podman (substitua 1000 pelo seu 'id -u')
      - /run/user/1000/podman/podman.sock:/var/run/podman/podman.sock:z
    networks:
      - proxy

  whoami:
    image: docker.io/traefik/whoami
    container_name: whoami
    labels:
      - "traefik.enable=true"
      # O roteamento deve considerar a porta exposta (8081)
      -
        "traefik.http.routers.whoami.rule=Host(`whoami.podman.localhost`)"
```

```
      - "traefik.http.routers.whoami.entrypoints=web"
    networks:
      - proxy

networks:
  proxy:
    name: proxy
```

A mágica acontece nas `labels` do serviço `whoami`:

- `traefik.enable=true`: "Olá Traefik, por favor, gerencie este contêiner."
- `...rule=Host(`whoami.podman.localhost`)`: "Se uma requisição chegar com o domínio `whoami.podman.localhost`, envie-a para mim."

Após subir este compose, podemos acessar o dashboard do Traefik em `http://localhost:8081` e nossa aplicação em `http://whoami.podman.localhost:8081` (após adicionar este domínio ao nosso `/etc/hosts` local).

3.8 Recursos Avançados: Pods e Manifestos Kubernetes

O Podman possui dois recursos fundamentais que o diferenciam do Docker e o aproximam do Kubernetes.

3.8.1 O Conceito de "Pod"

Emprestado diretamente do Kubernetes, um **Pod** é a menor unidade de implantação. É um grupo de um ou mais contêineres que compartilham os mesmos namespaces de rede e IPC.

Isso significa que contêineres dentro do mesmo pod podem se comunicar usando `localhost`, como se estivessem na mesma máquina. Isso é mais eficiente do que criar uma rede virtual. Em nosso exemplo Nextcloud/WordPress, poderíamos colocar os serviços `app` e `db` no mesmo pod. O `podman-compose` não gerencia pods nativamente, mas o `podman` sim.

```
# Cria um Pod que expõe a porta 8080 (rootless)
$ podman pod create --name minha-app-pod -p 8080:80

# Executa os contêineres DENTRO do pod
$ podman run -d --pod minha-app-pod --name redis_db
   docker.io/redis
$ podman run -d --pod minha-app-pod --name webapp minha-webapp
```

Neste cenário, a `webapp` se conectaria ao Redis simplesmente em `localhost:6379`.

3.8.2 Gerando Manifestos Kubernetes

A funcionalidade mais poderosa do Podman é sua capacidade de atuar como uma ponte entre o desenvolvimento local e a produção em Kubernetes. O Podman pode inspecionar um pod em execução e gerar um manifesto `.yml` do Kubernetes que o descreve.

```
# Gere o YAML a partir do Pod que criamos
$ podman kube generate pod minha-app-pod > minha-app.yml
```

O arquivo `minha-app.yml` resultante é um recurso Kubernetes válido que pode ser implantado em qualquer cluster (como Minikube, GKE, ou OpenShift) com `kubectl apply -f minha-app.yml`. Isso unifica drasticamente o fluxo de trabalho de desenvolvimento e produção.

Capítulo 4

Introdução ao Docker: O Padrão da Indústria

Neste capítulo, voltamos nossa atenção para o **Docker**, a plataforma que popularizou os contêineres de aplicação e definiu o padrão da indústria. Embora o Podman ofereça uma arquitetura *daemonless* inovadora, é fundamental compreender o Docker, pois sua arquitetura, ferramentas (como o Docker Compose) e o próprio formato do `Dockerfile` são a base do ecossistema de contêineres moderno.

4.1 Instalação

Ao contrário do Podman, o Docker opera em uma arquitetura cliente-servidor. O componente central é o **daemon Docker** (`dockerd`), um processo que roda com privilégios de `root` e é responsável por construir, executar e gerenciar os contêineres. A ferramenta de linha de comando `docker` (o cliente) se comunica com a API deste daemon.

A instalação no Debian envolve adicionar o repositório oficial do Docker para garantir que recebamos as versões mais recentes.

Primeiro, como superusuário, configuramos o `apt` para confiar no repositório do Docker:

```
# Adicionar o repositório do Docker
apt-get update
apt-get install -y ca-certificates curl gnupg lsb-release
mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg
   --dearmor -o /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch=$(dpkg --print-architecture)
     signed-by=/etc/apt/keyrings/docker.gpg]
```

```
    https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" | tee
    /etc/apt/sources.list.d/docker.list > /dev/null
```

Com o repositório configurado, atualizamos o índice de pacotes e instalamos o Docker Engine, a CLI e o plugin do Compose:

```
# Instalar o Docker Engine
apt-get update
apt-get install -y docker-ce docker-ce-cli containerd.io
   docker-compose-plugin
```

Uma vez instalado, podemos verificar a versão com:

```
$ docker -v
```

4.2 Executando o Docker como um Usuário Não-Root

Por padrão, apenas o usuário `root` (ou usuários com `sudo`) pode se comunicar com o daemon do Docker. Para permitir que seu usuário comum execute comandos `docker` sem `sudo`, você deve adicioná-lo ao grupo `docker` (criado durante a instalação).

*Nota de Segurança: Adicionar um usuário ao grupo `docker` é equivalente a dar a ele privilégios de `root`, pois ele pode usar o Docker para montar qualquer diretório do host ou executar comandos privilegiados. A abordagem **rootless** do Podman, discutida anteriormente, é a solução para esta vulnerabilidade.*

4.3 Operações Básicas de Contêineres

A sintaxe de comandos do Docker é o padrão que o Podman imitou. O ciclo de vida de um contêiner é gerenciado com comandos idênticos:

- `docker run hello-world`: O comando canônico para testar a instalação.
- `docker exec -it <ID> /bin/bash`: Entra em um contêiner em execução.
- `docker stop <ID>`: Para um contêiner.
- `docker rm <ID>`: Remove um contêiner.
- `docker stats`: Monitora o uso de recursos.
- `docker update --cpus 0.5 <ID>`: Atualiza recursos de um contêiner em execução.

4.4 Aprofundando em Dockerfiles

O `Dockerfile` é o "projeto"ou a "receita"de uma imagem de contêiner. É um script de texto que contém uma sequência de comandos que o daemon do Docker utiliza para montar, de forma automatizada e reprodutível, uma imagem.

Cada instrução em um `Dockerfile` cria uma nova "camada"(layer) na imagem. O Docker armazena essas camadas em cache, um recurso que acelera drasticamente as *builds* futuras, pois o Docker só reconstrói as camadas que mudaram.

4.4.1 Anatomia de um Dockerfile: Instruções Essenciais

Vamos detalhar as instruções mais comuns e sua finalidade:

- **`FROM`**: Define a imagem base a partir da qual a nova imagem será construída. Todo `Dockerfile` deve começar com `FROM`. A escolha de uma base pequena (como `alpine` ou `debian:slim`) é a melhor prática para imagens leves.
- **`WORKDIR`**: Define o diretório de trabalho para todas as instruções subsequentes (`RUN`, `COPY`, `CMD`, etc.). É uma prática muito superior a usar `RUN cd /meu-app`.
- **`COPY`**: Copia arquivos ou diretórios do contexto do build (a máquina local) para dentro do sistema de arquivos da imagem.
- **`RUN`**: Executa um comando shell *durante o processo de build*. É usado para instalar pacotes (`RUN apt-get install -y ...`), compilar código ou criar diretórios. Cada `RUN` cria uma nova camada.
- **`CMD`**: Define o comando padrão que será executado quando um contêiner for iniciado *a partir* da imagem. Só pode haver uma instrução `CMD`. Se o usuário especificar um comando ao iniciar o contêiner (ex: `docker run minha-imagem /bin/bash`), o `CMD` padrão será ignorado.
- **`EXPOSE`**: Documenta quais portas de rede o contêiner escuta em tempo de execução. É importante notar que `EXPOSE` **não** publica a porta; ele apenas informa ao operador humano (e a algumas ferramentas) quais portas são importantes. A publicação real é feita com `-p` no comando `docker run`.

4.4.2 Otimização: Encadeando Comandos `RUN`

Como cada `RUN` cria uma camada, `Dockerfiles` não otimizados podem ficar inchados.

Não otimizado (cria 3 camadas):

```
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y git
```

A forma otimizada é encadear os comandos com `&&` e
, o que os agrupa em uma única instrução `RUN` e, portanto, em uma única camada. Além disso, limpamos o cache do `apt` na mesma camada, garantindo que o cache não seja incluído desnecessariamente no tamanho final da imagem.

Otimizado (cria 1 camada):

```
RUN apt-get update && apt-get install -y \
    curl \
    git \
    && rm -rf /var/lib/apt/lists/*
```

4.4.3 Tópicos Avançados de Dockerfile

Para criar imagens prontas para produção, dominamos três conceitos adicionais: `ENTRYPOINT`, Builds Multi-Stage e `HEALTHCHECK`.

`ENTRYPOINT` vs. `CMD`

Este é um dos conceitos mais confusos para iniciantes. A melhor maneira de entendê-los é:

- **`ENTRYPOINT`**: Define o executável principal, o "ponto de entrada"da imagem. Não é feito para ser sobrescrito pelo usuário.
- **`CMD`**: Define os argumentos *padrão* para o `ENTRYPOINT`.

Um exemplo clássico é a imagem do `apachectl`:

```
ENTRYPOINT ["/usr/sbin/apachectl"]
CMD ["-D", "FOREGROUND"]
```

Ao executar `docker run <imagem>`, o contêiner executa `/usr/sbin/apachectl -D FOREGROUND`. Se o usuário executar `docker run <imagem> -X`, ele estará sobrescrevendo apenas o `CMD`, e o comando final será `/usr/sbin/apachectl -X`.

Builds Multi-Stage

Builds multi-stage são a técnica mais eficaz para criar imagens pequenas e seguras. A ideia é usar uma imagem grande e cheia de ferramentas (como `golang` ou `maven`)

para compilar a aplicação e, em seguida, copiar *apenas o binário compilado* para uma imagem final mínima (como `alpine` ou `scratch`).

Isso separa o ambiente de build do ambiente de produção, resultando em uma imagem final drasticamente menor, que não contém código-fonte, compiladores ou ferramentas de build.

```
# Estágio 1: Build
FROM golang AS buildando
WORKDIR /app
ADD . /app
RUN go build -o meugo

# Estágio 2: Imagem Final
FROM alpine
WORKDIR /new
# Copia apenas o executável do estágio anterior
COPY --from=buildando /app/meugo /new/
ENTRYPOINT ./meugo
```

HEALTHCHECK

A instrução `HEALTHCHECK` define um comando que o Docker executa periodicamente *dentro* do contêiner para verificar se ele está funcionando corretamente (ou seja, "saudável"). Isso é crucial para orquestradores, que podem usar essa informação para reiniciar automaticamente um contêiner "doente"(`unhealthy`).

```
HEALTHCHECK --interval=1m --timeout=3s \
 CMD curl -f http://localhost/ || exit 1
```

O status da verificação (ex: `starting`, `healthy`, `unhealthy`) aparecerá na saída do `docker ps`.

4.5 Gerenciamento de Dados com Volumes

Por padrão, contêineres são **efêmeros**. Seus sistemas de arquivos são voláteis; quaisquer dados escritos dentro de um contêiner são perdidos quando ele é removido. Para aplicações que precisam manter estado (como bancos de dados, uploads de usuários ou arquivos de configuração), precisamos de uma forma de persistir dados. O mecanismo preferido pelo Docker para isso são os **Volumes**.

4.5.1 Tipos de Persistência

Existem duas formas principais de persistir dados no Docker:

- **Named Volumes (Volumes Nomeados)**: Esta é a abordagem recomendada. Os volumes são gerenciados diretamente pelo Docker e armazenados em uma área específica no host (ex: `/var/lib/docker/volumes/`). Eles são desacoplados do ciclo de vida do contêiner. Podemos criar um volume com `docker volume create meusdados`.
- **Bind Mounts**: Mapeiam um diretório ou arquivo existente no sistema de arquivos do host para dentro de um contêiner (ex: `-v /opt/meu-app:/app`). São úteis em desenvolvimento para refletir mudanças no código-fonte em tempo real, mas em produção são menos flexíveis que os volumes nomeados.

Para usar um volume nomeado, o criamos e o anexamos no `docker run`:

```
# Crie um volume nomeado
$ docker volume create meusdados

# Execute um contêiner usando o volume
$ docker container run -ti --mount
   type=volume,src=meusdados,dst=/dados debian
```

Agora, qualquer coisa escrita em `/dados` dentro do contêiner será salva no volume `meusdados` no host. Se removermos o contêiner, o volume (e seus dados) permanecerá intacto.

4.6 Orquestração com Docker Compose

Similar ao `podman-compose`, o **Docker Compose** é a ferramenta do Docker para definir e executar aplicações multi-contêiner. Ele usa um arquivo YAML (por padrão, `compose.yml`) para declarar todos os serviços, redes e volumes que compõem uma aplicação.

4.6.1 Instalando o Docker Compose

Desde 2021, o Docker Compose foi reescrito em Go e integrado diretamente ao Docker Engine como um plugin (`docker-compose-plugin`). O comando moderno é `docker compose` (sem o hífen), que instalamos no primeiro capítulo.

4.6.2 Orquestrando o Portainer

Vamos usar o Compose para implantar o **Portainer**, uma popular interface gráfica de gerenciamento para o Docker.

Primeiro, criamos nosso arquivo `compose.yml`:

```
services:
  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer
    ports:
      - "9443:9443"
      - "9000:9000"
    volumes:
      # Mapeia o socket do Docker para que o Portainer possa
         gerenciar o Docker
      - /var/run/docker.sock:/var/run/docker.sock
      # Volume para persistir os dados do Portainer
      - portainer_data:/data
    restart: always

volumes:
  portainer_data:
```

Os dois mapeamentos de volume aqui são cruciais:

- `/var/run/docker.sock:/var/run/docker.sock`: Este é um *bind mount* que mapeia o socket da API do Docker do host para dentro do contêiner. É assim que o Portainer ganha a capacidade de controlar o Docker.
- `portainer_data:/data`: Este é um *volume nomeado* que garante que os dados do Portainer (configurações, senhas) persistam.

Para iniciar a aplicação, navegamos até o diretório do arquivo e executamos:

```
$ docker compose up -d
```

O Portainer estará acessível em `https://localhost:9443`.

4.6.3 Expandindo o Compose: Profiles e .env

O Docker Compose possui recursos avançados para gerenciar ambientes complexos.

Profiles (Perfis)

Os perfis permitem agrupar serviços no `compose.yml` e ativá-los seletivamente. Isso é ideal para separar serviços de produção (padrão) de serviços de desenvolvimento ou depuração (`debug`).

Por exemplo, podemos adicionar um visualizador de logs como o `Dozzle` ao nosso compose, mas associá-lo a um perfil `debug`:

```
services:
  portainer:
    # ... (configuração do portainer) ...

  dozzle:
    image: amir20/dozzle:latest
    container_name: dozzle
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
      - "8081:8080"
    profiles:
      - debug  # Este serviço só iniciará se o perfil 'debug'
         for ativado
```

Ao executar `docker compose up -d`, apenas o Portainer iniciará. Para iniciar ambos, executamos: `docker compose -profile debug up -d`.

Arquivos `.env` para Variáveis

É uma má prática "chumbar"(hardcode) valores como senhas, portas ou nomes de usuário no `compose.yml`. A solução é usar um arquivo `.env` no mesmo diretório. O Docker Compose o carrega automaticamente.

Arquivo `.env`:

```
# .env
PORTAINER_WEB_PORT=9443
```

Arquivo `compose.yml`:

```
services:
  portainer:
    # ...
    ports:
      # Usando a variável do arquivo .env
      - "${PORTAINER_WEB_PORT}:9443"
    # ...
```

Isso torna a configuração mais segura e flexível.

4.7 Estudos de Caso: Nextcloud e WordPress

A implantação do Nextcloud e do WordPress com Docker Compose segue exatamente o mesmo padrão de dois serviços (aplicação + banco de dados) que vimos no

capítulo do Podman, demonstrando a portabilidade dos arquivos `compose.yml` entre os ecossistemas.

A única diferença notável é o nome do driver de rede padrão (Docker cria uma rede `bridge`, enquanto Podman usa `netavark` ou `CNI`), mas para o usuário final, a descoberta de serviço baseada no nome do serviço (`db`) funciona de forma idêntica.

4.8 Orquestração de Cluster: Docker Swarm

O Docker Compose é excelente para gerenciar múltiplos contêineres em um *único host*. No entanto, para produção, precisamos de resiliência e escala, o que significa distribuir nossos contêineres por *múltiplos hosts* (nós).

O **Docker Swarm** é a ferramenta de orquestração nativa do Docker para gerenciar um cluster de nós como se fossem um único sistema.

4.8.1 Arquitetura: Managers e Workers

Um cluster Swarm consiste em dois tipos de nós:

- **Managers**: Responsáveis por gerenciar o estado do cluster, agendar serviços e manter a consistência. Para alta disponibilidade, recomenda-se um número ímpar de managers (ex: 3 ou 5) para formar um quórum.
- **Workers**: Executam os contêineres (chamados de `tasks`) que são atribuídos pelos managers.

Para inicializar um cluster, vamos ao nó que será o primeiro manager e executamos:

```
$ docker swarm init
```

Este comando torna o nó atual um manager e gera um token. Nos outros nós, executamos o comando `docker swarm join <token>` para que eles entrem no cluster como workers.

4.8.2 Serviços no Swarm

No Swarm, não executamos contêineres diretamente; nós criamos **Serviços**. Um serviço define o estado desejado de uma aplicação, incluindo a imagem, o número de réplicas e as portas. O Swarm então garante que o número correto de réplicas (tasks) esteja sempre em execução em algum lugar do cluster.

```
# Cria um serviço chamado 'webserver' com 3 réplicas da imagem
   nginx
```

```
$ docker service create --name webserver --replicas 3 -p
   8080:80 nginx
```

O Swarm agora garantirá que 3 contêineres `nginx` estejam rodando. Se um nó falhar, o Swarm automaticamente reagendará as tasks daquele nó em outros nós saudáveis.

4.8.3 Escalando e Gerenciando Nós

A principal vantagem de um orquestrador é a capacidade de escalar e gerenciar falhas.

Podemos escalar um serviço instantaneamente:

```
$ docker service scale webserver=10
```

O Swarm tratará de criar 7 novas réplicas e distribuí-las pelo cluster.

Para manutenção de um nó (ex: `node01`), podemos drená-lo. O `drain` remove todas as tarefas do nó, reagendando-as em outros nós ativos, sem interromper o serviço:

```
$ docker node update --availability drain node01
```

Após a manutenção, retornamos o nó ao estado ativo: `docker node update -availability active node01`.

4.9 Gerenciamento de Rede Avançado com Traefik

Assim como no Podman, o **Traefik** brilha como um proxy reverso para o Docker, especialmente em um ambiente Swarm. Sua capacidade de descoberta de serviço nos permite expor aplicações à internet de forma dinâmica, sem reconfiguração manual.

A configuração é quase idêntica à do Podman, com uma diferença chave: em vez de usar o provedor `podman`, usamos o provedor `docker`.

4.9.1 Configuração do Traefik com Docker Compose

Em um ambiente Docker (seja single-host ou Swarm), implantamos o Traefik como um serviço, geralmente via Docker Compose.

```
# docker-compose.yml
services:
  traefik:
    image: traefik:v2.11
    container_name: traefik
    command:
      - "--api.dashboard=true"
```

```
      - "--providers.docker=true" # Usando o provedor Docker
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"     # Porta para o tráfego HTTP
      - "8080:8080" # Porta para o Dashboard
    volumes:
      # Monta o socket do Docker para que o Traefik possa ouvir
         os eventos
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - proxy

  whoami:
    image: traefik/whoami
    container_name: whoami
    labels:
      - "traefik.enable=true"
      -
         "traefik.http.routers.whoami.rule=Host(`whoami.localhost`)"
      - "traefik.http.routers.whoami.entrypoints=web"
    networks:
      - proxy

networks:
  proxy:
    name: proxy
```

O volume `/var/run/docker.sock:/var/run/docker.sock:ro` é a chave. É através dele que o Traefik monitora a API do Docker e detecta novos contêineres (ou serviços Swarm) que possuem as *labels* `traefik.enable=true`. Ao detectar um, ele lê a *label* `rule` e cria a rota de acesso automaticamente.

Capítulo 5

Introdução à Automação com Ansible

Nos capítulos anteriores, focamos em como *empacotar* e *executar* aplicações de forma isolada e consistente usando contêineres. No entanto, ainda resta um desafio fundamental: como preparar e gerenciar a infraestrutura subjacente onde esses contêineres irão rodar?

A preparação de um servidor (o "provisionamento") envolve tarefas como instalar pacotes, configurar serviços, gerenciar usuários e garantir que os arquivos de configuração estejam corretos. Fazer isso manualmente é lento, propenso a erros e impossível de escalar.

Neste capítulo, introduzimos o **Ansible**, uma poderosa ferramenta de automação de TI que simplifica radicalmente o gerenciamento de configuração e a implantação de aplicações.

5.1 O que é o Ansible?

O Ansible é um motor de automação de código aberto que opera em um paradigma *push-based* (baseado em "empurrar"configurações). Sua característica mais marcante é sua arquitetura **agentless** (sem agentes).

Diferente de outras ferramentas como Puppet ou Chef, que exigem que um "agente"de software seja instalado e mantido em cada servidor gerenciado, o Ansible não requer nada além de uma conexão **SSH** padrão e um interpretador **Python** (que já vem instalado na maioria das distribuições Linux modernas).

Essa simplicidade reduz a complexidade de gerenciamento e a superfície de ataque da sua infraestrutura.

5.2 Conceitos Fundamentais

Para trabalhar com o Ansible, precisamos entender sua terminologia:

- **Control Node (Nó de Controle)**: A máquina onde o Ansible está instalado e de onde você executa os comandos.
- **Managed Nodes (Nós Gerenciados)**: Os servidores que o Ansible gerencia.
- **Inventory (Inventário)**: Um arquivo (em formato INI ou YAML) que lista e agrupa os nós gerenciados. É o "catálogo de endereços"do Ansible.
- **Playbook**: O coração do Ansible. É um arquivo YAML que define uma lista de *tarefas* a serem executadas em um grupo de servidores.
- **Task (Tarefa)**: Uma única ação, como "instalar o pacote nginx"ou "copiar um arquivo".
- **Module (Módulo)**: O código que o Ansible envia via SSH para o nó gerenciado executar uma tarefa. Por exemplo, o módulo `apt` gerencia pacotes no Debian, e o módulo `service` gerencia serviços.

5.3 Instalação e Configuração Prática

Vamos configurar um ambiente básico no Nó de Controle.

5.3.1 Instalação do Ansible

O Ansible é facilmente instalado via gerenciador de pacotes. Em um sistema baseado em Debian/Ubuntu, executamos:

```
# Atualiza o índice de pacotes e instala o Ansible
$ sudo apt-get update
$ sudo apt-get install -y ansible
```

Podemos verificar a instalação com `ansible -version`.

5.3.2 Criando um Inventário

O inventário define *quais* servidores o Ansible irá gerenciar. Vamos criar um diretório para nosso projeto e um arquivo de inventário chamado `hosts`:

```
$ mkdir ansible-lab && cd ansible-lab
$ nano hosts
```

Dentro do arquivo `hosts`, definimos um grupo de servidores. Para este exemplo, vamos assumir que queremos gerenciar um servidor em `192.168.1.100`:

```
# Arquivo: hosts

[webservers]
server1 ansible_host=192.168.1.100
```

- `[webservers]`: Define um grupo de hosts.
- `server1`: É um apelido (alias) para o host.
- `ansible_host`: É uma variável que informa ao Ansible o IP real para conexão.

5.3.3 Testando a Conexão (Comandos Ad-Hoc)

Antes de escrever um playbook complexo, sempre testamos a conectividade. Usamos um comando "ad-hoc"para executar o módulo `ping` em todos os hosts do inventário.

```
# -i especifica o inventário
# -m especifica o módulo (ping)
# 'all' é um grupo especial que significa "todos os hosts"
$ ansible all -i hosts -m ping
```

Se a conexão SSH (geralmente por chaves) estiver funcionando, o Ansible retornará uma resposta `SUCCESS` com um `"ping": "pong"`. Caso contrário, é possível que você encontre uma mensagem semelhante a esta:

```
[server1] UNREACHABLE! => {"msg": "Failed to connect to the
   host via
ssh: (publickey).", "unreachable": true} Permission denied.
```

Casa máquina é um caso que deve ser diagnosticado com atenção, todavia, uma das soluções mais comuns é verificar se a chave pública dos hosts/nodes foram corretamente adicionadas no arquivo `authorized_keys` do host/node de controle. Outra possibilidade de solução é passar o nome do usuário pelo arquivo de inventário do Ansible, no nosso caso `hosts`. Uma terceira possibilidade é adicionar o caminho da chave privada do host-alvo para facilitar o processo de autenticação com `ansible_ssh_private_key_file=` logo após especificar o usuário:

```
# Arquivo: hosts

[webservers]
server1 ansible_host=192.168.1.100  ansible_user=fulani
   ansible_ssh_private_key_file=~/.ssh/id_ed25519
```

5.4 Seu Primeiro Playbook: A Idempotência

Agora, vamos automatizar uma tarefa real: garantir que o servidor web Nginx esteja instalado e rodando. Criamos um arquivo `install_nginx.yml`:

```
# install_nginx.yml
---
- name: Instalar e configurar o Nginx
  hosts: webservers
  become: yes  # Indica que as tarefas devem ser executadas com
     sudo

  tasks:
    - name: Atualizar o cache do apt
      apt:
        update_cache: yes

    - name: Instalar o Nginx
      apt:
        name: nginx
        state: present
```

Vamos analisar este playbook:

- `hosts: webservers`: Define que este *play* será executado no grupo `[webservers]` do nosso inventário.
- `become: yes`: Informa ao Ansible para escalar privilégios (usar `sudo`) para executar as tarefas.
- `tasks`: A lista de ações. Cada tarefa chama um módulo.
- `state: present`: Esta é a chave do gerenciamento de configuração. Estamos dizendo ao Ansible: "Eu não me importo como, apenas garanta que o Nginx *esteja presente*".

Executamos o playbook com o comando:

```
$ ansible-playbook -i hosts install_nginx.yml --ask-become-pass
```

Na primeira execução, o Ansible verá que o Nginx não está instalado e o instalará. A saída da tarefa mostrará `changed`. Se executarmos o *mesmo playbook* uma segunda vez, o Ansible verificará o estado, verá que o Nginx *já está presente* e não fará nada. A saída mostrará `ok`.

Esse conceito é chamado de **Idempotência** e é o pilar do Ansible: um playbook descreve o *estado final desejado*, e o Ansible de forma inteligente só realiza as ações necessárias para alcançá-lo.

5.5 Playbooks Avançados: Handlers e Templates

Instalar pacotes é apenas o começo. O verdadeiro poder do Ansible está em gerenciar arquivos de configuração e o estado dos serviços.

5.5.1 Gerenciando Arquivos e Reiniciando Serviços com Handlers

Um desafio comum é que um serviço (como o Nginx) só deve ser reiniciado se seu arquivo de configuração for realmente alterado. Reiniciá-lo a cada execução do playbook é ineficiente e pode causar indisponibilidade.

O Ansible resolve isso com **Handlers**. Um Handler é uma tarefa especial que só é executada se outra tarefa a "notificar".

Vamos aprimorar nosso playbook para copiar um arquivo `index.html` personalizado e notificar um handler para reiniciar o Nginx apenas se o arquivo for alterado.

Primeiro, criamos o arquivo local:

```
$ mkdir files
$ echo "<h1>Site gerenciado pelo Ansible!</h1>" >
   files/index.html
```

Agora, modificamos nosso playbook:

```
# install_nginx.yml
---
- name: Instalar e configurar o Nginx
  hosts: webservers
  become: yes

  tasks:
    - name: Garantir que o Nginx esteja instalado
      apt:
        name: nginx
        state: present

    - name: Copiar a pagina index.html personalizada
      copy:
        src: files/index.html            # Origem no Control
           Node
        dest: /var/www/html/index.html   # Destino no Managed
           Node
      # ATENÇÃO: A indentação do notify deve estar no mesmo
         nível do módulo copy
      # O nome deve ser EXATAMENTE igual ao definido no handler
         abaixo
      notify: Reiniciar Nginx
```

```
  # Bloco especial para handlers
  handlers:
    - name: Reiniciar Nginx
      service:
        name: nginx
        state: restarted
```

Na primeira execução, o módulo `copy` copiará o arquivo, verá uma mudança (`changed=true`), e notificará o handler `Reiniciar Nginx`, que será executado no final do play. Na segunda execução, o `copy` verá que os arquivos são idênticos (`ok=true`), não notificará o handler, e o Nginx *não* será reiniciado.

5.5.2 Gerando Configurações Dinâmicas com Templates

"Chumbar"arquivos de configuração estáticos não é escalável. Ambientes diferentes (desenvolvimento, produção) precisam de configurações diferentes. O Ansible resolve isso com o módulo `template` e o motor de templates **Jinja2**.

O módulo `template` funciona como o `copy`, mas antes de enviar o arquivo, ele o processa, substituindo variáveis (marcadas com `{{ ... }}`) por valores definidos no playbook.

Vamos transformar nossa página `index.html` em um template. A convenção é usar a extensão `.j2`.

```
$ mkdir templates
$ echo "<h1>{{ mensagem_da_pagina }}</h1>" >
   templates/index.html.j2
```

Agora, modificamos o playbook para usar `template` e definir a variável:

```
# install_nginx.yml
---
- name: Instalar e configurar o Nginx com Templates
  hosts: webservers
  become: yes

  # Define variáveis para este play.
  # CUIDADO: A indentação de 'vars' deve estar alinhada com
     'tasks' e 'hosts'
  vars:
    mensagem_da_pagina: "Site dinâmico com Ansible!"

  tasks:
    - name: Garantir que o Nginx esteja instalado
      apt:
        name: nginx
```

```
        state: present

    - name: Gerar a pagina index.html a partir do template
      template:
        # Garanta que a pasta 'templates' existe no diretório
           onde roda o comando
        src: templates/index.html.j2
        dest: /var/www/html/index.html
      notify: Reiniciar Nginx

  handlers:
    - name: Reiniciar Nginx
      service:
        name: nginx
        state: restarted
```

Ao executar, o Ansible lerá o `index.html.j2`, substituirá `{{ mensagem_da_pagina }}` pelo valor em `vars`, e enviará o arquivo final resultante para o servidor. Agora, podemos gerenciar o conteúdo do nosso site (ou configurações complexas do Nginx) simplesmente alterando as variáveis em nosso playbook, e não os arquivos em si.

Capítulo 6

Introdução ao Kubernetes com Minikube

Nos capítulos anteriores, exploramos como criar contêineres (com Docker e Podman), como gerenciá-los em um único host (com Compose) e como provisionar a infraestrutura (com Ansible). Finalmente, chegamos ao desafio da orquestração em larga escala: como gerenciar, escalar e manter milhares de contêineres distribuídos por um cluster de dezenas ou centenas de máquinas? A resposta para essa pergunta é o **Kubernetes** (comumente abreviado como **K8s**).

6.1 O que é o Kubernetes?

O Kubernetes é um sistema de orquestração de contêineres de código aberto, originalmente desenvolvido pelo Google. Ele automatiza a implantação, o dimensionamento (escalabilidade) e o gerenciamento de aplicações em contêineres.

Ele agrupa os contêineres que compõem uma aplicação (como o servidor web e o banco de dados) em unidades lógicas para facilitar o gerenciamento e a descoberta de serviços. Mais importante, o Kubernetes opera em um nível de *cluster*. Ele abstrai a infraestrutura subjacente (sejam máquinas virtuais, bare-metal ou nuvem pública) e a apresenta como um único e vasto pool de recursos computacionais.

Suas principais funções incluem:

- **Automação de Implantação (Deploy)**: Define o estado desejado da aplicação e o Kubernetes trabalha para alcançá-lo.
- **Balanceamento de Carga e Descoberta de Serviço**: Expõe contêineres na rede e distribui o tráfego entre eles.
- **Auto-healing (Auto-reparação)**: Reinicia automaticamente contêineres que falham, substitui nós problemáticos e garante que o estado desejado seja mantido.

- **Auto-escalabilidade**: Ajusta automaticamente o número de contêineres em execução com base no uso de CPU ou memória.

6.2 Arquitetura de um Cluster Kubernetes

Um cluster Kubernetes é composto por dois tipos de recursos principais: o *Control Plane* (Plano de Controle) e os *Nodes* (Nós).

6.2.1 Control Plane (Manager)

O Control Plane é o "cérebro"do cluster. Ele toma as decisões globais, como agendar contêineres e responder a eventos. É composto por vários componentes, como o `api-server` (o front-end para o cluster), o `etcd` (o banco de dados de estado) e o `scheduler` (que decide em qual nó um contêiner deve rodar).

6.2.2 Nodes (Workers)

Os Nodes, ou "workers", são as máquinas (virtuais ou físicas) que executam as aplicações. Cada nó executa dois processos principais: o `kubelet` (que se comunica com o Control Plane) e um `container runtime` (como o Docker ou `containerd`) que é responsável por, de fato, iniciar e parar os contêineres.

6.3 O que é o Minikube?

Um cluster Kubernetes completo é complexo de configurar. Para fins de aprendizado, desenvolvimento e teste local, usamos o **Minikube**.

O Minikube é uma ferramenta que cria um cluster Kubernetes *local* de forma simples e rápida, geralmente rodando todos os componentes do Control Plane e um nó Worker dentro de uma única máquina virtual ou contêiner Docker em sua máquina. Ele nos permite experimentar a API completa do Kubernetes sem a complexidade de provisionar uma infraestrutura de múltiplos servidores.

6.4 Instalando o Cluster Minikube

Neste capítulo, prepararemos nosso ambiente Debian 12 para executar um cluster Minikube. Isso envolve a instalação de três componentes: o `docker` (que servirá como o "driver"ou a base para o nó do Minikube), o `kubectl` (a ferramenta de linha de comando para interagir com o cluster) e o próprio `minikube`.

6.4.1 Instalando o Driver: Docker

O Minikube precisa de um ambiente para criar seu "nó"de cluster. A opção mais comum é usar o Docker. Se você ainda não o instalou (conforme o Capítulo 5), o processo envolve adicionar o repositório oficial do Docker:

```
# Adiciona o repositório Docker (comandos de curl e gpg
   omitidos por brevidade)
$ echo \
  "deb [arch=$(dpkg --print-architecture)
     signed-by=/etc/apt/keyrings/docker.asc]
     https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt-get update
```

Com o repositório pronto, instalamos o Docker Engine:

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
$ docker --version
```

Configurando o Driver para o Systemd

Para garantir que o Minikube e o Docker interajam corretamente no gerenciamento de recursos (cgroups), é crucial configurar o `containerd` (o runtime de baixo nível do Docker) para usar o `systemd` como seu driver de cgroup.

Geramos o arquivo de configuração padrão do `containerd` e, em seguida, usamos o `sed` para alterar a diretiva `SystemdCgroup` de `false` para `true`:

```
$ containerd config default | sudo tee
   /etc/containerd/config.toml >/dev/null 2>&1
$ sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \=
   true/g' /etc/containerd/config.toml
```

Finalmente, reiniciamos e habilitamos o serviço `containerd` para aplicar a mudança:

```
$ sudo systemctl restart containerd
$ sudo systemctl enable containerd
```

6.4.2 Instalando Minikube e Kubectl

O `kubectl` é a CLI (Command Line Interface) universal para interagir com *qualquer* cluster Kubernetes, seja ele local (Minikube) ou na nuvem. O `minikube` é o executável que *cria* o cluster local.

Instalamos o `kubectl` usando `snap` ou `curl`, e o `minikube` baixando seu pacote `.deb`:

```
$ curl -LO
   https://storage.googleapis.com/minikube/releases/latest/minikube_latest_a
$ sudo dpkg -i minikube_latest_amd64.deb
$ minikube version
$ sudo snap install kubectl --classic
$ kubectl version --client
```

6.4.3 Preparando o Host para o Kubernetes

O Kubernetes tem requisitos estritos sobre o ambiente do host, principalmente em relação à memória e rede.

Desativando a SWAP

O Kubernetes espera que os recursos de memória sejam previsíveis. A SWAP (memória de troca em disco) interfere no agendador (*scheduler*), que precisa saber exatamente quanta memória um nó possui. Se a SWAP estiver ativa, o agendador pode alocar um *Pod* (unidade de trabalho) em um nó que está com a memória física esgotada, levando a instabilidade.

Por isso, devemos desativá-la permanentemente:

```
$ sudo swapoff -a
# Comenta a linha da SWAP no /etc/fstab para desabilitar no boot
$ sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
$ sudo systemctl daemon-reload
```

Carregando Módulos do Kernel

O Kubernetes precisa de dois módulos do kernel para gerenciar a rede de contêineres e os sistemas de arquivos em camadas:

- `overlay`: Permite o sistema de arquivos em camadas usado pelas imagens de contêiner.
- `br_netfilter`: Permite que o tráfego de rede entre Pods seja filtrado e roteado corretamente pelas regras do `iptables`.

Carregamos esses módulos e os tornamos permanentes no boot:

```
$ sudo tee /etc/modules-load.d/containerd.conf <<EOF
overlay
```

```
br_netfilter
EOF

$ sudo modprobe overlay
$ sudo modprobe br_netfilter
```

Também habilitamos o encaminhamento de IP para que a rede do cluster funcione:

```
$ sudo tee /etc/sysctl.d/kubernetes.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
$ sudo sysctl --system
```

6.4.4 Iniciando o Cluster Minikube

Com o ambiente preparado, iniciar o cluster é um único comando. Podemos solicitar múltiplos nós (`--nodes=2`) para simular um ambiente mais realista com um control-plane e um worker:

```
$ minikube start --nodes=2
minikube v1.36.0 on Debian 12.11 (arm64)
Using the docker driver based on user configuration
Starting "minikube" primary control-plane node in "minikube"
   cluster
Creating docker container (CPUs=2, Memory=1975MB) ...
Preparing Kubernetes v1.33.1 on Docker 28.1.1 ...
Enabled addons: default-storageclass, storage-provisioner
Starting "minikube-m02" worker node in "minikube" cluster
Creating docker container (CPUs=2, Memory=1975MB) ...
Done! kubectl is now configured to use "minikube" cluster...
```

Após alguns instantes, o Minikube configura o `kubectl` automaticamente. Podemos verificar o status do nosso cluster:

```
$ kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE   VERSION
   INTERNAL-IP    OS-IMAGE
minikube       Ready    control-plane   115s  v1.33.1
   192.168.49.2   Ubuntu 22.04
minikube-m02   Ready    <none>          93s   v1.33.1
   192.168.49.3   Ubuntu 22.04
```

Nosso cluster de dois nós está pronto para ser usado.

6.4.5 Interagindo com o Cluster e Serviços

Com nosso cluster Minikube em execução, podemos começar a interagir com ele e implantar aplicações.

6.4.6 Dashboard e Métricas

O Kubernetes oferece um Dashboard gráfico (Web UI) para inspecionar o cluster. O Minikube o fornece como um "addon". Para que o dashboard mostre informações de uso (CPU/Memória), precisamos habilitar também o `metrics-server`:

```
$ minikube addons enable metrics-server
The 'metrics-server' addon is enabled

$ minikube dashboard
Launching proxy ...
Opening
   http://127.0.0.1:37853/api/v1/namespaces/kubernetes-dashboard...
```

O comando `minikube dashboard` inicia um proxy e abre a interface no navegador. Como estamos em um servidor, podemos usar um túnel SSH (conforme visto em laboratórios anteriores) para acessar essa porta `127.0.0.1` a partir da nossa máquina física.

6.5 Namespaces

Antes de implantar aplicações, devemos introduzir o conceito de **Namespaces**. Em vez de lançar todos os nossos recursos (Pods, Serviços, etc.) no namespace `default`, uma boa prática é criar um namespace separado para cada aplicação ou projeto.

Namespaces fornecem:

- **Isolamento lógico**: Recursos com o mesmo nome podem existir em namespaces diferentes.
- **Controle de Acesso (RBAC)**: Podemos definir permissões por namespace (ex: Time A só acessa o namespace `dev`).
- **Gerenciamento de Recursos**: É possível definir cotas de CPU, memória e storage por namespace.

Criamos um namespace com `kubectl create namespace` e podemos definir nosso contexto `kubectl` para atuar dentro dele:

```
$ kubectl create namespace nextcloud
namespace/nextcloud created

# Alterando para um ns específico
$ kubectl config set-context --current --namespace=nextcloud
```

6.6 Instanciando Serviços: WordPress

Vamos implantar uma aplicação WordPress completa. No Kubernetes, não criamos "Pods"diretamente. Nós definimos *objetos* de nível superior, como **Deployments**, **Services** e **PersistentVolumeClaims**, e o Control Plane se encarrega de criar os Pods para nós.

A forma mais comum de fazer isso é através de um arquivo de manifesto YAML, que descreve o estado final desejado.

Primeiro, criamos um namespace para o projeto:

```
$ kubectl create namespace wordpress
```

Em seguida, criamos um único arquivo `wp-mysql.yml` que define todos os recursos necessários:

```
# wp-mysql.yml
apiVersion: v1
kind: Namespace
metadata:
  name: wordpress
---
apiVersion: v1
kind: Secret
metadata:
  name: mysql-pass
  namespace: wordpress
type: Opaque
stringData:
  # A senha real é definida aqui (o K8s fará o encode base64
     automaticamente com stringData)
  password: senha-super-secreta
---
# Serviço para o MySQL (ClusterIP - Interno)
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: wordpress
spec:
  ports:
```

```
  - port: 3306
  selector:
    app: mysql
  clusterIP: None # Headless service é comum para DBs, mas
     ClusterIP normal funciona
---
# PVC para o Banco de Dados
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  namespace: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
# Deployment do MySQL
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: wordpress
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7 # Versão estável para WP
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
```

```
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
---
# Serviço para o WordPress (NodePort ou LoadBalancer)
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  namespace: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
  type: LoadBalancer # No Minikube, isso requer 'minikube
     tunnel' ou apenas NodePort
---
# PVC para arquivos do WordPress
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  namespace: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
# Deployment do WordPress
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  namespace: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
  template:
```

```
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - image: wordpress:latest
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: mysql # Nome do Serviço do MySQL definido acima
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

Com o arquivo pronto, aplicamos o manifesto ao cluster:

```
$ kubectl apply -f wp-mysql.yml
namespace/wordpress created
persistentvolumeclaim/wordpress-pvc created
secret/mysql-pass created
deployment.apps/mysql created
service/mysql created
...
```

O Kubernetes agora trabalhará para criar tudo isso. Podemos verificar o status dos Pods (as unidades de execução) dentro do namespace `wordpress`:

```
$ kubectl get pods -n wordpress
NAME                         READY   STATUS    RESTARTS   AGE
mysql-65d8c54c47-abcde       1/1     Running   0          5m
wordpress-7f58f555d4-fghij   1/1     Running   0          5m
```

6.7 Acessando o Serviço via Minikube

O WordPress foi exposto através de um *Service*. O Minikube fornece um comando de atalho para expor este serviço em uma URL acessível:

```
$ minikube service wordpress -n wordpress --url
http://192.168.49.2:30080
```

Podemos então usar este IP e porta (novamente, com um túnel SSH se necessário) para acessar a tela de instalação do WordPress em nosso navegador, concluindo a implantação.

Capítulo 7

Introdução ao Terraform

7.1 O que é Terraform?

O **Terraform** é uma ferramenta para gerenciamento de infraestrutura de aplicações via descrição de código (**IaC**). Ela é mantida e criada pela empresa **HashiCorp** e apresenta uma documentação bem amigável com tutoriais de instalação em diversos provedores Cloud como AWS, Azure, Oracle, Docker e Google Cloud, por exemplo, além de tutoriais mais específicos da própria ferramenta, também trazendo casos de uso.

Com ela é possível instanciar componentes de baixo nível – **servidores, bancos de dados, balanceadores de carga e redes**–, bem como aqueles de alto nível, como **entradas de DNS, CDN, Serveless services, Simple Queue Service, Simple Notification Service, Monitoramento e Logs**, entre outras funcionalidades de SaaS.

Aqui vamos continuar seguindo a documentação para implementação de uma infraestrutura com **Docker** para **Linux**, mas caso queira, você pode seguir os passos para Windows ao longo do tutorial.

7.1.1 Vantagens

- **Gestão centralizada** da infraestrutura em diversos provedores de plataformas na nuvem (**Cloud**) via arquivos de configuração.
- **Linguagem declarativa** e de alto nível para escrita rápida da infraestrutura.
- **Controle dos estados** permite acompanhar as alterações dos recursos ao longo das implantações.

7.1.2 Ciclo de deploy

Para realizar o deploy com o terraform, vamos seguir as seguintes etapas:

- *Scope*: identificar a infraestrutura do projeto.
- *Author*: escrever a configuração que define a infraestrutura.
- *Inicialize*: instalar os provedores necessários.
- *Plan*: visualizar as mudanças que o Terraform vai fazer.
- *Apply*: aplicar as mudanças na infraestrutura.

7.1.3 Arquivos de configurações e suas funções

- `main.tf`: arquivo de configuração de infraestrutura da aplicação.
- `terraform.tfstate`: responsável por guardar o **estado** das alterações ao longo do tempo, contendo mais detalhes sobre os recursos. Deve ser armazenado com cuidado por conter **informações sensíveis** como IDs, *hashs* e outros atributos dos recursos.

7.2 Instalação

Para instalar a ferramenta, consulte a documentação no site oficial da HashiCorp e escolha o tutorial de acordo com a sua máquina. Verifique se a instalação foi bem sucedida com `terraform -version` ou dê uma olhada nos comandos da ferramenta com `terraform -help`. Caso queira saber mais de um determinado comando basta incluí-lo no comando: `terraform plan -help`. Você pode habilitar o *auto-complete* de comandos com `terraform -install-autocomplete`.

7.3 Build

Cada arquivo de configuração do terraform deve estar organizado em um diretório de trabalho específico. Vamos

```
mkdir build-nginx && cd build-nginx
```

```
touch main.tf
```

Adicione a configuração como no arquivo `main.tf` e depois inicialize o *deploy* com `terraform init`.

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
```

```
      version = "~> 3.0.1"
    }
  }
}

provider "docker" {}

resource "docker_image" "nginx" {
  name         = "nginx:latest"
  keep_locally = false
}

resource "docker_container" "nginx" {
  image = docker_image.nginx.image_id
  name  = "tutorial"
  ports {
    internal = 80
    external = 8000
  }
}
```

Aqui o terraform vai baixar o **docker** e instalar em um subdiretório escondido chamado `.terraform`. Ele também vai criar um arquivo de "trava"especificando a versão e o provedor exato que foi utilizado. Não é recomendado realizar alterações manuais nele, pois pode resultar em perdas futuras.

7.3.1 Criando a infraestrutura

Ao executar `terraform apply`, o terraform vai mostrar o planejamento a ser executado descrevendo as ações a serem tomadas para subir a infraestrutura. Ele vai esperar você aprovar a aplicação e dentro de alguns segundos você terá seu `nginx` ativo em `http://localhost:8000`. Você pode verificar o estado atual da infraestrutura com `terraform show`.

7.4 Fazendo alterações na infraestrutura

No arquivo `main.tf` altere a porta externa de `8000` para `8888`. Em seguida execute `terraform apply` como anteriormente e você verá que ele vai mostrar as alterações semelhante ao `git`. Verifique em `http://localhost:8888`.

7.4.1 Destruindo recursos

Para destruir recursos, basta executar `terraform destroy`, o que executa exatamente o procedimento inverso do `terraform apply`.

7.4.2 Criando variáveis

Uma boa prática dessa ferramenta é criar um arquivo de variáveis `variables.tf` para configurar os nomes de uma forma **flexível e segura**. Aqui vamos criar uma variável para o nome do container.

```
variable "container_name" {
  description = "Value of the name for the Docker container"
  type        = string
  default     = "Ngineco"
}
```

Em seguida, na `main.tf` adapte para o nome que deseja no recurso do *container* e altere o nome de `"tutorial"` para `var.container_name`.

```
resource "docker_container" "nginx" {
  image = docker_image.nginx.image_id
  name  = var.container_name
  ports {
    internal = 80
    external = 8888
  }
```

Aplique as alterações com `terraform apply`. Você também pode aplicar isso diretamente na CLI com a *flag* `-var "container_name=OutroNome"`.

7.4.3 Objetificando outputs

Crie um arquivo chamado `outputs.tf` e insira os blocos de id do *container* e da imagem, como está no arquivo. Aplique as alterações novamente com `terraform apply` e você verá os valores dos respectivos IDs. De forma alternativa você pode verificar com `terraform output`.

```
output "container_id" {
  description = "ID of the Docker container"
  value       = docker_container.nginx.id
}

output "image_id" {
  description = "ID of the Docker image"
  value       = docker_image.nginx.id
}
```

Como saída teremos algo semelhante ao seguinte prompt:

```
...
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

Outputs:

container_id =
   "e5fff27c62e04d21980543f21161225ab483a1e534a98311a677b9453a"
image_id =
   "sha256:d1a364dc548d5357f0da3268594f1d61c6fdeenginx:latest"
```

Entre os benefícios de utilizá-lo está a possibilidade de conectar os recursos de outros projetos a sua infraestrutura de modo a automatizar o *workflow* da sua aplicação.