Sumário

[Introdução aos Containers e Docker](#_Toc188361712)

[O que são Containers](#_Toc188361713)

[Principais Características dos Containers](#_Toc188361714)

[Containers vs. Máquinas Virtuais](#_Toc188361715)

[Docker como Container Runtime](#_Toc188361716)

[Funcionamento do Docker](#_Toc188361717)

[Observações sobre a Docker Inc.](#_Toc188361718)

[Open Container Initiative (OCI)](#_Toc188361719)

[Principais Objetivos da OCI](#_Toc188361720)

[Importância da OCI](#_Toc188361721)

[Conclusão](#_Toc188361722)

[Referências](#_Toc188361723)

[Manipulando Containers](#_Toc188361724)

[Executando o Primeiro Container (docker run hello-world)](#_Toc188361725)

[Nomeando Containers e Entendendo Diferentes Execuções](#_Toc188361726)

[Executando um Container com um Nome Personalizado](#_Toc188361727)

[Executando um Container em Segundo Plano (d)](#_Toc188361728)

[Mapeando Portas com a Flag p](#_Toc188361729)

[Parando, Iniciando e Removendo Containers de Forma Forçada](#_Toc188361730)

[Listando Containers em Execução e Parados](#_Toc188361731)

[Parando um Container](#_Toc188361732)

[Iniciando um Container Parado](#_Toc188361733)

[Removendo um Container](#_Toc188361734)

[Attach e Detach](#_Toc188361735)

[Conectando-se a um Container em Execução (docker attach)](#_Toc188361736)

[Saindo do Container sem Parar (CTRL + P, CTRL + Q)](#_Toc188361737)

[Executando Comandos e Removendo Containers Automaticamente](#_Toc188361738)

[Executando Comandos em um Novo Container](#_Toc188361739)

[Entrando no Container com Bash](#_Toc188361740)

[Diferença entre docker run e docker exec](#_Toc188361741)

[Removendo Containers Automaticamente (-rm)](#_Toc188361742)

[Removendo Todos os Containers com Subcomandos](#_Toc188361743)

[Publicação de Portas](#_Toc188361744)

[Execução Interativa e Acesso ao Shell](#_Toc188361745)

[Acessando o Shell de um Container com docker exec -it](#_Toc188361746)

[Diferença entre docker exec e docker attach](#_Toc188361747)

[Resumo dos Comandos](#_Toc188361748)

[Persistência de Dados e a Natureza Efêmera dos Containers](#_Toc188361749)

[Entendendo a Perda de Dados Após a Remoção do Container](#_Toc188361750)

[Camadas de Leitura e Escrita nos Containers](#_Toc188361751)

[Introdução ao OverlayFS e Funcionamento das Camadas](#_Toc188361752)

[Introdução à Persistência de Dados](#_Toc188361753)

[Por que Precisamos Persistir Dados?](#_Toc188361754)

[Conceitos de Volumes e Bind Mounts no Docker](#_Toc188361755)

[Utilizando Bind Mounts para Compartilhar Diretórios](#_Toc188361756)

[Configurando Bind Mounts com a Flag v ou -mount](#_Toc188361757)

[Usando a Flag v](#_Toc188361758)

[Usando a Flag -mount](#_Toc188361759)

[Exemplo Prático: Executando Nginx com Bind Mount](#_Toc188361760)

[Editando Arquivos no Host e Vendo as Alterações no Container](#_Toc188361761)

[Demonstrando a Efemeridade sem Bind Mounts](#_Toc188361762)

[Demonstrando a Persistência com Bind Mounts](#_Toc188361763)

[Dicas sobre Bind Mounts](#_Toc188361764)

[Gerenciando Volumes Docker](#_Toc188361765)

[Criando e Gerenciando Volumes com docker volume](#_Toc188361766)

[Criar um Volume](#_Toc188361767)

[Listar Volumes](#_Toc188361768)

[Inspecionar um Volume](#_Toc188361769)

[Principais Linhas do inspect:](#_Toc188361770)

[Remover um Volume](#_Toc188361771)

[Montando Volumes em Containers](#_Toc188361772)

[Comparação entre Volumes e Bind Mounts](#_Toc188361773)

[Volumes](#_Toc188361774)

[Bind Mounts](#_Toc188361775)

[Backup e Restauração de Volumes](#_Toc188361776)

[Backup de um Volume](#_Toc188361777)

[Restauração de um Volume](#_Toc188361778)

[Entendendo Imagens Docker](#_Toc188361779)

[O que São Imagens e Como Elas Funcionam](#_Toc188361780)

[Conceito de Camadas em Imagens Docker](#_Toc188361781)

[Imagens Oficiais vs. Imagens de Terceiros](#_Toc188361782)

[Gerenciamento de Imagens](#_Toc188361783)

[Listando Imagens Locais com docker images](#_Toc188361784)

[Removendo Imagens com docker rmi](#_Toc188361785)

[Inspecionando Imagens com docker inspect](#_Toc188361786)

[Trabalhando com Registries](#_Toc188361787)

[Introdução ao Docker Hub e Outros Registries](#_Toc188361788)

[Buscando e Fazendo Pull de Imagens com docker pull](#_Toc188361789)

[Diferença entre docker pull e Download Automático com docker run](#_Toc188361790)

[Comandos Úteis](#_Toc188361791)

[Uso de docker image prune para Limpeza de Recursos Não Utilizados](#_Toc188361792)

[Remoção de Múltiplas Imagens ao Mesmo Tempo](#_Toc188361793)

[Comandos Avançados para Gerenciamento Eficiente](#_Toc188361794)

[Filtrando Imagens](#_Toc188361795)

[Verificar o Espaço em Disco Usado pelo Docker](#_Toc188361796)

[Limpar Recursos Não Utilizados](#_Toc188361797)

[Docker Hub](#_Toc188361798)

[Recursos do Docker Hub](#_Toc188361799)

[Criação de Conta e Namespace](#_Toc188361800)

[Imagens Oficiais, Verificadas e de Comunidade](#_Toc188361801)

[Organização de Tags](#_Toc188361802)

[Aspectos de Segurança e SBOM](#_Toc188361803)

[Repositórios Públicos e Privados](#_Toc188361804)

[Automação de Builds](#_Toc188361805)

[Webhooks e Integração com CI/CD](#_Toc188361806)

[Visualização de Histórico de Builds e Tags](#_Toc188361807)

[Introdução ao Docker e Dockerfile](#_Toc188361808)

[Estrutura Básica de um Dockerfile](#_Toc188361809)

[Instruções Detalhadas do Dockerfile](#_Toc188361810)

[FROM e ARG](#_Toc188361811)

[Estratégia para Reaproveitamento de Cache:](#_Toc188361812)

[Uso do .dockerignore](#_Toc188361813)

[CMD e ENTRYPOINT](#_Toc188361814)

[USER e Segurança](#_Toc188361815)

[HEALTHCHECK](#_Toc188361816)

[EXPOSE](#_Toc188361817)

[VOLUME](#_Toc188361818)

[LABEL](#_Toc188361819)

[Multi-stage Builds e Otimização de Imagens](#_Toc188361820)

[ONBUILD e Imagens Base Personalizadas](#_Toc188361821)

[Passando Variáveis e Argumentos no docker run](#_Toc188361822)

[Passando Variáveis de Ambiente com e](#_Toc188361823)

[Passando Argumentos para o CMD ou ENTRYPOINT](#_Toc188361824)

[Sobrescrevendo o ENTRYPOINT com -entrypoint](#_Toc188361825)

[Especificando o Usuário com u](#_Toc188361826)

[Exemplos Práticos Passo a Passo](#_Toc188361827)

[Aplicação Node.js](#_Toc188361828)

[Aplicação Go](#_Toc188361829)

[Resumo e Melhores Práticas](#_Toc188361830)

[Introdução ao Docker Buildx](#_Toc188361831)

[Conceitos Fundamentais](#_Toc188361832)

[Builders e Contextos](#_Toc188361833)

[Drivers: docker vs docker-container](#_Toc188361834)

[Criando e Gerenciando Builders](#_Toc188361835)

[Criando um Builder com docker-container](#_Toc188361836)

[Inicializando o Builder com -bootstrap](#_Toc188361837)

[Listando e Selecionando Builders](#_Toc188361838)

[Removendo Builders e Contextos](#_Toc188361839)

[Executando Builds Multiplataforma](#_Toc188361840)

[Usando qemu para Emulação de Arquitetura](#_Toc188361841)

[Builds Multiplataforma com docker-container](#_Toc188361842)

[Comparação com o Driver docker](#_Toc188361843)

[Gerenciamento de Cache no Buildx](#_Toc188361844)

[Cache Local](#_Toc188361845)

[Cache Remoto em Registries](#_Toc188361846)

[Limpeza de Cache com buildx prune](#_Toc188361847)

[Exemplos Práticos Passo a Passo](#_Toc188361848)

[Introdução ao Docker Networking](#_Toc188361849)

[Conceitos Básicos de Rede no Docker](#_Toc188361850)

[Tipos de Redes no Docker](#_Toc188361851)

[Como os Containers se Conectam às Redes](#_Toc188361852)

[Comunicação entre Containers na Mesma Rede](#_Toc188361853)

[Comunicação entre Containers e o Host](#_Toc188361854)

[Utilizando host.docker.internal](#_Toc188361855)

[Gerenciando Múltiplas Redes e Isolamento](#_Toc188361856)

[Introdução ao Docker Compose](#_Toc188361857)

[Criando os Primeiros Serviços](#_Toc188361858)

[Serviço Node.js](#_Toc188361859)

[Serviço Nginx](#_Toc188361860)

[Definindo Dependências e Healthchecks](#_Toc188361861)

[Trabalhando com Volumes e Bind Mounts](#_Toc188361862)

[Persistindo Dados com Volumes](#_Toc188361863)

[Variáveis de Ambiente e env\_file](#_Toc188361864)

[Gerenciando Redes no Docker Compose](#_Toc188361865)

[Criando Redes Personalizadas](#_Toc188361866)

[Definindo Redes no docker-compose.yml](#_Toc188361867)

[Conectando Serviços às Redes](#_Toc188361868)

[Utilizando extra\_hosts para Acessar o Host](#_Toc188361869)

[Atualizando o Serviço Node.js com extra\_hosts](#_Toc188361870)

[Introdução ao Compose Watch](#_Toc188361871)

[Configurando o compose-watch](#_Toc188361872)

[Adicionando o Serviço nodeapp-watch](#_Toc188361873)

[Utilizando o Compose Watch](#_Toc188361874)

[O que é o docker init](#_Toc188361875)

[Projeto Go com docker init](#_Toc188361876)

[Análise detalhada do Dockerfile Go](#_Toc188361877)

[Análise do compose.yaml Go](#_Toc188361878)

[Projeto Node.js com docker init](#_Toc188361879)

[Análise detalhada do Dockerfile Node.js](#_Toc188361880)

[Análise do compose.yaml Node.js](#_Toc188361881)

[Projeto de Propósito Geral com docker init](#_Toc188361882)

[Análise detalhada do Dockerfile Propósito Geral](#_Toc188361883)

[Entendendo o Uso de Secrets no docker-compose.yml](#_Toc188361884)

[Uso de Secrets no Docker Compose:](#_Toc188361885)

[Segurança e Criptografia de Secrets:](#_Toc188361886)

# Introdução aos Containers e Docker

## O que são Containers

Um **container** é uma unidade padronizada de software que empacota o código e todas as suas dependências, permitindo que uma aplicação seja executada de forma rápida e consistente em qualquer ambiente. Isso significa que, independentemente do sistema operacional ou das configurações específicas do host, o container fornecerá um ambiente isolado e consistente para a aplicação.

## Principais Características dos Containers

* **Imutabilidade**: Containers são construídos a partir de imagens que são essencialmente snapshots imutáveis. Isso garante que o ambiente de execução seja o mesmo em qualquer lugar que o container seja implantado.
* **Isolamento de Processos e Recursos Computacionais**: Cada container opera em um espaço de usuário isolado, com seu próprio sistema de arquivos, rede e recursos de processamento. Isso permite que múltiplos containers sejam executados simultaneamente sem interferência mútua.
* **Leveza**: Ao contrário das máquinas virtuais, os containers são executados como processos no sistema operacional do host, sem a necessidade de um sistema operacional completo para cada instância. Isso resulta em menor uso de recursos e maior eficiência.
* **Utilização dos Recursos do Kernel do SO**: Containers compartilham o kernel do sistema operacional do host. Eles não necessitam de um sistema operacional completo em cada instância, o que reduz significativamente o overhead.
* **Ilusão de SO Próprio**: Através de tecnologias como namespaces e cgroups, o container é "enganado" para acreditar que possui seu próprio sistema operacional completo, com recursos e processos independentes.
* **Visibilidade Limitada de Processos**: Um container só pode ver e interagir com processos dentro do seu próprio espaço. Isso aumenta a segurança e o isolamento entre aplicações.
* **Inicialização Rápida**: Como não há necessidade de inicializar um sistema operacional completo, os containers podem ser iniciados e parados quase instantaneamente, facilitando a escalabilidade e a implantação contínua.
* **Utilização de Imagens Imutáveis**: Containers são criados a partir de imagens que definem o estado inicial do sistema de arquivos do container. Essas imagens são imutáveis e podem ser versionadas, garantindo consistência entre ambientes.
* **Predominância no Linux**: Embora existam implementações de containers em outros sistemas operacionais, a maioria das tecnologias de containerização, incluindo o Docker, foi inicialmente desenvolvida para o Linux, aproveitando recursos nativos do kernel.
* **"Na Minha Máquina Funciona"**: Com containers, o clássico problema de discrepâncias entre ambientes de desenvolvimento e produção é mitigado, já que o ambiente dentro do container é o mesmo em qualquer máquina.

## Containers vs. Máquinas Virtuais

**Máquinas Virtuais (VMs)**

* **Sistema Operacional Completo**: Cada VM executa um sistema operacional completo, incluindo seu próprio kernel, sobre um hypervisor que gerencia os recursos do hardware físico.
* **Overhead Maior**: Devido à necessidade de virtualizar todo o hardware e executar um SO completo, as VMs consomem mais recursos de memória e CPU.
* **Tempo de Inicialização Mais Longo**: Inicializar uma VM é similar a ligar um computador físico, o que pode levar minutos.
* **Isolamento Forte**: Oferecem um alto nível de isolamento, já que cada VM é completamente separada das demais e do host.

**Containers**

* **Compartilhamento do Kernel do Host**: Containers utilizam o kernel do sistema operacional do host, isolando apenas o aplicativo e suas dependências.
* **Leveza e Eficiência**: Sem a necessidade de virtualizar o hardware ou executar um SO completo, os containers são significativamente mais leves e eficientes.
* **Inicialização Instantânea**: Containers podem ser iniciados em segundos ou até milissegundos, permitindo rápida escalabilidade.
* **Isolamento de Aplicativos**: Embora compartilhem o kernel do host, os containers ainda oferecem isolamento suficiente para a maioria das aplicações, graças a tecnologias como namespaces e cgroups.
* **Alta Densidade**: Devido à sua leveza, é possível executar um grande número de containers em um único host, maximizando o uso dos recursos disponíveis.

## Docker como Container Runtime

**História do Docker**

* **Origem**: O Docker foi criado em 2013 por uma empresa chamada dotCloud, que oferecia serviços de Plataforma como Serviço (PaaS). Inicialmente, o Docker foi desenvolvido para uso interno, visando facilitar a implantação de aplicações.
* **Open Source**: Reconhecendo o potencial da tecnologia, a dotCloud tornou o Docker Engine open source, permitindo que a comunidade contribuísse e expandisse suas capacidades.
* **Evolução para Docker Inc.**: Com a crescente popularidade do Docker, a dotCloud mudou seu foco principal para o desenvolvimento e suporte da tecnologia Docker, renomeando-se para Docker Inc.

**Docker Engine, Docker CE e Docker Desktop**

* **Docker Engine**: É o coração do Docker, responsável por criar, gerenciar e executar containers. Ele é composto pelo daemon Docker (dockerd) e pela interface de linha de comando (CLI).
* **Docker CE (Community Edition)**: É a versão gratuita e open source do Docker Engine, destinada a desenvolvedores e pequenas equipes.
* **Docker Desktop**: É um produto da Docker Inc. que fornece um ambiente completo de desenvolvimento Docker para macOS e Windows. Inclui o Docker Engine, Docker CLI, Docker Compose e outras ferramentas.
* **Diferença entre Docker CE/Engine e Docker Desktop**: Enquanto o Docker CE/Engine é a tecnologia base para criação e execução de containers, o Docker Desktop oferece uma interface integrada e fácil de usar para desenvolvedores em ambientes não Linux, além de incluir recursos adicionais.

## Funcionamento do Docker

* **Utilização de Recursos do Linux**: O Docker aproveita funcionalidades nativas do kernel Linux, como namespaces e cgroups, para fornecer isolamento e gerenciamento de recursos aos containers.
* **Ferramenta 360**: Além de gerenciar containers, o Docker oferece funcionalidades para gerenciamento de redes, volumes (discos) e criação de imagens, tornando-se uma solução completa para ambientes containerizados.
* **Arquitetura Cliente-Servidor**:
  + **Daemon Docker**: O dockerd é o daemon que executa em segundo plano, gerenciando os containers, imagens, redes e volumes.
  + **Cliente Docker**: A interface de linha de comando (CLI) que os usuários interagem para enviar comandos ao daemon.
* **Ponto Único de Falha (SPoF)**: Como o daemon Docker gerencia todos os containers, se ele falhar, todos os containers no host podem ser afetados. Isso torna o daemon um ponto único de falha que deve ser considerado em ambientes críticos.
* **Root vs. Rootless**:
  + **Modo Root**: Por padrão, o daemon Docker requer privilégios de root para executar, o que pode representar riscos de segurança.
  + **Modo Rootless**: O Docker também pode ser executado em modo rootless, permitindo que usuários não privilegiados executem containers sem acesso root, aumentando a segurança.

## Observações sobre a Docker Inc.

* **Contribuição para a Comunidade**: A Docker Inc. tem sido fundamental na popularização dos containers e continua contribuindo com projetos open source.
* **Modelos de Negócio**: Além das versões gratuitas, a Docker Inc. oferece soluções comerciais com suporte empresarial e recursos avançados.
* **Ecosistema Rico**: A empresa mantém o Docker Hub, um repositório público de imagens Docker, e desenvolve ferramentas adicionais que complementam o Docker Engine.

## Open Container Initiative (OCI)

A **Open Container Initiative (OCI)** é um projeto colaborativo aberto, fundado em 2015 pela Docker Inc. e outras empresas líderes na indústria de tecnologia. O objetivo principal da OCI é criar padrões abertos para formatos de containers e runtimes, garantindo a portabilidade e interoperabilidade entre diferentes plataformas e ferramentas.

## Principais Objetivos da OCI

* **Estabelecer Padrões**: Definir especificações padronizadas para formatos de imagens de containers e runtimes, promovendo um ecossistema unificado.
* **Interoperabilidade**: Permitir que containers criados em uma plataforma possam ser executados em outra sem modificações significativas.
* **Neutralidade**: Garantir que os padrões não sejam controlados por uma única entidade, mas sim pela comunidade e pela indústria como um todo.

## Importância da OCI

* **Evitar Lock-in**: Com padrões abertos, desenvolvedores e empresas não ficam presos a uma única ferramenta ou fornecedor.
* **Facilitar a Inovação**: Ao estabelecer bases comuns, a OCI permite que diferentes projetos e ferramentas inovem em cima dos padrões, beneficiando todo o ecossistema.
* **Adoção Ampla**: Muitas das principais tecnologias de containerização, incluindo o Docker e o Kubernetes, aderem aos padrões da OCI, promovendo compatibilidade.

## Conclusão

Os containers revolucionaram a forma como desenvolvemos, implantamos e executamos aplicações, oferecendo consistência, eficiência e escalabilidade. O Docker desempenhou um papel crucial nessa transformação, fornecendo ferramentas acessíveis e poderosas para trabalhar com containers. Com a padronização promovida pela Open Container Initiative, o futuro dos containers promete ser ainda mais integrado e interoperável, permitindo que desenvolvedores e empresas aproveitem ao máximo essa tecnologia em constante evolução.

## Referências

* **Docker Documentation**: <https://docs.docker.com/>
* **Open Container Initiative**: <https://opencontainers.org/>
* **Docker vs. Virtual Machines**: <https://www.docker.com/resources/what-container/>

# Manipulando Containers

## Executando o Primeiro Container (docker run hello-world)

Para verificar se o Docker está instalado corretamente, execute:

docker run hello-world

Este comando baixa a imagem hello-world (se ainda não estiver no seu sistema) e executa um container que exibe uma mensagem de confirmação.

## Nomeando Containers e Entendendo Diferentes Execuções

### Executando um Container com um Nome Personalizado

Por padrão, o Docker atribui nomes aleatórios aos containers. Você pode especificar um nome usando a flag --name:

docker run --name mynginx nginx

### Executando um Container em Segundo Plano (d)

Para executar um container em modo "detached" (segundo plano), use a flag -d:

docker run -d --name mynginx nginx

### Mapeando Portas com a Flag p

Para mapear a porta do container para a porta do host, use -p:

docker run -d -p 8080:80 nginx

Isso mapeia a porta 80 do container para a porta 8080 do host.

## Parando, Iniciando e Removendo Containers de Forma Forçada

### Listando Containers em Execução e Parados

* **Containers em execução:**
  + docker ps
* **Todos os containers (incluindo parados):**
  + docker ps -a

*Diferença:* docker ps lista apenas os containers em execução, enquanto docker ps -a lista todos os containers existentes no sistema.

### Parando um Container

docker stop mynginx

### Iniciando um Container Parado

docker start mynginx

### Removendo um Container

* **Remoção normal:**

docker rm mynginx

* **Remoção forçada (para containers em execução):**

docker rm -f mynginx

*Diferença:* docker rm remove apenas containers parados. Para remover um container em execução, use docker rm -f.

## Attach e Detach

### Conectando-se a um Container em Execução (docker attach)

docker attach mynginx

Este comando conecta seu terminal ao processo principal do container.

### Saindo do Container sem Parar (CTRL + P, CTRL + Q)

Para sair do modo attach sem parar o container, pressione CTRL + P seguido de CTRL + Q.

## Executando Comandos e Removendo Containers Automaticamente

### Executando Comandos em um Novo Container

Você pode executar um comando diretamente em um novo container:

docker run nginx ls -la

Isto executa ls -la no container nginx e exibe o resultado no seu terminal.

### Entrando no Container com Bash

Para acessar o shell bash dentro de um container:

docker run -it nginx bash

Isto inicia um container nginx e abre uma sessão interativa do bash.

### Diferença entre docker run e docker exec

* **docker run:** Cria e inicia um novo container.
* **docker exec:** Executa um comando em um container já em execução.

Exemplo com docker exec:

docker exec -it mynginx bash

Isto abre uma sessão bash em um container mynginx já em execução.

### Removendo Containers Automaticamente (-rm)

Para remover automaticamente um container após sua execução:

docker run --rm nginx ls -la

## Removendo Todos os Containers com Subcomandos

Para remover todos os containers parados:

docker rm $(docker ps -a -q)

* **Explicação:**
  + docker ps -a -q lista todos os IDs de containers.
  + $(...) insere essa lista no comando docker rm.

Para remover todos os containers, incluindo os em execução, use:

docker rm -f $(docker ps -a -q)

## Publicação de Portas

Para executar um servidor Nginx em um container e publicar a porta:

docker run -d -p 8080:80 nginx

Agora, o Nginx está acessível em http://localhost:8080.

## Execução Interativa e Acesso ao Shell

### Acessando o Shell de um Container com docker exec -it

Se você já tem um container em execução e deseja acessar seu shell:

docker exec -it mynginx bash

### Diferença entre docker exec e docker attach

* **docker exec:** Executa um novo processo dentro de um container em execução (ex.: abrir uma nova sessão bash).
* **docker attach:** Anexa seu terminal ao processo principal do container (ex.: ver logs em tempo real).

### Resumo dos Comandos

* **Executar um container:**
  + docker run [opções] imagem [comando]
* **Listar containers:**
  + docker ps # Em execução
  + docker ps -a # Todos
* **Parar, iniciar e remover containers:**
  + docker stop <nome|id>
  + docker start <nome|id>
  + docker rm <nome|id>
  + docker rm -f <nome|id> # Forçado
* **Executar comandos em containers:**
  + docker exec -it <nome|id> <comando>
* **Acessar shell bash:**
  + docker exec -it <nome|id> bash
* **Remover todos os containers:**
  + docker rm $(docker ps -a -q)
  + docker rm -f $(docker ps -a -q) # Incluindo em execução

# Persistência de Dados e a Natureza Efêmera dos Containers

## Entendendo a Perda de Dados Após a Remoção do Container

Por padrão, tudo o que acontece dentro de um container Docker é efêmero. Ao remover um container, todos os dados e alterações feitas dentro dele são perdidos. Isso é útil para manter a consistência e portabilidade, mas pode ser problemático quando precisamos persistir dados, como em bancos de dados ou aplicações que armazenam informações de usuários.

### Camadas de Leitura e Escrita nos Containers

Os containers Docker utilizam um sistema de arquivos em camadas, conhecido como **OverlayFS**. Quando um container é iniciado a partir de uma imagem, o Docker cria uma camada de leitura e escrita sobre as camadas somente leitura da imagem. Todas as alterações feitas no container são gravadas nessa camada superior.

### Introdução ao OverlayFS e Funcionamento das Camadas

* **OverlayFS:** É um sistema de arquivos unificador que permite sobrepor múltiplos sistemas de arquivos.
* **Camadas de Imagem:** São camadas somente leitura que compõem a imagem Docker.
* **Camada de Container:** É a camada de leitura e escrita criada quando o container é iniciado.

Esse sistema permite que múltiplos containers compartilhem a mesma imagem base sem interferir uns nos outros, economizando espaço e recursos.

## Introdução à Persistência de Dados

### Por que Precisamos Persistir Dados?

Em muitos casos, precisamos que os dados sobrevivam além do ciclo de vida de um container. Por exemplo:

* Bancos de dados que armazenam informações críticas.
* Aplicações que geram logs importantes.
* Sites que permitem uploads de arquivos.

### Conceitos de Volumes e Bind Mounts no Docker

* **Volumes:**
  + Gerenciados pelo Docker.
  + Armazenados em um local específico no sistema de arquivos do Docker.
  + Podem ser locais ou remotos (usando drivers de volume).
  + São a maneira recomendada para persistir dados em produção.
* **Bind Mounts:**
  + Montam um diretório ou arquivo do sistema de arquivos do host no container.
  + Oferecem mais flexibilidade durante o desenvolvimento.
  + Dependem da estrutura do host, o que pode afetar a portabilidade.

## Utilizando Bind Mounts para Compartilhar Diretórios

### Configurando Bind Mounts com a Flag v ou -mount

## Usando a Flag v

A sintaxe básica para um bind mount com -v é:

docker run -v [caminho\_do\_host]:[caminho\_do\_container] [imagem]

## Usando a Flag -mount

A sintaxe com --mount é mais explícita:

docker run --mount type=bind,source=[caminho\_do\_host],target=[caminho\_do\_container] [imagem]

### Exemplo Prático: Executando Nginx com Bind Mount

**Passo 1: Criar um Diretório no Host**

Crie um diretório para os arquivos do Nginx:

mkdir ~/my\_nginx\_html

**Passo 2: Criar um Arquivo index.html**

Crie um arquivo HTML simples:

echo "<h1>Hello Docker!</h1>" > ~/my\_nginx\_html/index.html

**Passo 3: Executar o Nginx com Bind Mount**

Usando -v e $(pwd) para obter o caminho atual:

docker run -d -p 8080:80 -v $(pwd)/my\_nginx\_html:/usr/share/nginx/html nginx

**Nota:** Certifique-se de estar no diretório onde o my\_nginx\_html está localizado ao usar $(pwd).

**Passo 4: Verificar no Navegador**

Acesse http://localhost:8080 e você deverá ver **"Hello Docker!"**.

## Editando Arquivos no Host e Vendo as Alterações no Container

**Passo 1: Alterar o index.html no Host**

Edite o arquivo:

echo "<h1>Content Updated!</h1>" > ~/my\_nginx\_html/index.html

**Passo 2: Atualizar o Navegador**

Recarregue a página em http://localhost:8080 e veja a alteração refletida.

## Demonstrando a Efemeridade sem Bind Mounts

**Passo 1: Remover o Container**

docker rm -f [container\_id ou nome]

**Passo 2: Recriar o Container sem Bind Mount**

docker run -d -p 8080:80 nginx

**Passo 3: Verificar se as Alterações Foram Perdidas**

Acesse http://localhost:8080 e note que as alterações feitas anteriormente não estão mais presentes.

## Demonstrando a Persistência com Bind Mounts

**Passo 1: Remover o Container**

docker rm -f [container\_id ou nome]

**Passo 2: Recriar o Container com Bind Mount**

docker run -d -p 8080:80 -v $(pwd)/my\_nginx\_html:/usr/share/nginx/html nginx

**Passo 3: Verificar a Persistência das Alterações**

Acesse http://localhost:8080 e confirme que o conteúdo alterado permanece.

### Dicas sobre Bind Mounts

* **Caminho Absoluto:** Sempre use caminhos absolutos ou $(pwd) para evitar problemas.
* **Permissões:** Certifique-se de que o Docker tem permissão para acessar o diretório.
* **Compatibilidade:** Os bind mounts dependem do sistema de arquivos do host, o que pode afetar a portabilidade entre diferentes sistemas.

## Gerenciando Volumes Docker

**Tipos de Volumes**

* **Volumes Locais:** Armazenados no sistema de arquivos do host, gerenciados pelo Docker.
* **Volumes Remotos:** Utilizam drivers de volume para armazenar dados em soluções de rede, como NFS, Azure File Storage, AWS EFS, etc.

## Criando e Gerenciando Volumes com docker volume

### Criar um Volume

docker volume create my\_volume

### Listar Volumes

docker volume ls

### Inspecionar um Volume

docker volume inspect my\_volume

### Principais Linhas do inspect:

* **"Mountpoint":** Local no sistema de arquivos onde o volume está armazenado.
* **"Driver":** O driver usado (geralmente "local").
* **"Labels":** Metadados que podem ser atribuídos ao volume.

Exemplo de saída:

[

{

"CreatedAt": "2023-10-10T12:34:56Z",

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/my\_volume/\_data",

"Name": "my\_volume",

"Options": {},

"Scope": "local"

}

]

### Remover um Volume

docker volume rm my\_volume

## Montando Volumes em Containers

**Passo 1: Executar Nginx com um Volume**

docker run -d -p 8081:80 -v my\_volume:/usr/share/nginx/html nginx

**Passo 2: Copiar Arquivos para o Volume**

Copie o arquivo index.html para o container:

docker cp ~/my\_nginx\_html/index.html [container\_id ou nome]:/usr/share/nginx/html

**Nota:** Como estamos usando um volume, o arquivo será persistido mesmo após a remoção do container.

**Passo 3: Verificar no Navegador**

Acesse http://localhost:8081 para ver o conteúdo.

**Passo 4: Remover e Recriar o Container**

docker rm -f [container\_id ou nome]

docker run -d -p 8081:80 -v my\_volume:/usr/share/nginx/html nginx

**Passo 5: Confirmar Persistência dos Dados**

Acesse novamente http://localhost:8081 e veja que o conteúdo permanece.

### Comparação entre Volumes e Bind Mounts

### Volumes

* **Gerenciados pelo Docker:** O Docker controla onde os dados são armazenados.
* **Independência do Host:** Não dependem da estrutura de diretórios do host.
* **Segurança:** Oferecem melhor isolamento, já que o host não acessa diretamente os dados.
* **Backup e Restauração:** Facilmente gerenciáveis para backup e migração.
* **Compatibilidade com Drivers de Volume:** Podem utilizar drivers para armazenar dados em locais remotos ou em rede.

### Bind Mounts

* **Dependência do Host:** Dependem da estrutura de diretórios e sistema de arquivos do host.
* **Flexibilidade:** Permitem acesso direto aos arquivos do host, útil em ambientes de desenvolvimento.
* **Risco de Segurança:** Maior risco se não configurados corretamente, pois dão ao container acesso direto ao sistema de arquivos do host.
* **Desempenho:** Podem ter desempenho variável dependendo do sistema de arquivos e configuração.

**Quando Usar Cada Um**

* **Volumes:**
  + **Produção:** Maior segurança e independência do host.
  + **Persistência de Dados Sensíveis:** Bancos de dados, dados de aplicação.
  + \*Quando se deseja usar drivers de volume para armazenamento em rede ou em nuvem.
* **Bind Mounts:**
  + **Desenvolvimento:** Facilita a edição de arquivos e atualização em tempo real.
  + **Casos Especiais:** Quando é necessário acessar arquivos específicos do host.

## Backup e Restauração de Volumes

Para garantir a persistência dos dados armazenados em volumes, é uma prática recomendada criar backups regulares, especialmente em ambientes de produção.

### Backup de um Volume

docker run --rm -v my\_volume:/data -v $(pwd):/backup busybox tar czf /backup/backup.tar.gz /data

* **docker run**: Executa um novo container.
* **-rm**: Remove o container automaticamente ao final do processo.
* **v my\_volume:/data**: Monta o volume my\_volume no caminho /data dentro do container.
* **v $(pwd):/backup**: Monta o diretório atual do host ($(pwd)) no caminho /backup dentro do container.
* **busybox**: Uma imagem de contêiner leve usada para executar comandos Unix básicos.
* **tar czf /backup/backup.tar.gz /data**:
  + Cria um arquivo compactado backup.tar.gz com o conteúdo do volume my\_volume e o armazena no diretório atual do host.

### Restauração de um Volume

docker run --rm -v my\_volume:/data -v $(pwd):/backup busybox tar xzf /backup/backup.tar.gz -C /

* **docker run**: Executa um novo container.
* **-rm**: Remove o container automaticamente ao final do processo.
* **v my\_volume:/data**: Monta o volume my\_volume no caminho /data dentro do container.
* **v $(pwd):/backup**: Monta o diretório atual do host, onde o arquivo de backup está localizado.
* **tar xzf /backup/backup.tar.gz -C /**:
  + Extrai o conteúdo de /backup/backup.tar.gz no diretório raiz do container.
  + O arquivo backup.tar.gz inclui o caminho /data, que restaura os dados diretamente no volume my\_volume.

# Entendendo Imagens Docker

## O que São Imagens e Como Elas Funcionam

Uma **imagem Docker** é um pacote imutável que contém tudo o que é necessário para executar uma aplicação: código, bibliotecas, dependências, ferramentas e configurações. As imagens são usadas para criar containers, que são instâncias em execução dessas imagens.

As imagens atuam como um modelo ou snapshot de um sistema de arquivos. Quando você executa um container, ele é iniciado a partir de uma imagem específica.

## Conceito de Camadas em Imagens Docker

As imagens Docker são construídas em camadas. Cada instrução em um Dockerfile (quando criamos imagens personalizadas) cria uma nova camada na imagem. Embora não estejamos cobrindo a criação de imagens personalizadas neste tutorial, é importante entender que as imagens são compostas por múltiplas camadas empilhadas.

Essas camadas são compartilhadas entre imagens, o que economiza espaço em disco e acelera downloads. Por exemplo, se duas imagens compartilham a mesma camada base, essa camada é baixada apenas uma vez.

### Imagens Oficiais vs. Imagens de Terceiros

* **Imagens Oficiais**: São mantidas pela equipe Docker ou pela comunidade e são consideradas confiáveis. Elas geralmente são encontradas no Docker Hub sem um nome de usuário, por exemplo, nginx, redis, mysql.
* **Imagens de Terceiros**: Criadas e mantidas por usuários individuais ou organizações. Elas podem ter personalizações específicas. O nome da imagem geralmente inclui o nome de usuário ou organização, por exemplo, usuario/imagem-personalizada.

# Gerenciamento de Imagens

## Listando Imagens Locais com docker images

Para ver todas as imagens Docker armazenadas localmente em seu sistema, use:

docker images

Este comando exibirá uma lista com:

* **REPOSITORY**: Nome do repositório ou imagem.
* **TAG**: Tag da imagem, que geralmente indica a versão.
* **IMAGE ID**: Identificador único da imagem.
* **CREATED**: Quando a imagem foi criada.
* **SIZE**: Tamanho da imagem.

## Removendo Imagens com docker rmi

Para remover uma imagem que você não precisa mais, use:

docker rmi [OPTIONS] IMAGE [IMAGE...]

Exemplo:

docker rmi nginx

**Nota**: Você não pode remover uma imagem que está sendo usada por um container em execução. Será necessário parar e remover o container primeiro.

## Inspecionando Imagens com docker inspect

Para obter detalhes sobre uma imagem específica, use:

docker inspect [OPTIONS] NAME|ID [NAME|ID...]

Exemplo:

docker inspect nginx

Isso retornará um objeto JSON com informações detalhadas sobre a imagem, como:

* **ID da Imagem**
* **Tags**
* **Camadas (Layers)**
* **Variáveis de Ambiente**
* **Comando de Entrada (Entrypoint)**
* **Volumes**

Você pode filtrar informações específicas usando a flag --format. Por exemplo, para obter apenas o ID da imagem:

docker inspect --format='{{.Id}}' nginx

# Trabalhando com Registries

## Introdução ao Docker Hub e Outros Registries

Um **registry** é um serviço onde as imagens Docker são armazenadas e distribuídas. O **Docker Hub** é o registry público oficial, mantido pela Docker, onde milhões de imagens estão disponíveis.

Outros registries populares incluem:

* **Amazon Elastic Container Registry (ECR)**
* **Google Container Registry (GCR)**
* **Azure Container Registry (ACR)**
* **Registries Privados**: Empresas podem hospedar seus próprios registries para imagens internas.

### Buscando e Fazendo Pull de Imagens com docker pull

Para buscar uma imagem no Docker Hub, você pode usar o comando docker search:

docker search ubuntu

Para baixar (fazer pull) de uma imagem, use:

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

Exemplo:

docker pull nginx

Isso baixará a imagem nginx com a tag latest (padrão). Para especificar uma tag:

docker pull nginx:1.21

## Diferença entre docker pull e Download Automático com docker run

* **docker pull**: Apenas baixa a imagem para o seu sistema local. Use quando você deseja apenas obter a imagem sem executá-la imediatamente.
* **docker run**: Se a imagem especificada não estiver presente localmente, o Docker automaticamente faz o pull da imagem antes de executá-la.

Exemplo:

docker run nginx

Se você ainda não tiver a imagem nginx localmente, o Docker fará o download e, em seguida, iniciará um container a partir dessa imagem.

# Comandos Úteis

## Uso de docker image prune para Limpeza de Recursos Não Utilizados

Com o tempo, você pode acumular imagens não utilizadas que ocupam espaço em disco. Para remover imagens dangling (imagens que não estão associadas a nenhuma tag):

docker image prune

Para remover **todas** as imagens não utilizadas (imagens não referenciadas por nenhum container), use:

docker image prune -a

**Atenção**: Use o parâmetro -a com cuidado, pois ele removerá todas as imagens não utilizadas, mesmo que você possa precisar delas no futuro.

## Remoção de Múltiplas Imagens ao Mesmo Tempo

Você pode remover várias imagens especificando seus IDs ou nomes:

docker rmi imagem1 imagem2 imagem3

Ou usando um comando para listar e remover:

docker rmi $(docker images -q)

Esse comando removerá todas as imagens listadas por docker images -q, que retorna apenas os IDs das imagens.

## Comandos Avançados para Gerenciamento Eficiente

### Filtrando Imagens

Você pode filtrar as imagens listadas com docker images usando a flag --filter ou -f.

Exemplo: Listar imagens antes de uma determinada data:

docker images --filter before=nginx:1.19

### Verificar o Espaço em Disco Usado pelo Docker

Para verificar quanto espaço em disco o Docker está utilizando:

docker system df

### Limpar Recursos Não Utilizados

Para liberar espaço removendo containers parados, imagens não utilizadas, volumes não utilizados e networks não referenciadas:

docker system prune

# Docker Hub

## Recursos do Docker Hub

O Docker Hub oferece vários recursos que facilitam o gerenciamento e distribuição de imagens:

### Criação de Conta e Namespace

* Qualquer pessoa pode criar uma conta gratuita no Docker Hub.
* Ao criar uma conta, você obtém um **namespace** exclusivo (geralmente seu nome de usuário).
* Esse namespace é usado como prefixo para todas as imagens que você criar e enviar para o Docker Hub.
* Exemplo: Se seu nome de usuário é joaosilva, suas imagens terão o formato joaosilva/nome-da-imagem.

### Imagens Oficiais, Verificadas e de Comunidade

* **Imagens Oficiais**: Mantidas pela equipe Docker ou pela comunidade, sem namespace no nome. Exemplo: nginx, mysql.
* **Imagens Verificadas**: Criadas por fornecedores de software e verificadas pelo Docker. Elas possuem um selo de verificação e garantias adicionais de segurança e suporte.
* **Imagens de Comunidade**: Criadas por usuários do Docker Hub. Podem ser úteis, mas é importante verificar a confiabilidade do mantenedor.

### Organização de Tags

* As imagens podem ter várias **tags** que geralmente representam diferentes versões ou variações.
* A tag padrão é latest, mas é recomendável especificar uma tag específica para garantir consistência.
* Exemplo: nginx:1.21 refere-se à versão 1.21 do Nginx.

### Aspectos de Segurança e SBOM

* **SBOM (Software Bill of Materials)**: Muitas imagens verificadas fornecem um SBOM, que é uma lista detalhada de todos os componentes e dependências incluídos na imagem.
* O SBOM ajuda na transparência e segurança, permitindo que os usuários conheçam exatamente o que está dentro da imagem.
* Imagens verificadas passam por verificações adicionais de segurança, incluindo varreduras de vulnerabilidades.

### Repositórios Públicos e Privados

* **Repositórios Públicos**: Qualquer pessoa pode acessar e baixar as imagens.
* **Repositórios Privados**: Acesso restrito; requer autenticação. Útil para armazenar imagens internas ou confidenciais.

### Automação de Builds

O Docker Hub pode ser integrado a repositórios de código (como GitHub ou Bitbucket) para automatizar o processo de build de imagens quando há novas alterações no código.

### Webhooks e Integração com CI/CD

Webhooks podem ser configurados para notificar serviços externos quando eventos ocorrem no Docker Hub, como quando uma nova imagem é enviada. Isso facilita a integração com pipelines de CI/CD.

### Visualização de Histórico de Builds e Tags

O Docker Hub mantém um histórico de builds e permite que você gerencie as diferentes tags de suas imagens, facilitando o controle de versões e atualizações.

# Introdução ao Docker e Dockerfile

O Docker é uma plataforma que permite empacotar, distribuir e executar aplicações em containers, garantindo que elas funcionem de maneira consistente em diferentes ambientes. Um **Dockerfile** é um script que contém uma série de instruções para construir uma imagem Docker. Ele define todos os componentes necessários para executar sua aplicação, desde a imagem base até as dependências e configurações.

## Estrutura Básica de um Dockerfile

Vamos começar analisando o Dockerfile fornecido para a aplicação Node.js:

ARG NODE\_VERSION=21.1.0

FROM node:${NODE\_VERSION}

RUN apt-get update \

&& apt-get install -y vim \

&& rm -rf /var/lib/apt/lists/\*

ENV PORT=3001

ENV MESSAGE="Hello Docker!"

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

HEALTHCHECK --interval=10s --timeout=5s --start-period=5s --retries=3 \

CMD [ "curl","-f","http://localhost:3001" ]

EXPOSE 3001

CMD ["node", "index.js"]

Este Dockerfile contém várias instruções importantes que exploraremos em detalhes a seguir.

## Instruções Detalhadas do Dockerfile

### FROM e ARG

* **FROM**: Especifica a imagem base a partir da qual sua imagem será construída.

FROM node:${NODE\_VERSION}

* **ARG**: Define variáveis de build que podem ser passadas durante a construção da imagem.

ARG NODE\_VERSION=21.1.0

**Explicação:**

* Estamos usando o **ARG** NODE\_VERSION para permitir a parametrização da versão do Node.js. Isso facilita a atualização ou alteração da versão sem modificar o Dockerfile.
* **Uso durante o build:**

docker build --build-arg NODE\_VERSION=16.13.0 -t my-node-app .

**RUN**

* Executa comandos durante a construção da imagem.

RUN apt-get update \

&& apt-get install -y vim \

&& rm -rf /var/lib/apt/lists/\*

**Explicação:**

* **apt-get update**: Atualiza a lista de pacotes disponíveis.
* **apt-get install -y vim**: Instala o editor vim. Isso pode ser útil para depuração dentro do container, mas em ambientes de produção, recomenda-se minimizar a instalação de pacotes adicionais para reduzir o tamanho da imagem.
* **rm -rf /var/lib/apt/lists/**: Remove os arquivos de lista de pacotes para reduzir o tamanho da imagem.

**ENV**

* Define variáveis de ambiente que estarão disponíveis no container em tempo de execução.

ENV PORT=3001

ENV MESSAGE="Hello Docker!"

**Explicação:**

* **PORT** e **MESSAGE** são usadas pela aplicação para configurar a porta de escuta e a mensagem a ser exibida.

**WORKDIR**

* Define o diretório de trabalho dentro do container.

WORKDIR /app

**Explicação:**

* Define /app como o diretório onde os comandos subsequentes serão executados, facilitando o gerenciamento dos arquivos da aplicação.

**COPY e Reaproveitamento de Cache**

* **COPY**: Copia arquivos do sistema host para o sistema de arquivos do container.

COPY package\*.json ./

RUN npm install

COPY . .

### Estratégia para Reaproveitamento de Cache:

* **Por que copiar primeiro o package\*.json antes do restante do código?**
  + Ao copiar apenas os arquivos de dependência (package.json e package-lock.json) e executar npm install, o Docker pode armazenar em cache esta camada.
  + Se o código da aplicação mudar, mas as dependências permanecerem as mesmas, o Docker reutilizará o cache desta camada, acelerando o processo de build.

**Exemplo:**

* **Etapa 1:** Copia os arquivos de dependência.

COPY package\*.json ./

* **Etapa 2:** Instala as dependências.

RUN npm install

* **Etapa 3**: Copia o restante do código.

COPY . .

## Uso do .dockerignore

Crie um arquivo .dockerignore para excluir arquivos e diretórios desnecessários do contexto de build.

**Exemplo de .dockerignore:**

node\_modules

npm-debug.log

Dockerfile

.dockerignore

.git

**Explicação:**

* **node\_modules**: Evita copiar dependências já instaladas, garantindo que elas sejam instaladas dentro do container.
* **Dockerfile e .dockerignore**: Não são necessários dentro da imagem.
* **.git**: Exclui os arquivos de controle de versão.

## CMD e ENTRYPOINT

**CMD**

* Define o comando padrão a ser executado quando um container é iniciado a partir da imagem.

CMD ["node", "index.js"]

**ENTRYPOINT**

* Define o executável padrão do container que não é facilmente sobrescrito.

**Diferença entre CMD e ENTRYPOINT:**

* **CMD** pode ser sobrescrito ao passar um comando no docker run.
* **ENTRYPOINT** é fixo e qualquer argumento passado será adicionado após o ENTRYPOINT.

**Exemplo na Aplicação Go:**

No Dockerfile da aplicação Go:

FROM scratch

USER 1001

COPY --from=builder /app/server /server

ENTRYPOINT ["./server"]

**Explicação:**

* **ENTRYPOINT ["./server"]**: Define o executável principal do container.
* Ao executar docker run my-go-app, o container executará ./server.
* Se passarmos argumentos, por exemplo docker run my-go-app 8080, o container executará ./server 8080

## USER e Segurança

* Define o usuário sob o qual os processos dentro do container serão executados.

RUN useradd -m mynode

RUN chown -R mynode /app

USER mynode

**Explicação:**

* **useradd**: Cria um novo usuário mynode.
* **chown**: Altera a propriedade dos arquivos para o usuário mynode.
* **USER mynode**: Define que os comandos subsequentes e o processo principal serão executados como mynode.
* **Nota:** Estas linhas estão comentadas no Dockerfile fornecido. Para melhorar a segurança, recomenda-se descomentá-las e usar um usuário não root.

## HEALTHCHECK

* Permite que o Docker verifique a saúde do container em intervalos regulares.

HEALTHCHECK --interval=10s --timeout=5s --start-period=5s --retries=3 \ CMD [ "curl","-f","http://localhost:3001" ]

**Explicação:**

* **-interval=10s**: Verifica a cada 10 segundos.
* **-timeout=5s**: Timeout de 5 segundos para cada tentativa.
* **-start-period=5s**: Aguarda 5 segundos antes de iniciar as verificações.
* **-retries=3**: Considera o container unhealthy após 3 falhas consecutivas.
* **CMD [ "curl","-f","**[**http://localhost:3001**](http://localhost:3001/)**" ]**: Executa o comando para verificar se a aplicação está respondendo.

**Correção:**

* Certifique-se de que o comando curl está disponível na imagem. Se estiver usando uma imagem mínima que não inclui curl, você pode usar o comando wget ou um script em sh.

## EXPOSE

* Documenta a porta em que a aplicação estará escutando.

EXPOSE 3001

**Explicação:**

* **EXPOSE** não publica a porta, apenas documenta. Para publicar a porta, você deve usar o p ou P ao executar o container.

## VOLUME

* Declara um volume que deve ser montado em um diretório específico.

VOLUME [“/data”]

**Explicação:**

* Esta linha está comentada. Se sua aplicação precisa persistir dados, você pode descomentá-la para criar um volume em /data.

## LABEL

* Adiciona metadados à imagem.

LABEL maintainer="Wesley Willians"

**Explicação:**

* Ajuda na documentação e manutenção da imagem, permitindo que informações sejam recuperadas com docker inspect.

## Multi-stage Builds e Otimização de Imagens

O Docker permite otimizar suas imagens utilizando multi-stage builds, o que ajuda a reduzir o tamanho da imagem final, especialmente para linguagens compiladas como Go.

**Dockerfile da Aplicação Go:**

FROM golang:latest AS builder

LABEL maintainer="Wesley Willians"

WORKDIR /app

COPY . .

RUN CGO\_ENABLED=0 go build -ldflags="-s -w" -o server main.go

Final stage

FROM scratch

USER 1001

COPY --from=builder /app/server /server

ENTRYPOINT ["./server"]

**Explicação:**

* **Primeiro Estágio (builder):**
  + Usa a imagem golang:latest que contém todas as ferramentas necessárias para compilar a aplicação.
  + Compila o binário estático com otimizações (ldflags="-s -w" reduz o tamanho do binário).
* **Segundo Estágio (imagem final):**
  + Usa a imagem scratch, que é vazia, resultando em uma imagem muito pequena.
  + Copia apenas o binário compilado do estágio anterior.
  + Define o usuário e o entrypoint.

**Observação sobre Imagens Otimizadas:**

* **Imagens Alpine e Slim:**
  + Você pode usar golang:alpine ou golang:slim no primeiro estágio para reuzir ainda mais o tamanho da imagem de build
  + Exemplo:

FROM golang:latest-alpine AS builder

**Benefícios:**

* **Redução de Tamanho:** A imagem final contém apenas o binário necessário para executar a aplicação.
* **Segurança:** Menor superfície de ataque, pois não inclui ferramentas desnecessárias.

## ONBUILD e Imagens Base Personalizadas

O ONBUILD permite que você adicione instruções ao Dockerfile que só serão executadas quando sua imagem for usada como imagem base para outra.

Dockerfile.base:

ARG NODE\_VERSION=21.1.0

FROM node:${NODE\_VERSION}

WORKDIR /app

ONBUILD COPY package\*.json ./

ONBUILD RUN npm install

ONBUILD COPY . .

CMD ["node", "index.js"]

Dockerfile.child:

FROM wesleywillians/docker-node-base:latest

**Explicação:**

* **Dockerfile.base:**
* Define uma imagem base personalizada com instruções ONBUILD.
* Quando outra imagem usa esta imagem como base, as instruções ONBUILD são executadas.

**Uso Prático:**

Facilita a padronização de ambientes em equipes ou projetos.

## Passando Variáveis e Argumentos no docker run

Você pode sobrescrever variáveis de ambiente e argumentos definidos no Dockerfile ao executar o container.

### Passando Variáveis de Ambiente com e

docker run -e MESSAGE="Hello from Docker!" -e PORT=4000 -p 4000:4000 my-node-app

* **e MESSAGE="Hello from Docker!"**: Define a variável de ambiente MESSAGE.
* **e PORT=4000**: Define a variável de ambiente PORT.

## Passando Argumentos para o CMD ou ENTRYPOINT

**Aplicação Go:**

* O Dockerfile define:

ENTRYPOINT ["./server"]

* Para passar a porta como argumento:

docker run -p 8080:8080 my-go-app 8080

* Isso executará ./server 8080.

## Sobrescrevendo o ENTRYPOINT com -entrypoint

docker run --entrypoint /bin/sh my-go-app

* Permite executar um shell dentro do container.

## Especificando o Usuário com u

docker run -u 1001 my-node-app

Executa o container com o usuário de ID 1001.

## Exemplos Práticos Passo a Passo

### Aplicação Node.js

**Estrutura de Diretórios:**

- app/

- Dockerfile

- .dockerignore

- package.json

- package-lock.json

- index.js

**Conteúdo do .dockerignore:**

node\_modules

npm-debug.log

Dockerfile

.dockerignore

.git

**Dockerfile Otimizado:**

ARG NODE\_VERSION=21.1.0

FROM node:${NODE\_VERSION}

ENV PORT=3001

ENV MESSAGE="Hello Docker!"

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

HEALTHCHECK --interval=10s --timeout=5s --start-period=5s --retries=3 \

CMD ["curl", "-f", "http://localhost:${PORT}"] || exit 1

EXPOSE ${PORT}

CMD ["node", "index.js"]

**Explicações Adicionais:**

* Uso de Variáveis no HEALTHCHECK e EXPOSE:
  + Utilizamos ${PORT} para referenciar a variável de ambiente definida anteriormente.

**Código da Aplicação (index.js):**

const express = require("express");

const app = express();

const port = process.env.PORT || 3000;

const message = process.env.MESSAGE || "Hello World!";

app.get("/", (req, res) => {

res.send(message);

});

app.get("/health", (req, res) => {

res.sendStatus(200);

});

app.listen(port, () => {

console.log(`Server is running on port ${port}`);

});

**Observação:**

* Adicionamos o endpoint /health para o HEALTHCHECK funcionar corretamente.

#### Comandos para Build e Run:

* Build da imagem:

docker build -t my-node-app .

* Executando o Container:

docker run -p 3001:3001 my-node-app

* Passando Variáveis de Ambiente:

docker run -e MESSAGE="Hello from Docker!" -e PORT=4000 -p 4000:4000 my-node-app

### Aplicação Go

#### Dockerfile com Multi-stage Build:

FROM golang:latest AS builder

LABEL maintainer="Wesley Willians"

WORKDIR /app

COPY . .

RUN CGO\_ENABLED=0 go build -ldflags="-s -w" -o server main.go

# Final stage

FROM scratch

USER 1001

COPY --from=builder /app/server /server

ENTRYPOINT ["./server"]

#### Código da Aplicação (main.go):

package main

import (

"fmt"

"net/http"

"os"

)

func main() {

args := os.Args[1:]

port := "8080"

if len(args) > 0 {

port = args[0]

}

http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "Hello, Docker!")

})

http.HandleFunc("/health", func(w http.ResponseWriter, r \*http.Request) {

w.WriteHeader(http.StatusOK)

})

fmt.Printf("Starting server on port :%s\n", port)

http.ListenAndServe(":"+port, nil)

}

**Observação:**

* Modificamos o código para aceitar argumentos de linha de comando para a porta e adicionamos o endpoint /health.

#### Comandos para Build e Run:

* **Build da Imagem:**

docker build -t my-go-app .

* **Executando o Container:**

docker run -p 8080:8080 my-go-app

* **Passando Argumentos para Especificar a Porta:**

docker run -p 9090:9090 my-go-app 9090

## Resumo e Melhores Práticas

* **Reaproveitamento de Cache:**
  + Copie os arquivos de dependência antes do código-fonte para maximizar o uso do cache.
* **Uso do .dockerignore:**
  + Exclua arquivos e diretórios desnecessários para reduzir o contexto de build.
* **CMD vs ENTRYPOINT:**
  + **CMD** pode ser sobrescrito e define comandos padrão.
  + **ENTRYPOINT** define o executável principal e é menos suscetível a ser sobrescrito.
* **Passando Variáveis e Argumentos:**
  + Use e ou -env no docker run para passar variáveis de ambiente.
  + Passe argumentos diretamente para sobrescrever o **CMD**.
* **Multi-stage Builds:**
  + Utilize para reduzir o tamanho da imagem final.
  + Use imagens base otimizadas (e.g., alpine) no estágio de build quando apropriado.
* **HEALTHCHECK:**
  + Implemente um endpoint /health para verificar a saúde da aplicação.
  + Certifique-se de que as ferramentas usadas no HEALTHCHECK (e.g., curl) estão disponíveis na imagem.
* **Segurança com USER:**
  + Execute processos com usuários não root para melhorar a segurança.
* **LABEL e Metadados:**
  + Adicione labels para facilitar a manutenção e automação.

# Introdução ao Docker Buildx

O Docker Buildx é uma ferramenta avançada que estende o comando docker build com funcionalidades poderosas, permitindo:

* **Builds Multiplataforma**: Construa imagens para várias arquiteturas (e.g., amd64, arm64) a partir de uma única máquina.
* **Ambiente Isolado**: Use um ambiente de build separado, evitando conflitos com o Docker daemon principal.
* **Emulação com qemu**: Permite construir imagens para arquiteturas diferentes da do host.
* **Gerenciamento Avançado de Cache**: Reutilize camadas de build para acelerar o processo.
* **Docker-in-Docker**: Executa builds dentro de um container com seu próprio Docker daemon.

## Conceitos Fundamentais

### Builders e Contextos

* **Builder**: Um ambiente onde os builds são executados. Pode ser local ou remoto e possui configurações específicas.
* **Contexto**: Define qual Docker daemon o Docker CLI usará. Pode apontar para um daemon local ou remoto.

**Nota**: Um builder pode estar associado a um contexto, e isso afeta como ele é gerenciado.

### Drivers: docker vs docker-container

**Driver docker**

* Usa o Docker daemon local.
* Builds são executados diretamente no daemon principal.
* Suporte limitado para builds multiplataforma sem emulação.
* Não cria um ambiente isolado.

**Driver docker-container**

* Cria um container separado que executa um Docker daemon interno (Docker-in-Docker).
* Suporta builds multiplataforma completos usando qemu para emulação.
* Oferece um ambiente de build isolado, evitando interferências com o daemon principal.
* Permite configurações avançadas de cache e builds distribuídos.

**Ambiente Isolado e Docker-in-Docker**

Quando você usa o driver docker-container, o Buildx cria um **container** que roda um **daemon Docker interno**. Isso significa:

* O build ocorre em um ambiente separado, não afetando o Docker daemon principal.
* O container tem seu próprio cache, configurações e pode usar qemu para emulação.
* É essencialmente um "Docker-in-Docker", onde um Docker daemon está rodando dentro de um container Docker.

**O que é qemu e por que é importante?**

* **QEMU**: **Quick Emulator**, uma ferramenta de emulação que permite executar programas compilados para uma arquitetura em outra diferente.
* No contexto do Docker Buildx, qemu permite construir imagens para arquiteturas diferentes da do host (e.g., construir uma imagem amd64 em um host arm64).
* O Buildx com driver docker-container configura automaticamente o qemu dentro do ambiente isolado, facilitando os builds multiplataforma.

## Criando e Gerenciando Builders

### Criando um Builder com docker-container

Para criar um novo builder usando o driver docker-container:

docker buildx create --name meu-builder --driver docker-container --use

* **-name**: Nome do builder.
* **-driver docker-container**: Especifica o driver.
* **-use**: Define o builder como o ativo.

### Inicializando o Builder com -bootstrap

Após criar o builder, inicialize-o:

docker buildx inspect --bootstrap

* O -bootstrap configura o ambiente, incluindo a configuração do qemu para emulação.
* Se o builder não estiver ativo, ele será inicializado, e o container correspondente será criado.

### Listando e Selecionando Builders

* **Listar builders**:

docker buildx ls

Exibe todos os builders, seus drivers, contextos e status.

* **Selecionar um builder existente**:

docker buildx use meu-builder

Define o builder especificado como o ativo.

### Removendo Builders e Contextos

* **Remover um builder**:

docker buildx rm meu-builder

* **Se o builder estiver associado a um contexto**:

Se você receber um erro semelhante a:

failed to remove desktop-linux: context builder cannot be removed, run `docker context rm desktop-linux` to remove this context

Isso significa que desktop-linux é um contexto, não apenas um builder. Nesse caso, remova o contexto:

docker context rm desktop-linux

Depois, remova o builder se necessário.

**Nota**: Um builder associado a um contexto não pode ser removido até que o contexto seja removido.

## Executando Builds Multiplataforma

### Usando qemu para Emulação de Arquitetura

* **qemu** permite que você construa imagens para arquiteturas diferentes da do host.
* O Buildx com docker-container configura o qemu automaticamente dentro do ambiente isolado.
* Isso é essencial para builds multiplataforma confiáveis.

### Builds Multiplataforma com docker-container

Exemplo: Construir uma imagem para amd64 em um host arm64 (e vice-versa).

1. **Crie e inicialize o builder** (como mostrado anteriormente).
2. **Execute o build multiplataforma**:

docker buildx build --platform linux/amd64,linux/arm64 -t meu-repositorio/minha-imagem:latest --push .

- \*\*`-platform`\*\*: Especifica as arquiteturas alvo.

- \*\*`-push`\*\*: Necessário para builds multiplataforma, pois a imagem resultante precisa ser enviada a um registry.

### Comparação com o Driver docker

* **Driver docker**:
  + Usa o Docker daemon local.
  + Não suporta emulação com qemu por padrão.
  + Builds multiplataforma são limitados à arquitetura do host.
* **Limitações**:
  + Em um host arm64, você não conseguirá construir imagens amd64 corretamente sem qemu.
  + O driver docker não configura qemu automaticamente.

## Gerenciamento de Cache no Buildx

O Buildx oferece opções avançadas de cache para acelerar seus builds.

### Cache Local

Para usar cache local:

docker buildx build --cache-to type=local,dest=./cache --cache-from type=local,src=./cache -t minha-imagem .

* **-cache-to**: Define onde o cache será salvo.
* **-cache-from**: Especifica de onde carregar o cache existente.

### Cache Remoto em Registries

Para armazenar o cache em um registry remoto (como o Docker Hub):

docker buildx build \ --cache-to type=registry,ref=meu-repositorio/minha-imagem:cache,mode=max \ --cache-from type=registry,ref=meu-repositorio/minha-imagem:cache \ -t minha-imagem .

* Isso permite compartilhar o cache entre diferentes ambientes e máquinas.

### Limpeza de Cache com buildx prune

Para liberar espaço em disco removendo caches antigos:

* **Remover todo o cache**:

docker buildx prune -a

* **Remover cache não utilizado nas últimas 24 horas**:

docker buildx prune --filter=until=24h

## Exemplos Práticos Passo a Passo

**Exemplo 1: Criando e Usando um Builder**

1. **Criar um builder com docker-container**:

docker buildx create --name meu-builder --driver docker-container --use

1. **Inicializar o builder**:

docker buildx inspect --bootstrap

1. **Verificar se o builder está ativo**:

docker buildx ls

**Exemplo 2: Build Multiplataforma de uma Aplicação Go**

Usando o Dockerfile da aplicação Go fornecida:

**Dockerfile:**

FROM golang:latest AS builder

LABEL maintainer="Wesley Willians"

WORKDIR /app

COPY . .

RUN CGO\_ENABLED=0 go build -ldflags="-s -w" -o server main.go

# Final stage

FROM scratch

USER 1001

COPY --from=builder /app/server /server

ENTRYPOINT ["./server"]

**Passos:**

1. **Criar e inicializar o builder** (se ainda não o fez).
2. **Executar o build multiplataforma**:

docker buildx build --platform linux/amd64,linux/arm64 -t meu-repositorio/minha-aplicacao-go:latest --push .

1. **Verificar a imagem no registry**:

As imagens para ambas as arquiteturas estarão disponíveis.

**Exemplo 3: Gerenciamento de Builders e Contextos**

1. **Listar builders e contextos**:

docker buildx ls

docker context ls

1. **Remover um builder associado a um contexto**: Primeiro, remover o contexto:

docker context rm meu-contexto

Depois, remover o builder:

docker buildx rm meu-builder

# Introdução ao Docker Networking

O Docker fornece um sistema de rede completo que permite conectar containers entre si e com o mundo externo. Compreender como o Docker lida com redes é essencial para desenvolver aplicações distribuídas e escaláveis.

## Conceitos Básicos de Rede no Docker

### Tipos de Redes no Docker

O Docker oferece vários tipos de redes, mas os mais comuns são:

1. **Bridge (Ponte)**:
   * Rede padrão para containers independentes.
   * Fornece isolamento entre containers e o host.
   * Containers na mesma rede bridge personalizada podem se comunicar entre si por nome.
2. **Host**:
   * O container compartilha a interface de rede com o host.
   * Útil para casos onde o desempenho é crítico ou quando o container precisa ouvir em portas privilegiadas.
   * **Nota**: No macOS e Windows, o modo de rede host tem limitações. No Docker Desktop, essa opção pode ser habilitada, mas é recomendado usar o modo bridge para maior compatibilidade.

### Como os Containers se Conectam às Redes

* **Interfaces Virtuais**: O Docker cria interfaces de rede virtuais para cada container.
* **Endereços IP**: Cada container recebe um endereço IP privado dentro da rede à qual está conectado.
* **Resolução de Nomes**: Em redes bridge personalizadas, o Docker configura um serviço de DNS interno para resolução de nomes de containers.

## Comunicação entre Containers na Mesma Rede

**Exemplo Prático: Aplicação Node.js e MongoDB**

Vamos criar uma aplicação Node.js que se comunica com um banco de dados MongoDB, ambos executando em containers Docker.

**Passo 1: Preparar a Aplicação Node.js**

**Arquivo index.js:**

const mongoose = require("mongoose");

mongoose

.connect("mongodb://db:27017/test")

.then(() => console.log("Connected to MongoDB"))

.catch((err) => console.log(err));

* A aplicação tenta conectar-se ao MongoDB no host db na porta 27017.
* O nome db será resolvido dentro da rede Docker personalizada.

**Passo 2: Criar o Dockerfile**

**Arquivo Dockerfile:**

dockerfile

Copy code

FROM node:lts-alpine

WORKDIR /app

COPY package\*.json .

RUN npm install

COPY . .

CMD ["node", "index.js"]

* Utilizamos a imagem node:lts-alpine para um container leve.
* Instalamos as dependências e copiamos o código da aplicação.

**Passo 3: Criar o .dockerignore**

**Arquivo .dockerignore:**

node\_modules

* Evita copiar a pasta node\_modules local para o container.

**Passo 4: Construir a Imagem Docker**

docker build -t mynode\_app\_network .

* Constrói a imagem e a nomeia como mynode\_app\_network.

**Passo 5: Criar uma Rede Bridge Personalizada**

docker network create app-network

* Cria uma rede chamada app-network.

**Passo 6: Iniciar o Container do MongoDB**

docker run -d --name db --network app-network mongo:latest

* Inicia o MongoDB na rede app-network com o nome db.

**Passo 7: Iniciar o Container da Aplicação Node.js**

docker run -d --name app --network app-network mynode\_app\_network

* Inicia a aplicação Node.js na mesma rede app-network.

**Passo 8: Verificar a Comunicação entre os Containers**

* Verifique os logs da aplicação:
* docker logs app
* Você deverá ver a mensagem:
* Connected to MongoDB
* Isso confirma que a aplicação Node.js conseguiu resolver o nome db e conectar-se ao MongoDB.

## Comunicação entre Containers e o Host

Às vezes, um container precisa acessar um serviço que está rodando no host. Vamos ver como fazer isso usando host.docker.internal.

### Utilizando host.docker.internal

* host.docker.internal é um hostname especial que resolve para o endereço IP do host.
* **Nota**: No Linux, host.docker.internal não está disponível por padrão. Porém, a partir do Docker 20.10, podemos usar o placeholder host-gateway.

**Exemplo Prático: Acessando um Serviço no Host a partir de um Container**

**Passo 1: Preparar a Aplicação Node.js no Host**

Crie uma aplicação Node.js simples no host que escuta na porta 3000.

**Arquivo app\_host.js:**

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello from the Host");

});

app.listen(3000, "0.0.0.0", () => {

console.log("App listening on port 3000");

});

Inicie a aplicação:

node app\_host.js

Teste no host:

curl http://localhost:3000

# Deverá retornar: Hello from the Host

**Passo 2: Iniciar um Container Nginx e Acessar o Serviço do Host**

Inicie o container do Nginx, adicionando host.docker.internal ao /etc/hosts do container.

docker run -d --name nginx --add-host=host.docker.internal:host-gateway nginx

* O host-gateway é resolvido automaticamente pelo Docker para o endereço IP do host.

**Passo 3: Testar a Conexão a partir do Container**

Entre no container do Nginx:

docker exec -it mynginx sh

Teste a conexão com o serviço no host:

curl http://host.docker.internal:3000

* Você deverá ver:
* Hello World

**Passo 4: Verificar a Configuração da Rede**

Inspecione a rede bridge:

docker network inspect bridge

* Verifique o gateway e certifique-se de que o host-gateway está sendo resolvido corretamente.

## Gerenciando Múltiplas Redes e Isolamento

É possível conectar um container a mais de uma rede, permitindo comunicação seletiva entre serviços.

**Criando e Conectando Containers a Múltiplas Redes**

**Passo 1: Criar Redes Personalizadas**

docker network create backend-net

docker network create db-net

**Passo 2: Iniciar os Containers nas Redes Apropriadas**

Inicie o container do MongoDB na rede db-net:

docker run -d --name db --network db-net mongo:latest

Inicie o container da aplicação Node.js na rede backend-net:

docker run -d --name app --network backend-net mynode\_app\_network

**Passo 3: Verificar que a Aplicação Não Consegue Conectar ao MongoDB**

Verifique os logs da aplicação:

docker logs app

Você deverá ver um erro de conexão, pois a aplicação não consegue resolver o nome db.

**Passo 4: Conectar o Container da Aplicação à Rede db-net**

Conecte o container app também à rede db-net:

docker network connect db-net app

Agora, o container app está conectado a duas redes: backend-net e db-net.

**Passo 5: Verificar que a Aplicação Consegue Conectar ao MongoDB**

Verifique novamente os logs da aplicação:

docker logs app

Você deverá ver:

Connected to MongoDB

**Exemplo Prático: Isolamento de Serviços com Redes Personalizadas**

* **Objetivo**: Isolar o acesso ao MongoDB, permitindo que apenas a aplicação Node.js se conecte a ele.
* **Benefício**: Melhora a segurança, evitando que outros containers acessem o banco de dados inadvertidamente.

**6. Modos de Rede: Bridge vs. Host**

**Modo Bridge (Padrão)**

* **Recomendado**: Fornece isolamento entre containers e do host.
* **Comunicação entre Containers**: Através de redes bridge personalizadas, com resolução de nomes automática.
* **Compatibilidade**: Funciona de forma consistente em Linux, macOS e Windows.

**Modo Host**

* **Funcionamento**: O container compartilha a interface de rede do host.
* **Uso no Linux**:
* docker run -d --network host mynode\_app\_network
* **Considerações**:
  + **Desempenho**: Pode melhorar o desempenho em aplicações que requerem alta taxa de transferência de rede.
  + **Portas**: Não é necessário mapear portas; o container usa as mesmas portas do host.
  + **Segurança**: Menos isolado; pode representar riscos se não for usado corretamente.
* **Uso no macOS e Windows**:
  + O suporte ao modo host no Docker Desktop está disponível, mas pode ter limitações.
  + **Recomendação**: Prefira usar o modo bridge para maior compatibilidade.

# Introdução ao Docker Compose

O Docker Compose é uma ferramenta que permite definir e executar aplicações Docker multicontainer. Com ele, podemos utilizar um arquivo YAML para configurar todos os serviços necessários para a aplicação e, com um único comando, criar e iniciar todos os containers definidos.

**História:**

**Docker Compose V1**: Lançado em 2014 como uma ferramenta separada, permitia orquestrar containers usando o arquivo docker-compose.yml.

**Docker Compose V2+:** Introduzido para melhorar a integração e desempenho, agora faz parte do Docker CLI como docker compose (note o espaço).

Por que usar o Docker Compose?

* Simplicidade: Gerencia múltiplos containers facilmente.
* Reprodutibilidade: Facilita a configuração e compartilhamento de ambientes.
* Escalabilidade: Permite escalonar serviços de forma simples.

## Criando os Primeiros Serviços

Começaremos criando dois serviços básicos:

Aplicação Node.js que se conecta a um MongoDB.

Servidor Nginx para servir conteúdo estático.

### Serviço Node.js

**Estrutura de Arquivos:**

* **Diretório**: ./node/
  + Dockerfile
  + index.js
  + package.json
  + .env
  + .dockerignore

**Código da Aplicação (index.js):**

const mongoose = require("mongoose");

mongoose

.connect("mongodb://mongo:27017/test")

.then(() => console.log("Connected to MongoDB!"))

.catch((err) => console.log(err));

A aplicação tenta conectar ao MongoDB no host mongo na porta 27017.

**Dockerfile da Aplicação:**

FROM node:lts-alpine

WORKDIR /app

COPY package\*.json .

RUN npm install

COPY . .

CMD ["node", "index.js"]

Usa a imagem node:lts-alpine para uma imagem leve.

Instala as dependências e copia o código da aplicação.

**Criando o Serviço no docker-compose.yml:**

services:

nodeapp:

build:

context: ./node

dockerfile: Dockerfile

**build**: Define que o serviço será construído a partir do Dockerfile no diretório ./node.

### Serviço Nginx

**Estrutura de Arquivos**

* **Diretório**: ./nginx-html/
  + index.html

**Arquivo index.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Nginx Container</title>

</head>

<body>

Hello World

</body>

</html>

**Atualizando o docker-compose.yml:**

services:

nodeapp:

# ... (configuração anterior)

nginx:

image: nginx:latest

ports:

- "8080:80"

**image**: Usa a imagem oficial do Nginx.

**ports**: Mapeia a porta 8080 do host para a porta 80 do container.

## Definindo Dependências e Healthchecks

Nossa aplicação Node.js depende do MongoDB. Vamos adicionar o serviço do MongoDB e configurar dependências e healthchecks.

**Adicionando o Serviço MongoDB**

services:

# ... (nodeapp e nginx)

mongo:

image: mongo:latest

Configurando depends\_on e healthcheck

Atualizando o Serviço MongoDB com Healthcheck

services:

mongo:

image: mongo:latest

healthcheck:

test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]

interval: 5s

retries: 5

start\_period: 10s

timeout: 5s

* **healthcheck:** Verifica se o MongoDB está pronto.
* **test:** Comando para testar a saúde do serviço.

Atualizando o Serviço Node.js com depends\_on

services:

nodeapp:

build:

context: ./node

dockerfile: Dockerfile

depends\_on:

mongo:

condition: service\_healthy

* **depends\_on:** Define que nodeapp depende de mongo.
* **condition: service\_healthy:** Garante que o nodeapp só inicie após o mongo estar saudável.

## Trabalhando com Volumes e Bind Mounts

### Persistindo Dados com Volumes

Para garantir que os dados do MongoDB persistam entre reinicializações, vamos adicionar um volume.

**Atualizando o Serviço MongoDB com Volumes**

services:

mongo:

# ... (configurações anteriores)

volumes:

- mongo\_data:/data/db

volumes:

mongo\_data:

* **volumes:** Monta o volume nomeado mongo\_data no caminho /data/db dentro do container.
* **volumes no topo:** Declaração do volume nomeado.

**Utilizando Bind Mounts para Desenvolvimento**

Para facilitar o desenvolvimento, podemos montar nosso código local dentro do container.

**Atualizando o Serviço Node.js com Bind Mount**

services:

nodeapp:

# ... (configurações anteriores)

volumes:

- ./node:/app

* **volumes:** Monta o diretório local ./node no caminho /app dentro do container.

## Variáveis de Ambiente e env\_file

**Definindo Variáveis no Compose**

Podemos definir variáveis de ambiente diretamente no docker-compose.yml.

**Adicionando Variáveis ao Serviço Node.js**

services:

nodeapp:

# ... (configurações anteriores)

environment:

- MY\_VAR=123456

* **environment:** Define variáveis de ambiente para o container.

**Usando Arquivos .env**

Para evitar expor informações sensíveis e facilitar a gestão de variáveis, podemos usar um arquivo .env.

**Criando o Arquivo .env na Pasta ./node/**

Conteúdo de ./node/.env:

API\_KEY=abcdef12345

**Atualizando o Serviço Node.js para Usar o env\_file**

services:

nodeapp:

# ... (configurações anteriores)

env\_file:

- ./node/.env

* **env\_file:** Especifica o arquivo contendo variáveis de ambiente.

## Gerenciando Redes no Docker Compose

### Criando Redes Personalizadas

Vamos criar redes personalizadas para isolar nossos serviços.

### Definindo Redes no docker-compose.yml

networks:

backend:

db-net:

frontend:

* **networks:** Define as redes que serão usadas pelos serviços.

### Conectando Serviços às Redes

**Atualizando o Serviço Node.js com Redes**

services:

nodeapp:

# ... (configurações anteriores)

networks:

- backend

- db-net

* **networks:** Conecta o serviço nodeapp às redes backend e db-net.

**Atualizando o Serviço MongoDB com Rede**

services:

mongo:

# ... (configurações anteriores)

networks:

- db-net

* O MongoDB estará na rede db-net, permitindo que o nodeapp se comunique com ele.

**Atualizando o Serviço Nginx com Rede**

services:

nginx:

# ... (configurações anteriores)

networks:

- frontend

* O Nginx estará isolado na rede frontend.

## Utilizando extra\_hosts para Acessar o Host

Para que os containers possam acessar serviços rodando no host, usamos extra\_hosts com host.docker.internal.

### Atualizando o Serviço Node.js com extra\_hosts

services:

nodeapp:

# ... (configurações anteriores)

extra\_hosts:

- "host.docker.internal:host-gateway"

* **extra\_hosts:** Adiciona uma entrada no /etc/hosts do container, permitindo que o nome host.docker.internal seja resolvido.

## Introdução ao Compose Watch

O **Compose Watch** é uma ferramenta que permite monitorar alterações nos arquivos e sincronizar com o container em tempo real, facilitando o desenvolvimento.

### Configurando o compose-watch

### Adicionando o Serviço nodeapp-watch

services:

nodeapp-watch:

build:

context: ./node

dockerfile: Dockerfile

depends\_on:

mongo:

condition: service\_healthy

develop:

watch:

- path: ./node

target: /app

action: sync

ignore:

- "./node/node\_modules"

- path: ./node/package.json

action: rebuild

- path: ./node/index.js

target: /app/index.js

action: sync+restart

networks:

- backend

- db-net

* + **develop.watch:** Configurações para o Compose Watch.
    - **action: sync**: Sincroniza arquivos entre host e container.
    - **action: rebuild**: Reconstrói a imagem quando o arquivo muda.
    - **action: sync+restart**: Sincroniza e reinicia o serviço.

### Utilizando o Compose Watch

Comando para Iniciar o Compose Watch:

* docker compose watch

O Compose Watch irá monitorar as alterações nos arquivos e atualizar o container automaticamente.

**Arquivo docker-compose.yml Completo**

services:

nodeapp:

build:

context: ./node

dockerfile: Dockerfile

depends\_on:

mongo:

condition: service\_healthy

environment:

- MY\_VAR=123456

env\_file:

- ./node/.env

volumes:

- ./node:/app

networks:

- backend

- db-net

extra\_hosts:

- "host.docker.internal:host-gateway"

nodeapp-watch:

build:

context: ./node

dockerfile: Dockerfile

depends\_on:

mongo:

condition: service\_healthy

develop:

watch:

- path: ./node

target: /app

action: sync

ignore:

- "./node/node\_modules"

- path: ./node/package.json

action: rebuild

- path: ./node/index.js

target: /app/index.js

action: sync+restart

networks:

- backend

- db-net

nginx:

image: nginx:latest

ports:

- "8080:80"

volumes:

- ./nginx-html:/usr/share/nginx/html

networks:

- frontend

mongo:

image: mongo:latest

healthcheck:

test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]

interval: 5s

retries: 5

start\_period: 10s

timeout: 5s

volumes:

- mongo\_data:/data/db

networks:

- db-net

volumes:

mongo\_data:

networks:

db-net:

backend:

frontend:

# O que é o docker init

O docker init é um comando introduzido para simplificar a criação de arquivos Dockerfile e docker-compose.yml em projetos que ainda não os possuem. Ele analisa o código-fonte do seu projeto e gera automaticamente templates otimizados, facilitando o processo de containerização de aplicações.

**Principais benefícios:**

* **Automação:** Gera arquivos de configuração Docker automaticamente.
* **Boas práticas:** Os templates seguem as melhores práticas recomendadas pela Docker.
* **Otimizações:** Inclui otimizações de cache e suporte a builds multiplataforma.
* **Flexibilidade:** Suporta várias linguagens e permite personalização posterior.

**Como usar:**

Basta navegar até o diretório do seu projeto e executar:

docker init

O comando irá interagir com você para coletar informações sobre o projeto e, em seguida, gerar os arquivos necessários.

## Projeto Go com docker init

Vamos criar um projeto simples em Go e usar o docker init para gerar os arquivos de configuração.

**Passo a Passo:**

1. **Crie um diretório para o projeto e navegue até ele:**

mkdir my-go-app

cd my-go-app

1. **Inicie um módulo Go:**

go mod init my-go-app

1. **Crie um arquivo main.go:**

package main

import (

"fmt"

"net/http"

)

func handler(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "Hello, Docker!")

}

func main() {

http.HandleFunc("/", handler)

http.ListenAndServe(":8080", nil)

}

1. **Execute docker init:**

docker init

**Durante a interação:**

* + **Select the application platform:** Go
  + **Name of the service:** server (padrão)
  + **Which port does your server application listen on?:** 8080
  + **Would you like to include Compose files?:** Yes
  + **Which version of Go do you want to use?:** (padrão ou escolha a versão desejada)

### Análise detalhada do Dockerfile Go

Vamos analisar o Dockerfile gerado linha por linha.

# syntax=docker/dockerfike:1

* **Linha 1:** Define a sintaxe da linguagem de Dockerfile a ser usada. Especifica que usaremos os recursos da versão 1 do Dockerfile, permitindo usar recursos mais avançados como RUN --mount.

ARG GO\_VERSION=1.23.1

FROM --platform=$BUILDPLATFORM golang:${GO\_VERSION} AS build

WORKDIR /src

* **ARG GO\_VERSION=1.23.1:** Define uma variável de build GO\_VERSION com a versão do Go a ser usada.
* **FROM --platform=$BUILDPLATFORM golang:${GO\_VERSION} AS build:** Especifica a imagem base, permitindo builds multiplataforma com -platform. Usa a variável GO\_VERSION para selecionar a versão do Go.
* **WORKDIR /src:** Define o diretório de trabalho dentro do container.

RUN --mount=type=cache,target=/go/pkg/mod/ \

--mount=type=bind,source=go.mod,target=go.mod \

go mod download -x

* **RUN --mount=type=cache,target=/go/pkg/mod/:** Utiliza um cache para o diretório de módulos do Go, acelerando builds subsequentes.
* **-mount=type=bind,source=go.mod,target=go.mod:** Faz um bind mount do arquivo go.mod para evitar copiá-lo para o container, aproveitando o cache.
* **go mod download -x:** Baixa as dependências definidas em go.mod.

ARG TARGETARCH

* **ARG TARGETARCH:** Recebe a arquitetura alvo do build (por exemplo, amd64, arm64), permitindo builds multiplataforma.

RUN --mount=type=cache,target=/go/pkg/mod/ \

--mount=type=bind,target=. \

CGO\_ENABLED=0 GOARCH=$TARGETARCH go build -o /bin/server ./cmd/server

* **-mount=type=cache,target=/go/pkg/mod/:** Reutiliza o cache dos módulos Go durante o build.
* **-mount=type=bind,target=.:** Faz um bind mount do diretório atual, evitando copiar o código para o container.
* **CGO\_ENABLED=0 GOARCH=$TARGETARCH go build -o /bin/server ./cmd/server:** Compila o binário Go para a arquitetura alvo, desabilitando CGO para builds estáticos.

FROM alpine:latest AS final

* **FROM alpine:latest AS final:** Inicia uma nova etapa de build chamada final, usando a imagem mínima alpine.

RUN --mount=type=cache,target=/var/cache/apk \

apk --update add \

ca-certificates \

tzdata \

&& \

update-ca-certificates

* **-mount=type=cache,target=/var/cache/apk:** Utiliza cache para os pacotes APK, acelerando builds futuros.
* **apk --update add ...:** Instala certificados raiz e dados de fuso horário necessários.

ARG UID=10001

RUN adduser \

--disabled-password \

--gecos "" \

--home "/nonexistent" \

--shell "/sbin/nologin" \

--no-create-home \

--uid "${UID}" \

appuser

USER appuser

* Cria um usuário não privilegiado appuser com UID 10001 para executar o aplicativo, seguindo boas práticas de segurança.

COPY --from=build /bin/server /bin/

* Copia o binário compilado da etapa build para a imagem final.

EXPOSE 8080

ENTRYPOINT [ "/bin/server" ]

* **EXPOSE 8080:** Documenta que o container expõe a porta 8080.
* **ENTRYPOINT [ "/bin/server" ]:** Define o executável que será iniciado quando o container for executado.

### Análise do compose.yaml Go

services:

server:

build:

context: .

target: final

ports:

- 8080:8080

* **services:** Define os serviços que compõem a aplicação.
* **server:** Nome do serviço principal.
* **build:**
  + **context: .:** Define o diretório atual como contexto de build.
  + **target: final:** Especifica que o alvo do build é o estágio final do Dockerfile (multi-stage build).
* **ports:**
  + **8080:8080:** Mapeia a porta 8080 do host para a porta 8080 do container.

# Projeto Node.js com docker init

Agora, vamos criar um projeto simples em Node.js e usar o docker init.

**Passo a Passo:**

1. **Crie um diretório para o projeto e navegue até ele:**

mkdir my-node-app

cd my-node-app

1. **Inicie um projeto Node.js:**

npm init -y

1. **Instale dependências necessárias (exemplo: express):**

npm install express

1. **Crie um arquivo index.js:**

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Hello, Docker!');

});

app.listen(3000, () => {

console.log('App listening on port 3000');

});

* **Importante:** O node\_modules está no .dockerignore para não ser copiado para o contexto de build, garantindo que as dependências sejam instaladas dentro do container.

1. **Execute docker init:**

Docker init

**Durante a interação:**

* **Select the application platform:** Node.js
* **Name of the service:** server (padrão)
* **Which port does your server application listen on?:** 3000
* **Would you like to include Compose files?:** Yes
* **Which version of Node.js do you want to use?:** (padrão ou escolha a versão desejada)

## Análise detalhada do Dockerfile Node.js

# syntax=docker/dockerfile:1

* **Linha 1:** Define a sintaxe da linguagem de Dockerfile versão 1.

ARG NODE\_VERSION=21.1.0

FROM node:${NODE\_VERSION}-alpine

* **ARG NODE\_VERSION=21.1.0:** Define a versão do Node.js a ser usada.
* **FROM node:${NODE\_VERSION}-alpine:** Usa a imagem base do Node.js na versão especificada, baseada no Alpine Linux para uma imagem leve.

ENV NODE\_ENV production

WORKDIR /usr/src/app

* **ENV NODE\_ENV production:** Define a variável de ambiente NODE\_ENV como production.
* **WORKDIR /usr/src/app:** Define o diretório de trabalho.

RUN --mount=type=bind,source=package.json,target=package.json \

--mount=type=bind,source=package-lock.json,target=package-lock.json \

--mount=type=cache,target=/root/.npm \

npm ci --omit=dev

* **-mount=type=bind,source=package.json,target=package.json:** Faz bind mount do package.json.
* **-mount=type=bind,source=package-lock.json,target=package-lock.json:** Faz bind mount do package-lock.json.
* **-mount=type=cache,target=/root/.npm:** Utiliza cache para o diretório do NPM, acelerando builds futuros.
* **npm ci --omit=dev:** Instala as dependências de produção, garantindo builds reprodutíveis.

USER node

* Executa o restante das instruções como o usuário não privilegiado node.

COPY . .

* Copia o código da aplicação para o container. Como node\_modules está no .dockerignore, essa pasta não será copiada, evitando conflitos.

EXPOSE 3000

CMD node index.js

* **EXPOSE 3000:** Documenta que o container expõe a porta 3000.
* **CMD node index.js:** Comando que será executado quando o container iniciar.

## Análise do compose.yaml Node.js

services:

server:

build:

context: .

environment:

NODE\_ENV: production

ports:

- 3000:3000

* **services:** Define os serviços.
* **server:** Nome do serviço.
* **build:**
  + **context: .:** Usa o diretório atual como contexto de build.
* **environment:**
  + **NODE\_ENV: production:** Define variável de ambiente.
* **ports:**
  + **3000:3000:** Mapeia a porta 3000 do host para o container.

Projeto de Propósito Geral com docker init

Vamos criar um projeto sem especificar uma linguagem, para ver como o docker init lida com ele.

**Passo a Passo:**

1. **Execute docker init:**

docker init

**Durante a interação:**

1. **Select the application platform:** Other
2. **Name of the service:** (padrão)
3. **Would you like to include Compose files?:** Yes

### Análise detalhada do Dockerfile Propósito Geral

# syntax=docker/dockerfile:1

* Define a sintaxe da linguagem de Dockerfile versão 1.

FROM alpine:latest as base

* **FROM alpine:latest as base:** Usa a imagem alpine como base para as etapas seguintes.

FROM base as build

RUN echo -e '#!/bin/sh\n\

echo Hello world from $(whoami)! In order to get your application running in a container, take a look at the comments in the Dockerfile to get started.'\

> /bin/hello.sh

RUN chmod +x /bin/hello.sh

* **FROM base as build:** Inicia a etapa de build.
* Cria um script hello.sh que imprime uma mensagem.
* **chmod +x /bin/hello.sh:** Torna o script executável.

FROM base AS final

ARG UID=10001

RUN adduser \

--disabled-password \

--gecos "" \

--home "/nonexistent" \

--shell "/sbin/nologin" \

--no-create-home \

--uid "${UID}" \

appuser

USER appuser

COPY --from=build /bin/hello.sh /bin/

ENTRYPOINT [ "/bin/hello.sh" ]

* Inicia a etapa final usando a imagem base.
* Cria um usuário não privilegiado appuser.
* Copia o script hello.sh da etapa de build.
* Define o comando de entrada para executar o script.

## Entendendo o Uso de Secrets no docker-compose.yml

Nos exemplos dos arquivos compose.yaml, há uma seção comentada que demonstra como adicionar um banco de dados PostgreSQL com uso de secrets.

**Seção de Exemplo no compose.yaml:**

services:

server:

# ... (configurações do serviço)

depends\_on:

db:

condition: service\_healthy

db:

image: postgres

restart: always

user: postgres

secrets:

- db-password

volumes:

- db-data:/var/lib/postgresql/data

environment:

- POSTGRES\_DB=example

- POSTGRES\_PASSWORD\_FILE=/run/secrets/db-password

expose:

- 5432

healthcheck:

test: [ "CMD", "pg\_isready" ]

interval: 10s

timeout: 5s

retries: 5

volumes:

db-data:

secrets:

db-password:

file: db/password.txt

**Análise e Explicação:**

* **secrets:** Define um secret chamado db-password que será carregado a partir do arquivo db/password.txt.
* **db:** Serviço do PostgreSQL.
  + **secrets:** O serviço usa o secret db-password.
  + **environment:**
    - **POSTGRES\_PASSWORD\_FILE=/run/secrets/db-password:** Informa ao PostgreSQL para usar o arquivo de secret para obter a senha.
* **depends\_on:** O serviço server depende do db e aguarda até que ele esteja saudável.

### Uso de Secrets no Docker Compose:

* **No Docker Swarm:**
  + **Secrets** são armazenados de forma segura e distribuídos para os nós do Swarm.
  + **Segurança:** Os secrets são criptografados e gerenciados pelo Swarm, garantindo que apenas serviços autorizados tenham acesso.
* **No Docker Compose (Standalone):**
  + **Limitações:** O uso de secrets no Docker Compose tradicional não fornece o mesmo nível de segurança que no Swarm.
  + **Implementação:** Os secrets são montados como arquivos simples no container, sem criptografia adicional.
  + **Finalidade:** Facilita a gestão de credenciais em ambiente de desenvolvimento, mas não deve ser considerado seguro para produção.

### Segurança e Criptografia de Secrets:

* **Docker Swarm:**
  + **Criptografia:** Secrets são criptografados em trânsito e em repouso.
  + **Protocolos:** Usa TLS para comunicação segura entre nós.
  + **Armazenamento:** Mantidos no Raft log do Swarm, que é criptografado.
  + **Acesso Controlado:** Apenas serviços que declaram necessidade podem acessar os secrets.
* **Docker Compose (Standalone):**
  + **Sem Criptografia Avançada:** Os secrets são montados como arquivos nos containers.
  + **Recomendação:** Não é recomendado para armazenar informações sensíveis em produção.