



Comandos básicos do Docker

Este documento explora os comandos básicos do Docker, uma plataforma de containers essencial para o desenvolvimento moderno. Abordamos conceitos fundamentais como imagens e containers, além de comandos práticos para gerenciar ambientes Docker. O guia inclui exemplos, explicações detalhadas e um exercício prático para consolidar o aprendizado.

editado por Everton Coimbra de Araújo

O que é Docker e sua importância

Docker é uma plataforma de containers que permite empacotar uma aplicação e suas dependências em um ambiente isolado e portátil. Ele ajuda a garantir que a aplicação funcione da mesma forma em qualquer sistema, resolvendo o famoso problema do "funciona na minha máquina, mas não no servidor". Docker é amplamente utilizado para criar ambientes consistentes de desenvolvimento e produção, simplificar o gerenciamento de dependências e tornar o processo de deploy mais eficiente.

Cada comando que veremos a seguir será explicado em detalhes, com exemplos práticos e forneceremos também as razões pelas quais cada comando é importante, além dos pontos positivos e negativos de seu uso. Assim, nosso objetivo é ensinar de maneira prática e didática como você pode utilizar o Docker para otimizar seu trabalho e melhorar suas habilidades no desenvolvimento de software.

O que é uma Imagem Docker e um Container?

Imagem Docker

Uma imagem Docker é um pacote que contém tudo o que é necessário para executar uma aplicação: código, dependências, bibliotecas, variáveis de ambiente e instruções para a execução. É como um "modelo" a partir do qual os containers são criados. As imagens são imutáveis, o que significa que não podem ser modificadas depois de criadas. Isso garante consistência e confiabilidade durante o desenvolvimento e o deploy.

Por exemplo, uma imagem pode ser de um servidor web como **Nginx**, um banco de dados como **MySQL** ou até mesmo uma aplicação específica que você desenvolveu. As imagens são armazenadas em **registries**, como o **Docker Hub**, onde você pode buscar imagens públicas ou armazenar as suas próprias.

Container

Um container é uma instância de uma imagem em execução. Ele pode ser visto como um ambiente de execução isolado, criado a partir de uma imagem. Imagine que a imagem seja a receita e o container seja o prato preparado; um container é, portanto, a materialização da imagem em um ambiente que pode ser executado.

Os containers são leves e compartilham o kernel do sistema operacional do host, o que os torna mais eficientes do que máquinas virtuais. Cada container criado a partir de uma imagem é independente e pode ser manipulado individualmente, proporcionando uma maneira ágil e escalável de rodar aplicações.



Comandos Docker: docker run e docker ps

1

docker run hello-world

O comando `docker run hello-world` é utilizado para baixar e executar uma imagem de teste chamada `hello-world`. Ele é uma excelente maneira de verificar se a instalação do Docker foi bem-sucedida e se está funcionando corretamente. Quando você executa esse comando, ele baixa a imagem do Docker Hub (caso ainda não esteja armazenada localmente) e cria um container a partir dessa imagem. O Docker Hub é um serviço de registro de imagens onde os desenvolvedores podem compartilhar e armazenar imagens Docker. Ele é essencial para facilitar o acesso a imagens pré-configuradas e para compartilhar suas próprias com a comunidade ou em equipes.

2

docker ps

Outro comando muito útil é o `docker ps`. Esse comando permite listar todos os containers em execução no momento. Com ele, você pode monitorar quais containers estão ativos, além de obter informações como o ID do container, o nome e o status. Esse é um comando fundamental para gerenciar o seu ambiente Docker, permitindo visualizar o que está acontecendo no sistema e monitorar containers ativos. No entanto, ele não mostra containers que já foram parados, limitando um pouco a visão geral do que foi executado.

3

docker ps -a

Para obter uma visão mais completa, o comando `docker ps -a` é essencial. Ele lista todos os containers, incluindo aqueles que não estão em execução. Isso significa que você pode ver o histórico completo de containers que foram criados, seja para diagnosticar problemas, seja para saber quais containers ainda estão armazenados no sistema, mas não estão ativos. Isso pode ser extremamente útil ao depurar falhas ou ao tentar entender a sequência de operações que foi realizada. Por outro lado, quando há muitos containers, a listagem pode ficar bastante extensa e difícil de analisar, exigindo o uso de filtros para melhorar a organização.

Comandos Docker: Executando e gerenciando o Nginx

Levante um servidor web com um único comando

Quando você precisa de um servidor web rápido e fácil de configurar, o comando **docker run nginx** é a solução perfeita. Com apenas uma linha de código, você baixa e executa a imagem do servidor web Nginx em um container Docker. Seu servidor web já está no ar, pronto para receber tráfego na porta 80 padrão. É uma abordagem incrível para testes e desenvolvimento, permitindo que você se concentre no que realmente importa.

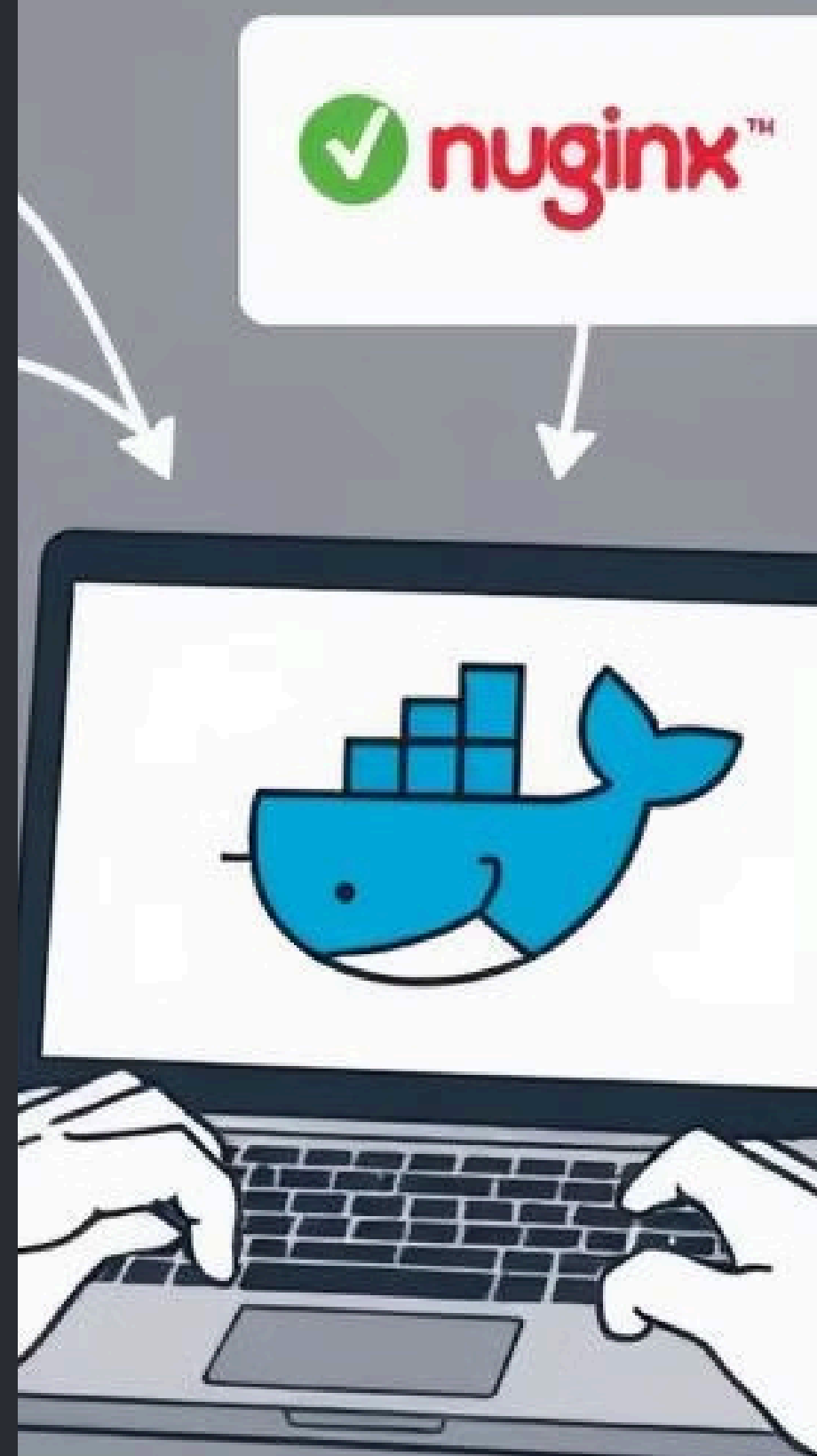
Evite conflitos de porta com mapeamento personalizado

Às vezes, pode haver outros serviços rodando em sua máquina na porta 80. Nesse caso, o comando **docker run -p 8082:80 nginx** é sua solução. Ele mapeia a porta 8082 do seu host para a porta 80 do container Nginx. Agora você pode acessar seu servidor web em **http://localhost:8082** sem nenhum conflito. Essa é uma técnica poderosa para executar múltiplos serviços em um único host, mantendo tudo organizado e isolado.

Execute o servidor web em segundo plano

Quando você precisa manter o servidor web Nginx em execução, mas também liberar o terminal para outras tarefas, o comando **docker run -d -p 8082:80 nginx** é a resposta. Ele roda o container em modo desanexado (-d), o que significa que o processo continua rodando em segundo plano. Você pode continuar usando o terminal para outras atividades, enquanto seu servidor web Nginx continua atendendo requisições na porta 8082. Embora você não veja os logs diretamente no terminal, é possível acessá-los a qualquer momento com o comando **docker logs**.

inial ce proger
tee mesalof te



Comandos Docker: Gerenciando containers

`docker stop <container_id>`

Depois de ter containers em execução, pode chegar um momento em que você precise parar um container. O comando `docker stop <container_id>` é a forma mais segura de fazer isso, pois envia um sinal para que o container seja encerrado de maneira adequada. O container é parado, mas continua existindo no sistema, o que significa que pode ser reiniciado a qualquer momento. Essa é uma abordagem útil quando você precisa liberar recursos, mas não quer perder o estado do container. Contudo, é importante lembrar que ele ainda ocupa espaço no disco.

`docker rm <container_id>`

Finalmente, para limpar containers que não são mais necessários, o comando `docker rm <container_id>` é utilizado para removê-los do sistema permanentemente. Isso é muito importante para liberar espaço e manter seu ambiente Docker organizado. É uma boa prática remover containers que não serão mais utilizados, para evitar acúmulo desnecessário. No entanto, uma vez removido, o container não pode ser recuperado, e todos os dados armazenados nele serão perdidos.



`docker start <container_id>`

Caso precise reiniciar um container que foi previamente parado, o comando `docker start <container_id>` permite fazer isso sem precisar recriar o container. Isso economiza tempo e mantém todas as configurações e dados intactos. A principal vantagem aqui é a rapidez com que você pode colocar o container de volta em operação. No entanto, o estado do container no momento em que foi parado pode não ser completamente restaurado, especialmente se ele estiver no meio de uma tarefa.

Comandos Docker: Operações avançadas

Limpeza Radical

Imagine ter um container teimoso, que simplesmente se recusa a parar. Nesse caso, o comando **docker rm -f <container_id>** é sua solução definitiva. Esse comando força a remoção do container, encerrando qualquer processo em execução e garantindo que ele seja removido do sistema. Embora essa seja uma medida drástica, às vezes é necessário adotar uma abordagem mais firme para lidar com containers problemáticos ou travados. No entanto, é importante lembrar que, uma vez removido, o container e todos os seus dados serão perdidos para sempre.

Reinício Rápido


Quando um container precisa de uma "reinicialização", o comando **docker restart <container_id>** é a solução perfeita. Diferente de ter que parar e iniciar o container manualmente, o restart faz tudo isso em uma única etapa. Isso é especialmente útil quando um serviço parou de responder ou você precisa aplicar pequenas alterações que não exigem a recriação completa do container. Com apenas um comando, você pode ter seu container de volta em ação, pronto para atender suas necessidades.

Monitorando Atividades

Para acompanhar o que está acontecendo dentro de um container, especialmente quando ele está rodando em segundo plano, o comando **docker logs <container_id>** é sua melhor ferramenta. Esse comando exibe os logs de saída do container, permitindo que você acompanhe o comportamento da sua aplicação em tempo real. Você pode adicionar a flag **-f** para seguir os logs continuamente, como se estivesse monitorando a atividade do container de forma ininterrupta. Essa é uma funcionalidade essencial para diagnosticar problemas e entender o que está acontecendo dentro de seus ambientes Docker.

Conclusão e FAQ

Nesta seção, vimos os principais comandos básicos do Docker, desde a execução de um container com `docker run` até o gerenciamento de containers com `docker ps`, `docker stop`, `docker rm` e outros. Esses comandos são fundamentais para o uso eficiente do Docker e ajudam a entender como criar, gerenciar e remover containers. A prática desses comandos permitirá a criação de ambientes de desenvolvimento consistentes e o gerenciamento eficaz de aplicações containerizadas.



O que é Docker?

Docker é uma plataforma de containers que permite empacotar uma aplicação e suas dependências em um ambiente isolado.





Imagem vs Container

Uma imagem é um pacote que contém tudo o necessário para executar uma aplicação, enquanto um container é uma instância em execução dessa imagem.



Liberando espaço

Use `docker rm <container_id>` para remover containers e `docker rmi <image_id>` para remover imagens não utilizadas.

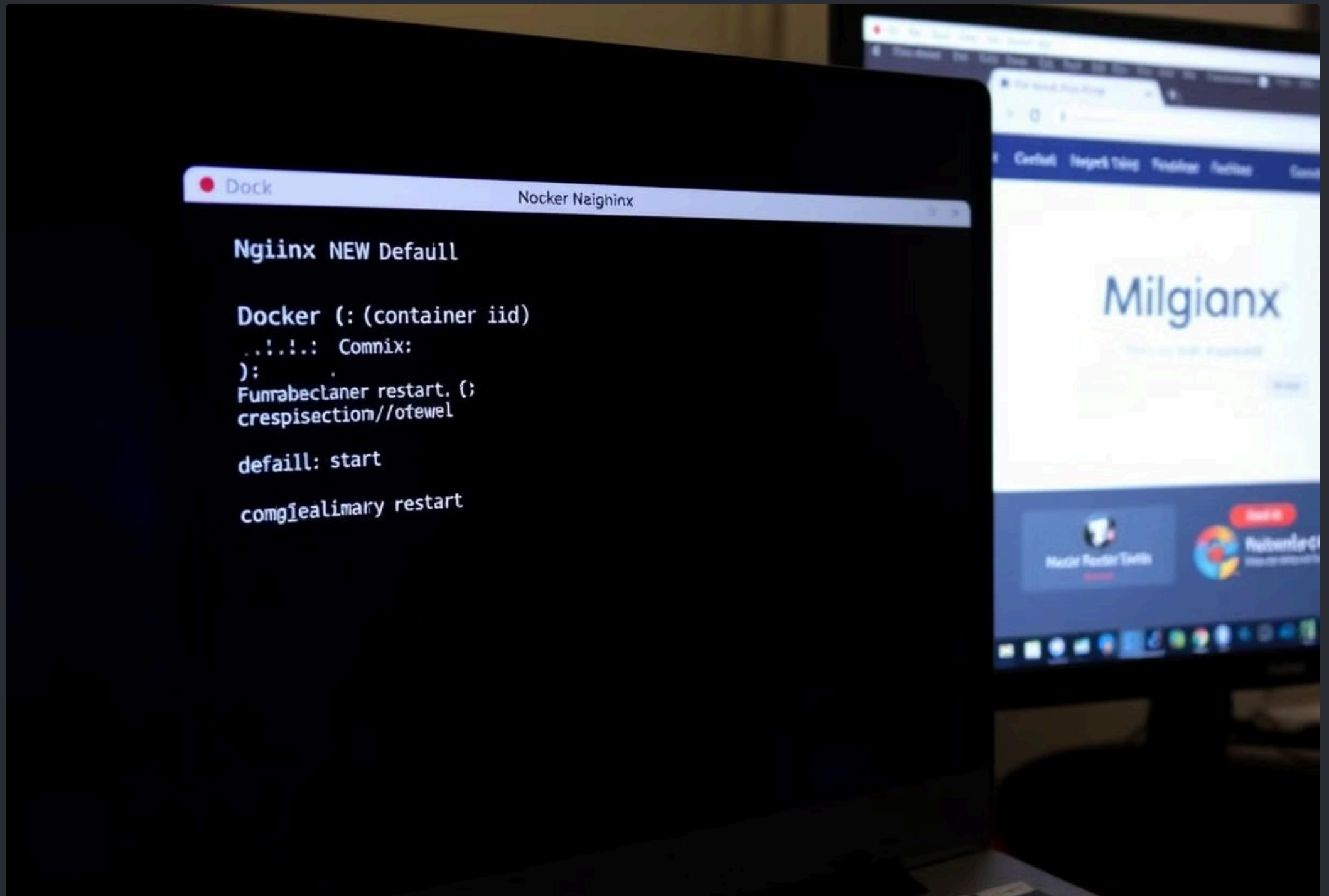
Exercício Prático

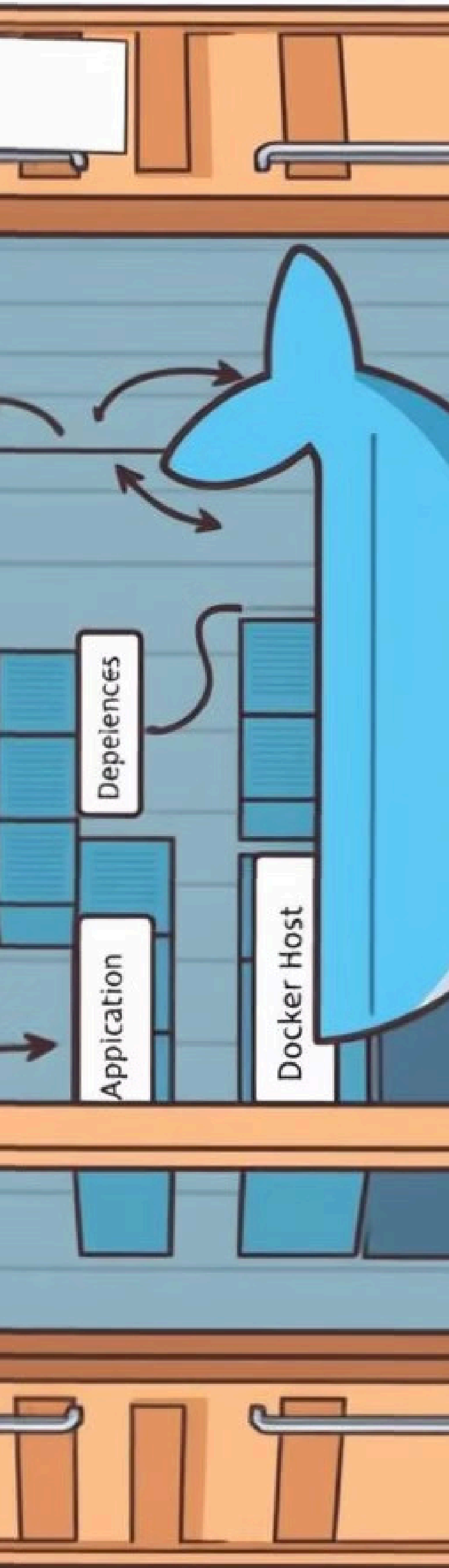
Para consolidar o conhecimento adquirido, siga as instruções abaixo para realizar alguns exercícios práticos com Docker:

1. Execute o comando `docker run hello-world` para verificar se a instalação do Docker está funcionando corretamente.
2. Crie um container Nginx usando `docker run -d -p 8080:80 nginx` e acesse-o no navegador através de `http://localhost:8080`.
3. Liste todos os containers em execução usando `docker ps` e, em seguida, pare o container Nginx com `docker stop <container_id>`.
4. Reinicie o container Nginx utilizando `docker start <container_id>` e verifique se ele está novamente acessível no navegador.
5. Remova o container Nginx usando `docker rm <container_id>` e confirme que ele não está mais presente utilizando `docker ps -a`.

Resolução Comentada dos Exercícios:

1. **Verificação do Docker:** O comando `docker run hello-world` baixará e executará uma imagem de teste que exibirá uma mensagem no terminal. Isso confirma que a instalação do Docker está funcionando corretamente.
2. **Criando um container Nginx:** O comando `docker run -d -p 8080:80 nginx` criará um container em segundo plano, que executará o servidor web Nginx, mapeando a porta 8080 do host para a porta 80 do container. Ao acessar `http://localhost:8080`, você verá a página padrão do Nginx, indicando que o container está funcionando.
3. **Listando e parando o container:** Com o comando `docker ps`, você poderá ver o ID do container Nginx. Em seguida, use `docker stop <container_id>` para parar o container, interrompendo o servidor web.
4. **Reiniciando o container:** O comando `docker start <container_id>` reiniciará o container parado, e você poderá acessar novamente o servidor Nginx pelo navegador para verificar se ele está funcionando corretamente.
5. **Removendo o container:** Para liberar recursos, use `docker rm <container_id>` e, em seguida, `docker ps -a` para confirmar que o container foi removido do sistema.





Roteiro para Aula: Docker - Volumes na Prática

Este documento apresenta um roteiro detalhado para uma aula sobre Docker, com foco em volumes e persistência de dados. A aula abrange desde conceitos básicos até práticas avançadas de gerenciamento de volumes, incluindo exemplos práticos e uma tarefa para os alunos.



por **Everton Coimbra de Araújo**

Introdução ao Docker e Conceitos Iniciais

Docker é uma plataforma que permite criar, distribuir e executar aplicações em containers. Containers são ambientes isolados que contêm tudo que é necessário para executar um software, proporcionando uma forma de garantir que o software rode da mesma maneira em qualquer lugar. Você pode imaginar containers como "mini-máquinas" que rodam sua aplicação com todas as dependências necessárias.

O foco da nossa aula será trabalhar com volumes. Volumes são um componente crítico no Docker para armazenar dados de forma persistente. Eles são a solução que o Docker oferece para garantir que os dados não se percam quando um container é removido ou recriado.

Docker e Persistência de Dados

Na prática, você pode imaginar volumes como "pastas" que estão fora do ciclo de vida do container, garantindo que os dados possam ser usados por outros containers e existam independente de um container específico.

Durante a execução de containers, é comum precisar armazenar dados gerados pela aplicação. Se não houver uma solução de persistência, ao remover um container, todos os dados armazenados também são perdidos. Aqui entram os volumes. Eles são áreas especiais do sistema de arquivos que são montadas dentro de containers e permitem armazenar dados que sobreviverão à recriação dos containers.

Tipos de Volumes no Docker

Volumes Anônimos

Criados automaticamente pelo Docker, geralmente sem que o usuário precise intervir diretamente.

Volumes Nomeados

Volumes que recebem um nome para facilitar seu gerenciamento.

Bind Mounts

Utilizados para mapear uma pasta específica do sistema hospedeiro para o container, oferecendo mais controle sobre onde os dados estão fisicamente armazenados.

Vamos discutir quando faz sentido utilizar cada um desses volumes e suas diferenças. Por exemplo, volumes anônimos são muito utilizados em containers temporários, enquanto bind mounts são comuns em ambientes de desenvolvimento, onde queremos alterar arquivos no hospedeiro e refletir diretamente no container.

Exemplo Prático: Nginx e Volumes

Vamos colocar em prática o exemplo do **nginx**. Primeiro, criaremos um container **nginx** sem volume e faremos uma alteração no arquivo **index.html** para entender a falta de persistência de dados.

Abra o terminal e execute o seguinte comando para criar um container **nginx**:

```
$ docker run -d --name nginx_container -p 8082:80 nginx
```

Com esse comando, estamos criando um container a partir da imagem **nginx** e mapeando a porta **8082** do sistema hospedeiro para a porta **80** do container.

Agora, vamos acessar o container e modificar o arquivo **index.html**. Primeiro, precisamos navegar até o diretório onde o arquivo está localizado e usar um editor de texto. No nosso caso, utilizaremos o **vim**. No entanto, o **vim** pode não estar instalado no container por padrão, então vamos instalá-lo primeiro.

Acesse o container:

```
$ docker exec -it nginx_container bash
```

Instale o **vim** (é necessário que o container tenha acesso à internet para isso):

```
# apt-get update && apt-get install -y vim
```

Após a instalação, navegue até o diretório onde está o arquivo **index.html**:

```
# cd /usr/share/nginx/html
```

Agora, edite o arquivo **index.html** com o **vim**:

```
# vim index.html
```

No **vim**, pressione **i** para entrar no modo de inserção e altere o conteúdo para::

```
<h1>Alteração no index.html</h1>
```

Para salvar e sair do **vim**, siga os seguintes passos:

1. Pressione **Esc** para sair do modo de inserção.
2. Digite **:wq** e pressione **Enter** para salvar e sair.

Podemos sair do container digitando **exit**. Abra o navegador e acesse <http://localhost:8082> para ver a mensagem "**Alteração no index.html**". Agora, vamos remover o container e recriá-lo.

```
$ docker rm -f nginx_container  
$ docker run -d --name nginx_container -p 8082:80 nginx
```

Acesse novamente **http://localhost:8082** e você verá que a alteração foi perdida, pois o container foi recriado sem persistir os dados.

Aqui, uma alternativa ao uso do **vim** é utilizar o comando **echo** para alterar diretamente o conteúdo do arquivo **index.html**. Isso é útil quando você deseja uma modificação rápida e direta, sem a necessidade de um editor de texto.

Por exemplo, para alterar o arquivo **index.html**, execute o seguinte comando dentro do container:

```
# echo "Alteração rápida com echo" > /usr/share/nginx/html/index.html
```

Este comando irá substituir todo o conteúdo do arquivo **index.html** pela mensagem "**Alteração rápida com echo**". Diferente do **vim**, que permite edição parcial do conteúdo, o **echo** sobrescreve tudo, então deve ser usado com cuidado.

Se compararmos com o conteúdo anterior, que foi editado com **vim**, a principal diferença aqui é a simplicidade e rapidez do **echo**, mas ele não permite edições complexas ou manutenção do conteúdo existente. Se precisarmos apenas inserir uma nova linha, poderíamos usar **>>** ao invés de **>** para adicionar ao conteúdo existente:

```
# echo "Nova linha adicionada com echo" >> /usr/share/nginx/html/index.html
```

Este comando adiciona uma nova linha ao final do arquivo, sem remover o conteúdo já presente.

Mapeando Volumes Diretamente na Linha de Comando

Antes de criarmos volumes nomeados, vamos começar mapeando volumes diretamente na linha de comando. Esse método é prático e útil para situações rápidas em que desejamos garantir a persistência dos dados sem criar volumes explicitamente. Podemos fazer isso utilizando o parâmetro **-v** ao criar um container.

Por exemplo, vamos criar um container **nginx** e mapear um volume diretamente da seguinte forma:

```
$ docker run -d --name nginx_container -p 8082:80 -v $(pwd)/html:/usr/share/nginx/html nginx
```

Neste comando, estamos mapeando o diretório **html** do sistema hospedeiro (que deve existir previamente) para o diretório **/usr/share/nginx/html** dentro do container. Dessa forma, qualquer alteração que fizermos no diretório **html** do nosso computador será refletida diretamente no container.

Agora, acesse o diretório mapeado e crie um arquivo **index.html**:

```
$ echo "Conteúdo mapeado diretamente do sistema hospedeiro" > html/index.html
```

Acesse **http://localhost:8082** e veja o conteúdo renderizado diretamente do volume mapeado. Se fizermos qualquer alteração no arquivo **index.html** no sistema hospedeiro, a mudança será refletida imediatamente no container, sem a necessidade de qualquer modificação adicional no Docker.

Utilizando Volumes para Persistir Dados

Agora, vamos utilizar volumes nomeados para garantir a persistência da alteração no arquivo **index.html**. Diferente do mapeamento direto que vimos anteriormente, volumes nomeados oferecem uma solução mais organizada e replicável, sendo especialmente úteis em ambientes de produção, onde a consistência dos dados é essencial.

Crie um volume nomeado:

```
$ docker volume create nginx_volume
```

Em seguida, crie um container **nginx** associando o volume criado:

```
$ docker run -d --name nginx_container -p 8080:80 -v nginx_volume:/usr/share/nginx/html nginx
```

Acesse o container e faça a mesma modificação no arquivo **index.html**:

```
$ docker exec -it nginx_container bash
# apt-get update && apt-get install -y vim
# cd /usr/share/nginx/html # vim index.html
```

No **vim**, pressione **i** para entrar no modo de inserção e altere o conteúdo para:

Para salvar e sair do **vim**, pressione **Esc** e depois digite **:wq** seguido de **Enter**.

Bind Mounts na Prática

Agora vamos trabalhar com bind mounts, uma abordagem que permite mapear uma pasta do sistema hospedeiro diretamente para um container. Diferente dos volumes nomeados, que são gerenciados pelo Docker e mais indicados para ambientes de produção, os bind mounts proporcionam um controle mais direto, pois vinculam uma pasta específica do hospedeiro ao container. Essa abordagem é muito útil durante o desenvolvimento, onde as alterações feitas no hospedeiro precisam ser imediatamente refletidas no container, proporcionando agilidade e eficiência no teste e ajuste do código.

Há duas formas principais de utilizar bind mounts: com a opção **-v** ou com a opção **--mount**. A seguir, vamos explorar ambas.

Usando a opção -v

1

Este método é mais compacto e geralmente é usado em situações mais simples ou rápidas. Por exemplo:

```
$ docker run -d --name nginx_container -p 8082:80 -v $(pwd)/html:/usr/share/nginx/html nginx
```

Neste comando, estamos mapeando o diretório **html** do sistema hospedeiro (que deve existir previamente) para o diretório **/usr/share/nginx/html** dentro do container. Dessa forma, qualquer alteração que fizermos no diretório **html** do nosso computador será refletida diretamente no container.

2

Usando a opção --mount

Esta abordagem oferece uma sintaxe mais explícita e flexível, sendo útil quando há necessidade de múltiplas opções ou quando se quer mais clareza. Por exemplo:

```
$ docker run -d --name nginx_container -p 8082:80 --mount type=bind,source=$(pwd)/html,target=/usr/share/nginx/html nginx
```

Aqui, usamos **type=bind** para especificar que estamos utilizando um bind mount, seguido de **source=**, que indica o caminho do sistema hospedeiro, e **target=**, que define o destino no container. Essa forma pode ser mais fácil de entender e gerenciar quando se lida com configurações mais complexas.

Exemplos Comparativos

Em termos de escolha, a recomendação é utilizar **-v** para situações de desenvolvimento ou experimentação rápida, pois a sintaxe é mais curta e direta. Já o **--mount** deve ser preferido em ambientes onde a clareza e a precisão são fundamentais, como em ambientes de produção ou quando se trabalha com equipes, pois facilita o entendimento e o gerenciamento das configurações.

- A opção **-v** é mais concisa, o que a torna ideal para comandos rápidos e simples durante o desenvolvimento.
- A opção **--mount** é mais detalhada e explícita, o que é vantajoso quando precisamos configurar múltiplas propriedades, como permissões ou pontos de montagem mais específicos.

Podemos refletir sobre as diferenças e quando usar volumes ou bind mounts. Volumes são mais flexíveis e mais fáceis de gerenciar, sendo ideais para ambientes de produção, enquanto bind mounts são úteis em ambientes de desenvolvimento, onde você quer ter acesso direto aos arquivos no sistema hospedeiro.

Gerenciando Volumes e Containers

Para finalizar, é importante aprender a gerenciar volumes e containers, incluindo listar, obter informações e remover volumes.

Para listar todos os volumes existentes no Docker, podemos utilizar o comando:

```
$ docker volume ls
```

Este comando mostrará uma lista de todos os volumes criados, incluindo volumes anônimos e nomeados.

Para obter mais detalhes sobre um volume específico, como o caminho onde ele está armazenado no sistema hospedeiro, use o comando:

```
$ docker volume inspect meu_volume
```

O comando **inspect** retorna informações detalhadas, como o ponto de montagem, o driver utilizado e os containers que estão usando o volume.

Para remover um volume, podemos utilizar o comando:

```
$ docker volume rm meu_volume
```

É sempre bom lembrar que, ao remover um volume, todos os dados contidos nele serão apagados, por isso, é essencial garantir que não existem dados importantes antes de executar esse comando.

Conclusão

Encerrando nossa aula, podemos perceber que os volumes são fundamentais para manter a persistência dos dados nos containers Docker. Eles oferecem uma forma segura e eficiente de trabalhar com dados, principalmente em aplicações que precisam ser escaláveis e resilientes. O uso de bind mounts em desenvolvimento também proporciona agilidade na modificação de arquivos sem necessidade de rebuild constante.

FAQ - "Não Existem Perguntas Idiotas"

Por que precisamos de volumes no Docker?

Volumes garantem que os dados permaneçam mesmo após o container ser removido ou recriado, o que é essencial para aplicações que geram ou manipulam informações que não podem ser perdidas.

Quando utilizar bind mounts ao invés de volumes?

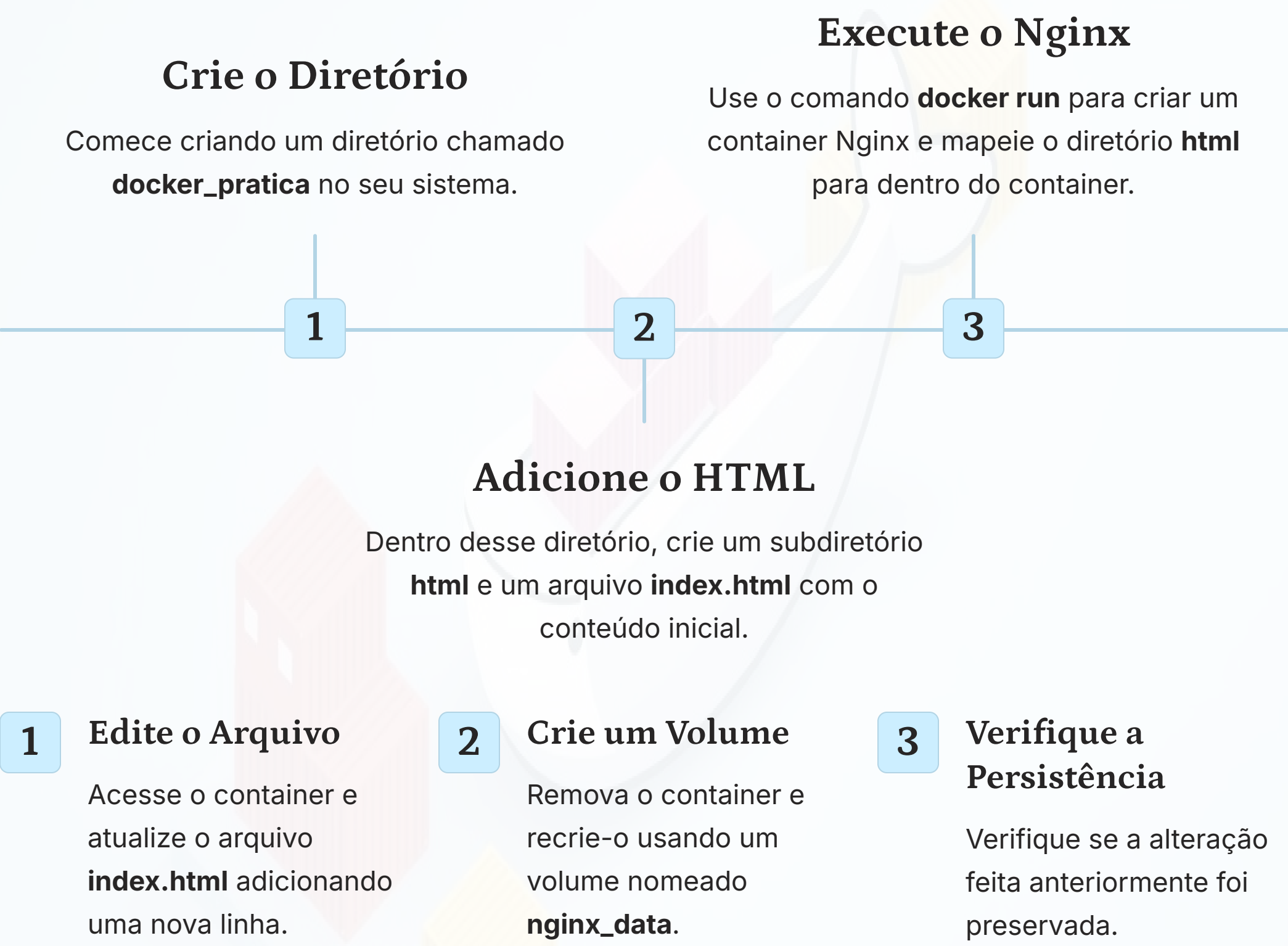
Bind mounts são ideais quando você precisa de acesso direto aos arquivos do sistema hospedeiro, principalmente durante o desenvolvimento.

É seguro usar volumes em produção?

Sim, volumes são seguros e recomendados para produção, especialmente porque o Docker os gerencia de forma eficiente, facilitando backups e migrações.

Tarefa Prática

Para consolidar tudo o que aprendemos sobre Docker e volumes, vamos realizar uma tarefa prática onde você deve aplicar os conceitos trabalhados:



Explore os Volumes	Limpe os Volumes
Liste todos os volumes disponíveis e inspecione o nginx_data .	Para remover um volume, use o comando apropriado.



Respostas Comentadas

1. Para criar o diretório e o subdiretório, utilize os seguintes comandos:

```
$ mkdir -p docker_pratica/html  
$ echo "<h1>Este é o conteúdo inicial</h1>" > docker_pratica/html/index.html
```

Esse passo prepara o ambiente local que será utilizado no container, garantindo que tenhamos um arquivo inicial.

2. Para criar o container nginx utilizando bind mounts, utilize o comando

```
$ docker run -d --name nginx_container -p 8082:80 -v $(pwd)/docker_pratica/html:/usr/share/nginx/html nginx
```

Esse comando cria um container nginx mapeando o diretório **html** do sistema hospedeiro. Assim, qualquer alteração feita localmente será refletida no container.

3. Acesse o container e edite o arquivo index.html:

```
$ docker exec -it nginx_container bash  
# echo "<p>Alteração feita no container</p>" >> /usr/share/nginx/html/index.html
```

Aqui, usamos o **echo** para adicionar uma nova linha ao arquivo **index.html** dentro do container. Essa alteração será temporária, pois estamos utilizando bind mounts.

4. Remova o container e recrie-o utilizando um volume nomeado:

```
$ docker rm -f nginx_container $ docker volume create nginx_data  
$ docker run -d --name nginx_container -p 8082:80 -v nginx_data:/usr/share/nginx/html nginx
```

Ao remover o container e recriá-lo utilizando um volume nomeado, garantimos que os dados persistam, mesmo após a remoção do container.

5. Acesse o container novamente para verificar o conteúdo do index.html:

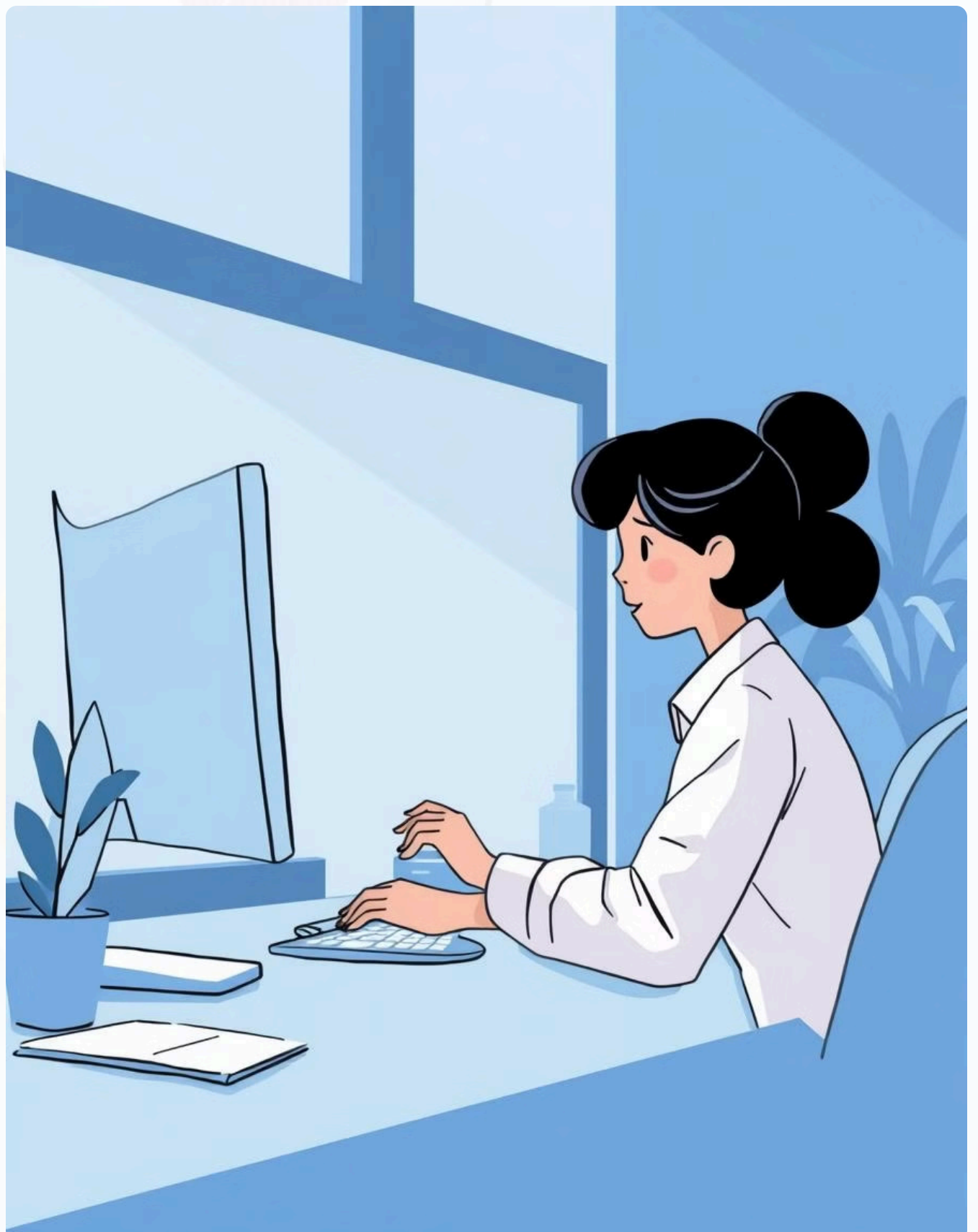
```
$ docker exec -it nginx_container bash  
# cat /usr/share/nginx/html/index.html
```

Você perceberá que o conteúdo anterior não foi preservado, pois o volume nomeado não estava associado inicialmente ao container onde fizemos a alteração. Agora podemos trabalhar com persistência de dados de maneira mais organizada.

6. Liste todos os volumes disponíveis e inspecione o volume nginx_data:

```
$ docker volume ls  
$ docker volume inspect nginx_data
```

Esses comandos permitem verificar todos os volumes criados e inspecionar detalhes como o ponto de montagem e quais containers estão utilizando o volume. É fundamental entender o gerenciamento de volumes para um uso eficiente do Docker.





Docker Hub e Imagens Docker: Explorando o Ecossistema Docker

Nesta aula, exploraremos o Docker Hub e o conceito de imagens Docker, entendendo como utilizar este repositório online essencial para armazenar, compartilhar e gerenciar imagens Docker.

Aprenderemos a trabalhar na prática com imagens Docker, criando, gerenciando e publicando conteúdo, facilitando a colaboração entre equipes e comunidades de desenvolvimento.



por Everton Coimbra de Araújo

Introdução ao Docker Hub e Conceitos Iniciais

Nesta aula, vamos explorar o Docker Hub e o conceito de imagens Docker. O Docker Hub é um repositório online onde você pode armazenar, compartilhar e gerenciar imagens Docker. Ele faz parte essencial do ecossistema Docker e facilita a distribuição e reutilização de imagens, ajudando equipes e comunidades a colaborar de forma mais eficiente. Vamos entender como utilizá-lo e trabalhar na prática com imagens Docker, criando, gerenciando e publicando conteúdo.

Docker Hub e Seus Benefícios

O Docker Hub é mais do que apenas um repositório. Ele nos oferece uma série de funcionalidades que tornam o desenvolvimento de software mais ágil e colaborativo:

1

Compartilhamento Fácil

Você pode compartilhar imagens com outras pessoas ou equipes.

2

Acesso a Imagens Prontas

Disponibiliza milhares de imagens pré-configuradas, facilitando o desenvolvimento.

3

Automatização

Possui integração com sistemas de CI/CD, permitindo builds automáticos.

4

Repositórios Privados

Possui planos pagos que oferecem repositórios privados, garantindo maior controle e segurança.

Empresas também podem optar por configurar repositórios privados para maior segurança, seja no Docker Hub ou por meio do Docker Registry.



Imagens Docker e Sua Importância

Uma imagem Docker é um pacote imutável que inclui tudo o que uma aplicação precisa para ser executada: código, runtime, bibliotecas e dependências. Ao utilizá-las, garantimos:

Portabilidade

Aplicativos funcionam da mesma forma em qualquer lugar.

Consistência

Eliminamos problemas de "funciona na minha máquina".

Eficiência

Imagens são rápidas de construir e iniciar, facilitando o desenvolvimento e implantações.

Trabalhando com Imagens Docker

Vamos aprender a trabalhar com imagens Docker utilizando alguns comandos essenciais:

- **docker pull ubuntu:** Este comando baixa a imagem oficial do Ubuntu do Docker Hub, servindo para obter uma imagem base para criar containers ou outras imagens.
- **docker images:** Este comando lista todas as imagens Docker armazenadas localmente, permitindo que você visualize as imagens disponíveis no sistema.
- **docker rmi <image>:** Utilizado para remover uma imagem Docker específica, liberando espaço no disco.

Construindo Imagens Personalizadas com Dockerfile

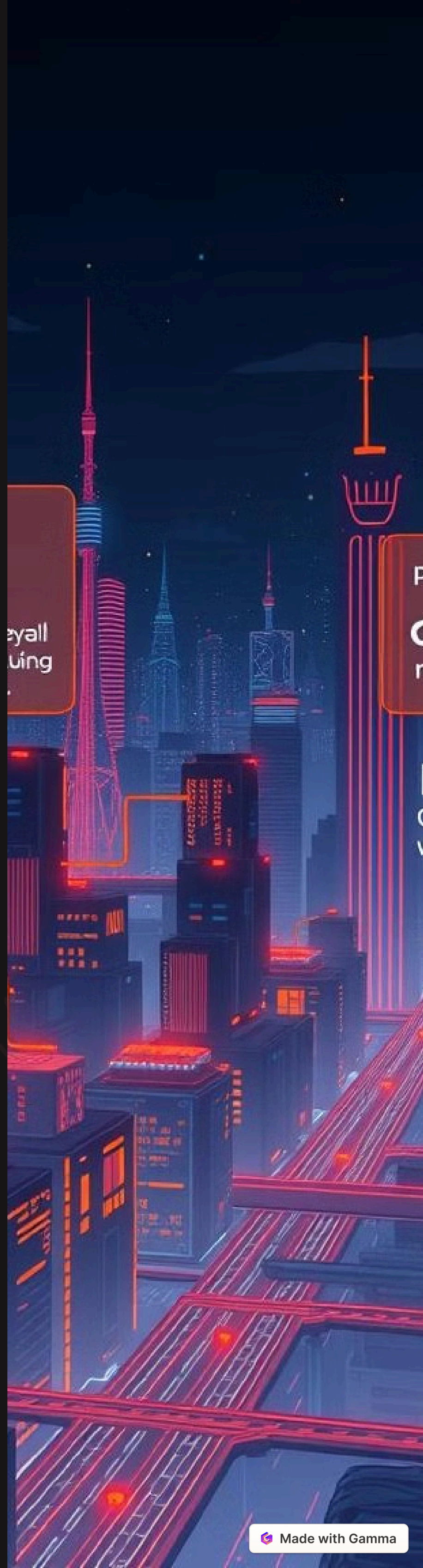
Uma imagem personalizada é importante quando precisamos adaptar um ambiente às necessidades específicas de uma aplicação. Ao criar uma imagem personalizada, podemos incluir apenas as dependências necessárias, otimizando o uso de recursos e garantindo que todas as ferramentas e configurações estejam presentes, proporcionando um ambiente controlado e consistente. Para criar imagens personalizadas, usamos um **Dockerfile**, um arquivo de texto que define cada passo para a construção de uma imagem. Vamos criar um exemplo prático onde instalamos o **Vim** em uma imagem baseada no **Nginx**:

```
# Use uma imagem base do Nginx
FROM nginx:latest

# Instale o Vim
RUN apt-get update && apt-get install -y vim

# Comando padrão para iniciar o Nginx
CMD ["nginx", "-g", "daemon off;"]
```

O comando **CMD** define o que será executado quando o container for iniciado. Neste caso, o Nginx é executado em primeiro plano para que o Docker possa gerenciar o ciclo de vida do container.



Criando e Executando uma Imagem Personalizada

Para construir a imagem personalizada a partir do Dockerfile, utilizamos o comando:

```
$ docker build -t evertoncoimbradearaujo/nginx-com-vim:latest .
```

Neste comando:

- **-t**: Nomeia e marca a imagem, facilitando o gerenciamento e a publicação.
- **evertoncoimbradearaujo/nginx-com-vim:latest**: Nome e tag atribuídos à imagem.
- **.**: Indica o diretório atual como o contexto de construção.

Depois de construir a imagem, podemos executar um container a partir dela usando o comando:

```
$ docker run -it -d -p 8082:80 --name nginx-com-vim evertoncoimbradearaujo/nginx-com-vim
```

Este comando cria um container interativo em segundo plano, mapeando a porta 80 do container para a porta 8082 do sistema hospedeiro. Assim, podemos verificar se o Nginx e o Vim estão funcionando conforme o esperado.

Para acessar o terminal de um container em execução, utilizamos:

```
$ docker exec -it nginx-com-vim /bin/bash
```

Este comando permite que você entre no container e verifique ou modifique arquivos manualmente.

Publicando uma Imagem no Docker Hub

Uma vez que criamos uma imagem, podemos publicá-la no Docker Hub para compartilhá-la com outras pessoas ou com nossa equipe:

1

Fazer login no Docker Hub

```
$ docker login
```

2

Marcar a imagem

```
$ docker tag evertoncoimbradearaujo/nginx-com-vim:latest  
evertoncoimbradearaujo/nginx-com-vim:latest
```

3

Enviar a imagem

```
$ docker push  
evertoncoimbradearaujo/nginx-com-vim:latest
```

Conclusão

Encerrando nossa aula, vimos que o Docker Hub é uma ferramenta essencial para armazenar e compartilhar imagens Docker, e que as imagens personalizadas permitem criar ambientes ajustados às necessidades específicas de nossas aplicações. Usar essas ferramentas de forma eficiente melhora a colaboração e garante consistência entre os ambientes de desenvolvimento, teste e produção.

FAQ – "Não Existem Perguntas Idiotas"

Por que precisamos do Docker Hub?

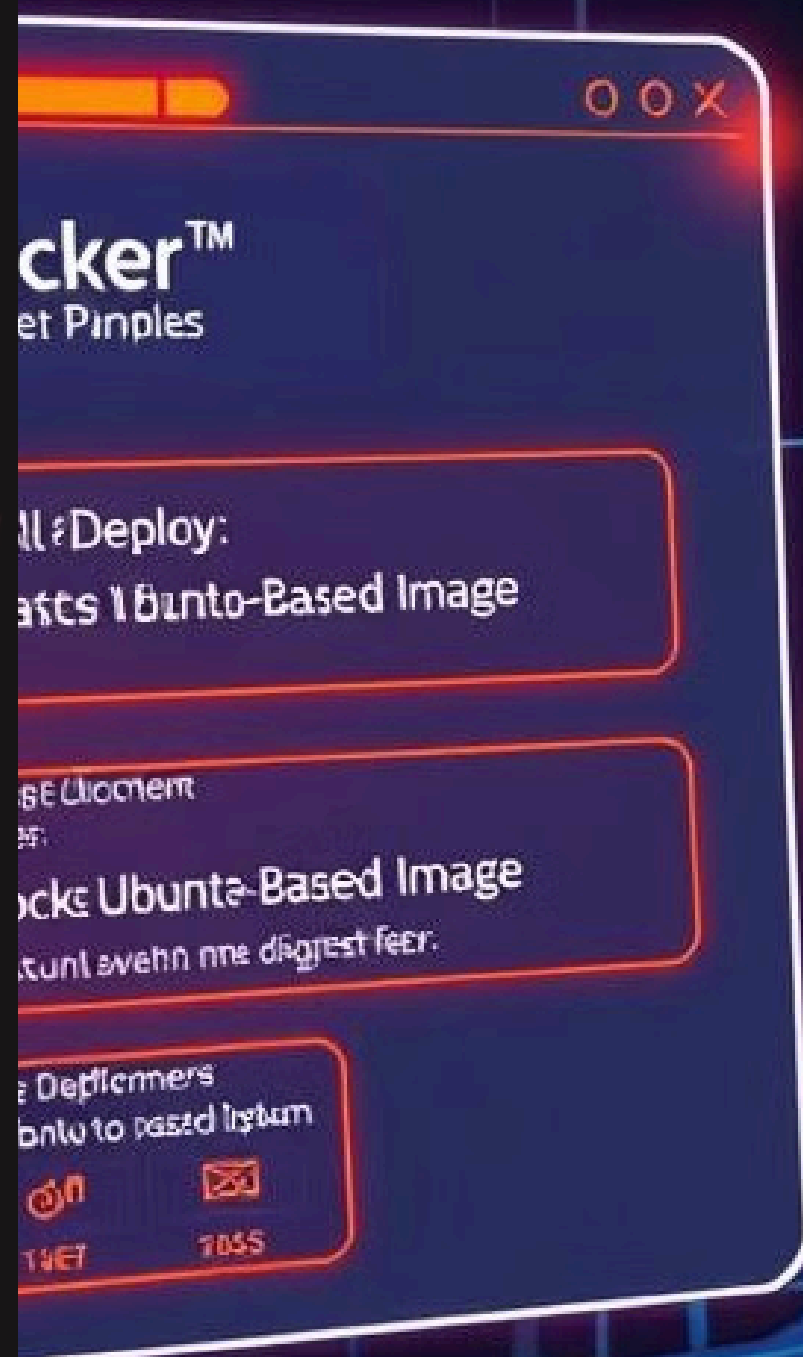
O Docker Hub permite compartilhar imagens, reutilizar soluções e colaborar de forma eficiente com outros desenvolvedores.

Qual a diferença entre uma imagem e um container?

A imagem é um modelo para criar containers. O container é uma instância em execução de uma imagem.

Quando usar repositórios privados?

Repositórios privados são recomendados para imagens que contêm informações sensíveis ou que não devem ser compartilhadas publicamente.



Tarefa Prática

Para consolidar o que aprendemos, vamos realizar uma tarefa prática:

Crie sua própria imagem Docker

Construa uma imagem personalizada baseada no Ubuntu, instalando o editor nano.

Verifique a instalação do nano

Acesse o bash do container e confirme que o nano foi instalado corretamente.

Limpe suas imagens locais

Liste as imagens locais e remova aquelas que não são mais necessárias.

1

2

3

4

5

Construa e execute sua imagem

Construa a imagem usando o comando docker build e execute um container a partir dela, mapeando a porta 8083 do sistema hospedeiro para a porta 80 do container.

Publique sua imagem no Docker Hub

Publique sua imagem personalizada no Docker Hub utilizando seu nome de usuário.



Respostas Comentadas

Para criar o Dockerfile com nano instalado, você deve usar:

```
FROM ubuntu:latest  
RUN apt-get update && apt-get install -y nano  
CMD ["/bin/bash"]
```

Para construir a imagem:

```
$ docker build -t meuusuario/ubuntu-com-nano:latest .
```

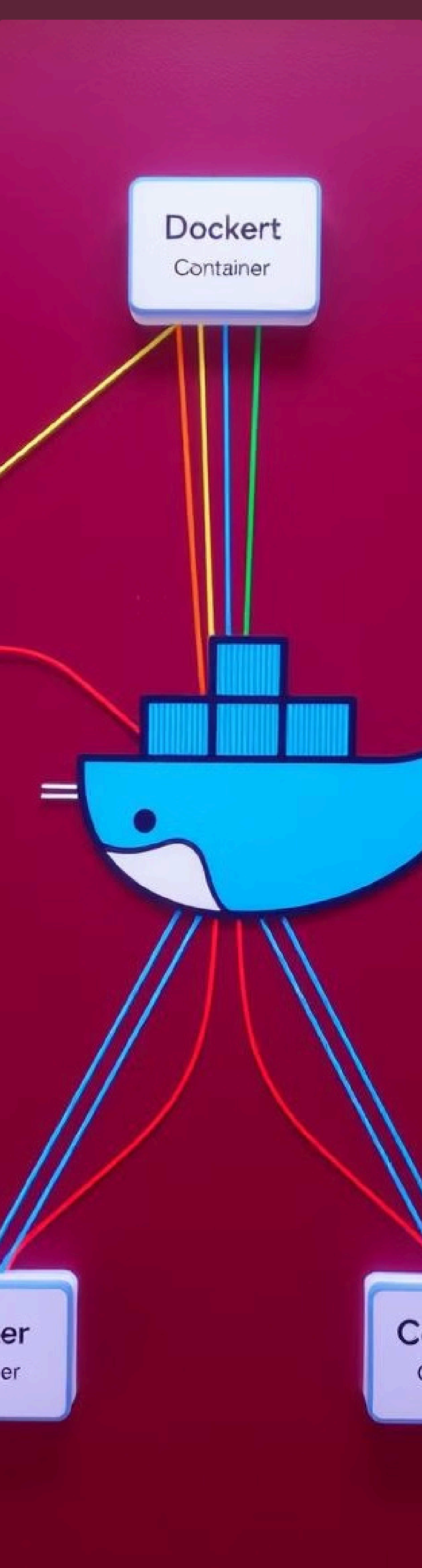
Para executar o container:

```
$ docker run -it -d -p 8083:80 --name ubuntu-com-nano meuusuario/ubuntu-com-nano
```

Para acessar o bash do container e verificar o nano:

```
$ docker exec -it ubuntu-com-nano /bin/bash
```

Com isso, reforçamos os conceitos trabalhados e praticamos a criação, execução e publicação de imagens Docker.



Introdução ao Uso de Networks em Docker

Este documento apresenta uma introdução abrangente ao uso de networks em Docker, cobrindo conceitos básicos, tipos de redes, comandos essenciais e exercícios práticos para entender a conectividade entre containers. O conteúdo aborda desde a definição de host no contexto do Docker até a criação e uso de redes personalizadas, fornecendo uma base sólida para o gerenciamento eficiente de redes em ambientes Docker.



por **Everton Coimbra de Araújo**

Introdução às Networks em Docker

Docker oferece um sistema de redes robusto e flexível que permite que containers se comuniquem entre si, com o host e com o mundo externo. Em termos simples, uma network em Docker é uma interface de comunicação que conecta containers e possibilita o tráfego de dados entre eles, garantindo que as aplicações funcionem de maneira coesa e integrada.

As networks são essenciais para a comunicação e conectividade em aplicações containerizadas, pois elas possibilitam a construção de arquiteturas de software complexas e escaláveis. Sem um sistema de redes bem definido, containers não poderiam se comunicar de forma eficiente, o que prejudicaria a funcionalidade e a performance das aplicações distribuídas.

Definição de Host no Contexto do Docker

No contexto do Docker, o termo "host" refere-se ao sistema operacional e hardware físico ou virtual em que o Docker Engine está instalado e em execução. O host é responsável por gerenciar os recursos do sistema, como CPU, memória, armazenamento e rede, e disponibilizá-los para os containers. Em outras palavras, o host é o ambiente que hospeda os containers, fornecendo a infraestrutura necessária para que eles possam ser executados.

Um exemplo prático de um host pode ser uma máquina física, como um servidor ou um computador pessoal, ou uma máquina virtual, como uma instância em uma nuvem pública (AWS, Azure, Google Cloud). O host Docker atua como um intermediário entre o sistema operacional subjacente e os containers, garantindo que cada container receba os recursos necessários de maneira isolada e segura.

Comandos Básicos de Network

Antes de explorar os diferentes tipos de redes disponíveis no Docker, é importante entender alguns comandos básicos que ajudam a gerenciar essas redes.

- **docker network:** Utilizado para gerenciar redes no Docker. Com ele, você pode criar, listar, inspecionar, remover e gerenciar redes de várias outras formas.
- **docker network ls:** Lista todas as redes Docker existentes no host. É útil para verificar quais redes estão disponíveis e entender a topologia de rede atual.
- **docker network prune:** Remove todas as redes não utilizadas no Docker. Este comando é útil para limpar redes desnecessárias e liberar recursos.

Tipos de Networks Docker

Bridge Network

Rede padrão criada pelo Docker quando nenhum parâmetro específico de rede é fornecido. Ela permite que containers em um único host se comuniquem entre si. A Bridge Network é útil para cenários onde você deseja conectar vários containers no mesmo host, permitindo que eles se comuniquem por meio de endereços IP internos ou usando o nome do container. Ela cria uma rede interna isolada do host e é adequada para ambientes de desenvolvimento ou aplicações simples que não precisam se comunicar com o mundo exterior diretamente.

Host Network

Remove a camada de isolamento de rede entre o container e o host, permitindo que o container compartilhe a rede do host. A Host Network é ideal para situações em que o desempenho da rede é essencial, já que elimina a sobrecarga de virtualização da camada de rede. É frequentemente utilizada em casos onde o container precisa acessar dispositivos de rede do host diretamente, ou em aplicações que necessitam de baixa latência. No entanto, isso também implica em menor isolamento, já que o container passa a ter acesso direto à rede do host.

Overlay Network

Usada para conectar múltiplos Docker daemons em um cluster, essencial para setups de Docker Swarm. A Overlay Network é fundamental quando se trabalha com clusters Docker, especialmente com Docker Swarm ou Kubernetes. Ela permite que containers em diferentes hosts se comuniquem como se estivessem na mesma rede local, o que é crucial para a criação de serviços distribuídos que precisam de alta disponibilidade e escalabilidade. A Overlay Network é frequentemente usada para serviços que precisam se comunicar entre diferentes máquinas em um cluster.

Macvlan Network e None Network

A Macvlan Network atribui um endereço MAC a cada container, fazendo com que ele apareça como um dispositivo físico na rede. É útil quando você deseja que seus containers sejam tratados como dispositivos físicos na rede. A None Network desativa completamente a rede do container, sendo utilizada quando você deseja isolar completamente um container, impedindo qualquer tipo de comunicação externa.

Antes de prosseguirmos, vale reforçar que, embora tenhamos todos estes tipos de redes disponíveis, nosso objetivo será trabalhar com a Bridge Network, por ser mais simples e atender bem ao nosso propósito nesta aula.

Explorando a Conectividade entre Containers em Docker

Neste exercício, vamos explorar a conectividade entre containers no Docker utilizando a rede bridge. Vamos criar dois containers, inspecionar a rede para verificar suas configurações e testar a comunicação entre eles.

Objetivo

O objetivo deste exercício é:

- Criar dois containers Docker em modo interativo.
- Inspecionar a rede bridge padrão do Docker.
- Verificar a conectividade entre os containers usando endereços IP e nomes de containers.

Procedimento



Criação e Uso de Redes Personalizadas no Docker

Além das redes padrão que o Docker cria automaticamente, podemos criar redes personalizadas para atender a necessidades específicas, garantindo maior flexibilidade e controle sobre a comunicação entre containers.

Criando uma Rede Personalizada

Podemos criar uma rede personalizada utilizando o comando `docker network create`. Isso nos permite especificar o driver e outras configurações de rede.

```
$ docker network create minha_rede_personalizada
```

Neste exemplo, criamos uma rede chamada **minha_rede_personalizada** usando o driver `bridge`.

Conectando Containers a uma Rede Personalizada

Depois de criar uma rede, podemos conectar containers a ela, permitindo que esses containers se comuniquem de forma mais eficiente.

```
$ docker run -d -it --name ubuntu1 --network  
minha_rede_personalizada ubuntu  
$ docker run -d -it --name ubuntu2 --network  
minha_rede_personalizada ubuntu
```

Com isso, os containers **ubuntu1** e **ubuntu2** estarão conectados à rede `minha_rede_personalizada` e poderão se comunicar utilizando o nome do container.



Adicionando um Container a uma Rede Existente

Também podemos adicionar um novo container a uma rede já existente, facilitando a expansão da infraestrutura de comunicação entre containers. Por exemplo, suponha que já tenhamos uma rede personalizada `minha_rede_personalizada` e queremos adicionar um novo container `ubuntu3` a essa rede.

1 Criar o Novo Container

Primeiro, crie o container `ubuntu3` em modo desacoplado:

```
docker run -d --name ubuntu3 ubuntu
```

2 Conectar o Novo Container à Rede Existente

Conecte o container `ubuntu3` à rede `minha_rede_personalizada`:

```
docker network connect minha_rede_personalizada ubuntu3
```

Agora, o container `ubuntu3` está conectado à rede **`minha_rede_personalizada`** e pode se comunicar com os outros containers na mesma rede.

Conclusão e Tarefa Prática

Nesta aula, exploramos como funcionam as redes em Docker, incluindo os tipos de networks disponíveis, como criar e utilizar redes personalizadas, e como adicionar containers a redes existentes. Compreender as diferentes opções de rede no Docker é fundamental para criar ambientes de aplicação que sejam escaláveis, eficientes e fáceis de gerenciar. As redes Docker nos permitem definir claramente como os containers devem se comunicar entre si, proporcionando um controle detalhado sobre conectividade, isolamento e performance.



FAQ - "Não Existem Perguntas Idiotas"

Por que precisamos de networks no Docker?

Networks garantem que containers possam se comunicar entre si, com o host e com o mundo externo, sendo fundamentais para aplicações distribuídas.

Qual a diferença entre Bridge e Host Network?

A Bridge Network cria uma rede isolada para os containers, enquanto a Host Network compartilha a rede do host com o container.

Tarefa Prática para os Alunos

- 1 Crie uma rede personalizada chamada `rede_fixa`.
- 2 Inicie dois containers (`container1` e `container2`) conectados à `rede_fixa`.
- 3 Adicione um terceiro container (`container3`) à rede `rede_fixa` após a sua criação.
- 4 Verifique a conectividade entre os containers usando o comando `ping`.
- 5 Inspecione a rede para verificar os containers conectados e suas configurações de IP.

Respostas Comentadas

Criação da Rede Personalizada

```
$ docker network create rede_fixa
```

Este comando cria uma nova rede personalizada chamada `rede_fixa` usando o driver `bridge`, permitindo que os containers conectados a ela se comuniquem de maneira mais organizada.

Iniciar os Containers e Conectar à Rede

```
$ docker run -d --name container1 --network rede_fixa ubuntu
```

```
$ docker run -d --name container2 --network rede_fixa ubuntu
```

Estes comandos iniciam dois containers chamados `container1` e `container2` e os conectam à rede `rede_fixa`. Isso garante que ambos estejam na mesma rede e possam se comunicar.

Adicionar um Terceiro Container à Rede

```
$ docker run -d --name container3 ubuntu
```

```
$ docker network connect rede_fixa container3
```

Primeiro, criamos o container `container3` sem conectá-lo a nenhuma rede. Em seguida, usamos o comando `docker network connect` para adicionar `container3` à rede `rede_fixa`. Isso permite adicionar containers a redes existentes mesmo após sua criação.

Verificar a Conectividade entre os Containers

```
$ docker exec -it container1 ping container2
```

```
$ docker exec -it container1 ping container3
```

Utilizamos o comando `ping` para testar a conectividade entre `container1` e os outros containers (`container2` e `container3`). Se a configuração da rede estiver correta, o ping deve responder sem problemas, indicando que os containers conseguem se comunicar.

Inspecionar a Rede

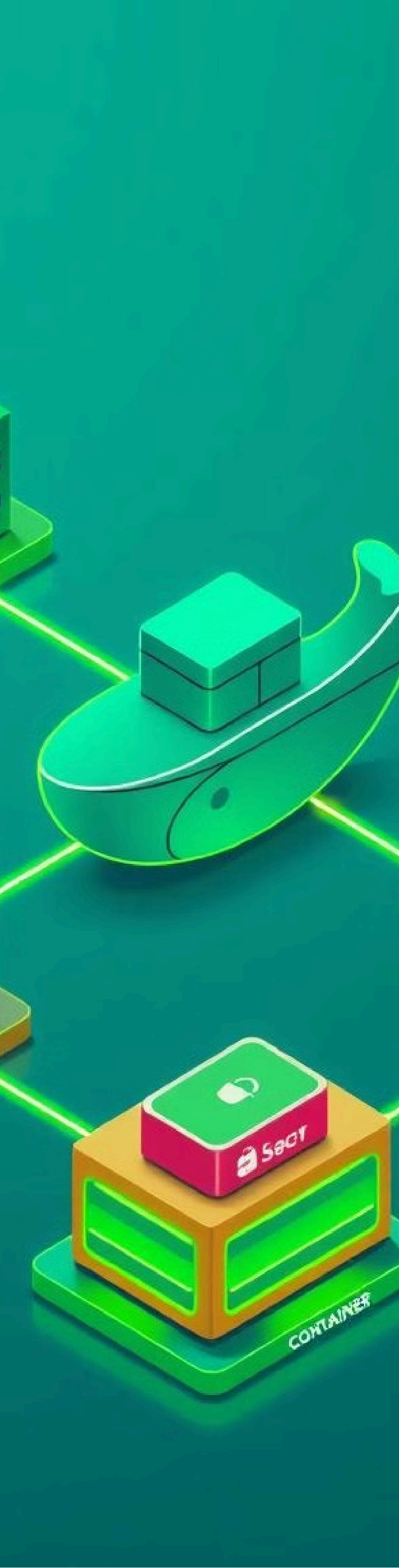
```
$ docker network inspect rede_fixa
```

O comando `docker network inspect` fornece detalhes sobre a rede `rede_fixa`, incluindo os containers conectados e seus endereços IP. Essa inspeção é útil para verificar a topologia da rede e garantir que todos os containers estejam corretamente conectados.

Docker Compose: Guia Completo

Este guia completo sobre Docker Compose aborda desde conceitos básicos até boas práticas e considerações de segurança. Ele explica como definir e gerenciar ambientes multi-container Docker de forma declarativa, utilizando um arquivo YAML para orquestrar serviços. O documento inclui exemplos práticos, comandos essenciais e um exercício para consolidar o aprendizado.

por Everton Coimbra de Araújo



Introdução ao Docker Compose

Docker Compose é uma ferramenta que permite definir e gerenciar ambientes multi-container Docker de forma declarativa. Com Docker Compose, você pode definir um ambiente multi-container em um único arquivo YAML, facilitando a orquestração e automação dos serviços que compõem a sua aplicação. É especialmente útil para configurar, iniciar e gerenciar aplicações que dependem de múltiplos containers, como um servidor web, um banco de dados e um serviço de cache.

Conceitos Básicos do Docker Compose

Serviços

Cada container em uma aplicação Docker Compose é definido como um serviço. Um serviço pode ser um banco de dados, um servidor web ou qualquer outro componente da sua aplicação.

Volumes

Volumes são usados para persistir dados entre os containers e o host. Eles permitem que os dados não sejam perdidos quando os containers são removidos ou recriados.

Redes

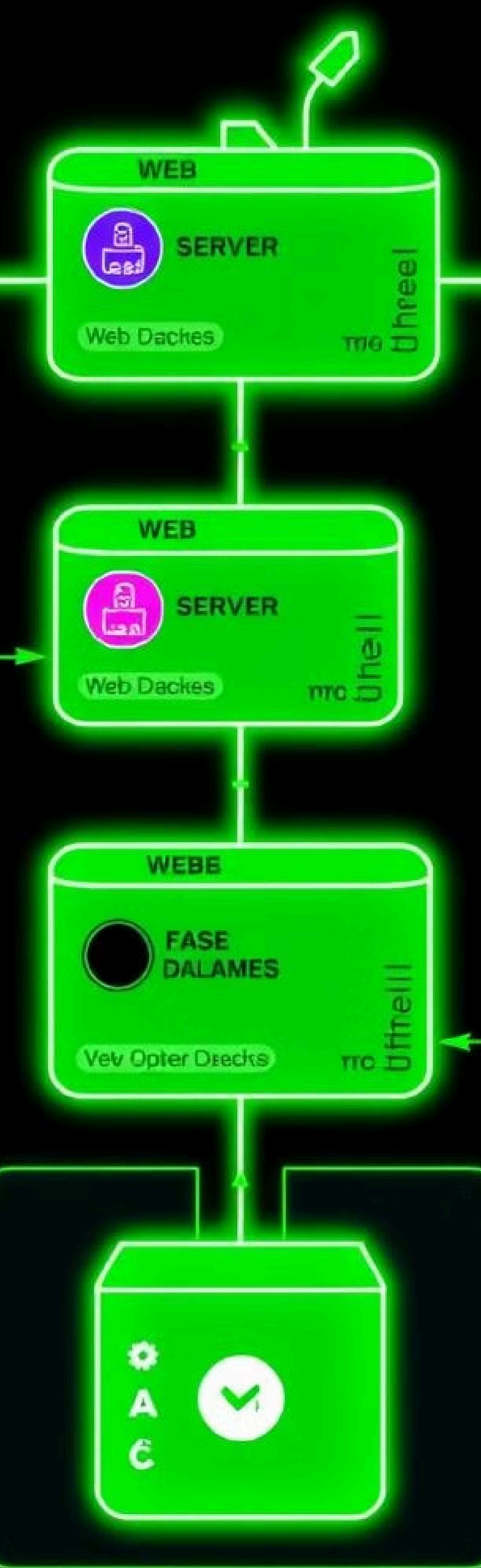
Docker Compose configura redes para permitir que os serviços se comuniquem entre si. Por padrão, os serviços na mesma rede podem se comunicar usando seus nomes de serviço.

Estrutura de um Arquivo docker-compose.yml

O arquivo docker-compose.yml é onde você define a configuração da sua aplicação. Aqui está um exemplo básico:

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "8082:80"
    volumes:
      - ./web:/usr/share/nginx/html
    networks:
      - mynetwork
  database:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - mynetwork
networks:
  mynetwork:
volumes:
  db_data:
```

O código acima define dois serviços, **web** e **database**, que utilizam o servidor web **Nginx** e o banco de dados **MySQL**, respectivamente. Além disso, cria uma rede chamada **mynetwork** para conectar os dois serviços, e um volume **db_data** para persistir os dados do MySQL, garantindo que eles não sejam perdidos se o container for removido ou recriado. Veja a seguir a descrição dos componentes do arquivo.



Descrição do Arquivo docker-compose.yml

- **version:** Define a versão do Docker Compose. Aqui usamos a versão 3.
- **services:** Define os serviços que compõem a aplicação.
 - **web:** Serviço para o servidor web Nginx.
 - **image:** Imagem Docker a ser usada.
 - **ports:** Porta mapeada do host para o container.
 - **volumes:** Volume montado do host para o container.
 - **networks:** Rede na qual o serviço está conectado.
 - **database:** Serviço para o banco de dados MySQL.
 - **image:** Imagem Docker a ser usada.
 - **environment:** Variáveis de ambiente para o serviço.
 - **volumes:** Volume montado do host para o container.
 - **networks:** Rede na qual o serviço está conectado.
- **networks:** Define as redes utilizadas pelos serviços.
- **volumes:** Define os volumes utilizados pelos serviços.

Comandos Básicos do Docker Compose

Os comandos básicos do Docker Compose são essenciais para gerenciar o ciclo de vida dos serviços definidos no arquivo docker-compose.yml. Com eles, você pode iniciar, parar, construir e monitorar os containers que compõem sua aplicação de forma prática e eficiente. A seguir, veremos alguns dos principais comandos usados para interagir com ambientes multi-container, explicando o que cada um faz, o motivo de utilizá-los e o que esperar de cada operação.

docker compose up

1

O comando docker compose up cria e inicia todos os serviços definidos no arquivo docker-compose.yml. Ele é utilizado para iniciar todo o ambiente de uma aplicação com apenas um comando, facilitando a automação e a orquestração dos containers.

docker compose build --no-cache

3

O comando docker compose build constrói ou reconstrói as imagens dos serviços definidos no arquivo docker-compose.yml. Ele é utilizado principalmente quando houve alterações no código ou nas dependências dos serviços, e você precisa refletir essas mudanças ao iniciar os containers.

A opção **--no-cache** garante que a construção das imagens seja feita sem utilizar o cache de etapas anteriores, o que é útil quando se deseja garantir que tudo seja baixado e recompilado do zero, evitando a reutilização de camadas que possam estar desatualizadas.

Utilizar este comando é essencial para garantir que os containers serão executados com as versões atualizadas do código ou das dependências, evitando problemas de inconsistência no ambiente.

2

docker compose down

O comando docker compose down é utilizado para parar e remover todos os containers, redes e volumes criados com o docker compose up. Ele é importante quando você quer liberar recursos, interromper o ambiente, ou garantir que tudo seja limpo após o uso.

Testando os Serviços do Arquivo docker-compose.yml

Para testar os dois serviços definidos no arquivo docker-compose.yml — o servidor web Nginx e o banco de dados MySQL — siga os passos abaixo:

1. Testar o Serviço Web (Nginx):

- Após executar o comando `docker compose up -d`, abra um navegador e acesse `http://localhost:8082`. Se tudo estiver correto, você deverá ver a página padrão do Nginx ou qualquer conteúdo que tenha colocado no diretório mapeado `./web`.

2. Testar o Serviço de Banco de Dados (MySQL):

- Verifique se o container do MySQL está em execução utilizando o comando:

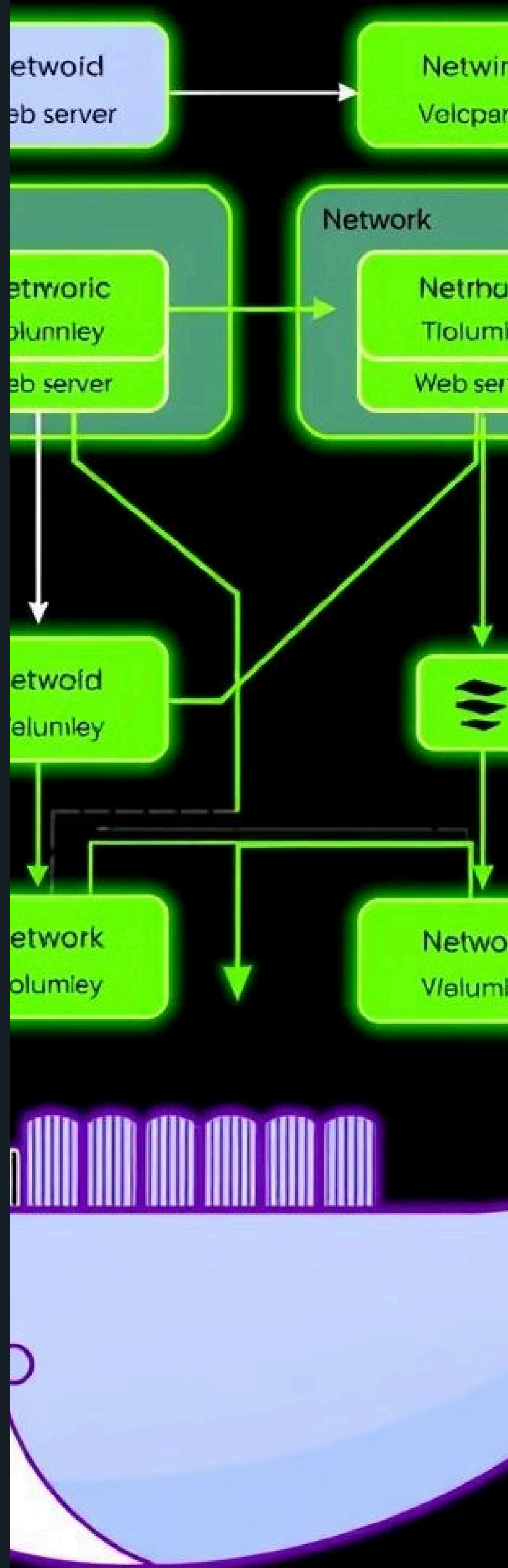
```
docker compose ps
```

- Em seguida, conecte-se ao container do MySQL para verificar se ele está funcionando corretamente:

```
docker exec -it mysql -p
```

Substitua pelo nome do container exibido pelo comando `docker compose ps`. Você será solicitado a inserir a senha que foi definida como **example**. Após a autenticação, você poderá executar comandos SQL para testar o funcionamento do banco de dados.

Ribcited sacapsie



Comandos para Testar o Banco de Dados MySQL

1. Listar Bancos de Dados:

```
SHOW DATABASES;
```

Este comando permite visualizar todos os bancos de dados disponíveis no MySQL. É útil para garantir que a conexão foi bem-sucedida e o banco de dados está acessível.

1. Criar um Novo Banco de Dados:

```
CREATE DATABASE test_db;
```

Este comando cria um novo banco de dados chamado test_db, ajudando a verificar se o MySQL está permitindo operações de escrita.

1. Usar um Banco de Dados:

```
USE test_db;
```

Com este comando, você muda o contexto para o banco de dados test_db criado anteriormente.

1. Criar uma Tabela:

```
CREATE TABLE users (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(100),  
  email VARCHAR(100)  
);
```

Este comando cria uma tabela chamada users no banco de dados test_db, permitindo testar a criação de estruturas dentro do banco.

1. Inserir Dados na Tabela:

```
INSERT INTO users (name, email) VALUES ('Alice', 'alice@example.com');
```

Este comando insere um registro na tabela users, verificando se o MySQL está aceitando inserções de dados corretamente.

1. Consultar Dados da Tabela:

```
SELECT * FROM users;
```

Este comando retorna todos os registros da tabela users, permitindo verificar se os dados foram inseridos corretamente.

Boas Práticas e Considerações de Segurança

- **Uso de Imagens Oficiais:** Sempre que possível, utilize imagens oficiais do Docker Hub.
- **Gerenciamento de Senhas e Credenciais:** Use variáveis de ambiente para gerenciar senhas e credenciais, e considere usar o Docker Secrets para informações sensíveis.
- **Versão de Compose:** Utilize a versão mais recente do Docker Compose que seja compatível com sua aplicação.
- **Limpeza de Recursos:** Utilize `docker compose down --volumes` para limpar recursos quando não forem mais necessários.
- **Documentação:** Documente o arquivo `docker-compose.yml` de forma clara, incluindo descrições sobre cada serviço e suas dependências. Isso facilita a manutenção futura e permite que outros desenvolvedores entendam rapidamente a configuração e o propósito de cada componente.

Docker Compose é uma ferramenta poderosa para definir e gerenciar ambientes multi-container. Com um único arquivo YAML, você pode orquestrar diversos serviços que compõem a sua aplicação, facilitando o desenvolvimento, testes e deploy. As boas práticas e exercícios aqui apresentados ajudam a consolidar o conhecimento necessário para utilizar o Docker Compose de forma eficaz e segura.

Conclusão

Docker Compose é uma ferramenta poderosa para definir e gerenciar ambientes multi-container. Com um único arquivo YAML, você pode orquestrar diversos serviços que compõem a sua aplicação, facilitando o desenvolvimento, testes e deploy. As boas práticas e exercícios aqui apresentados ajudam a consolidar o conhecimento necessário para utilizar o Docker Compose de forma eficaz e segura.

FAQ - "Não Existem Perguntas Idiotas"

Por que usar Docker Compose em vez de comandos Docker individuais?

Docker Compose facilita a automação e o gerenciamento de ambientes multi-container. Em vez de iniciar cada container individualmente, você define todos os serviços em um único arquivo `docker-compose.yml` e os gerencia com comandos simples, economizando tempo e reduzindo erros.

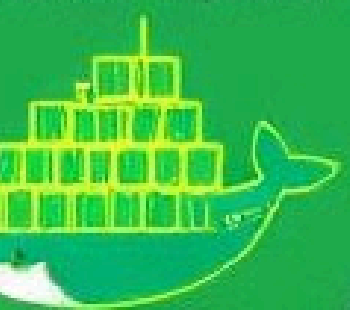
Docker Compose

ser
er compose

porter;
pose to
ose
s.

S

ker, serving.
rent ler detting
for
hencr emplher,



Updasterder
the asserts th
• Dockers cont
inschal pose
that fon the
the read en
poviler and

Perting

Docker composed
Docker Comnant

DOCKER SERVICE

- Docker compose: the command Docker readable the Comage.
- Extended decomp big estlets
- Leased complete indertsill ere. Dockere atome: docker that sponse.

Qual é a diferença entre **docker compose up** e **docker compose start**?

docker compose up cria e inicia os containers a partir do arquivo **docker-compose.yml**, enquanto **docker compose start** apenas inicia containers que já foram criados. Use **up** para iniciar o ambiente desde o início e **start** para reiniciar containers parados.

Posso usar Docker Compose em produção?

Sim, é possível usar Docker Compose em produção, especialmente para pequenos projetos ou em ambientes onde a simplicidade é desejada. No entanto, para ambientes mais complexos e com maior escalabilidade, ferramentas como Kubernetes são mais recomendadas.

Como faço para atualizar um serviço no Docker Compose?

Para atualizar um serviço, você pode modificar o arquivo **docker-compose.yml** e executar **docker compose up -d --build**. Isso reconstruirá a imagem do serviço e reiniciará o container com a nova configuração.

Qual é a função do **--no-cache** no comando **docker compose build**?

A opção **--no-cache** força o Docker a ignorar o cache durante a construção da imagem, garantindo que todos os pacotes e dependências sejam baixados novamente. Isso é útil para garantir que as últimas versões de todas as dependências sejam utilizadas.

Exercício Prático



Respostas Comentadas

Criação do Arquivo `docker-compose.yml`

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  database:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
  cache:
    image: redis:alpine
```

Neste arquivo, definimos três serviços (`web`, `database` e `cache`) usando imagens oficiais. Cada serviço é configurado com suas respectivas imagens e, no caso do MySQL, usamos uma variável de ambiente para definir a senha root.

Iniciar os Serviços

```
docker compose up -d
```

O comando inicia todos os serviços em segundo plano, garantindo que a aplicação esteja em execução e permitindo continuar utilizando o terminal para outros comandos.

Verificar Containers em Execução

```
docker compose ps
```

O comando lista todos os containers que estão em execução, confirmando que todos os serviços foram iniciados corretamente.

Conectar-se ao Container do MySQL

```
docker exec -it <nome_do_container_mysql> mysql -p
```

Utilizamos o comando `docker exec` para acessar o container do MySQL e verificar se ele está funcionando corretamente. Substitua `<nome_do_container_mysql>` pelo nome real do container exibido pelo `docker compose ps`.

Finalizar e Limpar os Serviços

```
docker compose down --volumes
```

Este comando para todos os containers e remove os volumes associados, garantindo que não haja resíduos de dados dos testes realizados.

Docker Compose Command

or staptivadl the

- Eptarits of the a

compasse waitlting

ker cther and

- Shep Dockffer C
Folcter is Becker
Plepcet lint, teet

flet morh
comnanly pacitions.

kestonitallity tine

Gondatter conmad.s.

SEPERTIHC: COMPOSE ENJLURI



UPDAT DOCKER COMPOSE

