IMERSÃO DevOps & Cloud

Aquecimento

E-BOOK

Docker

Primeiros passos

Docker

E-BOOK

Introdução

Nos últimos anos, a maneira como desenvolvemos, implantamos e executamos software passou por transformações drásticas. Hoje, a necessidade de entregar software mais rápido e de maneira mais eficiente nunca foi tão intensa. É aqui que os containers e o Docker entram em jogo.

Containers, na sua essência, são uma forma de empacotar software de maneira que ele possa funcionar de maneira idêntica, independentemente do ambiente em que é executado. Isso significa que se algo funciona no ambiente de desenvolvimento de um desenvolvedor, também funcionará no ambiente de produção, na máquina de outro desenvolvedor, ou em qualquer outra máquina onde seja implantado. E é isso que torna os containers tão poderosos.

O Docker, por outro lado, é a ferramenta que trouxe os containers para uma implementação mais simples, abstraindo a complexidades que era trabalhar com essa tecnologia. Ele oferece uma plataforma e um conjunto de ferramentas para facilitar o desenvolvimento, a distribuição e a execução de aplicativos em containers.

Neste e-book, faremos uma jornada desde o básico – entendendo o que são containers e o Docker, passando pela comparação com máquinas virtuais, explorando a arquitetura do Docker e, finalmente, aprendendo os principais comandos e conceitos associados ao Docker.

Se você é um profissional de desenvolvimento de software ou infraestrutura, mas ainda não teve a oportunidade de mergulhar no universo dos containers, este guia é para você. Ele foi criado para te ajudar a entender e dar os primeiros passos no fascinante mundo do Docker.

O que é container?

Quando falamos a palavra “container” pode vir na sua cabeça grandes estruturas usadas para transportar mercadorias em navios. Mas em tecnologia, no contexto de desenvolvimento e infraestrutura de software, os containers tem um significado completamente diferente, mas igualmente poderoso e com um objetivo bastante parecido.

Um container é, essencialmente, uma camada de isolamento pra executar processos de forma isolada em um ambiente. Nele você tem tudo que se precisa: código, tempo de execução, ferramentas de sistema, bibliotecas e configurações. Essa encapsulação garante que o software tenha consistência e possa ser executado da mesma forma, independentemente de onde o container é implementado.

O legal dos containers é a sua simplicidade e portabilidade. Eu posso te falar algumas caracteristicas importantes nos containers:

- **Isolamento:** Cada container opera de forma independente, assegurando que não haja interferências entre diferentes aplicativos ou serviços.

- **Leveza:** Containers compartilham o mesmo sistema operacional do host, mas funcionam de maneira isolada. Isso os torna muito mais eficientes em termos de recursos em comparação com as máquinas virtuais.

- **Consistência:** Como mencionado, a natureza encapsulada de um container garante que o software funcione da mesma forma, independentemente do ambiente em que está sendo executado.

- **Portabilidade:** Um container pode ser movido facilmente entre diferentes ambientes, seja ele um laptop de desenvolvedor, um servidor de teste ou um cluster de produção na nuvem.

Para visualizar melhor, pense nos containers como “caixas” virtuais que têm tudo de que precisam para funcionar corretamente, não importando onde são abertas. Isso torna a vida dos desenvolvedores e administradores de sistemas muito mais fácil, pois elimina aquela desculpa de que “na minha máquina funciona”.

Containers vs Máquinas Virtuais

Uma coisa muito comum que acontece com quem começa a aprender containers é achar que container é igual máquinas virtuais. E isso é um erro é pode atrapalhar muito o seu aprendizado e evolução no mundo de containers. Ambos oferecem maneiras de isolar aplicações e executá-las em ambientes, mas eles fazem isso de maneiras totalmente diferentes.

Máquinas Virtuais (MVs):

- **Definição:** Uma MV é uma emulação completa de um sistema computacional, rodando um sistema operacional inteiro, mais o software associado.
- **Recursos:** MVs são mais pesadas, pois cada uma possui seu próprio sistema operacional, drivers e recursos associados.
- **Isolamento:** Oferecem isolamento completo, pois são executadas por hypervisors que as separam fisicamente do hardware abaixo.
- **Inicialização:** Podem levar vários minutos para iniciar, dado que todo um sistema operacional precisa ser carregado.

Containers:

- **Definição:** Containers são leves e rodam apenas os processos necessários para executar a aplicação específica.
- **Recursos:** Usam menos recursos, pois compartilham o sistema operacional do host e apenas encapsulam o aplicativo e suas dependências.
- **Isolamento:** Enquanto ainda oferecem isolamento, eles são mais leves e rápidos, pois não precisam carregar um sistema operacional inteiro.
- **Inicialização:** Containers são conhecidos por sua velocidade e podem iniciar em segundos.

Por que essa diferença é importante ?

Ao desenvolver e implantar aplicações, a escolha entre usar MVs ou containers geralmente se resume à eficiência, velocidade e ao tipo de isolamento necessário. Containers vão ser mais usados em ambientes de desenvolvimento ágil, onde a velocidade e a portabilidade são fundamentais. MVs, por outro lado, ainda encontram seu lugar em cenários que exigem isolamento robusto e ambientes mais tradicionais.
Mas é importante notar que containers e MVs não são mutuamente exclusivos e podem coexistir, permitindo que as organizações aproveitem o melhor dos dois mundos conforme as necessidades surgirem.
Entendido! Vamos então ao tópico “Imagem x Container” utilizando a formatação Markdown.

Imagem x Container

Os dois principais conceitos quando vamos utilizar containers, é o conceito de imagem e containers. Entender cada um deles e o seu papel vai fazer você ser capaz de criar soluções eficientes com Docker e você vai ter um entendimento melhor dos outros tópicos.

Imagem

Uma imagem contém tudo necessário para executar um container, incluindo: código, tempo de execução, bibliotecas, variáveis de ambiente e arquivos de configuração.
Você pode pensar na imagem como um blueprint de um projeto, onde a partir desse blueprint, desse projeto, você pode criar um ou mais containers.
As imagens possuem as seguintes caracteristicas:
Imutáveis: Uma vez criadas, não podem ser alteradas. Reutilizáveis: Podem ser usadas para criar múltiplos containers. Versionadas: Você pode ter várias versões da mesma imagem.

Container Docker

Um container é uma instância em execução de uma imagem. É o resultado da
construção de um projeto baseado no blueprint.
Os containers possuem as seguintes caracteristicas:
Efêmeros: São projetados para serem criados e destruídos rapidamente. Isolados: Operam de forma independente e não interferem em outros containers.
Portáveis: Podem ser movidos e executados em qualquer ambiente que suporte Docker.

Dica pra quem é de programação

Fazendo uma comparação de imagem e containers com programação orientada a objetos, você pode pensar na imagem como uma classe e o container como um objeto.

Arquitetura do Docker

O Docker utiliza uma arquitetura cliente-servidor. Ao compreender essa arquitetura, você está mais preparado para usar o Docker de forma eficiente e entender sua operação por trás dos panos. Vamos detalhar os componentes dessa arquitetura:

Docker Daemon (dockerd)

O Docker Daemon (`dockerd`) é o serviço em segundo plano que gerencia a construção, execução e gerenciamento de todos os objetos do Docker.

- Containers
- Imagens
- Networks
- Volumes

A máquina que executa o dockerd é chamada de Docker Host.

Docker Client (docker)

O Docker Client (`docker`) é o elemento responsável por se comunicar e enviar as instruções para o dockerd. Quando você usa comandos como `docker container run` , o cliente Docker envia essas instruções para o `dockerd` , que as executa.

Docker Registries

Um Docker Registry é um local de armazenamento para imagens Docker. O Docker Hub (https://hub.docker.com) é o registry mais utilizado e pertence à própria empresa Docker, mas você pode ter o seu próprio registry privado ou utilizar os que são oferecidos pelos Cloud Providers.

Conexão entre os componentes

1. O cliente Docker se comunica com o Docker Daemon.
2. O Docker Daemon constrói e gerencia imagens e containers.
3. Para encontrar imagens, o Docker Daemon se comunica com o Docker Registry.

Dica importante

Um ponto importante é que o Docker Client e o daemon Docker podem ser executados no mesmo host, ou você pode ter um cliente Docker se comunicando com um daemon.

Principais Comandos com container Docker

Agora que você conhece os conceitos em volta de containers e do Docker, vamos abordar os comandos mais importantes para quem está começando:

Executar o meu seu primeiro container

Para iniciar, vamos rodar o seguinte comando para criar um container:

```
docker container run hello-world
```

Como o container é baseado em uma imagem, no momento que eu executo um container, preciso dizer qual imagem vai ser usada. Com isso, o dockerd baixa a imagem "hello-world" (se ainda não estiver localmente disponível) e executa o container, mostrando uma mensagem de boas-vindas do Docker.

Listar containers de forma simples

Para ver containers que estão atualmente em execução:

```
docker container ls
```

Listar todos os containers, incluindo os parados

O parâmetro **-a** mostra todos os containers, não apenas os ativos:

```
docker container ls -a
```

Remover os containers

Para remover um container (que não está em execução):

```
docker rm CONTAINER_ID
```

Lembre-se de substituir "CONTAINER_ID" pelo ID real do container.

Executar um container em modo iterativo

Execute um container em modo iterativo (por exemplo, uma imagem Linux):

```
docker run -it ubuntu /bin/bash
```

Isso inicia uma sessão interativa do Bash dentro do container Ubuntu, permitindo que você trabalhe como se estivesse em uma instância comum do Ubuntu.

Executar e acessar uma aplicação de forma contínua

Agora vamos executar uma aplicação realmente no container, aqui, eu vou utilizar o nginx.
Para executar um container NGINX:

```
docker run -d -p 8080:80 nginx
```

O parâmetro **-d** executa o container em modo "desanexado", enquanto **-p 8080:80** faz um "port bind", ou seja, mapeia a porta 80 do container para a porta 8080 do host. Agora, você pode acessar o NGINX no seu navegador através de **http://localhost:8080**.

O que é um port bind ?

"Port binding" refere-se ao processo de mapear uma porta em seu host para uma porta em seu container. No exemplo anterior do NGINX, mapeamos a porta 80 do container para a porta 8080 do host, permitindo-nos acessar o servidor web NGINX através da porta 8080 do nosso computador.

Docker Volume

Algo muito comum com quem tá aprendendo a trabalhar com containers é utilizar containers para processos que persistem dados (Banco de dados por exemplo) e quando elimina o container, os dados são eliminados juntos.

Isso é comum porque os containers por padrão são efemeros, eles são feitos para serem destruidos e a melhor forma para você não perder nenhuma informação é usar volumes.

O gerenciamento de dados é uma parte fundamental quando se trabalha com containers. Docker Volumes são a ferramenta recomendada para gerenciar e armazenar dados gerados e usados por containers Docker. Vamos mergulhar um pouco mais nesse tópico.

O que é Docker Volume?

O **Docker Volume** é uma abstração que permite persistir e compartilhar dados entre containers e com o seu host. Ao contrário do sistema de arquivos de um container, que é efêmero e desaparece quando o container é removido, um volume é persistente e independente.

Por que usar Docker Volume?

- **Persistência de Dados:** Mesmo que um container seja removido, o volume permanece intacto.
- **Compartilhamento:** Volumes podem ser compartilhados e reutilizados entre containers.
- **Backup e Migração:** Facilita o backup e a migração de dados.

Tipos de Volumes no Docker

1. **Volume:** São criados e gerenciados pelo Docker e residem em uma parte do sistema de arquivos do host, geralmente em /var/lib/docker/volumes . Eles são ideais para manter dados que devem ser persistidos, mesmo que o container seja parado ou removido.

2. **Bind Mount:** Este é um volume que liga um local específico do sistema de arquivos do host a um ponto no container. É útil para desenvolvimento, pois permite que as alterações feitas no host se reflitam instantaneamente no container.

3. **tmpfs:** São armazenados na memória do host e não são persistentes, ou seja, os dados neles são perdidos quando o container é interrompido.

Bind Mounts na Prática

O bind mount permite que você compartilhe arquivos ou diretórios específicos do sistema de arquivos do host com o container. É uma ferramenta poderosa, pois permite que o container tenha acesso direto ao sistema de arquivos do host, permitindo uma maior flexibilidade no desenvolvimento.

Aqui estão exemplos de como usar o bind mount com o Docker:

Montar um diretório inteiro:

Se você deseja compartilhar, por exemplo, um diretório de HTML com um container NGINX, pode usar:

```
docker container run -d -p 8080:80 -v "$(pwd)"/html:/usr/share/nginx/html nginx
```

Neste comando, **`$(pwd)/html`** é o caminho para o diretório de HTML no seu sistema host, e **`/usr/share/nginx/html`** é o local no container onde o NGINX espera encontrar arquivos HTML.

Montar um arquivo específico:

Se, em vez de compartilhar um diretório inteiro, você só quer compartilhar um arquivo específico, como **`index.html`**, pode fazer:

```
docker container run -d -p 8080:80 -v "$(pwd)"/html/index.html:/usr/share/nginx/html/
index.html nginx
```

Isso assegura que apenas o arquivo **`index.html`** do host seja refletido no container.
Embora existam diferentes formas de persistência de dados no Docker, o "bind mount" é especialmente útil durante o desenvolvimento. Ele proporciona uma atualização imediata entre os arquivos no host e o que está sendo servido no container, facilitando a iteração e o desenvolvimento rápido.

Docker Image

Imagens Docker são a base para containers. Eles contêm o código-fonte do aplicativo, bibliotecas, dependências, ferramentas, e outras coisas necessárias para a aplicação funcionar. Quando você cria um container, você está apenas criando uma instância executável da imagem. Por isso, entender imagens é fundamental para dominar Docker.

Por que criar sua própria Imagem?

Enquanto o Docker Hub tem milhares de imagens disponíveis para uso, muitas vezes é necessário personalizar uma imagem para atender às necessidades específicas da sua aplicação ou ambiente. Isso pode incluir a adição de scripts, a alteração da configuração padrão, ou mesmo a adição de novas ferramentas.

Construindo uma Imagem Docker

Para criar uma imagem Docker, você precisa de um **`Dockerfile`** . Nele eu tenho todas as instruções necessárias para criar uma imagem, você pode comparar com uma receita de bolo.

Esse é um exemplo de Dockerfile:

```
FROM nginx
COPY html/index.html /usr/share/nginx/html
```

Neste **`Dockerfile`** , estamos fazendo o seguinte:

1. Usando uma imagem base **`nginx`** .
2. Copiando o arquivo **`index.html`** do diretório html no nosso host para o diretório **`/usr/share/nginx/html`** no container.

Construindo a Imagem

Agora, usando o Dockerfile acima, podemos construir uma imagem Docker com o seguinte comando:

```
docker build -t minha-imagem-nginx .
```

Este comando diz ao Docker para construir uma imagem usando o Dockerfile no diretório atual (indicado pelo **`.`**) e nomear essa imagem como **`minha-imagem-nginx`** .

Verificando a Imagem

Uma vez que a imagem é construída, você pode ver todas as suas imagens Docker com:

```
docker image ls
```

Deletando uma Imagem

Para deletar uma imagem específica:

```
docker image rm [NOME_DA_IMAGEM]
```

Ou usando o ID da imagem:

```
docker image rm [ID_DA_IMAGEM]
```

Nota: Antes de deletar uma imagem, assegure-se de que não há containers ativos utilizando essa imagem. Se houver, você precisa parar e remover esses containers primeiro.

Limpar Imagens Não Utilizadas

Com o tempo, você pode acumular várias imagens antigas ou não utilizadas.
Para limpar essas imagens de uma vez:

```
docker image prune
```

Docker Registry

Os registries são cruciais quando se trabalha com containers. Eles oferecem um local para armazenar e distribuir imagens Docker. Isso pode ser muito útil, tanto para distribuição de suas próprias imagens quanto para obter imagens criadas por terceiros.

Um Docker Registry é uma solução de armazenamento que guarda imagens Docker e permite que você as distribua, seja dentro de sua organização ou para o público em geral.

Docker Hub

O Docker Hub é o registry padrão onde o Docker procura imagens. Ele é operado pela própria Docker, Inc. e oferece imagens tanto públicas quanto privadas.

Comando para baixar uma imagem

Para baixar uma imagem, você deve executar o comando docker pull:

```
# Para baixar uma imagem do Docker Hub:
docker pull [NOME_DA_IMAGEM]
```

Padrão de nomenclatura de uma imagem no Docker Hub

Quando se refere a uma imagem no Docker Hub, ela normalmente segue o padrão:

`[NAMESPACE]/[NOME_DA_IMAGEM]:[TAG]`

- **NAMESPACE:** Diz a quem pertence essa imagem. É o nome do usuário ou organização no Docker Hub.
- **NOME_DA_IMAGEM:** O nome da imagem Docker.
- **TAG:** Um marcador para a versão da imagem, normalmente adicionamos a tag versionada e a `latest` se for a versão mais recente.

Por exemplo: `fabricioveronez/custom-nginx:v1`

Autenticando em um Docker Registry

Antes de enviar imagens para um Docker Registry, como o Docker Hub, você deve se autenticar:

```
docker login
```

Você será solicitado a fornecer seu nome de usuário e senha do Docker Hub. Se você se autenticou com sucesso, suas credenciais serão salvas no seu host local.

Enviando uma imagem para o Docker Hub

Depois de criar e marcar sua imagem, você pode enviá-la para o Docker Hub:

```
# Marcar a imagem
docker image tag [NOME_DA_IMAGEM_LOCAL] [NAMESPACE]/[NOME_DA_IMAGEM]:[TAG]

# Enviar a imagem para o Docker Hub
docker push [NAMESPACE]/[NOME_DA_IMAGEM]:[TAG]
```

Docker Compose

Ao trabalhar com aplicações que têm múltiplos serviços, como um backend, frontend, banco de dados, entre outros, gerenciar individualmente cada container pode se tornar complexo. O Docker Compose foi projetado para resolver este problema, permitindo definir e executar aplicações multi-container com facilidade.

O que é Docker Compose?

Docker Compose é uma ferramenta que permite definir e gerenciar aplicações que utilizam vários containers. Ele utiliza arquivos YAML para configurar todos os serviços de uma aplicação, tornando mais fácil e claro o processo de gerenciamento.

Exemplo Prático: Aplicação com PostgreSQL e MariaDB

Vamos supor que você está desenvolvendo uma aplicação que precisa se conectar a bancos de dados PostgreSQL e MariaDB. O uso de Docker Compose facilita a execução dessas duas instâncias de banco de dados simultaneamente. Aqui está um exemplo do **`docker-compose.yml`** esse cenário:

```
version: '3'
services:
  postgree:
    image: postgres:latest
    container_name: postgreeServer
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: Pg@123

  mariadb:
    image: mariadb
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: MySql@123
      MYSQL_DATABASE: defaultdatabase
      MYSQL_USER: User123
      MYSQL_PASSWORD: Pwd123
```

Neste arquivo:

- Definimos dois serviços: **`postgree`** e **`mariadb`** .
- Cada serviço utiliza uma imagem específica e possui sua própria configuração de porta e variáveis de ambiente.

Para executar a aplicação com Docker Compose, você utiliza os seguintes comandos:

```
# Iniciar os serviços definidos no docker-compose.yml
docker-compose up -d

# Parar os serviços
docker-compose down
```

Vantagens do Docker Compose

1. **Configuração centralizada:** Com o Docker Compose, você tem uma única fonte de verdade para sua configuração de aplicação.
2. **Isolamento:** Cada serviço é isolado em seu próprio container, assegurando que não interfira em outros serviços.
3. **Facilidade de Uso:** É muito mais fácil e claro gerenciar múltiplos serviços com Docker Compose do que manualmente iniciar cada container.