Containers e Docker

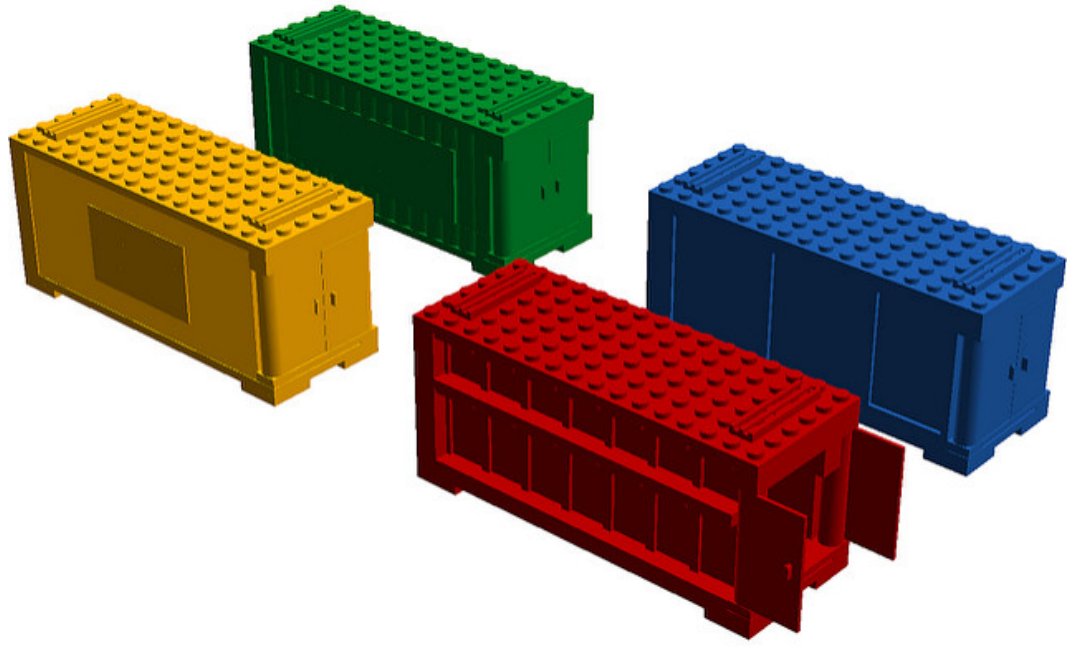
SENAI - Agile Infrastructure - 2019

Capitulo 1

Construindo imagens Docker com um Dockerfile

Anterior | Indice | Proxima

Construindo imagens Docker com um Dockerfile



Objetivos

Criaremos uma imagem de contêiner automaticamente, com um Dockerfile.

No final desta lição, você será capaz de:

- Escrever um Dockerfile.
- Criar uma imagem a partir de um Dockerfile.

Visão geral do Dockerfile

- Um Dockerfile é uma receita de compilação para uma imagem do Docker.
- Ele contém uma série de instruções informando ao Docker como uma imagem é construída.
- O comando docker build cria uma imagem a partir de um Dockerfile.

Escrevendo nosso primeiro Dockerfile

Nosso Dockerfile deve estar em um **novo diretório vazio** .

1. Crie um diretório para armazenar o nosso Dockerfile.

```
$ mkdir myimage
```

1. Crie um Dockerfile dentro deste diretório.

```
$ cd myimage
$ vi Dockerfile
```

Obviamente, você pode usar qualquer outro editor de sua escolha.

Coloque esse conteúdo no arquivo...

```
FROM ubuntu
RUN apt-get update
RUN apt-get install figlet
```

- `FROM` indica a imagem base para nossa compilação.
- Cada linha `RUN` será executada pelo Docker durante a compilação.
- Nossos comandos `RUN` devem ser não interativos.
  (Nenhuma entrada pode ser fornecida ao Docker durante a compilação.)
- Em muitos casos, adicionaremos a flag `-y` ao `apt-get`.

Construa!

Salve nosso arquivo e execute:

```
$ docker build -t figlet .
```

- `-t` indica a etiqueta a ser aplicada à imagem.
- `.` indica a localização do *contexto de construção* .

O que acontece quando construímos a imagem?

A saída do `docker build` é assim:

```
docker build -t figlet .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 ---> f975c5035748
Step 2/3 : RUN apt-get update
 ---> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
 ---> eb8d9b561b37
Step 3/3 : RUN apt-get install figlet
 ---> Running in c29230d70f9b
(...output of the RUN command...)
Removing intermediate container c29230d70f9b
 ---> 0dfd7a253f21
Successfully built 0dfd7a253f21
Successfully tagged figlet:latest
```

- A saída dos comandos `RUN` foi omitida.

Enviando o contexto de construção para o Docker

```
Enviando contexto de construção para o daemon do Docker 2.048 kB
```

- O contexto de construção é o diretório `.` fornecido ao `docker build`.
- É enviado pelo cliente do Docker para o daemon do Docker.
- Tenha cuidado (ou paciente) se esse diretório for grande e seu link estiver lento.
- Você pode acelerar o processo com um arquivo `.dockerignore`
  - Diz ao docker para ignorar arquivos específicos no diretório
  - Ignore apenas arquivos que você não precisará no contexto de construção!

Executando cada etapa

```
Step 2/3 : RUN apt-get update
 ---> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
 ---> eb8d9b561b37
```

- Um container (e01b294dbffd) é criado a partir da imagem base.
- O comando RUN é executado neste container.
- O contêiner é confirmado em uma imagem (eb8d9b561b37).
- O contêiner de construção (e01b294dbffd) é removido.
- A saída desta etapa será a imagem base para a próxima.

O sistema de armazenamento em cache

Se você executar a mesma compilação novamente, será instantânea. Por quê?

- Após cada etapa de criação, o Docker tira uma "foto" da imagem resultante.
- Antes de executar uma etapa, o Docker verifica se ele já criou a mesma sequência.
- O Docker usa as strings exatas definidas no seu Dockerfile, portanto:
  - `EXECUTAR o apt-get install figlet cowsay`
    é diferente de
    `EXECUTE o apt-get install cowsay figlet`

Você pode forçar uma reconstrução com `docker build --no-cache ...`.

Running the image

A imagem resultante não é diferente da imagem produzida manualmente.

```
$ docker run -ti figlet
root@91f3c974c9a1:/# figlet hello

 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

Yay! 🎉

Usando imagem e histórico de exibição

O comando history lista todas as camadas que compõem uma imagem.

Para cada camada, mostra seu tempo de criação, tamanho e comando de criação.

Quando uma imagem foi criada com um Dockerfile, cada camada corresponde a uma linha do Dockerfile.

```
$ docker history figlet
IMAGE          CREATED           CREATED BY                      SIZE
f9e8f1642759   About an hour ago   /bin/sh -c apt-get install fi   1.627 MB
7257c37726a1   About an hour ago   /bin/sh -c apt-get update       21.58 MB
07c86167cdc4   4 days ago          /bin/sh -c #(nop) CMD ["/bin    0 B
<missing>      4 days ago          /bin/sh -c sed -i 's/^#\s*\(    1.895 kB
<missing>      4 days ago          /bin/sh -c echo '#!/bin/sh'     194.5 kB
<missing>      4 days ago          /bin/sh -c #(nop) ADD file:b    187.8 MB
```

Introdução da syntax JSON

A maioria dos argumentos do Dockerfile pode ser transmitida de duas formas:

- string:
  `RUN apt-get install figlet`

- Lista JSON:
  `RUN ["apt-get", "install", "figlet"]`

Vamos mudar nosso Dockerfile para ver como isso afeta a imagem resultante.

Usando a syntax JSON no nosso arquivo Dockerfile

Vamos mudar o nosso Dockerfile da seguinte maneira!

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
```

Em seguida, crie o novo Dockerfile.

```
$ docker build -t figlet .
```

Syntax JSON vs string

Compare o novo histórico:

```
$ docker history figlet
IMAGE          CREATED              CREATED BY                      SIZE
27954bb5faaf   10 seconds ago       apt-get install figlet          1.627 MB
7257c37726a1   About an hour ago    /bin/sh -c apt-get update       21.58 MB
07c86167cdc4   4 days ago           /bin/sh -c #(nop) CMD ["/bin    0 B
<missing>      4 days ago           /bin/sh -c sed -i 's/^#\s*\(    1.895 kB
<missing>      4 days ago           /bin/sh -c echo '#!/bin/sh'     194.5 kB
<missing>      4 days ago           /bin/sh -c #(nop) ADD file:b    187.8 MB
```

- A syntax JSON específica um comando *exato* para executar.
- A syntax com String especifica um comando para ser chamado (wrapped) com `/bin/sh -c "..."`.

Avaliando quando usar a syntax JSON e quando usar string

- String:
  - é facil para escrever
  - interpola as variáveis de ambiente e outras expressões do shell
  - cria um processo extra (`/bin/sh -c ...`) para realizar o parser da string
  - necessita o `/bin/sh` existir no container
- JSON:
  - é mais difícil de escrever (e ler!)
  - todos os argumentos são passados sem processamento extra.
  - não cria um processo adicional
  - não necessita do `/bin/sh` existir no container

CMD e ENTRYPOINT

Anterior | Indice | Proxima

CMD e ENTRYPOINT

Objetivos

Nesta lição, aprenderemos sobre dois importantes Comandos do Dockerfile:

CMD eENTRYPOINT.

Esses comandos nos permitem definir o comando padrão para executar em um contêiner.

Definindo um comando padrão

Quando as pessoas executam nosso contêiner, queremos recebê-las com uma boa mensagem de saudação e usando uma fonte personalizada.

Para isso, executaremos:

```
figlet -f script hello
```

- `-f script` diz ao figlet para usar uma fonte sofisticada.
- `hello` é a mensagem que queremos que seja exibida.

Adicionando CMD ao nosso Dockerfile

Nosso novo Dockerfile ficará assim:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
CMD figlet -f script hello
```

- CMD define um comando padrão para executar quando nenhum é fornecido.
- Pode aparecer em qualquer ponto do arquivo.
- Cada CMD substituirá o anterior.
- Como resultado, embora você possa ter várias linhas CMD, isso é inútil.

Construa e teste nossa imagem

Vamos construí-lo:

```
$ docker build -t figlet .
...
Successfully built 042dff3b4a8d
Successfully tagged figlet:latest
```

E roda-lo:

```
$ docker run figlet
 _          _   _
| |        | | | |
| |    _   | | | |  __
|/ \   |/  |/  |/  /  \_
|   |_/|__/|__/|__/\__/
```

Substituindo CMD

Se quisermos colocar um shell em nosso contêiner (em vez de executar figlet), basta especificar um programa diferente para executar:

```
$ docker run -it figlet bash
root@7ac86a641116:/#
```

- Nós especificamos bash.
- Ele substituiu o valor de CMD.

Usando ENTRYPOINT

Queremos poder especificar uma mensagem diferente na linha de comando, mantendo o figlet e alguns parâmetros padrão.

Em outras palavras, gostaríamos de poder fazer isso:

```
$ docker run figlet salut
          _
         | |
 ,   __,  | |      _|_
/ \_/  |  |/  |   | |
 \/ \_/|_/|__/ \_/|_/|_/
```

Usaremos o verbo ENTRYPOINT no Dockerfile.

Adicionando ENTRYPOINT ao nosso Dockerfile

Nosso novo Dockerfile ficará assim:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
```

- ENTRYPOINT define um comando base (e seus parâmetros) para o contêiner.
- Os argumentos da linha de comando são anexados a esses parâmetros.
- Como CMD,ENTRYPOINT pode aparecer em qualquer lugar e substitui o valor anterior.

Por que usamos a sintaxe JSON para o nosso ENTRYPOINT?

Implicações da sintaxe JSON vs string

- Quando o CMD ou o ENTRYPOINT usam a sintaxe da string, eles são executado no contexto de `sh -c`.
- Para evitar esse "wrapping", podemos usar a sintaxe JSON.

E se usássemos `ENTRYPOINT` com sintaxe de string?

```
$ docker run figlet salut
```

Isso executaria o seguinte comando na imagem `figlet`:

```
sh -c "figlet -f script" salut
```

Construa e teste nossa imagem

Vamos construí-lo:

```
$ docker build -t figlet .
...
Successfully built 36f588918d73
Successfully tagged figlet:latest
```

E execute:

```
$ docker run figlet salut
          _
         | |
 ,   __,  | |       _|_
/ \_/  |  |/  |   |  |
 \/ \_/|_/|__/ \_/|_/|_/
```

Usando CMD e ENTRYPOINT juntos

E se quisermos definir uma mensagem padrão para nosso contêiner?

Em seguida, usaremos ENTRYPOINT e CMD juntos.

- ENTRYPOINT definirá o comando base para o nosso contêiner.
- CMD definirá o (s) parâmetro (s) padrão (s) para este comando.
- *Ambos* têm que usar a sintaxe JSON.

CMD eENTRYPOINT juntos

Nosso novo Dockerfile ficará assim:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

- ENTRYPOINT define um comando base (e seus parâmetros) para o contêiner.
- Se não especificarmos argumentos extras da linha de comando ao iniciar o contêiner, o valor de CMD é acrescentado.
- Caso contrário, nossos argumentos extras de linha de comando são usados em vez de CMD.

Construa e teste nossa imagem

Vamos construí-lo:

```
$ docker build -t myfiglet .
...
Successfully built 6e0b6a048a07
Successfully tagged myfiglet:latest
```

Execute-o sem parâmetros:

```
$ docker run myfiglet
 _        _   _                      _
| |      | | | |                    | |    |
| |    _ | | | | __          __    ,_   | |  __|
|/ \   |/  |/  |/  /  \_  | | |_/  \_/  | |/  /  |
|   |_/|__/|__/|__/\__/    \/ \/  \__/    |_/|__/\_/|_/
```

Substituindo os parâmetros padrão da imagem

Agora vamos passar argumentos extras para a imagem.

```
$ docker run myfiglet hola mundo
 _          _
| |        | |                               |
| |    ___ | | ___,    _  _  _        _  _    ___|   ___
|/ \   /  \_|/  /  |    / |/ |/ |  |   |  / |/ |  /  |  /  \_
|   |_/\___/ |__/\_/|_/    |  |  |_/ \_/|_/  |  |_/\_/|_/\___/
```

Substituímos o `CMD`, mas ainda usamos o `ENTRYPOINT`.

Substituindo ENTRYPOINT

E se quisermos executar um shell em nosso contêiner?

Não podemos simplesmente executar o `docker run myfiglet bash` porque isso apenas dizia ao figlet para exibir a palavra "bash".

Usamos o parâmetro `--entrypoint`:

```
$ docker run -it --entrypoint bash myfiglet
root@6027e44e2955:/#
```

Publicando imagens no Docker Hub

Anterior | Indice | Proxima

Publicando imagens no Docker Hub

Nós construímos nossas primeiras imagens.

Agora podemos publicá-lo no Docker Hub!

- Criar uma conta no Docker Hub é gratuito (e não requer cartão de crédito) e a hospedagem imagens públicas também são gratuitas. *

Como fazer login na nossa conta do Docker Hub

- Isso pode ser feito na CLI do Docker:

```
docker login
```

- Na maioria dos servidores Linux, as credenciais são criadas `~/.docker/config`

Como marcar uma imagem para colocá-la no Hub

- Vamos marcar a nossa imagem `xapi` (ou qualquer outra que seja do nosso agrado):

```
docker tag xapi dvriesman/xapi
```

- E fazer push para o github:

```
docker push dvriesman/xapi
```

- É isso!

Como marcar uma imagem para colocá-la no Hub

- Vamos marcar a nossa imagem `xapi` (ou qualquer outra que seja do nosso agrado):

```
docker tag xapi dvriesman/xapi
```

- E fazer push para o github:

```
docker push dvriesman/xapi
```

- É isso!
- Qualquer um pode agora rodar `docker run dvriesman/xapi` em qualquer lugar.

Exercício 01

- Crie uma imagem que executa o comando nmap da seguinte forma:

```
nmap -p 1-1000 -sV -sS -T4 <ip>
```

- O ip deverá ser informado como argumento na linha de comando de execução do docker, conforme abaixo:

```
docker run meusuario/minhaimagem 172.22.12.10
```

- Publique a imagem na sua conta do docker hub

Configuração da aplicação

Configuração da aplicação

Existem várias maneiras de fornecer configuração para as aplicações em contêiner.

Não existe o "melhor caminho" - depende de fatores como:

- tamanho da configuração
- parâmetros obrigatórios e opcionais,
- frequência de alterações na configuração.

Argumentos por linha de comando

```
docker run jpetazzo/hamba 80 www1:80 www2:80
```

- A configuração é fornecida através dos argumentos da linha de comando.
- No exemplo acima, o ENTRYPOINT é um script que irá:
  - analisar os parâmetros,
  - gerar um arquivo de configuração,
  - iniciar o serviço real.

Prós e contras dos argumentos na linha de comando

- Apropriado para parâmetros obrigatórios (sem os quais o serviço não pode ser iniciado).
- Conveniente para serviços de "canivete suiço" instanciados muitas vezes.

  (Como não há etapa extra: basta executá-lo!)
- Não é bom para configurações dinâmicas ou configurações maiores.

  (Essas coisas ainda são possíveis, mas mais complicadas.)

Variáveis de ambiente

```
docker run -e ELASTICSEARCH_URL=http://es42:9201/ kibana
```

- A configuração é fornecida através de variáveis de ambiente.
- A variável de ambiente pode ser usada diretamente pelo programa,
  ou por um script que gera um arquivo de configuração.

Prós e contras das Variáveis de ambiente

- Apropriado para parâmetros opcionais (já que a imagem pode fornecer valores padrão).
- Também conveniente para serviços instanciados muitas vezes.

  (É tão fácil quanto os parâmetros da linha de comando.)
- Ótimo para serviços com muitos parâmetros, mas você deseja especificar apenas alguns.

  (E use valores padrão para todo o resto.)
- Capacidade de examinar possíveis parâmetros e seus valores padrão.
- Não é bom para configurações dinâmicas.

Configuração hardcode na imagem

```
FROM prometheus
COPY prometheus.conf /etc
```

- A configuração é adicionada à imagem.
- A imagem pode ter uma configuração padrão; a nova configuração pode:
  - substituir a configuração padrão,
  - extender uma pré-existente (se o código puder ler vários arquivos de configuração).

Pros and contras da configuração hardcode

- Permite personalização arbitrária e arquivos de configuração complexos.
- Requer escrever um arquivo de configuração. (Obviamente!)
- Requer a construção de uma imagem para iniciar o serviço.
- Requer reconstruir a imagem para reconfigurar o serviço.
- Requer reconstruir a imagem para atualizar o serviço.
- Imagens configuradas podem ser armazenadas em registros.

  (O que é ótimo, mas requer um registro.)

Configuração através de volume

```
docker run -v appconfig:/etc/appconfig myapp
```

- A configuração é armazenada em um volume.
- O volume está anexado ao container.
- A imagem pode ter uma configuração padrão.

  (Isso resulta em uma configuração menos "óbvia", que precisa de mais documentação.)

Pros e contras das configurações por volume

- Permite personalização arbitrária e arquivos de configuração complexos.
- Requer a criação de um volume para cada configuração diferente.
- Serviços com configurações idênticas podem usar o mesmo volume.
- A configuração pode ser gerada ou editada através de outro contêiner.
- Viola o princípio III do 12 Factors (*Arquitetos de Software, fiquem ligados*)

Mantendo senhas

 Idealmente, você não deve colocar segredos (senhas, tokens ...) em:

- linha de comando ou variáveis de ambiente (qualquer pessoa com acesso à API do Docker pode obtê-las),
- imagens, especialmente armazenadas em um registro.

O gerenciamento de segredos é melhor tratado com um orquestrador (Kubernetes).

Gerenciar segredos com segurança sem um orquestrador pode ser "inventado".

Exemplo:

- a aplicação pode solicitar uma senha para um endpoint de uma API
- utilizar um cofre de senhas (hashcorp vault)

Exemplo COWSAY

```
docker run dvriesman/cowsay
```

```
docker run -e PALAVRA=Devops dvriesman/cowsay
```

```
docker run -e "PALAVRA=Agile Devops" dvriesman/cowsay
```

Desafio 01

- Com base no projeto cowsay disponivel em

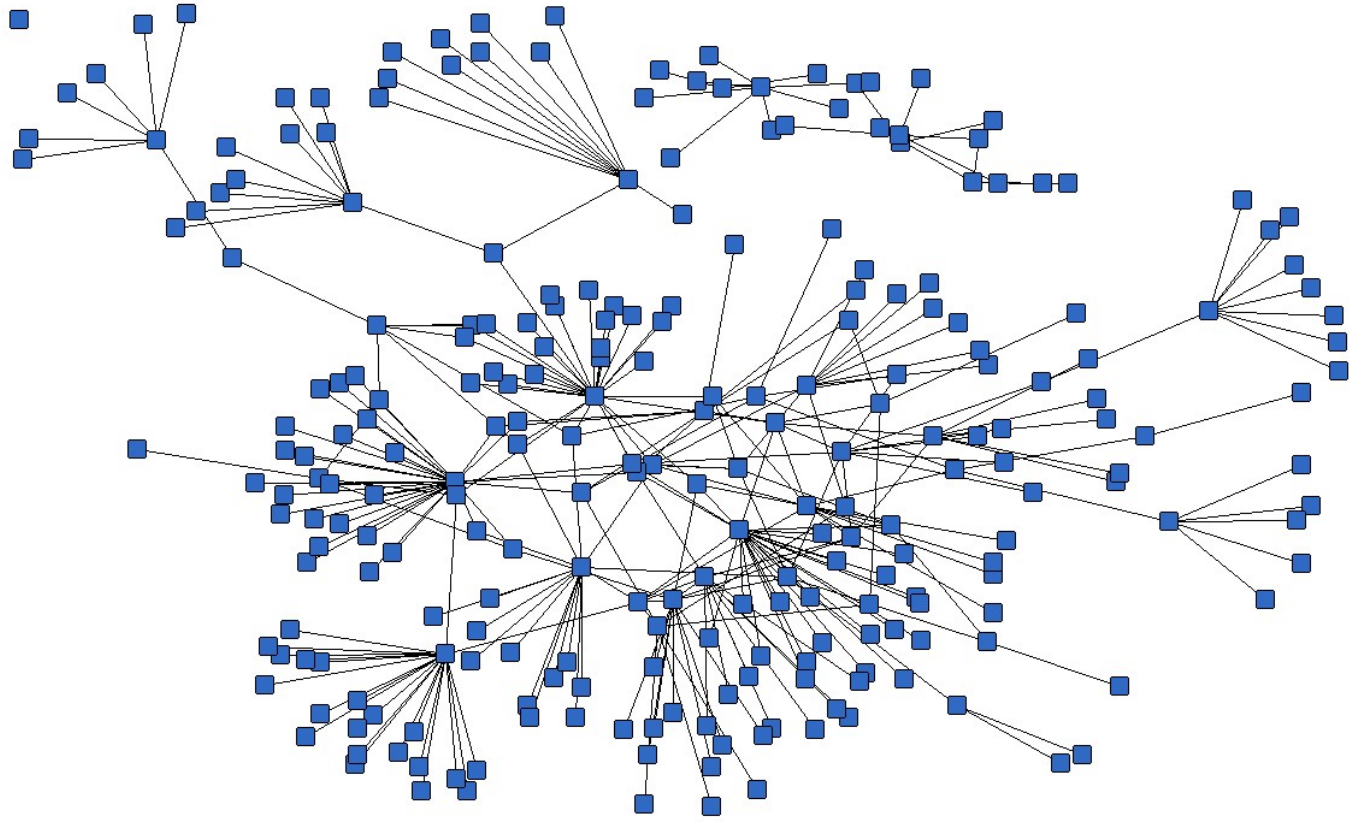
https://github.com/dvriesman/agile-devops/tree/master/week1/apps/cowsay

- Crie uma nova imagem com a aplicação "fortune" instalada. (Além da cowsay)
- Crie um novo parâmetro (variável de ambiente) que habilitará a vaca (cow) a falar o texto da sorte!.
- Dica 1: Pipes do Linux podem cair bem.
- Use caminhos absolutos para os programas chamados pelo shellscript.

O Basicão das redes no Docker

Anterior | Indice | Proxima

O Basicão das redes no Docker



Objetivos

Agora executaremos serviços de rede (aceitando solicitações) em contêineres.

No final desta seção, você será capaz de:

- Execute um serviço de rede em um contêiner.
- Manipular o básico da rede de contêineres
- Encontrar o endereço IP de um contêiner.

Um servidor web simples e estático

Execute a imagem do nginx do Docker Hub, que contém um servidor Web básico:

```
$ docker run -d -P nginx
66b1ce719198711292c8f34f84a7b68c3876cf9f67015e752b94e189d35a204e
```

- O Docker fará o download da imagem no Docker Hub.
- `-d` diz ao Docker para executar a imagem em segundo plano.
- `-P` diz ao Docker para tornar este serviço acessível a partir de outros computadores.

  (`-P` é a versão curta do `--publish-all`.)

Mas como nos conectamos ao nosso servidor web agora?

Localizando nossa porta do servidor web

Vamos usar o `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE  ...  PORTS                  ...
e40ffb406c9e  nginx  ...  0.0.0.0:32768->80/tcp  ...
```

- O servidor da web está sendo executado na porta 80 dentro do contêiner.
- Essa porta está mapeada para a porta 32768 em nosso host Docker.

Vamos explicar os porquês e comos desse mapeamento de portas.

Mas primeiro, vamos garantir que tudo funcione corretamente.

Conectando ao nosso servidor web (GUI)

Aponte seu navegador para o endereço IP do seu host Docker, na porta mostrado por `docker ps` para a porta 80 do contêiner.

Conectando ao nosso servidor web (CLI)

Você também pode usar o curl diretamente do host do Docker.

Certifique-se de usar o número da porta correto, se for diferente do exemplo abaixo:

```
$ curl localhost:32768
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Como o Docker sabe qual porta mapear?

- Existem metadados na imagem informando "esta imagem tem algo na porta 80".
- Podemos ver que os metadados com `docker inspect`:

```
$ docker inspect --format '{{.Config.ExposedPorts}}' nginx
map[80/tcp:{}]
```

- Esses metadados foram definidos no Dockerfile, com a palavra-chave `EXPOSE`.
- Podemos ver isso com o `docker history`:

```
$ docker history nginx
IMAGE          CREATED        CREATED BY
7f70b30f2cc6   11 days ago    /bin/sh -c #(nop)  CMD ["nginx" "-g" "…
<missing>      11 days ago    /bin/sh -c #(nop)  STOPSIGNAL [SIGTERM]
<missing>      11 days ago    /bin/sh -c #(nop)  EXPOSE 80/tcp
```

Por que estamos mapeando portas?

- Os contêineres não podem ter endereços IPv4 públicos.
- Eles têm endereços particulares.
- Os serviços devem ser expostos porta a porta.
- As portas precisam ser mapeadas para evitar conflitos.

Localizando a porta em um script

Analisar a saída do `docker ps` seria doloroso.

Existe um comando para nos ajudar:

```
$ docker port <containerID> 80
32768
```

Alocação manual de números de porta

Se você deseja definir os números de porta, não há problema:

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 8000:80 nginx
$ docker run -d -p 8080:80 -p 8888:80 nginx
```

- Estamos executando três servidores web NGINX.
- O primeiro está exposto na porta 80.
- O segundo está exposto na porta 8000.
- O terceiro está exposto nas portas 8080 e 8888.

Nota: o padrão é `port-on-host:port-on-container`.

Localizando o endereço IP do contêiner

Podemos usar o comando `docker inspect` para encontrar o endereço IP do recipiente.

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
172.17.0.3
```

- `docker inspect` é um comando avançado, que pode recuperar uma tonelada de informações sobre nossos contêineres.
- Aqui, fornecemos uma string de formato para extrair exatamente o endereço IP privado do contêiner.

Pingando nosso contêiner

Podemos testar a conectividade com o contêiner usando o endereço IP que acabamos de descobrir. Vamos ver isso agora usando a ferramenta ping.

```
$ ping <ipAddress>
64 bytes from <ipAddress>: icmp_req=1 ttl=64 time=0.085 ms
64 bytes from <ipAddress>: icmp_req=2 ttl=64 time=0.085 ms
64 bytes from <ipAddress>: icmp_req=3 ttl=64 time=0.085 ms
```

Contêineres em sua infraestrutura

Existem várias maneiras de integrar contêineres na sua rede.

- Inicie o contêiner, permitindo que o Docker aloque uma porta pública para ele.
  Em seguida, recupere o número da porta e alimente-o com a sua configuração.

- Escolha um número de porta fixo com antecedência, ao gerar sua configuração.
  Em seguida, inicie o contêiner configurando os números de porta manualmente.

- Use um plug-in de rede, conectando seus contêineres, por exemplo VLANs, túneis ...

Ao usar o Docker por meio de uma camada de gerenciamento extra, como Kubernetes, ele fornecerá seu próprio mecanismo para expor os contêineres.

Resumo da seção

Aprendemos como:

- Expor uma porta de rede.
- Manipular o básico da rede de contêineres.
- Encontrar o endereço IP de um contêiner.

Desafio 02

https://github.com/dvriesman/agile-devops/week1/apps/colored

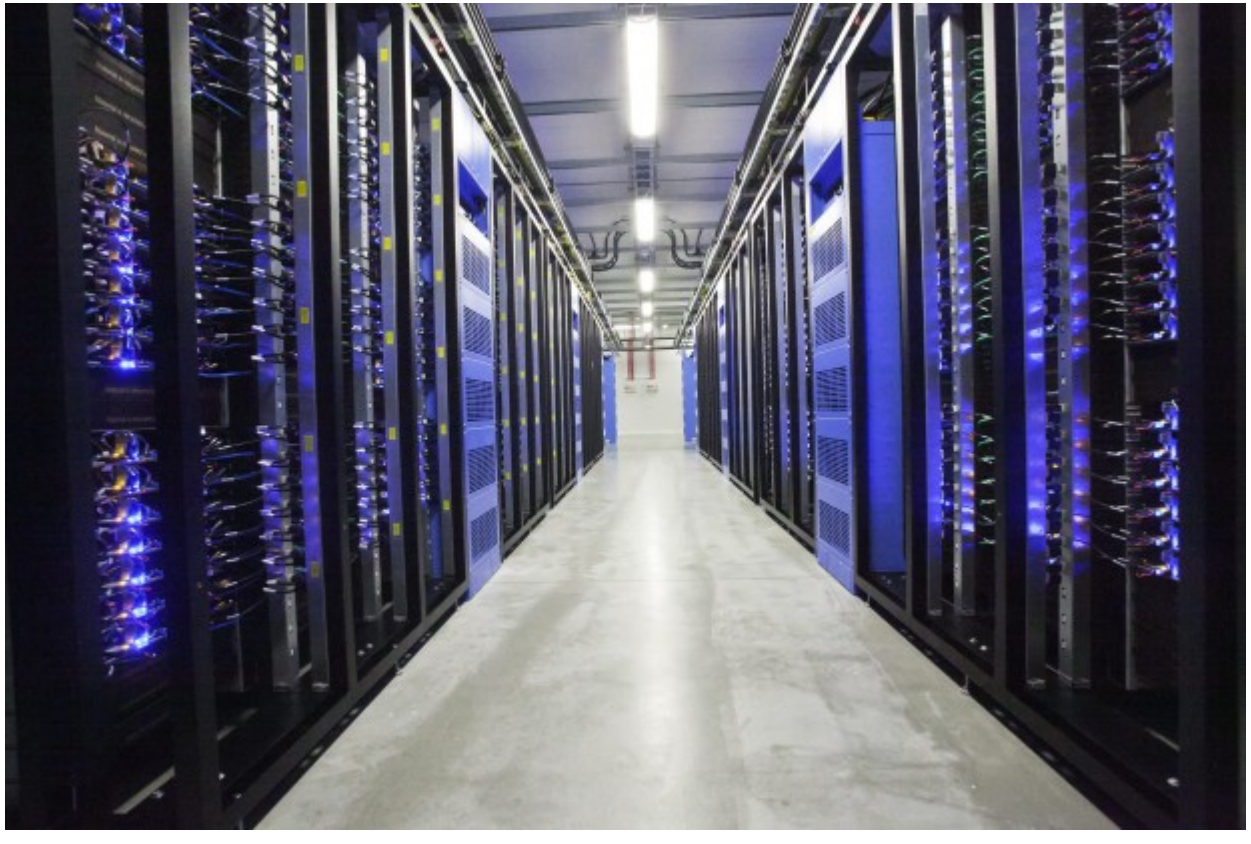
=> Valendo nota.

- Publicar o Dockerfile desse projeto no repositorio de vocês e me encaminhar o link no Slack !
- Publicar um arquivo readme.MD com instruções de como rodar o comando docker.

Trabalhando com volumes

Anterior | Indice | Proxima

Trabalhando com volumes



Objetivos

No final desta seção, você será capaz de:

- Criar contêineres contendo volumes.
- Compartilhar volumes entre contêineres.
- Compartilhar um diretório host com um ou vários contêineres.

Trabalhando com volumes

Os volumes do Docker podem ser usados para realizar muitas coisas, incluindo:

- Obter performance de disco I/O nativa.
- Deixar arquivos fora do docker commit.
- Compartilhar um diretório entre multiplos containers.
- Compartilhar um diretório entre o host e um contêiner.
- Compartilhar um *arquivo único* entre o host e um contêiner.

Volumes são diretórios especiais

Os volumes podem ser declarados de duas maneiras diferentes:

- Com um Dockerfile, ou com a instrução VOLUME.

```
VOLUME /uploads
```

- Na linha de comando, com a flag -v no docker run.

```
$ docker run -d -v /uploads myapp
```

Em ambos os casos, /uploads (dentro do container) será um volume.

Os volumes podem ser compartilhados entre contêineres

Você pode iniciar um contêiner com *exatamente os mesmos volumes* que outro.

O novo contêiner terá os mesmos volumes, nos mesmos diretórios.

Eles conterão exatamente a mesma coisa e permanecerão sincronizados.

Sob o capô, eles são realmente os mesmos diretórios no host de qualquer maneira.

Isso é feito usando a flag `--volumes-from` para `docker run`.

Compartilhando logs do servidor de aplicação com outro contêiner

Vamos começar um contêiner Tomcat:

```
$ docker run --name webapp -d -p 8080:8080 -v /usr/local/tomcat/logs tomcat
```

Agora, inicie um contêiner alpine acessando o mesmo volume:

```
$ docker run --volumes-from webapp alpine sh -c "tail -f /usr/local/tomcat/logs/*"
```

Em seguida, a partir de outra janela, envie solicitações para nosso contêiner Tomcat:

```
$ curl localhost:8080
```

Os volumes existem independentemente dos contêineres

Se um contêiner for parado ou removido, seus volumes ainda existirão e estarão disponíveis.

Os volumes podem ser listados e manipulados com os subcomandos docker volume:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               5b0b65e4316da67c2d471086640e6005ca2264f3...
local               pgdata-prod
local               pgdata-dev
local               13b59c9936d78d109d094693446e174e5480d973...
```

Alguns desses nomes de volume foram explícitos (pgdata-prod, pgdata-dev).

Os outros (os IDs hexadecimais) foram gerados automaticamente pelo Docker.

Nomeando volumes

- Os volumes podem ser criados sem um contêiner e depois usados em vários contêineres.

Vamos criar alguns volumes diretamente.

```
$ docker volume create webapps
webapps
```

```
$ docker volume create logs
logs
```

Os volumes não são ancorados a um caminho específico.

Usando nossos volumes nomeados

- Volumes são usados com a opção `-v`.
- Quando um caminho do host não contém um `/`, é considerado um nome de volume.

Vamos iniciar um servidor da web usando os dois volumes anteriores.

```
$ docker run -d -p 1234:8080 \
        -v logs:/usr/local/tomcat/logs \
        -v webapps:/usr/local/tomcat/webapps \
        tomcat
```

```
$ curl localhost:1234
```

Usando um volume em outro contêiner

- Faremos alterações no volume de outro contêiner.
- Neste exemplo, executaremos um editor de texto no outro contêiner.

  (Mas pode ser um servidor FTP, um servidor WebDAV, um receptor Git ...)

Vamos começar outro contêiner usando o volume `webapps`.

```
$ docker run -v webapps:/webapps -w /webapps -ti alpine vi ROOT/index.jsp
```

Vandalize a página, salve, saia.

Então rode `curl localhost:1234` para ver suas mudanças.

Usando "montagens de ligação" personalizadas

Em alguns casos, você deseja que um diretório específico no host seja mapeado dentro do contêiner:

- Você deseja gerenciar o armazenamento e os instantâneos por conta própria.

  (Com LVM, SAN, ou qualquer outra coisa!)

- Você tem um disco separado com melhor desempenho (SSD) ou resiliência do que o disco do sistema e você deseja colocar dados importantes nesse disco.

- Você deseja compartilhar seu diretório de origem entre seu host (onde o fonte é editada) e o contêiner (onde é compilado ou executado).

```
$ docker run -d -v /path/on/the/host:/path/in/container image ...
```

Ciclo de vida dos volumes

- Quando você remove um contêiner, seus volumes são mantidos.
- Você pode listá-los com `docker volume ls`.
- Você pode acessá-los criando um container com `docker run -v`.
- Você pode removê-los com `docker volume rm`

Por fim, você é o responsável pelo registro, monitoramento e backup de seus volumes.

Compartilhando um único arquivo

Anterior | Indice | Proxima

Compartilhando um único arquivo

O mesmo sinalizador `-v` pode ser usado para compartilhar um único arquivo (em vez de um diretório).

Um dos exemplos mais interessantes é compartilhar o soquete de controle do Docker.

```
$ docker run -it -v /var/run/docker.sock:/var/run/docker.sock docker sh
```

Nesse contêiner, agora você pode executar comandos `docker` se comunicando com o Docker Engine em execução no host. Experimente o `docker ps`!

Como esse contêiner tem acesso ao soquete do Docker, ele possui acesso de root ao host.

Volume plugins

Você pode instalar plug-ins para gerenciar volumes suportados por sistemas de armazenamento específicos, ou fornecendo recursos extras. Por exemplo:

- REX-Ray - (SAN or NAS) ou block storages (EBS).
- Portworx - Block storage.
- Gluster - armazenamento distribuído definido por software de código aberto que pode ser dimensionado para vários petabytes. Fornece interfaces para armazenamento de objetos, blocos e arquivos.

Exercícios

O time de desenvolvimento solicitou uma base de dados mysql - se o container reiniciar, os dados não podem ser perdidos de jeito nenhum !

(Se preferir pode ser um postgres)

Resumo da seção

Aprendemos como:

- Criar e gerencie volumes.
- Compartilhar volumes entre contêineres.
- Compartilhar um diretório host com um ou vários contêineres.

Limitando recursos

Limitando recursos

- O que acontece quando um contêiner tenta usar mais recursos do que o disponível?

  (RAM, CPU, uso de disco, ...)

- O que acontece quando vários contêineres competem pelo mesmo recurso?

- Podemos limitar os recursos disponíveis para um contêiner?

  (Alerta de spoiler: sim!)

Containers são procesos normais

```
  0  2662  0.2  0.3 /usr/bin/dockerd -H fd://
  0  2766  0.1  0.1  \_ docker-containerd --config /var/run/docker/containe
  0 23479  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
  0 23497  0.0  0.0      |   \_ nginx: master process nginx -g daemon off;
101 23543  0.0  0.0      |       \_ nginx: worker process
  0 23565  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
102 23584  9.4 11.3      |   \_ /docker-java-home/jre/bin/java -Xms2g -Xmx2
  0 23707  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
  0 23725  0.0  0.0          \_ /bin/sh
```

Por default, nada muda

- O que acontece quando um processo tenta usar muita memória num Linux ?

Por default, nada muda

- O que acontece quando um processo tenta usar muita memória num Linux ?
- Resposta simplificada:
  - swap é usada (se disponível);
  - se não tiver espaço disponivel, eventualmente, o out-of-memory killer é invocado;
  - O OOM killer usa heuristicas para matar o processo;
  - algumas vezes, as heuristicas mantam o cara errado.

Por default, nada muda

- O que acontece quando um processo tenta usar muita memória num Linux ?
- Resposta simplificada:
  - swap é usada (se disponível);
  - se não tiver espaço disponivel, eventualmente, o out-of-memory killer é invocado;
  - O OOM killer usa heuristicas para matar o processo;
  - algumas vezes, as heuristicas mantam o cara errado.
- O que acontece quando um contêiner usa muita memória?
- A mesma coisa!

Limitando a memória na prática

- `--memory` limita a quantidade de RAM física usada por um contêiner.
- `--memory-swap` limita a quantidade total (RAM + swap) usada por um contêiner.

Limitando a memória RAM

Exemplo:

```
docker run -ti --memory 100m python
```

Se o container tentar usar mais que 100 MB de RAM, e existir SWAP disponível:

- o container não será morto.

Limitando memória e swap

Example:

```
docker run -ti --memory 100m --memory-swap 100m python
```

Se tentar usar mais de 100 MB de memória, ele será morto killed.

Quando escolher qual estratégia?

- Serviços stateful (como bancos de dados) perderão ou danificarão dados quando mortos
- Permita que eles usem espaço de troca, mas monitore o uso do swap
- Serviços stateless podem ser eliminados com pouco impacto
- Limite o uso de mem + swap, mas monitore se eles são mortos

Limitando o uso da CPU

- Não há menos de três maneiras de limitar o uso da CPU:
  - definir uma prioridade relativa com `--cpu-shares`,
  - definindo um limite de% de CPU com `--cpus`,
  - fixando um contêiner a CPUs específicas com `--cpuset-cpus`.
- Eles podem ser usados separadamente ou juntos.

Definir prioridade relativa

- Cada contêiner tem uma prioridade relativa usada pelo scheduler do Linux.
- Por padrão, essa prioridade é 1024.
- Desde que o uso da CPU não seja maximizado, isso não terá efeito.
- Quando o uso da CPU é maximizado, cada contêiner recebe ciclos da CPU na proporção de sua prioridade relativa.
- Em outras palavras: um contêiner com `--cpu-shares 2048` receberá duas vezes mais que o padrão.

Definindo um limite de% de CPU

- Essa configuração garante que um contêiner não use mais do que um determinado% de CPU.
- O valor é expresso em CPUs; Portanto:

  `--cpus 0.1` significa 10% de uma CPU,

  `--cpus 1.0` significa 100% de uma CPU inteira,

  `--cpus 10.0` significa 10 CPUs inteiras.

Fixando contêineres nas CPUs

- Em máquinas com vários núcleos, é possível restringir a execução em um conjunto de CPUs.
- Exemplos:

  `--cpuset-cpus 0` força o contêiner a executar na CPU 0;

  `--cpuset-cpus 3,5,7` restringe o contêiner às CPUs 3, 5, 7;

  `--cpuset-cpus 0-3,8-11` restringe o contêiner às CPUs 0, 1, 2, 3, 8, 9, 10, 11.
- Isso não reservará as CPUs correspondentes!

  (Eles ainda podem ser usados por outros contêineres ou processos não contêineres.)

Limitando o uso do disco

- A maioria dos drivers de armazenamento não oferece suporte à limitação do uso de contêineres em disco.
- Isso significa que um único contêiner pode esgotar o espaço em disco para todos.
- Na prática, no entanto, isso não é uma preocupação, porque:
  - os arquivos de dados (para serviços com estado) devem residir em volumes
  - os recursos (por exemplo, blob, imagens, conteúdo gerado pelo usuário ...) devem residir em object stores ou em bases de dados.
  - os logs são gravados na saída padrão e coletados pelo mecanismo do contêiner.
- O uso do disco do contêiner pode ser auditado com `docker ps -s`
- Dica, usar o logrotate, ou algo do tipo `truncate -s 0 /var/lib/docker/containers/*/*-json.log`