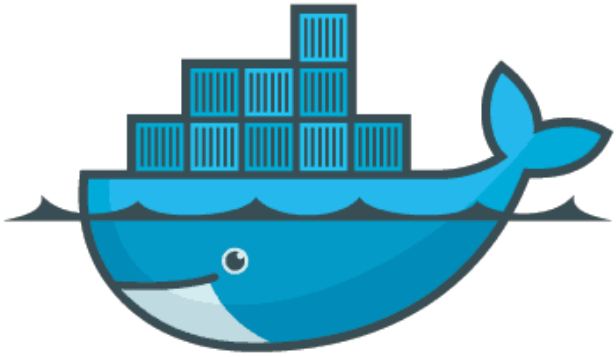
Docker



Conceitos sobre Docker

Principais Componentes

- Images
- containers
- Dockerfiles
- Registries

Images

- Imagens são templates para criação de containers, basicamente um filesystem e tipo de metadado sobre como o container deve ser construido e sobre o que o container deve rodar;
- Se você precisar mover ou transportar um container entre dois ambientes por exemplo, poderá fazer isso através do uso de imagens;
- O docker é responsável pela manipulação de imagens dentro de um host e fornecimento das ferramentas para transporte de imagens que por sua vez são armazenadas em Registries;

Dai a facilidade na manipulação de containers, a mesma imagem utilizada localmente pelo Developer é enviada a um registry e baixada para uso em produção, nem da mais pra usar a belissima desculpa: "Na minha máquina Rodou" s;

Containers

- Containers são instancias de uma imagem em execução, basicamente cópias a partir deles;
- É a partir dos containers que rodamos processos baseados nas especificações da imagem, por exemplo um node ou um tomcat;
- O Docker manipula os containers como se fossem processos sendo possível rodar, pausar ou parar um container via linha de comando;

Podemos executar alterações diretamente no container mas para que esses dados persistam é necessario que as alterações sejam replicadas na imagem

Dockerfiles

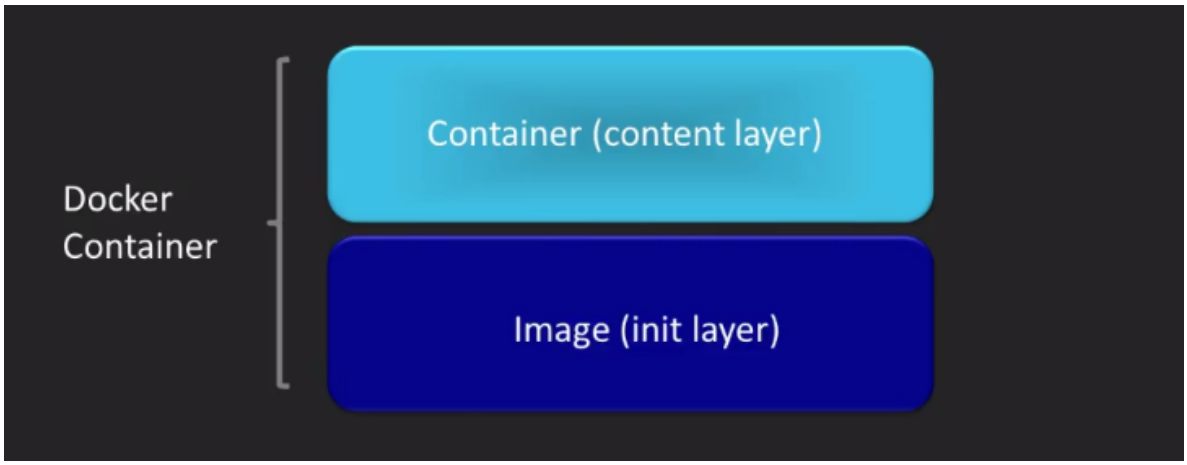
- Docker files são receitas ou arquivos de build contendo a informação sobre como um container deverá construir uma nova imagem (Execução de um build);
- O uso de dockerfile permite ao desenvolvedor manter a especificação da sua infra transformando isso em código, na maioria dos casos adicionado ao Github junto com o projeto como Neste Exemplo;

Registries

- Estruturas de Registries são utilizadas para armazenar imagens;
- Essas imagens são geradas via Dockerfile localmente durante o processo de Build e enviadas a um registry;
- O Docker mantém um registry no modelo "freemium" que pode ser usado para armazenamento de imagens, o acesso pode ser feito pela URL https://hub.docker.com/;

Layers

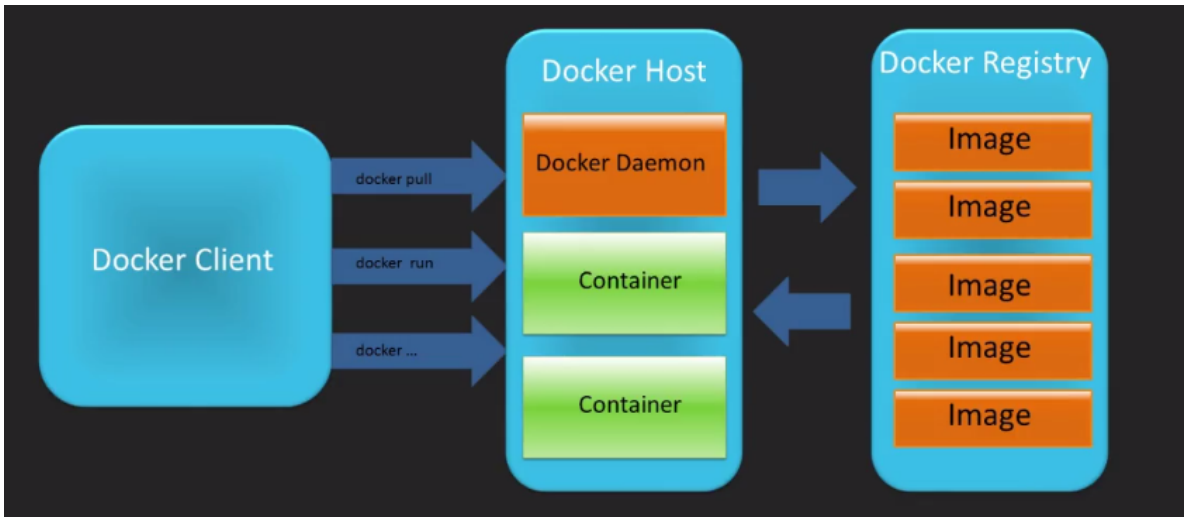
Quando iniciamos um container ele sempre é baseado em uma imagem, o container compõe essa imagem com base em layers, de acordo com a imagem original no registry;



Esse modelo facilita o processo de pull e armazenamento de imagens, uma vez que as layers são compartilhadas mantendo as imagens em tamanhos pequenos (Geralmente menos de 200mb)

Layers

Na pratica a relação entre o processo de pull o uso do registry e dos hosts que orquestram os containers pode ser descrita dessa forma:



Rodando Comandos

Existem três contextos para execução de containers:

- Execução de Comandos em Foreground;
- Execução de Comandos com interação em Foreground;
- Execução de Comandos em longos em Background;

Rodando Comandos

Execução de Comandos em Foreground:

Começando pelo básico:

```
$ docker run debian ls
```

No exemplo acima executamos um comando simples dentro do container Debian, você verá como retorno uma lista dos diretórios dentro do container, após executar a solicitação o docker devera parar o container

Rodando Comandos

Após a execução você conseguirá ver o container (Que neste momento já foi finalizado) com o comando do docker ps -a:

```
$ docker ps -a
```

Como trata-se de um container que já executou a tarefa solicitada (um simples comando ls) o container não mais encontra-se em execução, por isso o uso do "-a" para verificar todos os containers, inclusive os que não estão rodando

Rodando Comandos

Como o comando já foi finalizado o container pode ser removido executando o comando docker rm, para isso primeiro localizar o id do container:

```
$ docker ps -a

CONTAINER ID        IMAGE               COMMAND
8c52b4a18f3c        debian:latest       "ls"
```

Depois remova o container com base em seu id:

```
$ docker rm 8c52b4a18f3c
```

Rodando Comandos

É possível remover um container automaticamente após sua execução:

```
$ docker run --rm debian ping google.com
```

Finalize a execução utilizando um "Ctrl^C" procure pela imagem executando um "docker ps" o ping que estava sendo executado está rodando dentro do container

Rodando Comandos

Execução de Comandos interação em Foreground:

Outra possibilidade para a execução de comandos é o uso do formato interativo a partir da emulação de um terminal como o "bash" ou o "sh":

```
$ docker run --rm -i -t debian bash
```

Ao rodar comandos no modo interativo estamos efetivamente acessando o container, neste caso a partir do comando bash

Rodando Comandos

Execução de Comandos em longos em Background:

Finalizando é possível executar containers em Background utilizando a opção "-d" neste exemplo o container a ser executado é uma nginx:

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

A opção "-p 8080:80" refere-se ao bind da porta 8080 do host local na porta 80 do container

Como o container está sendo executado em uma camada de rede criada dentro do host hospedeiro é necessário export da porta para acesso a aplicação

Listando Containers

Abra um segundo terminal, visualize a lista de containers rodando:

```
$ docker ps

CONTAINER ID     IMAGE       COMMAND                  ...
7b058f5b8a2c     nginx...    "nginx -g 'daemon ..."    ...
```

A relação de parâmetros para esse comando pode ser consultada abaixo:

Parâmetro	Explicação
-a	Lista todos os containers, inclusive os desligados
-l	Lista os últimos containers, inclusive os desligados
-n	Lista os últimos N containers, inclusive os desligados
-q	Lista apenas os ids dos containers, ótimo para utilização em scripts

Exibição de Logs

É possível obter logs de containers em execução com o comando docker logs

sintaxe: docker logs ${OPTIONS} ${CONTAINER_NAME}

```
$ docker logs 7b058f5b8a2c
```

Uma boa prática ligada ao uso de containers indica que aplicações não devem gerenciar ou rotear arquivos de log, esse logs devem ser depositados sem qualquer esquema de buffer na saída padrão (STDOUT);

Fica por conta de uma infraestrutura externa à aplicação o armazenamento e gerenciamento desses dados;

Stateless vs Stateful

Ao utilizar plataformas baseadas em containers e modelos de aplicações nativas para cloud existe uma classificação relativa a maneira como essas aplicações lidam com dados persistentes:

Stateful: Chamamos de statefull as aplicações que possuem persistência de dados de alguma forma como por exemplo banco de dados, ou soluções que armazenam informações de usuários, como controladores transacionais ou serviços;

Stateless: Chamamos de stateless as aplicações que NÃO possuem persistência de dados como partes de microserviços ou páginas estáticas;

Dica:

Este artigo da robinsystems uma empresa que fornece paltaformas de cloud para bigadata possui um ótimo infografico sobre statefull x stateless;

Stateless vs Stateful

É muito comum que arquiteturas baseadas em containers e cloud-native sejam compostas por ambos os tipos stateless e stateful, por exemplo um serviço de autenticação poderia possuir um container estático com a página de login que utiliza um backend redis para persistência de dados da sessão;

No contexto descrito acima o uso de volume possibilitaria que vários containers rodando o redis utilizam uma mesma base, ou seja o ganho está na possibilidade de escalabilidade horizontal!

Usando Volumes

O Docker possui um mecanismo para montagem e gerenciamento de volumes por parte dos containers, ou seja além do processo de binding de arquivos usando COPY e ADD é possível gerenciar conteúdo estático usando volumes, o que trás algumas vantagens em determinadas situações:

- Volumes são mais fáceis de fazer backup ou migrar;
- Podem ser manipulados usando o cliente do proprio Docker;
- Podem facilamente ser compartilhados entre vários containers;
- O uso de drivers permite a integração com paltaformas de cloud como AWS ou Google;
- Volumes podém ser pré-configurados e populados, (o que provavelmente é sua maior vantagem);

Usando Volumes

Para demonstrar a montagem de um volume utilizaremos um exemplo simples montando o conteúdo de um container php:

Crie um diretório chamado ***php-vol***

```
$ mkdir php-vol ; cd php-vol/
```

Popule este diretório com o arquivo ***index.php***:

```
$ cat<<EOF > index.php

<?php
// Mostra todas as informações, padrão (INFO_ALL)
phpinfo();
?>
EOF
```

O phpinfo será utilizado para testar o uso do php mostrando seu atual estado e claro, será um ponto de montagem remota via volume para o documentroot do apache que por padrão lê arquivos index.*

Usando Volumes

Verifique se o arquivo foi cirado conforme esperado:

```
$ cat index.php

<?php
// Mostra todas as informações, padrão (INFO_ALL)
phpinfo();
?>
```

Para testar volumes utilizaremos a imagem php:7.0-apache

```
$ docker run -d -p 80:80 -v \
"$PWD":/var/www/html php:7.0-apache
```

Neste exemplo estamos montando o conteúdo do diretório corrente(nosso php-vol com o arquivo index.php) no diretório "/var/www/html"

Usando Volumes

IMPORTANTE: A instrução "-v" ou --"mount" do comando anterior especifica o diretório local a ser montado dentro do container, essa especificação deve ser um caminho absoluto, ou seja o caminho completo no filesystem do sistema operacional, logo caso não utiliza-se a variável $PWD a pesecificação seria algo conforme abaixo

```
$ docker run -d -p 80:80 -v \
"/home/<nome-usuario>/php-vol":/var/www/html php:7.0-apache
```

Porque funciona com $PWD?: A varíavel $PWD aponta o caminho completo para o diretório atual no sistema linux, no momento da execução este era o diretório php-vol, esse tipo de recurso é muito útil na hora de automatizar seus processos de build;

Dockerfiles

PRIVATE DOCKER CONTAINER IMAGES

Criando imagens

Como criar um Dockerfile para entregar seu código?

- A estratégia mais simples e eficiente para geração de uma imagem é a construção de um Dockerfile,
- Um Dockerfile é basicamente um arquivo texto contendo uma relação de instruções de linha de comando que farão a composição da imagem de seu container;
- A imagem é baseada em uma imagem base pré existente no Dockerhub, neste exemplo utilizaremos uma imagem com openjdk rodando Alpine;

Criando imagens

Como primeiro exemplo crie um arquivo com o nome "Dockerfile";

Adicione as duas linhas abaixo e salve o conteudo;

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
```

Criando imagens

- O processo de build é utilizado para criar uma imagem apartir de um Dockerfile;
- Neste exemplo utilizaremos uma imagem do apache com base na documentação https://hub.docker.com/_/httpd/;
- O Conteúdo do diretório public-html pode ser definido por você, se achar necessário utilize um template css do w3schools;

Criando imagens

Execute o comando docker build para criar sua imagem:

```
$ docker build -t web-css-sample .
```

```
$ docker images
```

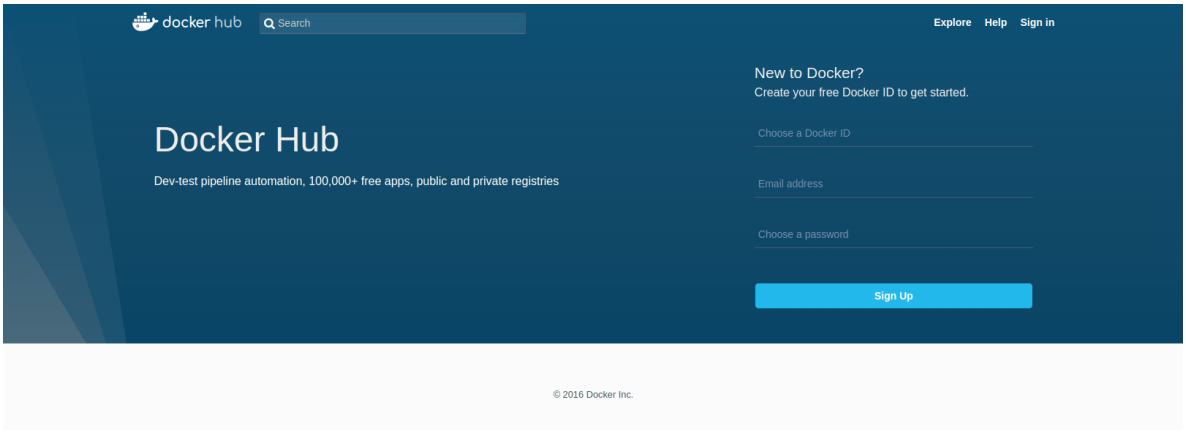
Teste seu novo container rodando o seguinte:

```
$ docker run -dit  -p 80:80 --name web-app web-css-sample
```

Executando o Upload de uma imagem

A imagem criada localmente pode ser enviada a um registry e disponibilizada para acesso de terceiros;

O Dockerhub é uma estrtura de registry publico usada para armazenamento e disponibilização de imagens;



Executando o Upload de uma imagem

Acesse o Dockerhub pelo endereço https://hub.docker.com/ e crie uma conta gratuita;

Voltando ao o docker faça o login no registry:

```
$ docker login
```

Crie uma nova tag para sua imagem com base em sua conta no dockerhub:

```
$ docker tag devfiap/web-css-sample:v0.1
```

Em seguida faça o upload da imagem:

```
$ docker push devfiap/web-css-sample:v0.1
```

Substitua "devfiap" pelo nome do SEU DOCKER HUB, acesse o dockerhub novamente e verifique se a imagem foi adicionada;

Criando uma imagem com Java

Criando um Dockerfile:

```
FROM openjdk:8-jdk-alpine
ADD target/gs-spring-boot-docker-0.1.0.jar app.jar
ENV JAVA_OPTS="-Xmx256m -Xms128m"
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -jar /app.jar" ]
```

Esse exemplo foi baseado no spring.io e pode ser baixado a partir Deste Link;

Não odeixe de ajustar os valores definidos na variável JAVA_OPTS de acordo com o sizing da sua aplicação.

Criando uma imagem com Java

Neste exemplo o Dockerfile utilizou quatro instruções:

- FROM: A instrução FROM determina qual a imagem base que será usada no Build, essa imagem pode ser qualquer imagem válida armazenada no Dockerhub ou em outra plataforma de Registry;

No exemplo anterior a imagem utilizada é uma imagem do openjdk um projeto que mantém um pacote com o Java Runtime, ela pode ser acessada diretamente no Repositório oficial do Projeto no dockerhub;

Tome um certo cuidado ao escolher quais as imagens a serem usadas em seu desenvolvimento, prefira sempre imagens oficiais de cada Projeto, verique a origem e documentação referentes que geralmente estão referenciadas na página da imagem no dockerhub

Criando uma imagem com Java

- ADD: Utilizamos essa instrução para copiar arquivos ou diretórios de uma origem <src> para dentro do Filesystem da imagem sendo executada em Docker <dest>

Sintaxe:

ADD <src> ... <dest>

- ENTRYPOINT: Esta instrução permite a configuração do container que executará seu código e comando exato a ser executado assim que o container for carregado;

No exemplo anterior o arquivo copiado foi um jar gerado via maven, este arquivo foi copiado de sua origem "target/gs-spring-boot-docker-0.1.0.jar" para o destino "app.jar

Em seguida com o arquivo criado no dirtetório / do container oepnjdk, utilizamos o Entrypoint para rodar o comando java -jar executando o conteudo do arquivo criado.

Criando uma imagem com Java

Acesse o diretório com o Projeto de Exemplo e em seguida faça o build do Projeto:

```
$ docker build -t hello_spring:version1.0 .
```

O container será disponibilizado no seu ambiente local:

```
$ docker images
```

O exemplo baseia-se no modelo proposto no ***spring.io***, Você encontrará a versão completa na [Documentação do Projeto](https://spring.io/guides/gs/spring-boot-docker/#initial)

Criando uma imagem com Java

Com a imagem criada podemos iniciar o container:

```
$ docker run -p 8080:8080 -t hello_spring:version1.0
```

Os parâmetros mais comuns na execução de containers foram listados abaixo;

Parâmetro	Explicação
-d	Execução do container em background
-i	Modo interativo. Mantém o STDIN aberto mesmo sem console anexado
-t	Aloca uma pseudo TTY
--rm	Automaticamente remove o container após finalização (**Não funciona com -d**)
--name	Nomear o container
-v	Mapeamento de volume
-p	Mapeamento de porta
-m	Limitar o uso de memória RAM
-c	Balancear o uso de CPU

Criando uma imagem com Node Js

Para complementar nossos exemplo criaremos uma imagem node usando Express 4 baseado no exemplo "Dockerizing a Node.js web app";

Para execução baixe os arquivos necessários a partir Deste Link;

Criando uma imagem com Node Js

Com o conteúdo do repositório a disposição verifique o arquivo ***Dockerfile***:

```
FROM node:carbon
WORKDIR /usr/src/app
COPY package.json ./
RUN npm install
COPY . /usr/src/app
EXPOSE 8080
CMD [ "npm", "start" ]
```

Criando uma imagem com Node Js

Comparando com o exemplo anterior temos algumas alterações:

- COPY: Neste exemplo utilizamos COPY ao invés de ADD, o conceito de copia de origem para destino é o mesmo porém a instrução COPY é mais "limitada" já que trabalha somente com origem e destino referenciados diretamente via caminho enquanto o ADD permite a expansão de arquivos compactados e o uso de Links simbólicos;

Sintaxe:

COPY <src> ... <dest>

> Embora a instrução ADD seja mais completa o manual de boas práticas para criação de Dockerfiles recomenda que sempre que possível utilize apenas o COPY, esse manual pode ser conultado abaixo

Best practices for writing Dockerfiles

Criando uma imagem com Node Js

- RUN: O run é um recurso muito util na construção de um Dockerfile, ele permite a execução de tarefas no processo de compilação:

Sintaxe:

RUN <command>

No exemplo do node a instrução RUN é necessária para executar etapas do processo de build como o uso do NPM para instalação de dependências.

Criando uma imagem com Node Js

- CMD: A instrução CMD é utilizada para especificar a instrução default a ser executada quando o container for chamado, isto é após o build, no processo de execução (docker run -d ...):

Sintaxe:

CMD ["executable","param1","param2"]

Assim como a questão entre COPY e ADD os comandos CMD e ENTRYPOINT possuem algumas similaridades, neste caso a diferença é que sempre existirá um enypoint que por padrão será "/bin/sh -c" sendo assim ao especificar apenas o CMD o comando especificado nele será executado por "/bin/sh -c"

Criando uma imagem com Node Js

Dockerfile: ENTRYPOINT vs CMD

> E se compararmos os exemplos? você verá que no Dockerfile do exemplo java o entrypoint utiliza o comando "sh -c", que já é o padrão de um container linux, logo o mesmo comando funcionaria se ao invés de ENTRYPOINT utilizassemos o CMD, ou seja no final não haveria diferença...

> Resumindo o conceito a diferença básica é que uma instrução do tipo CMD pode ser sobreposta por outra instrução passada pelo usuário, enquanto um ENTRYPOINT definará exatamente o que o cotainer irá rodar, e não será substituido sem que uma nova imagem seja criada.

Esse conceito está bem definido e exemplificado neste artigo do CenturyLink

Criando uma imagem com Node Js

- EXPOSE: O expose informa ao Docker em qual porta o container e consequentemente a aplicação fará o bind para receber conexões.

Sintaxe:

EXPOSE <port> [<port>/<protocol>...]

É importante entender que a instrução expose NÃO CRIA o vinculo com a porta de conexão para que seja publicada, na verdade ela atua como um tipo de documentação, ou um contrato sobre qual a porta a ser usada.

Do ponto de vista de execução do container a porta a publicada para que a aplicação receba conexões é definida pelo parâmetro "-p" passado no momento da execução conforme nosso próximo exemplo

Criando uma imagem com Node Js

Para validar nosso novo Dockerfile execute o build da imagem:

```
$ docker build . -t node-js-sample:version1.0
```

Em seguida execute seu novo container definindo qual a porta em listen:

```
$ docker run -d -p 80:8080 -e PORT=8080 node-js-sample:version1.0
```

Neste exemplo o uso da variavel de ambiente PORT altera o comportamento default da linguágem node que é fazer o bind na porta 5000,(verifique como essa config foi estabelecida na linha 4 do arquivo index.js)

Além disso para que a aplicação seja publicada foi usado a porta 80, o parâmetro -p 80:8080 especifica que o conteúdo da porta 8080 do cotainer será publicado na porta 80 do host/solução que roda o docker.

Debbuging

DOCKER PS

Os containers em execução podem ser listados utilizando o comando docker ps:

```
$ docker ps

CONTAINER ID        IMAGE           COMMAND              ...    NAMES
b4dcd19a02be        nginx:latest    "nginx -g 'daemon"   ...    pensive_mir
```

Neste exemplo existe apenas um container rodando o Nginx em execução

Pull de Imagens

Outra categoria de erros muito comuns são erros relacionados ao pull de imagens:

- Em situações onde estiver ocorrendo erros no processo de pull de imagem uma mensagem similar a mensagem abaixo será apresentada:

  @@@shel Unable to find image 'ubuntu:test' locally docker: Error response from daemon: manifest for ubuntu:test not found. See 'docker run --help'.

Verificar o caminho para imagem do container e as permissões de acesso no Registry (Em nosso caso o Dockerhub) pode ser um bom começo para entender esse erro

Outro processo útil é executar o pull manualmente em outro host utilizando docker, não se esqueça que, caso o repositório seja privado um token de autenticação será necessário para executar o pull da imagem

Verificação de Logs

Dentro das pods é possível verificar logs de containers de forma similar ao processo executado via Docker logs:

sintaxe:

```
$ docker logs ${CONTAINER_NAME}
```

Exemplo:

```
$ docker logs hello_spring:version1.0
```

Executando Comandos

Em alguns cenários pode ser necessário atuar dentro da Pod, esse processo pode ser executado via "docker exec":**

```
$ docker exec -it ${CONTAINER_NAME} {COMMAND}
```

Exemplo:opção

```
$ docker exec -it hello_spring:version1.0 ls
```

Substitua o /bin/sh pelo comando a ser executado dentro do container, caso a Pod possua apenas um container a -c ${CONTAINERNAME} pode ser omitida

Obrigado!