Play with Docker for ITs and Admins

# First Alpine Linux Containers

Primeira aula do tutorial Play with Docker.

Visitar o link, pois as imagens no tutorial ajudam no entendimento do conteúdo.

## Link

<https://training.play-with-docker.com/ops-s1-hello/>

## Containers

Executar um container:

docker container run *<image-name>*

Exemplo do Play with Docker:

docker container run hello-world

Ao rodar o comando no terminal o docker irá procurar localmente uma imagem (image) chamada “hello-world”.

Quando não encontra é procurado no Docker Store, o registro docker (docker registry) padrão. Se a image é encontrada, ela é baixada e executada num container.

* VM: abstração de *hardware*
* Container: abstração de *aplicação*

## Images

Baixar uma image:

docker image pull *<image-name>*

Exemplo do Play with Docker:

docker image pull alpine

Alpine é um linux leve disponível no docker registry padrão, o Docker Store.

Listando todas as images no sistema:

docker image ls

## Executando um container

Executando:

docker container run *<image-name> <extra-commands>*

Exemplo do Play with Docker:

docker container run alpine ls -l

Ao executar o comando são realizados os seguintes passos:

* O comando run procura um image com o nome indicado
* Cria um container
* Executa o comando no container
* Finaliza o container

Para utilizar o terminal interativo dentro de um container com Linux é necessário utilizar a flag “-it”. Exemplo:

docker container –it alpine /bin/sh

## Containers Isolados

Toda vez que o run é executado, é criado um novo container com a image indicada. Cada container é isolado, possuindo seu próprio filesystem e namespace. Por padrão, um container não possui meios de se comunicar com outro containers, mesmo que eles sejam criados de uma mesma image.

Exemplo de isolação de containers:

docker container run –it alpine /bin/ash

echo “hello world” > helloworld.txt

ls

exit

O trecho de código:

* Cria um container de alpine
* Abre o terminal do alpine dentro do container
* Cria um arquivo .txt com o conteúdo “hello world”
* Lista os arquivos no container
* Encerra o shell

Em seguida, executar o comando:

docker container run alpine ls

O código irá listar novamente as arquivos no container, no entanto não haverá o arquivo .txt, visto que é criado um novo container. Ou seja, é uma nova instância completamente separada do container criado anteriormente.

Esse conceito é importante para manter a segurança no sistema.

Executando novamente o container com o arquivo de texto:

docker container start *<container-id>*

O id do container pode ser identificado com o código que lista os containers:

docker container ls –a

Após executar o comando start, é possível verificar que o container continua ativo:

docker container ls

Isso ocorre porque o shell do tipo “/ash” fica aguardando um comando para ser executado.

Para enviar um comando nesse container, deve ser utilizado o comando “exec”:

docker container exec *<container-id>* *<command>*

No exemplo, para listar novamente os arquivos no container com o arquivo .txt:

docker container exec *<container-id>* *ls*

# Doing More With Docker Images

Customização de images.

## Link

<https://training.play-with-docker.com/ops-s1-images/>

## Criando um image a partir de um container (Commit)

Criando um container de exemplo: ubuntu com figlet.

docker container run –ti ubuntu bash

apt-get update

apt-get install –y figlet

figlet “hello docker”

exit

Para criar uma nova image é necessário saber o id do container:

docker container ls -a

Antes de criar a image é possível inspecionar as mudanças realizadas no container que foi criado. Utilizando o comando “diff”, é possível verificar todos os arquivos que foram criados ou modificados:

docker container diff *<container-id>*

Para criar uma image é necessário dar commit no container. Um commit cria uma image localmente no sistema que está executando do Docker.

docker container commit *<container-id>*

O comando commit cria um image sem repository e tag. É possível vê-lo com o comando que lista as images locais: docker image ls

### Tagging

Tagging é atribuir um valor à tag de uma image criada.

É necessário conhecer o id da image para poder fazer um tagging, verifica-se o id por meio da listagem de images:

docker image ls

Comando para tagging:

docker image tag *<image-id> <tag-name>*

Exemplo no Play with Docker:

docker image tag *<image-id>* ourfiglet

### Testando o exemplo

É possível executar o ourfiglet:

docker container run ourfiglet figlet Hello

## Criando image com Dockerfile

É possível utilizer o Dockerfile para criar images.

O resultado final em relação ao commit é o mesmo. No entanto, o Dockerfile é um conjunto de instruções de como construir a image. Sendo útil na hora de realizar mudanças à image.

Criando uma aplicação NodeJs de exemplo:

vi index.js

Código em javascript:

var os = require(“os”);

var hostname = os.hostname();

console.log(“hello from ” + hostname);

O alpine é o OS base do container, será adicionado também um runtime do NodeJS e uma cópia do código fonte (index.js).

Também será especificado qual o comando a ser usado na criação do container, “hello:v0.1”.

Criar um arquivo com o nome Dockerfile:

vi Dockerfile

Adicionar o código no Dockerfile:

# Utiliza o container alpine como base:

FROM alpine

# Executa “apk update” e “apk add nodejs” no alpine:

RUN apk update && apk add nodejs

# Copia o conteudo de host (index.js) para a pasta /app no alpine:

COPY . /app

# Altera o diretorio de inicialização para /app:

WORKDIR /app

# Atribui um comando para executar na inicialização do container:

# node index.js

CMD [“node”, “index.js”]

Build:

docker image build –t hello:v0.1 .

O Dockerfile permite especificar como construir o container, e o que o container deve fazer quando é executado. Uma vez que um container, construído pelo dockerfile, é colocado num repositório: outros usuários não precisam construi-lo por eles mesmos, nem precisam saber quais comandos foram utilizados para construí-lo.

## Image Layers

As imagens construídas no Docker são construídas em camadas (layers).

Comando que lista todos os containers intermediários que foram criados antes de criar o build final:

docker image history *<image-id>*

Alguns dos containers intermediários serão layers no build final.

O conceito de layers é interessante quando precisamos fazer mudanças no aplicativo. Pois essa mudança será em apenas nas camadas necessárias.

Exemplo com utilizando os arquivos anteriores: adicionando mais um comando em index.js

echo “console.log(\”this is v0.2\”);” >> index.js

Construindo uma nova imagem com o código atualizado:

docker image build –t hello:v0.2 .

A mensagem de saída exibe a palavra “Using cache” em alguns pontos, pois o Docker reconhece que já construiu algumas dessas layers anteriormente. Como não houve mudança nelas, o Docker usa a versão que já está no cache.

### Volume

Um layer especial que permite a pesistência de dados. Permite os dados seram compartilhados entre os containers.

## Inspecionando image

Retorna detalhes da image: comandos que executa, qual o OS, quais layers é composta ...

docker image inspect *<image-name>*

Como o retorno é um JSON, é possível filtrar as informações. Exemplo: layers da image.

docker image inspect --format “{{ json .RootFS.Layers }}” *<image-id>*

A image criada (hello), deve possuir 3 layers:

* Alpine: OS base (comando FROM)
* Comando RUN
* Comando COPY

Cada layer reprenta um comando no Dockerfile. Com excessão da última, cada layer existente na image é *read-only*.

# Swarm Mode Introduction

Introdução ao Swarm.

O Docker possui duas ferramentas para gerenciar vários containers numa rede: Docker Compose e Docker Swarm Mode.

* Compose: controla mútiplos containers num único sistema. Existe um arquivo de texto que descreve qual image utilizar, quantas instâncias, as conexões na rede.
* Swarm Mode: indica ao Docker que existem vários engines do Docker e que quer coordenar todas as operações que ocorrem entre eles. Possui as funções do Compose, mais função de definir e manter uma alta disponibilidade, escalonamente, balanceamento de carga...

## Iniciando o Swarm

Swarms em produção devem possuir no mínimo 3 nós (nodes) managers, para garantir um cluster com alta disponibilidade.

Um node manager pode executar as mesmas tarefas de um worker, no entanto não é recomendado.

Para iniciar o modo Docker Swarm deve ser utilizado o seguinte código:

docker swarm init --advertise-addr $(hostname -i)

Ao executar o código, ele criará o primeiro swarm manager e ficará escutando o ip retornado pelo comando (hostname -i).

Adicionando workers ao Swarm:

docker swarm join --token SWMTKN-1-*<token>* *<ip:port>*

O comando para criar um worker, as informações com o token e o ip são retornados na mensagem quando é criado o swarm.

## Listando os membros do swarm

A listagem dos nodes deve ser feita num node manager:

docker node ls

O node com manager-stat Líder (Leader) , é o controle principal que gerencia todos os nodes managers. Quando um Leader node cair, os outros node managers elegerão um novo Leader.

Um manager pode ser um worker também.

Um worker não pode ser um node manager, a não ser que seja explicitamente promovido.

## Aplicação de Exemplo

A aplicação de exemplo é um votador entre gatos e cachorros.

### Clonando o repositório

O repositório deve ser clonado no terminal do node manager:

git clone https://github.com/docker/example-voting-app

cd example-voting-app

### Deploy do stack

Um stack é um grupos de serviços que fazem o deploy em conjunto.