Docker Unleashed Documentation

Release 1.5

Gabriel Abdalla

jul 24, 2018

Sumário

Contents:

CAPÍTULO 1

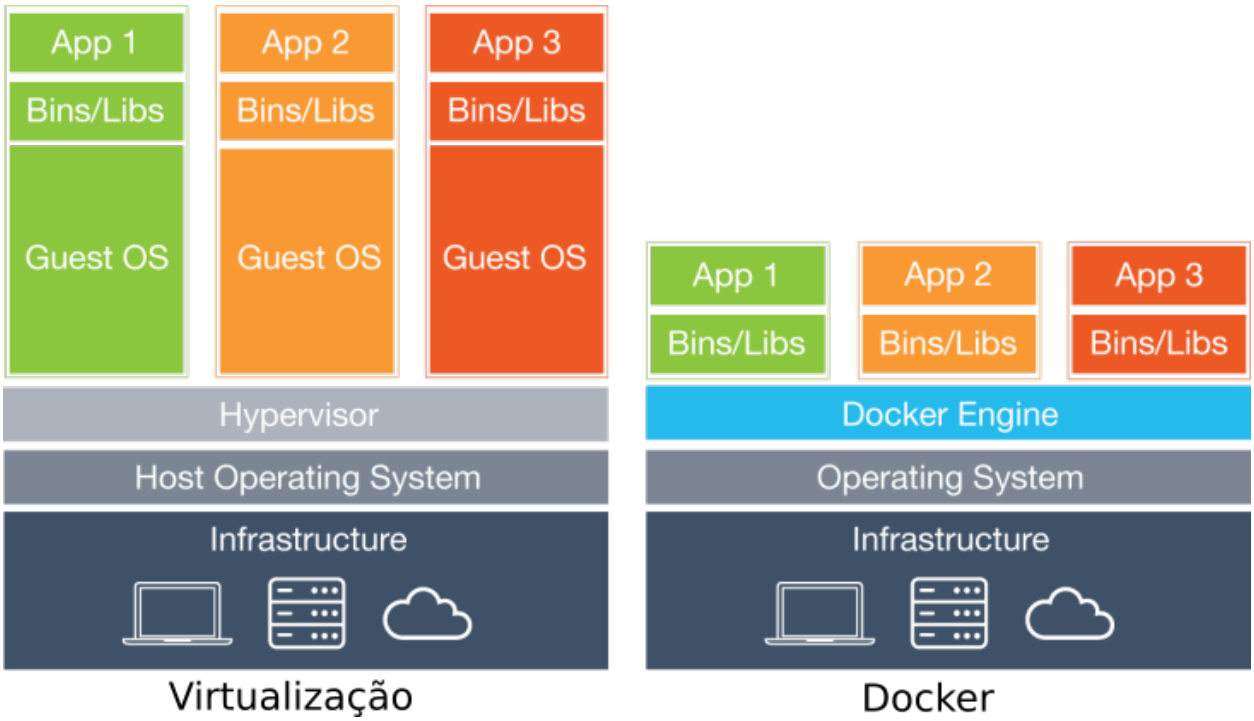
Aula 1: Docker e Microsserviços - Introdução

1.1 O que é o docker?

Docker é uma tecnologia de virtualização de processos, em que aplicações são encapsuladas em contêineres e iniciadas como um processo isolado no sistema operacional.

Um contêiner, por sua vez, contém todas as bibliotecas necessárias para o funcionamento de uma solução específica, sendo possível ter uma variedade de contêineres com diferentes versões do mesmo software ou bibliotecas rodando em um ambiente.

O docker se diferencia **principalmente** das tecnologias de virtualização no ponto em que cada contêiner se reflete em um grupo de processos separados do sistema operacional, mas que reproveitam a estrutura de software e hardware; por outro lado, as tecnologias de virtualização entregam desde o hardware virtual até o sistema operacional, refletido da maneira abaixo:

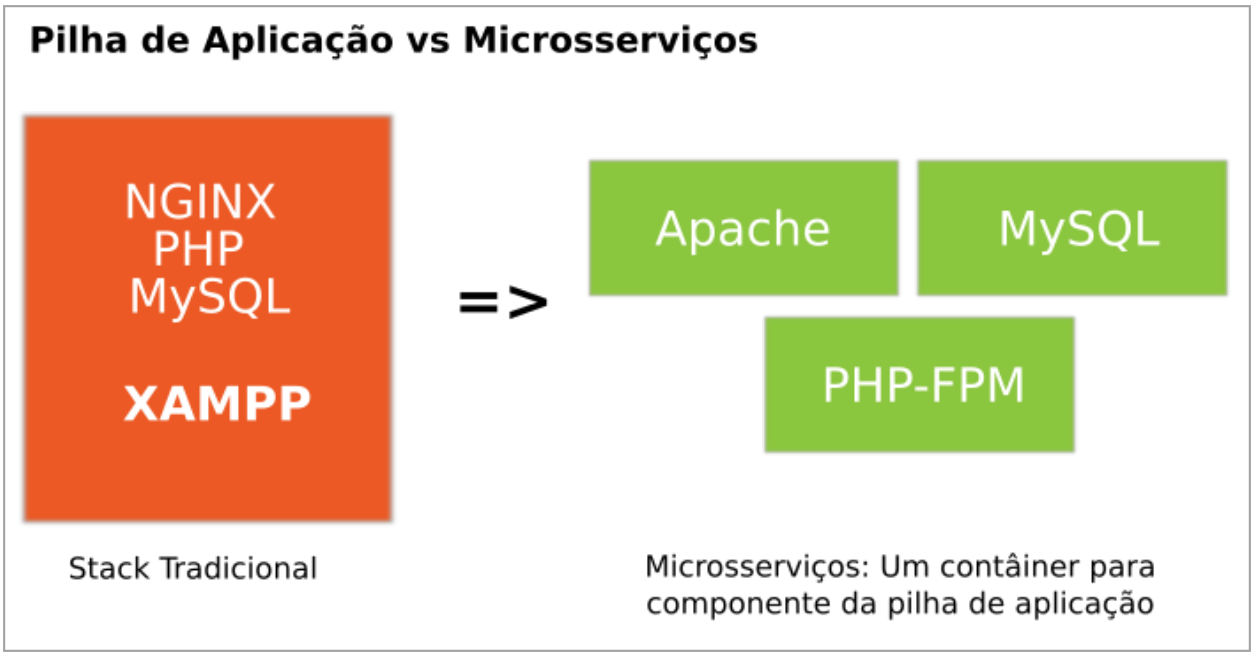


Dessa forma, contêineres do docker normalmente podem ser instalados em diferentes sistemas operacionais sem prejuízo de funcionamento, salvos os casos em que a aplicação precise de algum recurso específico que não esteja presente no kernel do host em que o docker está instalado.

1.2 O que são Microsserviços?

Uma arquitetura de microsserviços consiste numa abordagem que enfoca o desenvolvimento e manutenção de softwares em pequenas partes, ou microsserviços, em que cada processo funciona em um contêiner distinto e se comunica através de protocolos simples, como o HTTP.

Microsserviços são um contraponto direto da distribuição de pilhas de aplicação, como pode ser visto no exemplo abaixo:



1.3 Em quais Sistemas Operacionais o Docker funciona?

Atualmente o Docker funciona exclusivamente em sistemas operacionais de 64 bits, especificamente:

- Windows: Historicamente, era necessário realizar a criação de uma máquina virtual com Linux para então realizar a instalação do docker ou ainda utilizar o docker-machine para automatizar o processo. Atualmente encontra-se em testes uma versão com suporte nativo as versões Professional e Ultimate do sistema operacional (que é integrado ao Hyper-V) além da versão de produção que sairá em conjunto com o Windows Server 2016;
- MacOS: Nos mesmos moldes do Microsoft Windows, mas utilizando o xhyve como virtualizador.
- Linux: Qualquer versão não jurássica do Sistema Operacional: Debian 8, Ubuntu 14.04, Centos 7, Arch Linux, etc.

1.4 Instalação do Docker

A instalação do Docker é um processo relativamente simples; considerando como sabor Linux as distribuições Ubuntu (16.04) e Centos (7), os passos de instalação encontram-se disponíveis em:

https://docs.docker.com/engine/installation/linux/docker-ce/centos/ https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/

A instalação consiste em adicionar o repositório docker ao sistema operacional, bem como suas dependências e instalar o pacote **docker-ce**.

Algumas características dignas de nota são:

- Por padrão, o gerenciamento do docker somente pode ser realizado na máquina em que foi instalado ;
- O docker não escuta em portas tcp mas em um socket disponível /var/run/docker.sock;
- Somente o usuário root e o grupo docker possuem acesso ao socket.

Para visualizar essas características, utilizaremos alguns comandos próprios do docker, conforme exemplos abaixo:

```
$ docker ps
```

No exemplo acima, o retorno para o comando deve ser algo como:

```
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
```

O que acontece provavelmente, é que o usuário atual não possui acesso ao socket em /var/run/docker.sock, que é URI padrão que o cliente docker utiliza em seus comandos. Para remediar essa questão existem algumas possibilidades:

1. Rodar o comando como root;
2. Permitir a execução do comando docker como sudo para um usuário específico;
3. Incluir o usuário no grupo 'Docker';
4. Incluir a permissão para o usuário específico no socket do Docker.

É recomendável que apenas usuários específicos possam gerenciar o docker.

Antes de conceder o acesso, no entanto, faremos outras verificações:

O serviço está iniciado?

```
$ systemctl status docker.service
```

O retorno para este comando normalmente é algo como:

```
docker.service - Docker Application Container Engine
  Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: d
  Active: active (running) since Sáb 2016-06-18 09:08:16 BRT; 4h 25min ago
```

No entanto, se o serviço estiver parado, o retorno será o seguinte:

```
docker.service - Docker Application Container Engine
  Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: d
  Active: inactive (dead) since Sáb 2016-06-18 13:36:57 BRT; 2s ago
```

Nesse caso é necessário iniciar o serviço através do seguinte comando:

```
# systemctl start docker.service
```

Adicionalmente, para o que docker seja iniciado durante o processo de boot do computador é necessário usar o seguinte comando:

```
# systemctl enable docker.service
```

Para que o usuário comum possua acesso ao socket do Docker será necessário adicioná-lo ao grupo do docker através do seguinte comando:

```
# usermod -aG docker <USER>
```

Após o comando será necessário relogar ou aceder temporariamente como membro do grupo através do comando *newgrp docker*.

1.5 Imagens, Contêineres e seu funcionamento

Geralmente, ao se trabalhar com o Docker, passamos a tratar quase que unicamente com duas características do software: As Imagens e os Contêineres.

Uma imagem pode ser descrita como a implementação de um processo como um microsserviço; a **imagem funciona como uma fôrma** para a disponibilização potencialmente massiva desse microsserviço, cujo resultado são os contêineres. Efetivamente, ao se inserir um processo (mysql, apache, nginx, etc) na forma de uma imagem, torna-se muito simples realizar a disponibilização desses serviços.

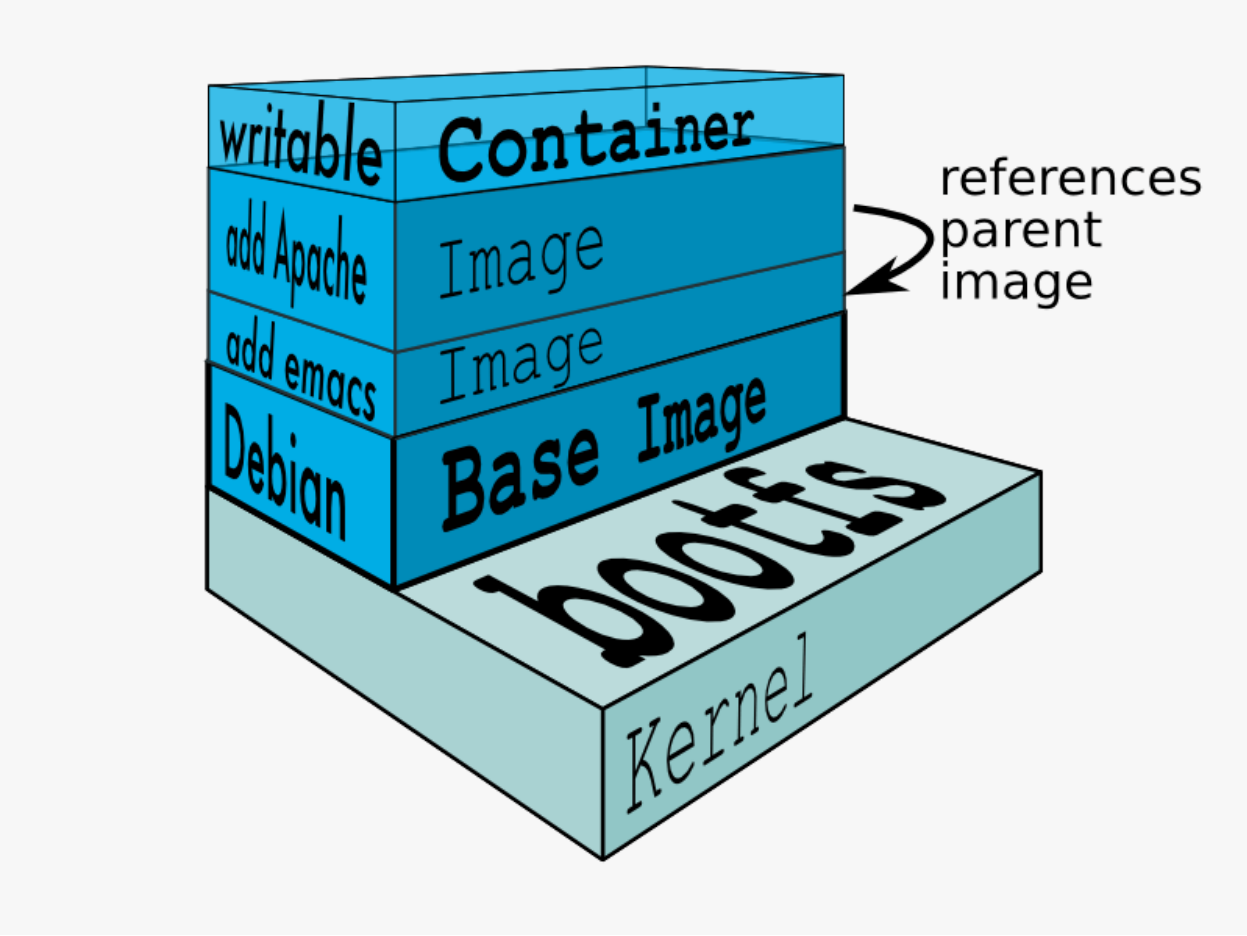
Um exemplo a ser pensado seria a necessidade de um colaborador da empresa Y precisar disponibilizar 5 instalações de MySQL para 5 diferentes clientes; no modelo atual, o colaborador instala 5 Máquinas virtuais, as configura e, por fim, instala e configura a aplicação alvo. No Docker, por outro lado, esse mesmo colaborador teria uma imagem do software "MySQL" e faria a criação de 5 contêineres a partir do mesmo da seguinte forma:

```
# docker run -d --name mysql-cliente1 -v /network/cliente1:/var/lib/mysql mysql:5.7
# docker run -d --name mysql-cliente2 -v /network/cliente2:/var/lib/mysql mysql:5.7
# docker run -d --name mysql-cliente3 -v /network/cliente3:/var/lib/mysql mysql:5.7
# docker run -d --name mysql-cliente4 -v /network/cliente4:/var/lib/mysql mysql:5.7
```

Nesse contexto, a imagem figura como uma **matriz** que contém o MySQL e todas as ferramentas, bibliotecas e configurações necessárias para seu funcionamento previamento empacotados e cada contêiner é o serviço em si.

1.6 Imagens e Contêineres

Como dito anteriormente, as imagens servem como fôrmas para a criação de inúmeros contêineres que irão disponibilizar os serviços desejados. Comumente, a arquitetura de uma imagem pode ser pensada (ainda que de forma simplória) da seguinte forma:



Conforme ilustrado na figura acima, uma imagem é um grupo de camadas que contém desde os software básico de um sistema operacional até os arquivos relacionados ao software que se deseja inserir na imagem. Por outro lado, o contêiner corresponde a parte volátil, ou seja, durante o funcionamento de um contêiner o gerenciamento de arquivos relacionado àquele contêiner somente se dá nessa camada, não permitindo a mudança dos arquivos relacionados a imagem.

Por conta dessa estrutura (que na verdade que é um CoW - Copy On Write) inicialmente só o espaço relacionado a imagem será gasto no sistema de arquivos; diferentemente do que acontece na virtualização convencional, 5 contêineres inicializados incorrem em uso total de espaço igual ao total da imagem (230MB por exemplo) e não 5 x 230MB. A medida que os arquivos dos contêineres forem mudando, aí sim, espaço adicional será utilizado.

Outra característica digna de nota é que, uma vez que uma imagem é criada, esta pode ser redistribuída e implantada em qualquer servidor que possua o docker instalado.

Nota: A especificação acerca da imagem utilizada pelo Docker está disponível em: https://github.com/docker/docker/blob/master/image/spec/v1.md

---

1.7 Adquirindo Novas Imagens

Atualmente, o docker conta com um repositório comunitário que é o **hub.docker.com**: o repositório contém imagens consideradas base (que possuem apenas o sistema operacional, utilizadas como ponto de partida para criação das imagens de aplicação), imagens oficiais das aplicações mais conhecidas (mysql, postgres, redis, etc) e imagens que são contribuições da própria comunidade.

Quando se deseja iniciar um novo contêiner precisamos, primeiramente, receber ou construir a imagem. Para saber quais imagens já estão disponíveis em um host, pode-se utilizar o seguinte comando:

```
$ docker images
```

O resultado varia de acordo com a quantidade de imagens instaladas em determinado computador, podendo ser algo como:

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
httpd               latest              0eb2069ce0d9        2 weeks ago         195.4 MB
[root@localhost ~]#
```

Ou, caso não haja imagens instaladas, o seguinte resultado:

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
[root@localhost ~]#
```

Como pode ser visto nas figuras anteriores, uma imagem possui os seguintes atributos:

- Repository (Repositório): Repositório de onde a imagem foi recebida. Quando uma imagem não possui um FQDN antes de seu nome (por exemplo 'postgres'), implica em dizer que essa imagem foi recebida a partir do repositório oficial do Docker, o hub.docker.com é uma imagem oficial. Outras informações no capítulo relacionado a **tags** posteriormente;
- Tag: Etiqueta atribuída a uma determinada imagem. Comumente relaciona a versão do software que foi encapsulada. Ex: postgres:9.6;
- Image Id: utilizado tanto para a verificação da imagem durante seu download quanto para identificação (além do nome).

- Created (Data de Criação): Data de criação da imagem;
- Size (Tamanho): Tamanho da imagem em múltiplos de Bytes (MB, GB).

Para realizar o recebimento de uma nova imagem, neste caso hospedada no site hub.docker.com, é possível utilizar o seguinte comando:

```
$ docker pull hello-world
```

Nota: O comando **docker pull** assume que a imagem será salva a partir do hub.docker.com, porém é possível ter os próprios repositórios de imagem. Mais informações acerca do capítulo sobre o "Registry".

1.8 Contêineres

Uma vez que já temos a imagem salva no host, basta agora realizar a criação novos contêineres a partir da mesma:

```
$ docker run hello-world
```

A execução do comando deve apresentar o seguinte resultado:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

No entanto, se verificarmos se o contêiner ainda está em funcionamento ...

```
$ docker ps
```

Veremos que o mesmo não é mais listado dentre os contêineres em funcionamento. Para visualizarmos todos os contêineres, inclusive aqueles que não estão iniciados ou que estão com algum outro *status*, é possível utilizar o seguinte comando:

    $ docker ps -a

Por que o contêiner encontra-se em estado **Parado**?

Conforme havíamos falado nos capítulos anteriores, um contêiner efetivamente disponibiliza um processo de uma aplicação; isso quer dizer que, caso a aplicação ou processo que esteja encapsulado no contêiner não esteja mais ativo

ou **esteja em background**, *o próprio docker assume que aquele contêiner já executou o serviço e coloca o contêiner dentre os não ativos usando o código de saída do mesmo.*

Um segundo exemplo, mas que mantém o contêiner funcionando seria o seguinte:

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

No caso acima, o contêiner executará eternamente o comando "echo hello world" ... que apesar de algo tolo, há de manter nosso contêiner em funcionamento:

```
$ docker ps
```

Ao iniciar o contêiner temos as mensagens de funcionamento da Aplicação, que utiliza o saída padrão no terminal atual impedindo que sejam feitas outras ações; parar o contêiner nesse caso pode ser feito através da combinação de teclas CTRL+C; o contêiner estará em um estado parado e poderá ser iniciado posteriormente de forma não interativa através do seguinte comando:

```
# docker start <CONTAINER_ID>
# docker start <CONTAINER_NAME>
```

É possível ainda definir um nome para o contêiner durante e após a sua criação, conforme exemplos abaixo:

```
# docker run --name apache httpd:2.4
# docker rename apache httpd24
```

Para remover um contêiner que esteja parado, pode-se utilizar o seguinte comando:

```
$ docker rm apache
```

Para a remoção de um contêiner que esteja em funcionamento pode-se utilizar o seguinte comando:

```
$ docker rm -f apache
```

É **imperativo** que estas funcionem em primeiro plano (em inglês **FOREGROUND**) para que o contêiner possa ser manter ativo e o serviço em si disponível. Um exemplo clássico é o funcionamento do Apache como um microsserviço: o comando executado no contêiner é o 'httpd -foreground', o que impede que o Apache vá para segundo plano e evitando que o contêiner seja finalizado.

Nota: A imagem será automaticamente baixada do repositório hub.docker.com sem a necessidade de realizar o **pull** antes quando esta não existir no servidor atual.

É possível ainda fazer a utilização de um contêiner e após a saída do processo aberto no mesmo o próprio docker pode fazer a remoção deste. Para estes casos pode-se utilizar o comando

```
$ docker run -it --rm debian bash
```

O comando acima utilizará uma imagem debian como base para criação do contêiner e abrirá um shell em modo interativo; quando da saída desse shell, através do comando exit, o contêiner será removido.

1.9 Contêineres: *Debugging* e Execução de Comandos 'Ad-Hoc'

Uma vez que um contêiner encontra-se iniciado é possível realizar o *debugging* de tarefas acessando o shell do mesmo ou enviando comandos diretamente:

```
$ docker exec <CONTAINER> <CMD>
$ docker exec <CONTAINER> python --version
```

É possível ainda realizar a execução do comando em *background* através do parâmetro "-d" conforme demonstrado abaixo:

```
$ docker exec -d <CONTAINER> <CMD>
```

Caso o comando a ser utilizado precise de interação ou ainda um pseudo terminal, então utilizar-se-á o parâmetro '-it':

```
$ docker exec -it <CONTAINER> bash
```

Em casos em que aplicações aceitem sinais para funcionamento (muitas aplicações aceitam o sinal HUP afim de verificar mudanças de configuração e aplicar sem a necessidade da interrupção do serviço), pode-se utilizar o comando *docker kill*,conforme sintaxe abaixo:

```
$ docker kill -s HUP <CONTAINER>
$ docker kill <CONTAINER> #Equivale a docker kill -S KILL <CONTAINER>
```

1.10 Criação de Imagens

Para então dar prosseguimento a criação de um novo microsserviço, deve-se criar primeiramente um DockerFile, que nada mais é do que a receita para construção de uma imagem. É possível ainda gerar uma nova imagem a partir de um contêiner em funcionamento (sem um Dockerfile) transformando-o em uma imagem.

Para ilustrar o uso de um DockerFile, faremos a criação do arquivo e a construção da imagem a partir deste; para tanto, execute os seguinte passos:

1. abra o prompt de comandos;
2. crie uma pasta chamada "stress";
3. acesse a pasta e crie um arquivo chamado "Dockerfile" (atenção para o "D" maiúsculo: o docker diferencia maiúsculas e minúsculas no momento da construção da imagem);
4. Copie e cole o seguinte texto dentro do arquivo:

```
FROM fedora:latest
RUN yum -y install stress && yum clean all
ENTRYPOINT ["stress"]
```

5. Salve e feche o arquivo;
6. Execute o comando "docker build -t stress ."

A diretiva final "docker build -t stress ." indica ao docker que uma imagem será construída a partir do diretório local, será etiquetada como "stress" (o que facilitará seu reconhecimento posterior), sendo que automaticamente o docker seleciona o arquivo Dockerfile no diretório atual.

Para visualizar a imagem, execute novamente o seguinte comando:

```
$ docker images
```

Vale lembrar que o Dockerfile é uma maneira transparente de compartilhar e averiguar a criação de uma imagem e é o método mais indicado de trabalho com o Docker.

No entanto, o próprio Docker disponibiliza um meio para visualizar a forma como determinada imagem foi declarada através do comando docker history, que pode ser utilizado da maneira abaixo:

```
$ docker history <IMAGE> --no-trunc
```

1.11 Docker Commit

Por vezes, temos o fato de que um contêiner poder ter evoluído ao ponto de que precisa ser várias vezes configurado ou implementado, o que implica em dizer que o mesmo deverá se tornar uma imagem; casos como a necessidade de inserção de uma licença de software após a sua inicialização é um dos mais comuns para esse uso. Nesse caso pode-se realizar as mudanças no contêiner e então realizar o **commit**, o que resultará em uma nova imagem. Quando da utilização de um commit para a criação de uma imagem, basta acessar ou criar um contêiner e realizar a configuração do mesmo como em um servidor comum.

Para ilustrar o processo, iniciaremos através da criação de um contêiner com base em Debian, cujo comando a ser executado será o shell Bash:

```
# docker run -it debian bash
```

Uma vez dentro do novo contêiner, podemos iniciar a configuração do contêiner:

```
# apt-get update && apt-get install wget -y
# exit
```

Ao sair do contêiner, o mesmo estará **parado**, pois o comando bash foi finalizado. Antes do commit será necessário saber o ID do mesmo:

```
# docker ps -a
```

Com ID em mãos, resta apenas criar uma nova imagem a partir do mesmo, através do seguinte comando:

```
# docker commit <ID> myimage:latest
```

Nota: Imagens criadas através de commits tem uma rastreabilidade e capacidade de reprodução muito baixas; algumas soluções no entanto tentam retornar quais comandos foram utilizados para a criação de uma imagem, como por exemplo em: https://github.com/CenturyLinkLabs/dockerfile-from-image.

1.12 Gerenciamento de Imagens

Além da criação e aquisição de novas imagens, o docker permite o gerenciamento daquelas presentes no host do docker.

A listagem das imagens presentes no host atual pode ser feita da seguinte forma:

```
# docker images
```

A opção de remoção é uma das mais comuns; para remoção de uma imagem, desde que a mesma não esteja em uso por algum contêiner, pode-se utilizar o seguinte comando:

```
# docker rmi <ID>
# docker rmi <NAME>:<TAG>
```

Por fim, é possível definir uma *etiqueta/tag* para uma imagem existente através do comando *docker tag*:

```
# docker tag centos:7 default-so:latest
```

Um outro ponto importante é o de que, a medida que as imagens e novas versões destas são baixadas, é necessário monitorar o espaço em disco utilizado pelas imagens, principalmente aquelas que não são mais utilizadas como base para nenhum contêiner ("stale"); para esse caso, atualmente há duas boas opções (USE COM CUIDADO):

1. Spotify-GC: Spotify GC é um contêiner que quando inicializado verifica todos os contêineres parados e imagens não utilizados a faz a remoção destes;
2. Docker Prune: através do comando *docker system prune* é possível recuperar espaço utilizados por imagens, contêineres e volumes do Docker.

Nota: Note que para as soluções que fazem a remoção de imagens *stale*, caso um contêiner esteja parado este será removido, mesmo que o motivo da parada tenha sido algum problema de software! Por conta desse fato, recomenda-se utilizar com bastante cautela qualquer uma das soluções apresentadas.

1.13 Principais instruções para criação de imagens

Pode-se dizer que em 95% dos casos as seguintes diretivas são utilizadas em um Dockerfile:

- ADD: Adiciona um arquivo ou diretório do sistemas de arquivo local para a imagem;
- COPY: Copia arquivos remotos e/ou locais para a imagem.
- CMD: Comando padrão a ser executado pela imagem;
- ENTRYPOINT: Permite configurar o contêiner ou apenas definir o comando a ser executado (Sobrepõe o CMD);
- ENV: Define variáveis de ambiente;
- EXPOSE: Informa ao docker que uma porta da rede do contêiner está disponível;
- FROM: Inicia a imagem a partir de outra imagem: Ex "FROM debian:8";
- RUN: Roda um comando no sistema operacional da imagem;
- ARG: Define variáveis de ambiente, mas permite que no momento da construção da imagem seja passado o valor para a variável especificada. Útil para quando se deseja permitir que o usuário construa imagens para mais de uma versão do mesmo software usando o mesmo DockerFile.

Para ilustrar o uso de cada uma dessa diretivas, visualizaremos o arquivo a seguir:

```
FROM gcavalcante8808/ansible

USER root
ARG SEMAPHORE_VERSION=2.0.4
ADD https://github.com/ansible-semaphore/semaphore/releases/download/v${SEMAPHORE_
↪VERSION}/semaphore_linux_amd64 /usr/bin/semaphore
RUN chmod +x /usr/bin/semaphore && \
    mkdir /tmp/semaphore && \
    chown webserver:webserver /tmp/semaphore

COPY ./semaphore.sh /docker-entrypoint-initdb.d/
USER webserver

ENV GIN_MODE release
WORKDIR /home/webserver
```

(continues on next page)

(continuação da página anterior)

```
EXPOSE 3000
ENTRYPOINT ["/docker-entrypoint-initdb.d/semaphore.sh"]
```

No exemplo acima, temos as seguintes definições:

1. A imagem é baseada na imagem "gcavalcante8808/ansible";
2. Muda-se o contexto para o usuário root (nesse caso, equivale a 'su');
3. Um arquivo da internet é baixado e inserido no contêiner pelo Docker sem a necessidade de instalação de pacotes de clientes Web como curl ou wget;
4. Alguns comandos são lançados no bash;
5. O arquivo "semaphore.sh" é copiado da pasta atual para dentro do contêiner;
6. Muda-se o contexto para o usuário webserver (sem privilégios administrativos);
7. Define-se uma variável de ambiente chamada "GIN_MODE" com o valor "release" (aplicações em Go que utilizam o framework GIN verificam essa variável para saber o modo de operação a ser inicializado);
8. O diretório de trabalho/inicial é definido para "/home/webserver";
9. A diretiva EXPOSE informa ao Docker quais são as portas em que o contêiner comumente escuta;
10. O *EntryPoint* para este contêiner será o script anteriormente copiado.

Nota: A instrução "EXPOSE" não faz a publicação da porta, sendo hoje uma boa prática para facilitar a visualização de quais portas do contêiner deveriam estar abertas no momento de sua execução. Instruções sobre cada uma das diretivas de construção podem ser vistas em: https://docs.docker.com/engine/reference/builder

---

1.14 Diferenças entre RUN, ENTRYPOINT e CMD

Muitas vezes, durante a confecção de um Dockerfile, a utilização das diretivas *RUN*, *ENTRYPOINT* e *CMD* parecem de certa forma nebulosas, pois estes parecem funcionar da mesma maneira. Para tentar esclarecer essa dúvida, pensemos da seguinte forma:

- RUN: Utilizado para rodar um comando dentro do contêiner, não é utilizado para definição do comando principal a ser executado pelo contêiner;
- CMD: Pode ser utilizado como comando principal a ser executado pelo contêiner;
- ENTRYPOINT: Pode e deve ser utilizado como comando principal a ser executado pelo contêiner.

A grande diferença aqui, na realidade, é que:

Quando existe um ENTRYPOINT e um CMD no mesmo DockerFile, os valores definidos na diretiva CMD se tornam **parâmetros** para o ENTRYPOINT, conforme o exemplo abaixo:

```
` FROM centos:7 ENTRYPOINT ["ls"] CMD ["-lha"] `
```

No caso acima, o entrypoint seria o comando *ls* e o CMD passaria o *-lha* como parâmetro para o entrypoint. Os casos mais comuns para utilização desse tipo é quando um determinado binário possui mais de uma função ou subcomando. Alguns exemplos:

1.15 Dicas para criação de imagens

Algumas dicas para a criação de imagens podem fazer com que tanto o tamanho final, quanto a complexidade destas diminuam potencialmente. Algumas das dicas são:

1.15.1 Sempre criar um diretório a parte para então realizar a edição do Dockerfile dentro do diretório

Durante o tempo de construção da imagem, todos os arquivos presentes no diretório são copiados para o **contexto** da imagem, que contribui para o tamanho final da mesma e permite que sejam utilizados através das diretivas *ADD* e *COPY*

1.15.2 Sempre utilizar um arquivo .dockerignore

Para evitar que pastas ou arquivos indesejados sejam copiados para dentro do contexto, crie um arquivo .dockerignore (no mesmo diretório do Dockerfile) contendo as pastas e arquivos (1 por linha) que serão ignorados no momento da construção da imagem.

1.15.3 Procure reduzir o número de camadas aninhando vários comandos RUN

Um erro comum dentre aqueles que estão iniciando no docker é utilizar uma diretiva *RUN* para cada comando que desejam que seja executado; ao invés disso é desejável que sempre que possível, vários comandos estejam aninhados em uma só diretiva *RUN*:

RUN apt-get update && apt-get install wget -y

1.15.4 Apt-get e Yum se mal utilizados aumentam e muito o tamanho das imagens

Arquivos de listas e caches desses gerenciadores de pacotes comumente aumentam muito o tamanho de uma imagem. Após instalar os pacotes necessário procure remover todos os arquivos criados por eles que não serão mais necessários.

Para o APT, utiliza-se a seguinte sintaxe:

**RUN apt-get update && ** apt-get install wget -y && \ rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* && \
apt-get clean

Para o Yum, utiliza-se a seguinte sintaxe:

**RUN yum install httpd -y && ** yum clean all

1.15.5 Jamais utilize Apt-get Upgrade ou Yum Upgrade

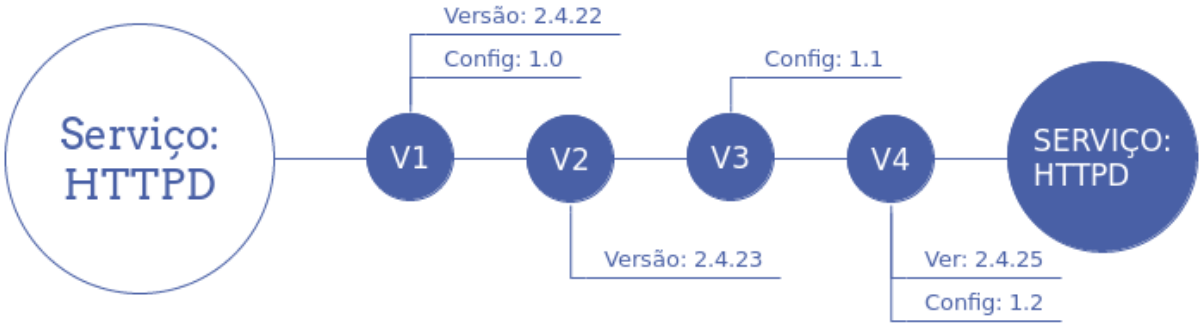
As imagens base do docker são regularmente atualizadas para que incluam as últimas versões dos pacotes (as vezes saem mais de uma imagem por semana). Jamais faça um 'upgrade' usando as ferramentas de pacotes.

1.15.6 Jamais instale o SystemD padrão nas imagens base

As imagens base contém a versão cloud do systemd, muito menor que o systemd original, sendo que estas são mutuamente exclusivas.

1.16 Ciclo de vida de Contêineres

Diferentemente das soluções tradicionais de TI, o ciclo de vida de um contêiner acaba no momento em que é necessário realizar a atualização do mesmo (versão da aplicação, configuração, etc): o contêiner antigo deve ser removido e um novo criado baseado na imagem atualizada. A image abaixo pode auxiliar na elucidação do processo:



Esse é mais dos motivos pelo qual se prega que os contêineres devem refletir a arquitetura de microsserviços: a atualização de cada componente de uma pilha de aplicação torna-se muito mais simples, considerando que cada componente pode ser atualizado em separado; também promulga as boas práticas de evitar **configuration drift** entre os vários contêineres, organizando e padronizando o ambiente de TI.

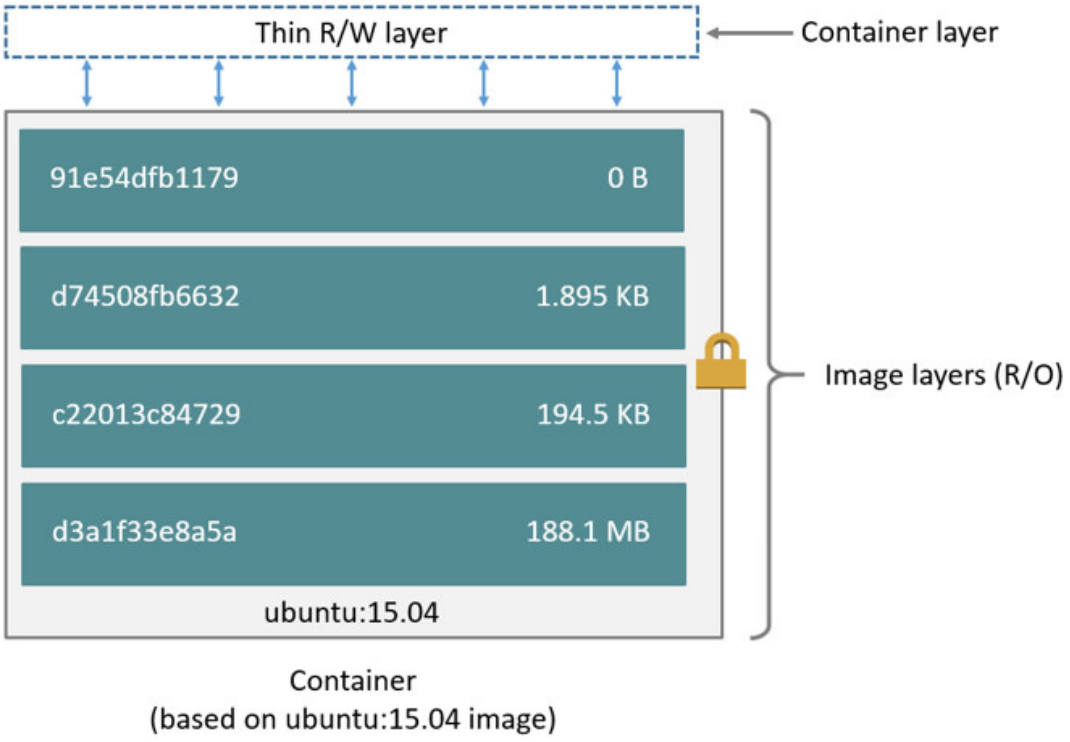
CAPÍTULO 2

Aula 2: Docker: Persistência de Dados e Configurações

2.1 Sistema de Arquivos dos Contêiners

Os sistemas de arquivos utilizados pelos contêiners são, comumente, reflexos do sistema de arquivos do host organizado por uma tecnologia de **UnionFS** e que suporta o conceito de **COW - Copy On Write**, organizando o sistema de arquivos em várias camadas com diferentes versões de arquivos e se apresentando de forma consolidada ao contêiner.

No caso das imagens, cada diretiva utilizada no momento de sua construção resulta em uma camada adicional no sistema de arquivos; isso permite que diferentes imagensreutilizem os dados de diretiva em comum além de**compartilhar as camadas existentes e evitar o uso adicional/intensivo de disco**, conforme ilustrado na imagem abaixo:



Assim, as camadas relativas à imagem permanecem inalteradas ao passo que uma camada relativa a um contêiner continua a ser alterável; no entanto, por conta da forma de funcionamento do **UnionFS**, o conjunto excessivo de operações de escrita na camada relativa a um contêiner devem ser evitadas, pois incorrem em diferentes níveis de perda de *throughput*.

Aviso: Todos os arquivos editados e/ou salvos no sistema de arquivos de um contêiner são perdidos quando da remoção do mesmo.

Após a criação de um contêiner, é possível visualizar quais dados que residem no sistema de arquivos do mesmo foram mudados, adicionados ou removidos, através do comando *docker diff* <*CONTAINER*>:

```
arthas@Host-001:~$ docker diff es_kibana_1
C /usr/share/kibana
C /usr/share/kibana/data
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/ChangeLog
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/LICENSE.BSD
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/README.md
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/bin
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/bin/phantomjs
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/arguments.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/child_process-examples.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/colorwheel.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/countdown.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/detectsniff.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/echoToFile.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/features.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/fibo.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/hello.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/injectme.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/loadspeed.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/loadurlwithoutcss.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/modernizr.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/module.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/netlog.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/netsniff.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/openurlwithproxy.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/outputEncoding.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/page_events.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/pagecallback.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/phantomwebintro.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/post.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/postjson.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/postserver.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/printenv.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/printheaderfooter.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/printmargins.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/rasterize.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/render_multi_url.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/responsive-screenshot.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/run-jasmine.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/run-jasmine2.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/run-qunit.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/scandir.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/server.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/serverkeepalive.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/simpleserver.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/sleepsort.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/stdin-stdout-stderr.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/universe.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/unrandomize.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/useragent.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/version.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/waitfor.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/examples/walk_through_frames.js
A /usr/share/kibana/data/phantomjs-2.1.1-linux-x86_64/third-party.txt
A /usr/share/kibana/data/uuid
C /usr/share/kibana/optimize
C /usr/share/kibana/optimize/.babelcache.json
arthas@Host-001:~$
```

No exemplo acima, há adições “A” e mudanças “C”; para remoções o *status* seria indicado pela letra “D” no início da linha.

2.2 Volumes e Tipos de Montagem

Ao contrário do sistema de arquivos do contêiner, que são removidos quando da exclusão do mesmo, os **volumes** são áreas de dados **persistentes**, normalmente diretórios do sistema de arquivos do *host* ou de um *storage* disponibilizados para um contêiner, de forma análoga a montagem de volumes que ocorre nos sistemas operacionais UNIX. Ao contrário dos sistemas de arquivos dos contêineres, volumes **não sofrem** *overheads* de escrita e também não são perdidos (a menos que se utilize o parâmetro -v na remoção do contêiner) com a exclusão/criação de contêineres, pois o acesso ao sistema de arquivos do volume é direto.

Existem 3 tipos de montagem de volumes que podem ser utilizados:

- *Bind*: Utiliza-se uma pasta presente no host que será montada em um alvo específico dentro do contêiner. Simplista, esse modo de montagem é muitas vezes utilizado para facilitar o acesso aos arquivos do volume;
- *Volume*: Comumente referido como **Named Volume**, utiliza-se a infraestrutura de volume do docker, que pode tanto utilizar drivers locais para servir o espaço para o contêiner, quanto utilizar plugins que permitem fazer a integração com outros serviços como o Amazon S3, Minio, CEPH, etc;
- *TmpFS*: Análogo ao tmpfs de sistemas posix, permite reservar uma área de memória para montagem em um contêiner, sendo útil para dados que possuem um alto nível de mutabilidade mas que não precisam persistir em caso de remoção do contêiner.

Nota: Existe ainda um sub-tipo, que seria um volume anônimo, criado como volume e um *guid* quando da inicialização de um contêiner que contenha a diretiva *VOLUME* mas que durante a sua criação não tenha sido incluído a montagem de um volume.

Para realizar a montagem de um volume em um contêiner, pode-se utilizar o parâmetro "–mount type=bind,source=/diretorio,target=/montagem" durante a execução do comando *docker run*, como no exemplo abaixo:

```
# docker run -d --mount type=bind,source=/data,target=/tmp/data httpd
```

No exemplo acima a pasta "/diretorio" será *montada* dentro do endereço "/pontodemontagem" do contêiner.

Nota: Caso a pasta a ser montada no contêiner não exista, a mesma será criada no sistema de arquivos.

Adicionalmente também é possível realizar a montagem em modo somente-leitura adicionando a diretiva ":ro" ao final da declaração:

```
$ docker run -d --mount type=bind,source=/data:/tmp/data:ro httpd:alpine
```

Nota: Um comportamento muito importante por parte dos volumes é o de que, se o volume estiver vazio, mas a pasta originalmente do contêiner possuir arquivos, estes serão copiados para o volume. **Para os demais tipos de montagem, esse comportamento não ocorre.**

2.2.1 Named Volumes

O *named Volume* é um volume inicializado durante a criação do contêiner e gerenciado pelo próprio Docker, sendo utilizado nas seguintes situações:

1. Quando da utilização de plugins do docker para suporte a volumes (NetApp, Convoy, etc);

2. Padronização dos volumes no ambiente.

A criação de um *named volume* normalmente ocorre através do seguinte comando:

```
$ docker volume create --driver local --name volume1
```

Após a criação de um volume, a lista com todos os volumes pode ser visualizada através do seguinte comando:

```
$ docker volume ls
```

A utilização do volume por um contêiner possui sintaxe parecida com a montagem de volumes do sistema de arquivos, conforme pode ser visualizado abaixo:

```
$ docker run -d --name postgres-default --mount type=volume,source=volume1,target=/
↪var/lib/postgresql/data postgres:alpine
```

A remoção de um *named volume* pode ser realizada através do seguinte comando:

```
$ docker volume rm volume1
```

Nota: A remoção de um volume só poderá se dar quando da não utilização do mesmo por um contêiner.

Por fim, é possível visualizar a utilização de espaço em disco para *named volumes* através do comando *docker system df -v*, que irá mostrar os espaço utilizados por volumes, imagens, contêineres e, para os recursos não utilizados, a percentual de espaço que pode ser recuperado.

Docker Compose

O *Docker Compose* é uma ferramenta para definição e inicialização de arquiteturas multi-container do docker. Com o compose, o administrador/desenvolvedor descreve a arquitetura e configuração de cada um dos contêineres e a relação entre eles em um arquivo do tipo "YAML" chamado **docker-compose.yml**.

O conjunto de definições disponível é o mesmo que seria relativo ao conjunto de opts/parâmetros na criação de um contêiner a partir do comando *docker run*; em verdade, a medida em que o docker evolui e passa a disponibilizar novas opções, o docker-compose também evolui afim de permitir que novos parâmetros possam ser usados para realizar a definição de novos conjuntos de microsserviços e suas configurações.

Nota: Uma visão geral das versões suportadas pelo docker compose em relação ao docker pode ser vista em https://docs.docker.com/compose/compose-file/compose-versioning/#compatibility-matrix

Antes de iniciar a utilização do docker-compose, é necessário realizar a instalação através dos seguintes comandos:

```
# sudo curl -L https://github.com/docker/compose/releases/download/1.20.1/docker-
↪compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
# sudo chmod +x /usr/local/bin/docker-compose
```

Nota: O Docker-compose é uma aplicação feita em Python, que também pode ser instalada via pip através do comando *pip install docker-compose*.

A seguir, realizamos a criação de uma aplicação simples com o framework *Flask*, em um arquivo chamado "app.py" contendo as seguintes diretivas:

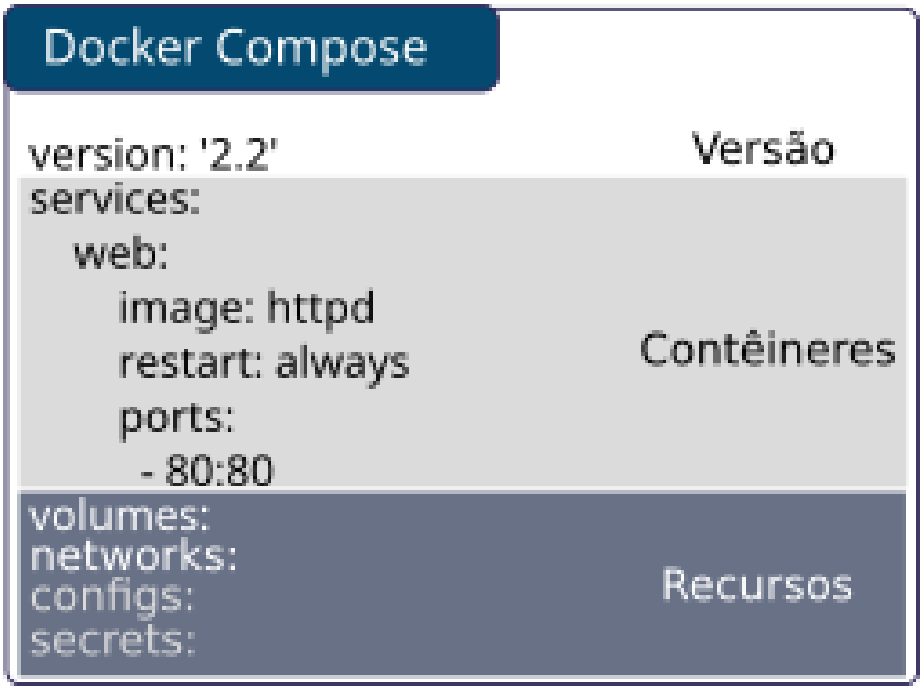
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Na continuação, criamos o Dockerfile relacionado a essa nova aplicação:

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ENV FLASK_APP=app.py
RUN pip install Flask
ADD . /code/
WORKDIR /code
CMD flask run --host=0.0.0.0
```

Após a instalação do mesmo, é necessário criar o arquivo **docker-compose.yml** que conterá todas as instruções necessárias para levantamento dos contêineres.

A estrutura de um arquivo docker-compose.yml, pode ser descrita da seguinte maneira:



Perceba que o arquivo se inicia com a definição da versão do docker-compose; neste caso 2.2, pois pode incluir apenas os recursos necessários para funcionamento dos serviços/contêineres em modo *standalone*, ou seja, não se aplica aos recursos específicos de cluster do docker.

Um exemplo de declaração válido está disponível abaixo:

```
version: '2'
services:
  db:
    image: postgres
```

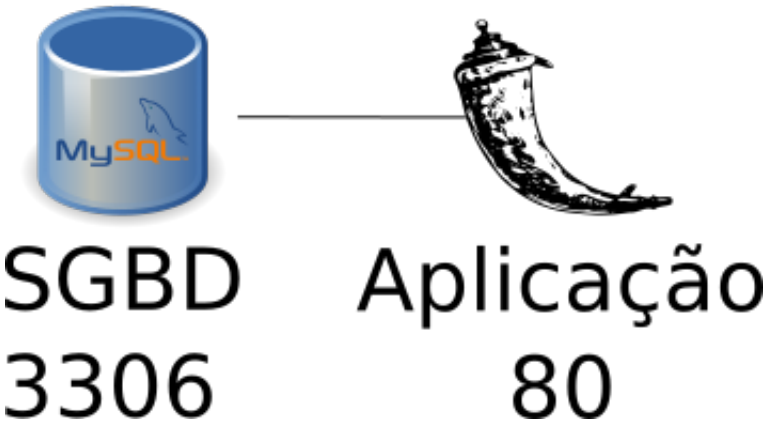
(continues on next page)

(continuação da página anterior)

```
  web:
    build: .
    volumes:
      - .:/code
    command: python manage.py runserver
    ports:
      - "8000:5000"
    depends_on:
      - db
```

Aviso: Arquivos dos tipo YAML se baseam na **indentação** dos itens para leitura das definições de arquivos. Assim sendo, recomenda-se a utilização de espaços para indentação ao invés de tabulações.

O arquivo anteriormente disponibilizado representa de forma simples, a seguinte relação:



Após a criação do arquivo docker-compose.yml, basta iniciar o conjunto de contêineres com o seguinte comando:

```
$ docker-compose up -d
```

Perceba que o docker-compose utiliza automaticamente o nome da pasta "recipiente" para a geração dos nomes dos contêineres; assim, o contêiner relativo ao serviço "db" passaria a ser "django_db_1" caso o nome da pasta fosse "django" e o contêiner da aplicação se chamaria "django_web_1". O número após o nome do contêiner permite ao compose controlar a quantidade de contêineres a serem criados para o "serviço" (o que dá o suporte a escalonamento).

Com os contêineres foram criados através do compose, é também possível gerenciá-los usando o docker-compose, conforme exemplos abaixo:

```
$ docker-compose stop
$ docker-compose start
$ docker-compose restart
$ docker-compose stop && docker-compose rm
$ docker-compose logs
```

Perceba que cada comando é executado sobre todos os contêineres presentes no docker-compose.yml; por fim, para remover o conjunto de contêineres pode-se utilizar o comando *docker-compose down*.

Para realizar o "escalonamento" dos serviços, pode-se utilizar o comando "docker-compose scale" seguido do nome do serviço e número de "réplicas":

```
$ docker-compose scale web=3
```

Assim, após a execução do comando supracitado, seriam criados os contêineres "django_web_2" e "django_web_3".

Nota: Diferentemente dos recursos de cluster, o docker-compose scale apenas cria novos contêineres para os serviços definidos dentro do arquivo docker-compose.yml, sendo o escopo do docker-compose local, ou seja, a criação de contêineres sempre ocorre apenas no host atual.

2.3 Definição de Volumes e uso de recursos já existentes

Além da definição de contêineres de forma propriamente dita, outros recursos como redes e volumes também podem ser definidos através de um arquivo do docker-compose. O exemplo abaixo contém exemplos de definição de redes e volumes:

```
version: '2'
volumes:
    data:

services:
    db:
      image: gcavalcante8808/zabbix-db-postgres
      restart: always
      networks:
       - databases
      environment:
        POSTGRES_DB: zabbix
        POSTGRES_USER: zabbix
        POSTGRES_PASSWORD: "zabbix"
      volumes:
        - data:/var/lib/postgresql/data

networks:
    databases:
        driver: bridge
```

No exemplo acima, temos um *named volume* chamado "data" e uma rede chamada "databases" que, considerando a pasta recipiente "zabbix", resultará na criação da rede "zabbix_databases" e do volume "zabbix_data" respectivamente.

Para o caso em que se deseja utilizar recursos previamente existentes, desde contêineres a redes pode-se utilizar a diretiva "external: true" conforme exemplo abaixo:

```
volumes:
  myvol:
    external: true
```

Considera-se como "external" quaisquer recursos que não serão gerenciados a partir do arquivo docker-compose.yml atual; podem ser recursos de qualquer espécie como volumes criados manualmente, redes, dentre outros.

A utilização de recursos de ambos os tipos (gerenciados e não gerenciados pelo compose) pode ser descrita da seguinte maneira:

```
volumes:
  db:
```

(continues on next page)

(continuação da página anterior)

```
    external: false
  backup:
    external: true
```

2.4 Docker Compose: Composição de configurações

Por fim, o docker-compose suporte a composição de configurações ou *overrides*, de modo que um arquivo docker-compose.yml pode servir de base para a configuração de um serviço e demais arquivos podem conter as especificidades a serem implementadas; em verdade, esse tipo é configuração é comum para casos em que se possui múltiplos ambientes, tais como desenvolvimento, homologação e produção.

Os requisitos para a composição de configurações através dos arquivos do docker-compose são:

- O arquivo de override ou definido pelo usuário precisa iniciar com o mesmo *version* do arquivo docker-compose.yml;
- Os serviços a serem configurados precisam ser aqueles já definidos no arquivo docker-compose.yml.

Por padrão, todas as configurações presentes em um arquivo "docker-compose.override.yml" são automaticamente lidos e aplicados pelo docker-compose no momento de sua execução; é possível ainda definir arquivos com nomes específicos que poderão ser analisados e aplicados pelo docker-compose.

O exemplo abaixo denota dois arquivos: o docker-compose.yml padrão e o docker-compose.override.yml:

```
version: '2'
services:
  db:
    image: postgres
  web:
    build: .
    volumes:
      - .:/code
    command: python manage.py runserver
    ports:
      - "8000:5000"
    depends_on:
      - db
```

```
version: '2'

volumes:
    pgdata:
        driver: netapp

services:
    db:
        image: postgres
        volumes:
         - pgdata:/var/lib/postgresql/data
```

No exemplo acima, no momento da execução do comando "docker-compose up -d", seria criado um volume chamado "django_pgdata" que seria utilizado como volume para guardar os dados do serviço "db", exemplificando uma possível guarda dos dados do banco de dados em um volume do storage netapp (possivelmente essa seria uma configuração de produção), além da aplicação das demais definições do arquivo docker-compose original.

A utilização de arquivos com nomes definidos pelo usuário (usualmente congruente aos ambientes de execução dos serviços) é plenamente suportada, bastando para tanto especificar a *flag* -f no momento de execução do docker-compose:

```
# docker-compose up -f docker-compose.devel.yml -d
# docker-compose up -f docker-compose.homolog.yml -d
# docker-compose up -f docker-compose.prod.yml -d
```

Sendo que no caso acima, cada arquivo conteria as configurações adequadas ao ambiente no qual estaria sendo executado.

Nota: O docker-compose sempre cria uma nova rede "default" e adiciona aos contêineres definidos no arquivo docker-compose.yml.

Nota: Mais informações acerca das diretivas do compose disponíveis em: https://docs.docker.com/compose/

CAPÍTULO 3

Aula 3: Repositório de Imagens, Conectividade Interna e Logging

3.1 Docker Registry

O Docker Registry provê um serviço para hospedagem de imagens do Docker análogo ao que está disponível no hub.docker.com, porém com a possibilidade de uso e hospedagem em uma rede interna.

Para criar um novo registry, crie uma nova pasta chamada 'registry' e então insira o seguinte conteúdo no arquivo docker-compose.yml:

```
version: '2'

volumes:
  data:

services:
  registry:
    image: registry:2
    restart: unless-stopped
    ports:
     - 5000:5000
    volumes:
     - data:/var/lib/registry
```

Crie o contêiner através do comando `docker-compose up -d`. O registry criado até esse momento utiliza a porta 5000 para comunicação, mas ainda não trabalha via TLS/HTTPS; Por padrão, o Docker não permite a comunicação sem TLS/HTTPS com um registry, a não ser que (por padrão) a url seja '127.0.0.1'.

Para verificar quais *insecure registries* são aceitos pelo daemon do docker, utilize o comando "docker info"; a informação desejada estará ao final do comando, abaixo da linha que se inicia com "Insecure Registries:".

Para registries externos, mesmo que estes utilizem um certificado auto assinado ou mesmo não suportem HTTPS, é possível configurar o Docker para aceitar *registries* adicionais. Para tanto, crie ou edite o arquivo "/etc/docker/daemon.json" inserindo as seguintes diretivas:

```
{
 "insecure-registries": ["url:5000"]
 }
```

E por fim reinicie o docker para aplicar as configurações:

```
$ sudo systemctl restart docker
```

Uma vez que o docker está preparado, realizar o envio de uma imagem requer que **você defina tags nas imagens atuais que contenham o nome do repositório no formato registry:porta/imagem:tag** e faça o *push*, conforme o exemplo abaixo:

```
$ docker tag ubuntu 127.0.0.1:5000/ubuntu:yak
$ docker push 127.0.0.1:5000/ubuntu:yak
```

Realizar o download de imagens a partir do registry é igualmente fácil:

```
$ docker pull 127.0.0.1:5000/ubuntu:yak
```

No entanto, até então, toda a comunicação com o registry vem sendo realizada através de HTTP, ou seja, sem a criptografia. Para ativar o suporte a TLS/HTTPS no acesso a aplicação, será necessário criar uma certificado, atualizar o docker-compose.yml para que fique da seguinte maneira:

```
version: '2'

volumes:
  data:
  registry_certs:
    external: true

services:
  registry:
    image: registry:2
    restart: unless-stopped
    ports:
     - 5000:5000
    volumes:
     - data:/var/lib/registry
     - registry_certs:/certs
    envinronment:
     - REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt
     - REGISTRY_HTTP_TLS_KEY=/certs/domain.key
```

Antes de inicializar a nova versão do contêiner, será necessário criar um volume para receber os certificados e criá-los através dos seguintes comandos:

```
$ docker volume create --driver local registry_certs
$ docker run --rm -e COMMON_NAME=localhost -e KEY_NAME=domain --mount type=volume,
↪source=registry_certs,target=/certs centurylink/openssl
```

Com os certificados já criados no volume, resta apenas inicializar a nova versão do contêiner através do comando:

```
$ docker-compose up -d
```

Nota: Para o ambiente de produção, solicite os certificados junto a área de infraestrutura.

Por fim, é desejável fazer a restrição de acesso ao registry através da utilização de credenciais no estilo htpasswd/*basic auth*. Para tanto, atualize o docker-compose.yml relativo ao registry para que fique com a seguinte conteúdo:

```
version: '2'

volumes:
  data:
  registry_certs:
    external: true
  registry_auth:
    external: true

services:
  registry:
    image: registry:2
    restart: unless-stopped
    ports:
     - 5000:5000
    volumes:
     - data:/var/lib/registry
     - registry_certs:/certs
     - registry_auth:/auth
    envinronment:
     - REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt
     - REGISTRY_HTTP_TLS_KEY=/certs/domain.key
     - "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm"
     - REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd
```

Antes de inicializar a nova versão do contêiner, será necessário criar um volume para receber o arquivo com as credenciais. Utilize os seguintes comandos para iniciá-los:

```
$ docker volume create --driver local registry_auth
$ docker run --entrypoint htpasswd --mount type=volume,source=auth,target=/auth␣
↪registry:2 -Bbn testuser testpassword > /auth/htpasswd
```

Com o arquivo htpasswd já criado no volume auth, resta apenas inicializar a nova versão do contêiner através do comando:

```
$ docker-compose up -d
```

Por conta da adição do suporte a credenciais, será necessário agora realizar o "login" para operar com o novo registry, que pode ser feito da seguinte maneira:

```
$ docker login 127.0.0.1:5000
```

Nota: As informações de login são guardadas como *base64* no arquivo ~/.docker/config.json.

Nota: Mais informações acerca do registry, incluindo melhores práticas para seu uso em produção, podem ser encontradas em: https://docs.docker.com/registry/configuration/.

3.2 Redes definidas por Software

Historicamente, o Docker possui três redes previamente configuradas que podem ser utilizadas:

- "docker0": Rede bridge padrão configurada para utilizar a subrede 172.17.0.0/16;
- "none": Para casos em que se deseja que um contêiner não possua suporte a Rede (os contêineres ainda terão o suporte a interface de loopback);
- "host": Espelha as mesmas conexões presentes no host para o contêiner.

Durante a criação de um contêiner este é automaticamente atrelado a interface "docker0" caso seja criado através do comando `docker run` sem configurações adicionais; para esse caso, uma regra de NAT é criada no firewall do host e o contêiner recebe um IP randômico dentro da faixa 172.17.0.0/16 e quaisquer portas expostas são acessíveis na forma IP:PORTA.

Nota: Os Endereços IP recebidos por um contêiner não possuem nenhuma garantia de continuidade; em verdade, os IP's são atribuídos na ordem em que os contêineres são iniciados, começando por 172.17.0.2, sendo que o endereço 172.17.0.1 é o gateway de acesso para a interface 'docker0'.

Para descobrir o Ip de um contêiner pode-se utilizar as seguintes formas:

```
$ docker inspect --format=" {{ .NetworkSettings }} " <CONTAINER>
$ docker exec -it <CONTAINER> ip a
```

Na primeira forma, utiliza-se o parâmetro inspect para retornar os metadados do contêiner enquanto que no segundo caso envia-se um comando em modo interativo para o contêiner "ip a", que irá retornar o endereço do contêiner, se esse tiver o pacote iputils instalado.

Para o caso em que dois ou mais contêineres estão conectados a rede 'docker0' (e também considerando a volatilidade da recepção dos endereços IP dos contêineres) é necessário fazer o uso de links entre os contêineres de forma que estes passem a referenciar um nome específico (mas que não precisa ser um FQDN); dessa forma, cada contêiner passa a 'conhecer' o endereçamento do outro contêiner, informação essa que pode ser usada em uma aplicação na forma "CONTAINER:NOME". Exemplo:

```
$ docker run -d --name postgres-principal postgres
$ docker run -d --name app --link postgres-principal:db httpd
```

No caso acima, o contêiner "app" reconhece o nome "db" e consegue resolver esse nome para o IP do contêiner "postgres-principal", mesmo que a ordem de inicialização e subsequentemente os IP's mudem.

Aviso: Fazer um link entre dois contêineres não impede que o primeiro contêiner seja parado ou reiniciado; em verdade, caso o primeiro contêiner seja reiniciado o segundo também precisará ser para que o endereço do primeiro seja 'atualizado' para o segundo.

Para os casos em que um determinado contêiner precisa ser acessível a outros computadores em uma rede pode-se utilizar o espelhamento de portas entre o contêiner e o host. Exemplo:

```
$ docker run -d --name postgres-default -p 5432:5432 postgres:alpine
```

No caso acima, a porta 5432 do host atual será vinculada na porta 5432 do contêiner via firewall(iptables); pode-se ainda realizar o vínculo de forma dinâmica, através do parâmetro -P (maiúsculo):

```
$ docker run -d --name postgres-default -P postgres:alpine
```

Os mapeamentos entre portas podem ser visualizados tanto através do comando `docker ports` quanto através do comando de listagem de contêineres ativos:

```
$ docker ps -a
$ docker ports <CONTAINER>
```

Nota: No caso do espelhamento dinâmico de portas, as portas começam a ser alocadas a partir da 32768 e seguem conforme a ordem de inicialização dos contêineres.

3.2.1 Definição de redes pelo usuário

Ao contrário da rede legada 'docker0', as redes criadas por um usuário possuem um número maior de recursos disponíveis; os principais são o suporte a resolução de nomes e a possibilidade de definir uma rede com range específico que poderá ser adicionada na criação e durante o funcionamento dos contêineres. Para tanto, utilize o seguinte comando:

```
$ docker network create --driver bridge --subnet 172.100.0.0/16 user_network
```

Após a criação da rede, é possível visualizar as informações gerais de quais redes estão definidas através do seguinte comando:

```
$ docker network ls
```

E informações específicas sobre a rede criada através do seguinte comando:

```
$ docker network inspect user_network
```

A partir desse ponto, a criação de contêineres passa a receber o parâmetro "–network" conforme o exemplo abaixo:

```
$ docker run -d --name db --network=user_network postgres
$ docker run -d --name app --network=user_network myapp
```

Para testar a resolução de nomes utilize o seguinte comando:

```
$ docker exec -it app ping db
```

Para adicionar a rede a um contêiner em funcionamento, utilize o seguinte comando:

```
$ docker network connect <NETWORK> <CONTAINER>
```

Analogamente é possível desconectar uma interface de um contêiner em funcionamento:

```
$ docker network disconnect <NETWORK> <CONTAINER>

Por fim, para remover uma rede utilize o seguinte comando:
```

```
$ docker network rm <NETWORK>
```

Nota: Antes de se realizar a remoção de uma rede é necessário desconectar a interface dos contêineres conectados a mesma.

Para o caso da utilização do docker-compose para gerenciamento dos contêineres, quando da inicialização dos contêineres, este cria uma rede automaticamente, normalmente com o padrão <PASTA>_default; da mesma forma, se

nenhum contêiner estiver conectado a esta rede, remover os contêineres via `docker-compose down` fará com que essa rede também seja removida.

3.3 Logging Drivers

A partir do momento em que uma aplicação é encapsulada em forma de um contêiner espera-se que seus logs estejam disponíveis na saída padrão (/dev/stdout), pois o próprio docker inclui os recursos necessários para a guarda e leitura dos logs através de **drivers/plugins**; assim, uma série de **backends** são suportados, sendo os principais:

- Json-File: padrão, envia todos os logs para um arquivo Json no sistema de arquivos do host;
- Syslog: envia todas as mensagens para um servidor SysLog;
- GELF: formato de dados compatível com o GrayLog 2;
- FluentD: formato de dados compatível com o FluentD.

A configuração de *log forwarding* pode ser definida em dois níveis: contêiner e do próprio Docker (o que inclui todos os contêineres que foram criados como padrão), mas mantendo a capilaridade ao ponto de que cada contêiner pode ter seu próprio método de logging.

Para o Docker, esta configuração é feita no arquivo "/etc/docker/daemon.json" ao passo que para um contêiner ela pode ser feita no "docker run" ou via docker-compose:

```
version: '2'

volumes:
  data:

services:
  registry:
    image: registry:2
    restart: unless-stopped
    ports:
     - 5000:5000
    volumes:
     - data:/var/lib/registry
    logging:
      driver: json-file
      options:
        max-size: "200k"
        max-file: "10"
```

Aviso: A utilização do comando `docker logs` ou `docker-compose logs` somente é possível quando da utilização dos logging drivers "json-file" ou "journald". Para os demais, os logs ficam indisponíveis pois são diretamente enviados às soluções conforme configuração.

3.3.1 Json-File

"Json-File" é o driver de logging padrão do Docker, onde um arquivo json passa a receber toda a saída advinda do contêiner. Inicialmente, para visualizar os logs de um contêiner utiliza-se o seguinte comando:

```
$ docker logs <CONTAINER>
$ docker logs -f <CONTAINER>
```

Em sua configuração padrão, este driver simplesmente recolhe e mantém toda a informação disponível no arquivo de log; para evitar o crescimento desenfreado de logs é recomendável adicionar o parâmetro "–log-opt max-size" à configuração do docker, no arquivo daemon.json:

```
{
 "insecure-registries": ["url:5000"],
 "log-driver": "json-file",
 "log-opts": {"max-size": "10m"},
}
```

Após fazer a mudança da configuração, reinicie o daemon do docker para aplicar as configurações:

```
$ sudo systemctl restart docker
```

Aviso: Arquivos que chegarem ao limite especificado de tamanho do log terão suas informações sobrescritas.

3.3.2 FluentD

O fluentD é um coletor de dados capaz de receber dados de diferentes níveis de infraestrutura e repassá-los a soluções específicas como o Apache Lucene/Elastic Search.

Para iniciar um novo contêiner com o fluentD, crie uma pasta com a seguinte declaração do arquivo docker-compose.yml:

```
version: '2.2'

services:
  fluentd:
    image: "fluent/fluentd"
    volumes:
     - ./stdout.conf:/fluentd/etc/fluent.conf
    ports:
     - 24224:24224
    restart: unless-stopped
```

A seguir, crie o arquivo "stdout.conf" no mesmo diretório em que o arquivo docker-compose.yml se encontra, com o seguinte conteúdo:

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<match docker.web-**>
  @type copy
  <store>
    @type stdout
  </store>
</match>
```

Por fim, inicialize o contêiner do fluentd através do comando `docker-compose up -d` a partir da pasta onde o arquivo docker-compose.yml reside.

Uma vez que o fluentd já está disponível, crie um novo contêiner através do seguinte comando:

```
$ docker run -d --name web-fluentd -p 8080:80 --log-driver=fluentd --log-opt fluentd-
↪address=localhost:24224 --log-opt tag="docker-web.{{.ImageName}}/{{.Name}}/{{.ID}}"␣
↪nginx:alpine
```

Após a criação do contêiner, realize algumas requisições http para o endereço 'http://localhost:8080' para que logs sejam gerados e, por fim, visualize a recepção destes através dos logs do próprio fluentd:

```
$ docker-compose logs -f
```

Nota: Informações acerca do FluentD podem ser obtidas na página do projeto: http://docs.fluentd.org/articles/quickstart, assim como informações acerca das opções de integração do mesmo com o docker: https://docs.docker.com/config/containers/logging/fluentd.

CAPÍTULO 4

Aula4: Gerenciamento de Recursos

4.1 CGROUPS: Gerenciamento de Recursos dos Containeres

É possível definir o uso de recursos do ambiente por contêineres durante a sua criação ou funcionamento.

Para tanto, o próprio recurso de CGroups contém descritores para cada tipo de recurso: CPU, Memória, *Throughput* de Disco e Rede. O número desses descritores atribui um peso/prioridade para cada contêiner, sendo que por padrão, todos os contêineres dividem os recursos disponíveis igualmente.

A definição da limitação de recursos pode acontecer durante a criação do contêiner, com parâmetros em conjunto com o comando "docker run" (também via declaração em um arquivo docker-compose.yml) ou após a criação do contêiner (inclusive durante seu funcionamento) utilizando-se o comando "docker update".

Nota: O gerenciamento/escalonamento da pilha de rede de um contêiner não é suportado pelo docker. Informações completas acerca do docker update disponíveis em: https://docs.docker.com/engine/reference/commandline/update.

4.1.1 CPU

O controle de uso de CPU por contêiner pode ocorrer em 3 níveis:

- Peso Relativo;
- Afinidade;
- Porcentagem de Uso do Recurso.

Para o caso de peso relativo, leva-se em conta que cada contêiner possui até 1024 descritores, o que define o uso mínimo do recurso de CPU. O exemplo abaixo ilustra a criação de um contêiner que possui acesso a pelo menos a 25% da CPU:

```
$ docker run -it --rm --cpu-shares 256 stress --cpu 1
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  stress:
    image: stress
    entrypoint: stress
    command: --cpu 1
    cpu_shares: 256
```

Dessa forma, em um cenário em que haja forte concorrência por uso da CPU, o contêiner acima terá resguardado pelo menos 25%.

No caso da definição da afinidade de um contêiner basta indicar as CPU's alvo através do parâmetro "–cpuset-cpus", conforme abaixo:

```
$ docker run -it --rm --cpuset-cpus=0,1 stress --cpu 2
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  gateway:
    image: stress
    entrypoint: stress
    command: --cpu 2
    cpuset: 0,1
```

Por fim, para definir um limite de uso em termos o uso de ciclos de cpu de um contêiner, deve-se utilizar o parâmetro "–cpuset-quota":

```
$ docker run -it --rm --cpu-quota=50000 stress --cpu 4
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  stress:
    image: stress
    entrypoint: stress
    command: --cpu 4
    cpu_quota: 50000
```

No caso acima, o contêiner estará limitado a utilizar até 50% do total de processamento do sistema em ciclos; como nesse caso não houve a definição de afinidade, o provável comportamento será o aparecimento de 4 processos, com ~13% de uso de cpu cada.

Nota: Demais informações acerca do controle de uso via quota para a CPU disponíveis em: https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt

---

4.1.2 Memória

O gerenciamento de uso do recurso de memória para um contêiner pode se dar em três níveis:

- Quantidade de Memória RAM;
- Quantidade de Uso de Swap;

- **Reserva de Memória**.

Para definir a quantidade de memória RAM que um determinado contêiner pode utilizar, adiciona-se o parâmetro "–memory" seguido da quantidade e unidade:

```
$ docker run -d --memory=1G --name httpd httpd
$ docker update --memory=512M httpd
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  gateway:
    image: httpd
    mem_limit: 512M
```

A **reserva de memória** (que funciona na prática como um **soft limit**) funciona de forma que, quando o ambiente estiver saturado, o docker tentará fazer com que contêiner alvo utilize o valor de memória definido. Vide o exemplo abaixo:

```
$ docker run -d --memory 1G --memory-reservation 100M --name httpd httpd
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  gateway:
    image: httpd
    mem_limit: 1g
    mem_reservation: 100mb
```

É interessante notar que, por padrão, o próprio Docker interromperá o funcionamento de um contêiner caso ele chegue ao topo de uso definido ou utilize toda a memória do sistema. A sintaxe para desabilitar o **oom-killer** para um determinado contêiner é:

```
$ docker run -it --rm -m 200M --oom-kill-disable ubuntu:16:04
```

Aviso: É possível desabilitar esse comportamento para um contêiner, porém isso só é recomendável para o caso em que ele possua um limite de RAM; desabilitar o **OOM-KILLER** para um contêiner que não possui um limite definido poderá fazer com que o Administrador do servidor precise matar os processos do host manualmente para liberar memória.

4.1.3 Uso de Disco - Throughput

O gerenciamento de uso do Disco para um contêiner pode se dar em três níveis:

- Peso Relativo (Ver https://github.com/moby/moby/issues/16173);
- Escrita e Leitura em bps (incluindo múltiplos);
- Escrita e Leitura em Operações por segundo (IOPS).

A gerenciamento de recursos de disco através do docker somente funcionará de facto caso o *Scheduler* de IO do Kernel seja o CFQ. Para descobrir o *scheduler* em uso, utilize o seguinte comando:

```
$ cat /sys/block/sda/queue/scheduler
```

Caso o scheduler não esteja definodo como CQF, utilize o seguinte comando para realizar a mudança:

```
# echo cfq > /sys/block/sda/queue/scheduler
```

Para a definição de uso de recursos através do peso relativo, deve-se levar em conta os valores de 100 (mínimo, maior restrição) a 1000 (máximo, sem restrições). Para visualizar os resultados do teste a seguir será necessário abrir dois terminais; o primeiro conterá um contêiner cujo parâmetro **–blkio-weight** será 100 e o segundo 600. Os comandos a serem inseridos em cada terminal são:

```
$ docker run -it --rm --blkio-weight 600 fedora sh -c 'time dd if=/dev/zero of=test.
↪out bs=1M count=512 oflag=direct'
$ docker run -it --rm --blkio-weight 100 fedora sh -c 'time dd if=/dev/zero of=test.
↪out bs=1M count=512 oflag=direct'
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  fedora1:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      weight: 600

  fedora2:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      weight: 100
```

É possível ainda realizar a configuração do peso relativo para dispositivos presentes no sistema operacional, que sejam indiretamente ou diretamente utilizados pelo contêiner. Exemplo:

```
$ docker run -it --rm --mount type=bind,source=/media/sdb,target=/mnt/rede --blkio-
↪weight-device "/dev/sdb:500" fedora sh -c 'time dd if=/dev/zero of=/mnt/rede/test.
↪out bs=1M count=512 oflag=direct'
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  fedora1:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      weight_device:
       - path: /dev/sdb
         weight: 500
```

Nota: Assim como no caso do peso relativo da CPU, a "limitação" do uso somente ocorrerá caso outro processo ou contêiner não esteja a fazer uso intensivo do I/O.

Aviso: Mudar o **IO Scheduler** do Sistema Operacional pode ter consequências indesejáveis em algumas aplicações ou contêineres. Verifique junto ao fabricante da solução alvo possíveis problemas que podem ser causados quando do uso do CFQ como **IO Scheduler**.

A seguir, quando da utilização do gerenciamento via **Escrita e Leitura em bps**, torna-se possível definir valores tanto para escrita quanto para a leitura. Os parâmetros utilizados para tanto são, respectivamente, "–device-read-bps" e "device-write-bps", que são utilizados em conjunto com o dispositivo ao quais os contêiner possuem acesso. Veja os exemplos abaixo:

```
$ docker run -it --rm --device-write-bps /dev/sda:10mb fedora sh -c 'time dd if=/dev/
↪zero of=test.out bs=1M count=512 oflag=direct'
$ docker run -it --rm --device-read-bps /dev/sda:10mb fedora sh -c 'time dd if=/dev/
↪zero of=test.out bs=1M count=512 oflag=direct'
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  fedora1:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      device_write_bps:
       - path: /dev/sda
         rate: '10mb'
      device_read_bps:
       - path: /dev/sda
         rate: '10mb'
  fedora2:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      device_write_bps:
       - path: /dev/sda
         rate: '5mb'
      device_read_bps:
       - path: /dev/sda
         rate: '5mb'
```

Por fim, é possível também realizar o gerenciamento de recursos baseados em operações por segundo (leitura ou escrita):

```
$ docker run -it --rm --device-write-iops /dev/sda:20 fedora sh -c 'time dd if=/dev/
↪zero of=test.out bs=1M count=512 oflag=direct'
```

Para uso via docker-compose, pode-se dispor o arquivo docker-compose.yml da seguinte maneira:

```
version: '2.2'
services:
  fedora1:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      device_write_iops:
       - path: /dev/sda
         rate: 20
```

(continues on next page)

(continuação da página anterior)

```
  fedora2:
    image: fedora
    command: sh -c 'time dd if=/dev/zero of=test.out bs=1M count=512 oflag=direct'
    blkio_config:
      device_write_iops:
       - path: /dev/sda
         rate: 40
```

4.2 Visualização de Recursos, Monitoramento & HealthChecks

A visualização de uso de Recursos do docker pode ser realizado através do seguinte comando:

```
$ docker stats
$ docker stats --no-stream
$ docker stats <CONTAINER>
```

Atualmente, algumas soluções do mercado já provêem suporte a estatísticas de funcionamento do docker, mas o CAdvisor destaca-se por ser uma aplicação criada pelo Google, simplista, que retorna as estatísticas de uso de recurso dos contêineres e do host. Para iniciar o Cadvisor em um Host com o docker utilize o seguinte comando:

```
docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

Por outro lado, o monitoramento do contêiner pode ocorrer através : para o segundo caso, um recurso muito útil é de **HealthCheck**, onde o próprio contêiner passa a conter uma instrução de checagem, que é automaticamente executada em segundo plano e que, a depender do resultado, irá mudar a chave "{{ Status.Health }}" e até mesmo parar o contêiner.

Um exemplo de instrução de checagem pode ser visto abaixo:

```
HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost/ || exit 1
```

4.3 Segurança

Além do fato dos processos serem isolados através de *CGROUPS*, o docker dispõe ainda do uso de outros mecanismos disponíveis em um kernel Linux tais como:

- Posix Caps: disponível a partir da versão 2.2 do Kernel Linux, Posix Capabilities são divisões unitárias de permissões que um superusuário possui, que sendo atreladas a um binário, evitam a necessidade de uso do root ou bit de execução como outro usuário(setuid e setgid) . Ex: Utiliza-se apenas CAP_SYS_CHROOT para usar um chroot e CAP_SYS_NICE para mudar o prioridade de um processo ao invés de dar acesso como 'root' via setuid;
- Seccomp: *Securing Compute Mode* é um recurso do kernel linux que permite restringir as chamadas do Kernel que podem ser executadas por um processo;

- Apparmor: Módulo de segurança que tem por objetivo proteger o sistema operacional das aplicações. Comumente, o AppArmor possui um *profile* relacionada a cada aplicação a ser rodada com ações permitidas e proibidas. Padrão para o sistema Ubuntu e possui suporte disponível no Debian através da instalação do pacote de mesmo nome;
- Selinux: *Secure Enhanced linux* ou *Selinux* é um módulo de segurança do Kernel que possui o mesmo objetivo do AppArmor: proteger o sistema operacional de ações danosas das aplicações. Padrão nas distribuições RedHat e seus derivados.

O Docker já trabalha com configurações de PCAPS, SecComp e AppArmor/Selinux por padrão em todas as imagens, pois já trás vários desses *profiles* junto com sua instalação padrão.

4.3.1 Posix Capabilities

O docker, por padrão, permite apenas uma pequena fatia de capabilities por padrão, como pode ser visto abaixo:

```
        s.Process.Capabilities = []string{
                "CAP_CHOWN",
                "CAP_DAC_OVERRIDE",
                "CAP_FSETID",
                "CAP_FOWNER",
                "CAP_MKNOD",
                "CAP_NET_RAW",
                "CAP_SETGID",
                "CAP_SETUID",
                "CAP_SETFCAP",
                "CAP_SETPCAP",
                "CAP_NET_BIND_SERVICE",
                "CAP_SYS_CHROOT",
                "CAP_KILL",
                "CAP_AUDIT_WRITE",
        }
```

Capabilities são adicionadas ou removidas de um contêiner no momento de sua criação:

```
$ docker run --cap-drop=NET_RAW --rm fedora bash
```

No exemplo acima, mesmo como root, não é possível utilizar o comando 'ping', pois o contêiner não possui a capability CAP_NET_RAW.

Nota: Uma lista completa de *capabilities* pode ser vista em `man capabilities`.

CAPÍTULO 5

Questões das Aulas

Contents:

5.1 Cap. 1 - Exercício1: Encapsulamento da Aplicação como Microsserviço

Para aplicar os demais conhecimentos nas demais aulas, será necessário criar uma imagem da aplicação 'curso', cujos detalhes são os seguintes:

1. A aplicação é feita em Python;
2. O projeto possui um arquivo *requirements.txt* que contém as bibliotecas python que devem ser instaladas através do PIP. Crie o arquivo 'requirements.txt' inclua o valor 'Flask' e insira a diretiva `RUN pip install -r requirements.txt` no Dockerfile;
3. Objetivando um uso inicial em desenvolvimento, apenas utilizaremos um servidor Werkzeug mais simples para servir as páginas do projeto. Para isso, o flask requer que **variável de ambiente** 'FLASK_APP' esteja definida com o nome do módulo da aplicação, nesse caso `app.py`, que pode ser expresso no Dockerfile através da diretiva `ENV FLASK_APP=app.py`;
4. O comando a ser utilizado para este contêiner é `flask run` que deve ser realizado a partir da pasta da aplicação;
5. Crie o módulo `app.py` com o código abaixo, salve-o e copie-o para o alvo '/usr/src' da a imagem (Diretiva `COPY . /usr/src/`):

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
```

6. Realiza a criação da imagem com o nome **oficina-docker** a etiqueta **aula-1** através do comando *docker build -t oficina-docker:imagem-aula-1 .*;
7. Crie um contêiner com o nome **aula-1** a partir da imagem com um vínculo da porta 5000 do host para a porta 5000 do contêiner para facilitar a visualização através do um browser, através do comando *docker run -d –name aula-1 -p 5000:5000 oficina-docker:imagem-aula-1*.

5.1.1 Informações adicionais

A imagem python disponibilizada pelo docker em suas diversas variantes (2, 3, 2-slim, 3-slim, 2-alpine, 3-alpine) já possui o PIP instalado, que pode ser utilizado para instalar bibliotecas a partir de um arquivo ou diretamente declarando o pacote a ser instalado, conforme os seguintes exemplos:

```
#Instalando a partir de um arquivo
pip install -r requirements.txt
#Instalando uma biblioteca diretamente
pip install Flask
```

Item 1: Se utilizar uma imagem mais simples como base (Debian, Centos, etc) e desejar instalar a última versão do PIP , usualmente a seguinte diretiva é suficiente para instalar o PIP (requer as bibliotecas mínimas do python já instaladas):

```
curl https://bootstrap.pypa.io/get-pip.py | python
```

Com base nos dados acima dispostos, realize o processo de encapsulamento da aplicação no modelo de contêineres.

5.1.2 Como Verificar se todas as ações ocorreram conforme o esperado

Após a criação da imagem, faça a criação de um contêiner a partir da mesma e verifique através de seu browser se é possível acessar a página inicial disponível em: 'http://localhost:5000'.

Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Inicializar o suporte ao versionamento
git init .
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 1"
# Criar uma etiqueta para esta aula.
git tag -a aula1 -m "Aula 1 - <ALUNO>"
# Enviar as mudanças para o repositório remoto.
git remote add https://github.com/<USUARIO>/docker-unleashed
git push -u origin master aula1
```

5.2 Cap. 2 - Exercício1: Gerenciamento e utilização de Volumes

Para aplicar os conhecimentos explanados nesta aula, será necessário atualizar a aplicação criando uma nova imagem e verificando o impacto da (não) utilização de volumes em conjunto com um contêiner do docker. Siga as seguintes instruções:

1. Atualize o código da aplicação, para que inclua agora o suporte a criação de um arquivo de log, conforme abaixo:

```
import logging
from flask import Flask, request
from logging.handlers import RotatingFileHandler

app = Flask(__name__)

handler = RotatingFileHandler('/tmp/foo.log', maxBytes=10000, backupCount=1)
handler.setLevel(logging.INFO)
app.logger.addHandler(handler)

@app.route("/")
def hello():
    app.logger.error(('The referrer was {}'.format(request.referrer)))
    return "Hello World!"
```

2. Realize a criação de uma imagem, cuja nome será **oficina-docker** e etiqueta será **aula2-volumes** através do comando “docker build -t oficina-docker:aula2-volumes . “;
3. Crie um contêiner com o nome **aula-2** a partir da imagem com um vínculo da porta 5000 do host para a porta 5000 do contêiner para facilitar a visualização através do um browser, através do comando `docker run -d --name aula-2 -p 5000:5000 oficina-docker:aula2-volumes`;
4. Acesse a aplicação a partir de um browser no endereço `http://localhost:5000` e realize algumas requisições para que estas possam ser inseridas pela aplicação no arquivo ‘foo.log’ dentro do contêiner;
5. Visualize o conteúdo do arquivo através do comando `docker exec -it aula-2 cat /tmp/foo.log`;
6. Remova o contêiner através do comando `docker rm -f aula-2`;
7. Repita os passos 3 e 5 (sem passar pelo passo 4) e tente visualizar possíveis diferenças no arquivo foo.log;
8. Crie um volume chamado `aula2-logs` através do comando: `docker volume create aula2-logs`;
9. Remova o contêiner através do comando `docker rm -f aula-2`;
10. Recrie o contêiner agora com suporte ao *named volume* ‘aula2-logs’ através do seguinte comando `docker run -d --name aula-2 -p 5000:5000 -v aula2-logs:/tmp oficina-docker:aula2-volumes`;
11. Repita os passos 4, 5, 6 e tente visualizar as possíveis diferenças no uso dos volumes.

5.2.1 Informações e/ou questões adicionais

Que outros efeitos colaterais temos ao utilizar um arquivo com alto nível de mudança de conteúdo fora de um volume?

Considerando que o objetivo atual é auxiliar no desenvolvimento da aplicação, existe alguma maneira mais simples de manter o arquivo de log ativo e acessível para o desenvolvedor?

Se removermos o contêiner o que acontecerá com o volume?

Se removermos o volume o que acontecerá o arquivo foo.log?

Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 2 - Exercicio 1"
# Criar uma etiqueta para esta aula.
git tag -a aula2_1 -m "Aula 2 - Exercicio 1"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula2_1
```

5.3 Cap. 2 - Exercício1: Persistência das configurações do contêiner

Para aplicar os conhecimentos explanados nesta aula, será necessário criar um arquivo *docker-compose.yml* que contenha todas as definições necessárias para que a aplicação possa ser construída e iniciada a partir do mesmo conforme especificações abaixo:

1. Crie um arquivo chamado docker-compose.yml dentro da pasta da aplicação;
2. Na declaração da versão do arquivo, utilize a versão '2';
3. A seguir, declare um volume chamado 'aula2-logs' que será um volume local, sem detalhes adicionais;
4. Na declaração dos serviços, crie um serviço chamado 'aula2';
5. Na declaração da imagem do serviço utilize a imagem `oficina-docker:aula2-volumes`;
6. Na declaração de volumes, mapeie o volume `aula2-logs` para `/tmp`;
7. Salve e feche o arquivo; em seguida, utilize o comando `docker-compose up -d` para criar o contêiner e o volume previamente declarados;
8. Através do comando *docker ps* verifique se o seu contêiner foi iniciado e se as informações dele são condizentes com o contêiner anterior;
9. Abra novamente o arquivo *docker-compose.yml* e insira a definição `container_name: aula-2` no escopo principal do serviço 'aula-2';
10. Repita o passo 7 e verifique as diferenças no novo contêiner criado.

5.3.1 Informações e/ou questões adicionais

Lembre-se que arquivos YAML utilizam a indentação para reconhecer os diferentes níveis de funcionalidades.

As várias formas de realizar a declaração do arquivo 'docker-compose.yml' afim de se realizar o mapeamento dos recursos do Docker estão disponíveis em https://docs.docker.com/compose/compose-file/compose-file-v2/.

É possível utilizar o comando docker-compose config para verificar se um arquivo docker-compose.yml é valido.

Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 2 - Exercicio 2"
# Criar uma etiqueta para esta aula.
git tag -a aula2_2 -m "Aula 2 - Exercicio 2"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula2_2
```

5.4 Cap. 3 - Exercício1: Persistências de Imagens em um Docker Registry

Para aplicar os conhecimentos explanados nesta aula, será necessário aplicar uma *tag* a imagem criada na aula anterior, de forma que seja possível enviar esta imagem para o registry central. As ações realizadas devem balizar o conjunto de especificações abaixo:

1. Com base na imagem "oficina-docker:aula2-volumes" atribua uma nova tag através do comando `docker tag oficina-docker:aula2-volumes <IP_REGISTRY_CENTRAL:5000>/<USUARIO_GITHUB>/oficina-docker:aula2-volumes`;
2. Realize o login para o registry "<IP_REGISTRY_CENTRAL:5000>" através do comando `docker login <IP_REGISTRY_CENTRAL:5000>` utilizando como credenciais (usuário/senha) o seu nome de usuário do github;
3. Envie a imagem ao registry central através do comando `docker push <IP_REGISTRY_CENTRAL:5000>/<USUARIO_GITHUB>/oficina-docker:aula2-volumes`;
4. Exclua a sua imagem "oficina-docker:aula2-volumes" local através do comando `docker rmi oficina-docker:aula2-volumes`;
5. Realize o download da imagem novamente, mas a partir do registry central através do comando `docker pull oficina-docker:aula2-volumes`.

5.4.1 Informações e/ou questões adicionais

Se uma determinada imagem estiver sendo utilizada por um contêiner, será necessário remover este contêiner antes de remover a imagem.

Que tipo de recursos seriam desejáveis caso fosse desejado ter um registry que atendesse a um ambiente corporativo?

Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
```

(continues on next page)

(continuação da página anterior)

```
git commit -m "Aula 3 - Exercicio 1"
# Criar uma etiqueta para esta aula.
git tag -a aula3_1 -m "Aula 3 - Exercicio 1"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula3_1
```

5.5 Cap. 3 - Exercício2: Uso e Comunicação entre contêineres em redes docker

Para aplicar os conhecimentos explanados nesta aula, será necessário atualizar a aplicação afim de que a mesma passe a utilizar um cache Redis. Para tanto, as ações abaixo balizam as mudanças necessárias:

1. Atualize o arquivo "app.py" da aplicação em python para que possua o seguinte conteúdo:

```
from flask import Flask, request
from logging.handlers import RotatingFileHandler

from flask.ext.redis import FlaskRedis

import logging
import datetime

#Create the App
app = Flask(__name__)

#Redis Connection URL
app.config['REDIS_URL'] = "redis://redis:6379/0"

#Bind Redis Connection to app
redis_store = FlaskRedis(app)

#Create logs
handler = RotatingFileHandler('/tmp/foo.log', maxBytes=10000, backupCount=1)
handler.setLevel(logging.INFO)
app.logger.addHandler(handler)

#Send Values to Redis.
redis_store.set('Start Time', datetime.datetime.now())

@app.route("/")
def hello():
    app.logger.error(('The referrer was {}'.format(request.referrer)))
    return "Hello World!"
```

2. A seguir, inclua a dependência "flask-redis" no arquivo requirements.txt relativo a aplicação;
3. Crie um arquivo docker-compose.yml com o seguinte conteúdo:

```
version: '2'
volumes:
  app_data:
  redis_data:

services:
```

(continues on next page)

(continuação da página anterior)

```
  app:
    image: oficina:aula3-network
    container_name: aula-3
    build: .
    volumes:
     - app_data:/tmp

  redis:
    image: redis:alpine
    volumes:
     - redis_data:/data
```

4. Inicialize os novos contêineres através do comando `docker-compose up -d`;
5. Verifique se a chave 'Start Time' foi criada no redis através do comando `docker-compose exec redis redis-cli keys '*'`.

5.5.1 Informações e/ou questões adicionais

O que acontecerá caso a declaração do serviço 'redis' mude de nome?

É possível refatorar a aplicação para que ela funcione com outros servidores redis?

Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 3 - Exercicio 2"
# Criar uma etiqueta para esta aula.
git tag -a aula3_2 -m "Aula 3 - Exercicio 2"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula3_2
```

5.6 Cap. 3 - Exercício 3: Centralização e Visualização dos Logs dos Contêineres

Para aplicar os conhecimentos explanados nesta aula, será necessário implementar um conjunto de soluções e integrações para guarda de logs de um contêiner de teste plenamente funcional. Para tanto, as ações abaixo balizam a implantação de um *stack* utilizando fluentd (coleta/recepção), elasticsearch (guarda) e Kibana (Visualização):

1. Remover o contêiner do fluentd previamente criado através do comando `docker-compose down`;
2. Como root, rodar o comando `sysctl -w vm.max_map_count=262144` necessário para funcionamento do ElasticSearch;
3. Criar uma nova pasta chamada "logs" e inserir o seguinte conteúdo para o arquivo docker-compose.yml:

```
version: '2.2'
volumes:
  esdata1:
  kibana-plugins:
  kibana-bundle:

services:
  fluentd:
    image: elastic-fluentd
    build: fluentd/
    restart: unless-stopped
    ports:
     - 24224:24224
    volumes:
     - ./fluentd/fluent.conf:/fluentd/etc/fluent.conf

  elasticsearch:
    image: "docker.elastic.co/elasticsearch/elasticsearch:6.1.2"
    volumes:
      - esdata1:/usr/share/elasticsearch/data

  kibana:
    image: "docker.elastic.co/kibana/kibana:6.1.2"
    ports:
     - 5601:5601
    volumes:
     - kibana-plugins:/usr/share/kibana/plugins
     - kibana-bundle:/usr/share/kibana/optimize

  web-test:
    image: nginx:alpine
    volumes:
     - ./nginx.conf:/etc/nginx/nginx.conf
    ports:
     - 8082:80
    logging:
      driver: fluentd
      options:
       fluentd-address: localhost:24224
       tag: "docker-web.{{.ImageName}}/{{.Name}}/{{.ID}}"
```

4. Criar uma pasta chamada “fluentd” dentro da pasta “logs” e, nela, criar o arquivo Dockerfile com o seguinte conteúdo:

```
FROM fluent/fluentd
RUN gem install fluent-plugin-elasticsearch --no-rdoc --no-ri
```

5. Ainda na pasta “fluentd”, proceda com a criação de um arquivo chamado “fluent.conf” contendo as seguintes configurações:

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>
```

(continues on next page)

(continuação da página anterior)

```
<filter docker-web**>
    @type parser
    format json
    key_name log
</filter>

<match docker-web**>
  @type copy
  format nginx
  <store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix docker-web
    logstash_dateformat %Y%m%d
    include_tag_key true
    tag_key @log_name
    flush_interval 1s
  </store>
</match>
```

5. Criar um arquivo chamado “nginx.conf” na pasta “logs”, contendo as seguintes configurações:

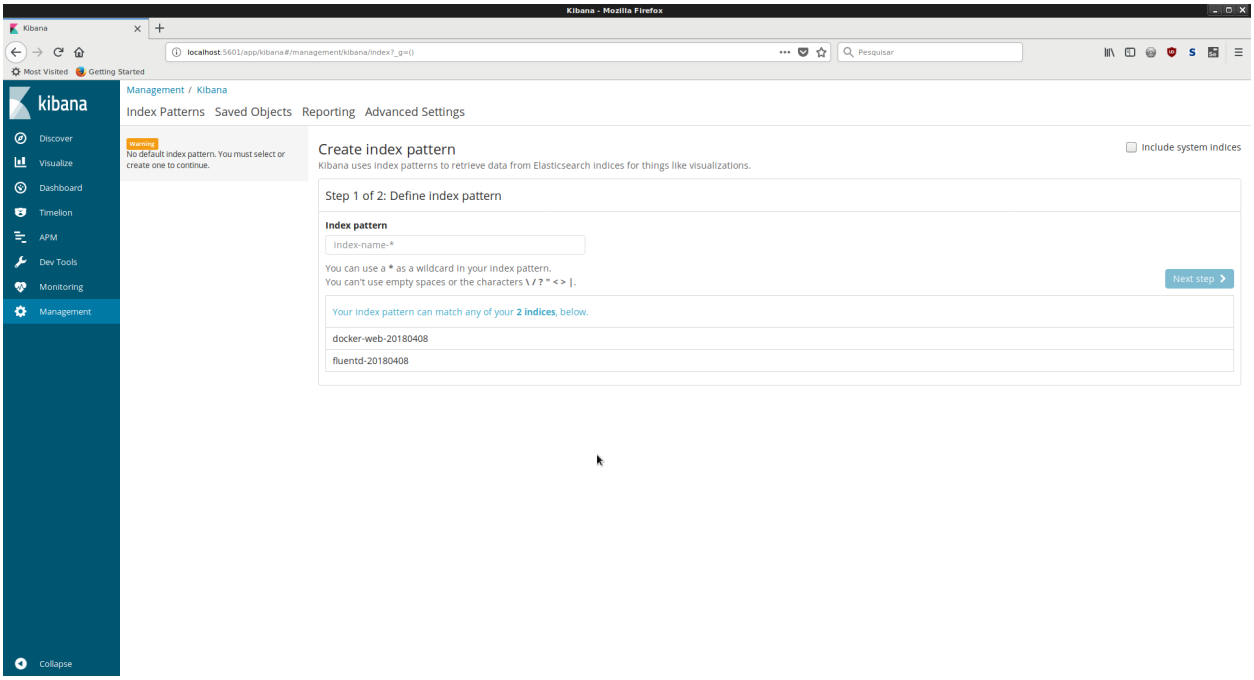
```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format main escape=json '{ "time_local": "$time_local", '
     '"remote_addr": "$remote_addr", '
     '"remote_user": "$remote_user", '
     '"request": "$request", '
     '"status": "$status", '
     '"body_bytes_sent": "$body_bytes_sent", '
     '"request_time": "$request_time", '
     '"http_referrer": "$http_referer", '
     '"http_user_agent": "$http_user_agent" }';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;
```

(continues on next page)

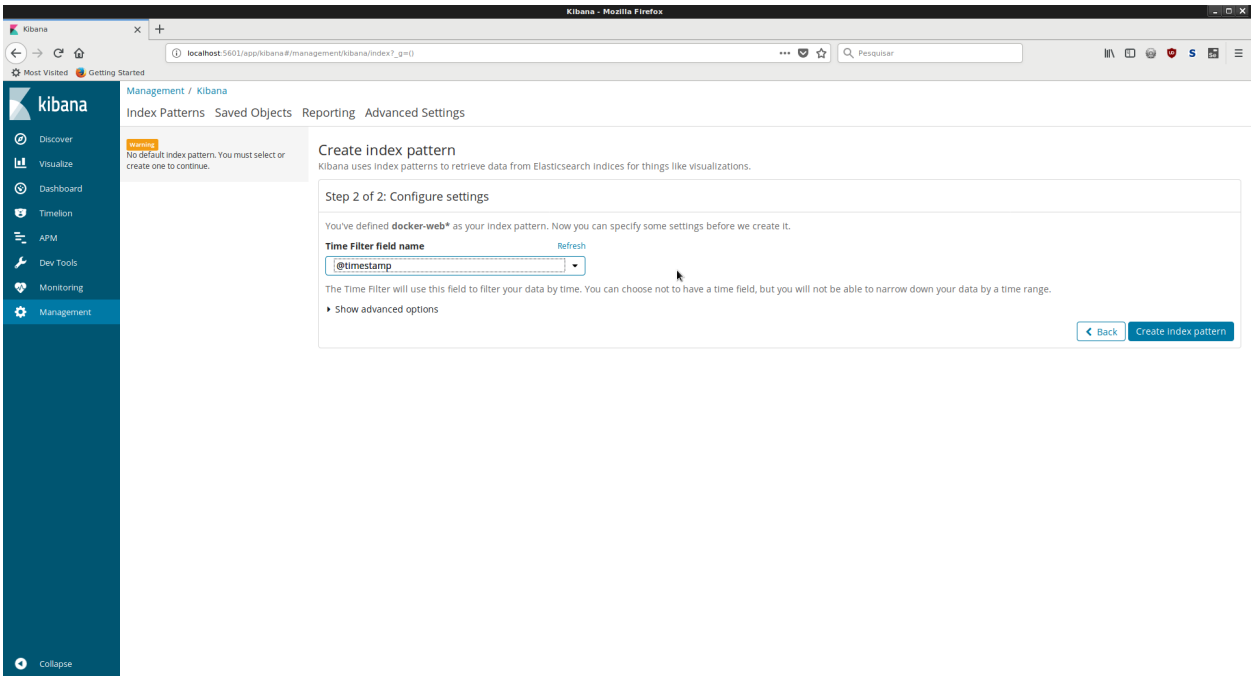
(continuação da página anterior)

```
    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}
```

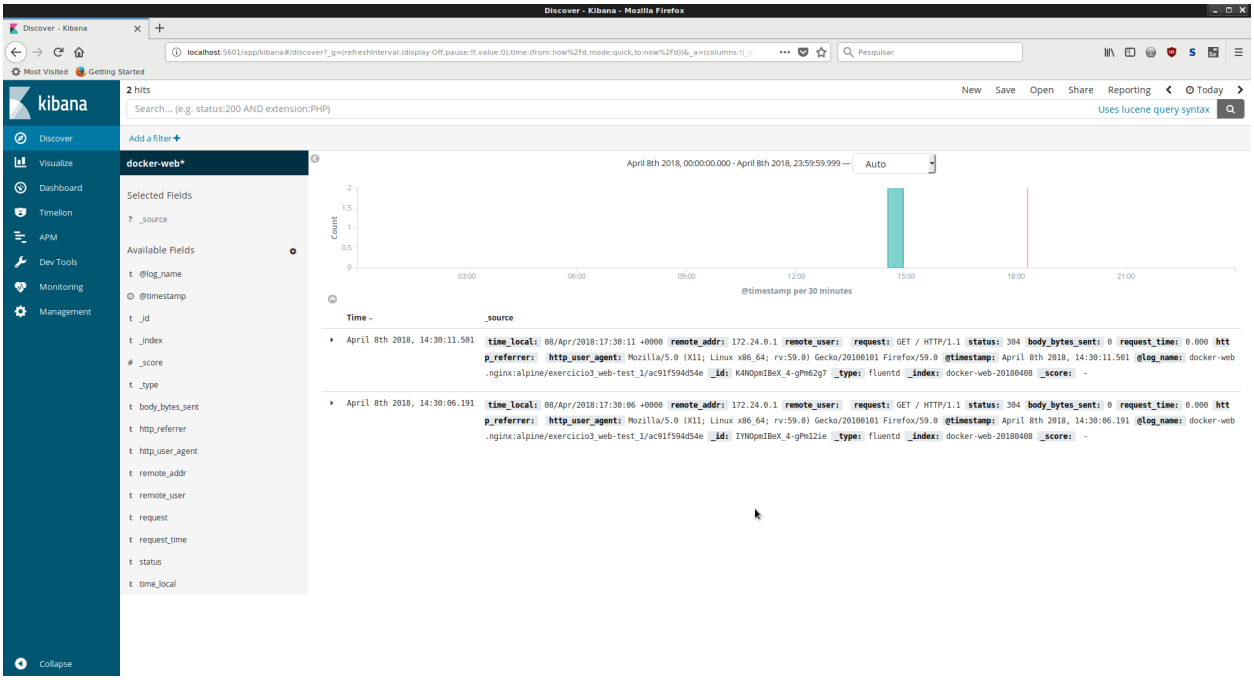
5. Inicializar os contêineres através do comando `docker-compose up -d`. Nesse ponto, o elasticsearch e o Kibana podem demorar de 1 a 2 minutos para serem inicializalidos a depender da configuração de hardware do host;
6. Realizar algumas requisições HTTP para o endereço `http://localhost:8082` afim de que logs sejam gerados e enviados ao fluentd e elasticsearch;
7. Acessar o Kibana através de um browser no endereço `http://localhost:5601` e clicar no item "Patterns", conforme figura abaixo:



8. No campo index pattern, incluir o valor "docker-web*" e em seguinte clicar no botão "Next Step";
9. Na tela seguinte, no campo "Time Filter field name", escolher a opção "@timestamp", conforme figura abaixo:



10. Por fim, basta visualizar os logs gerados clicando no item de menu "Discover" no painel lateral. A visualização há de ocorrer de forma parecida com a da figura abaixo:



5.6.1 Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 3 - Exercicio 3"
# Criar uma etiqueta para esta aula.
git tag -a aula3_3 -m "Aula 3 - Exercicio 3"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula3_3
```

5.7 Cap. 4 - Exercício 1: Envio e Visualização de Métricas via Beats e Kibana

Para aplicar os conhecimentos explanados nesta aula, será necessário implementar um conjunto de soluções e integrações para guarda das métricas de funcionamento dos contêineres.

Para tanto, a estrutura relativa ao exercício do Capítulo 3: Centralização e Visualização dos Logs dos Contêineres, precisará estar funcionando, pois o MetricBeat será conectado a rede anteriormente criada ("logs_default") e utiliza o ElasticSearch para realizar a guarda das coletas/métricas.

As ações abaixo balizam a implantação de um *stack* utilizando Elastic Beats (coleta), elasticsearch (guarda) e Kibana (Visualização):

1. Criar uma nova pasta chamada "monit" e inserir o seguinte conteúdo para o arquivo docker-compose.yml:

```
version: '2.2'
volumes:
  beats-data:

services:
  monit:
    image: docker.elastic.co/beats/metricbeat:6.1.2
    volumes:
     - beats-data:/usr/share/metricbeat
     - /var/run/docker.sock:/var/run/docker.sock
     - ./metricbeat.yml:/usr/share/metricbeat/metricbeat.yml
     - /proc:/hostfs/proc:ro
     - /sys/fs/cgroup:/hostfs/sys/fs/cgroup:ro
     - /:/hostfs:ro

    networks:
     - logs_default

networks:
  logs_default:
    external: true
```

2. Criar o arquivo de configuração do MetricBeat, chamado `metricbeat.yml` conforme definições abaixo:

```
metricbeat.config.modules:
  path: /usr/share/metricbeat/modules.d/*.yml
  reload.enabled: true

processors:
- add_cloud_metadata:
```

(continues on next page)

(continuação da página anterior)

```
output.elasticsearch:
  hosts: ['elasticsearch:9200']

setup.kibana:
  host: "kibana:5601"
```

3. Inicializar os contêineres através do comando `docker-compose up -d`;
4. Ativar o módulo responsável pelo monitoramento das métricas dos contêineres através do seguinte comando:

```
$ docker-compose exec monit metricbeat modules enable docker
```

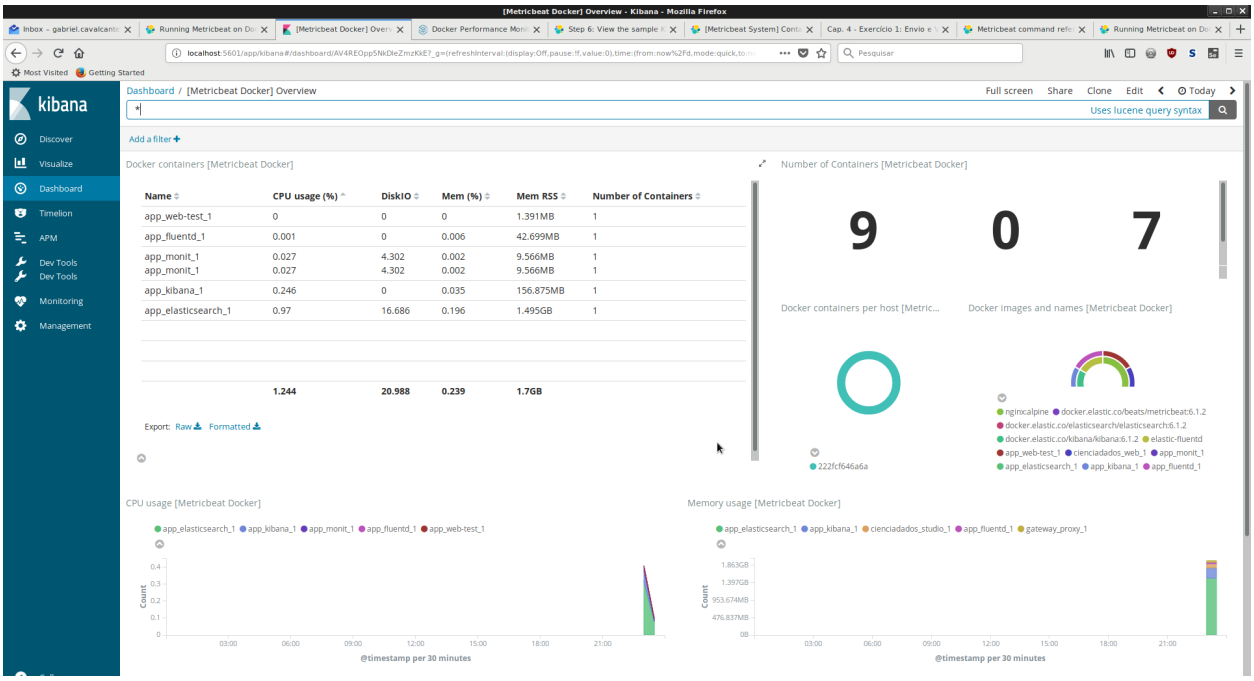
5. Se a api do Docker funcionar somente via socket, será necessário ajustar a permissão do socket no **host**, da seguinte forma:

```
$ sudo setfacl -m u:1000:rwx /var/run/docker.sock
```

6. A seguir, atualizar as configurações do Kibana para incluir as estruturas necessárias para recebimento das métricas e os *dashboards/visualizações*:

```
$ docker-compose exec monit metricbeat setup
```

7. Por fim, deve-se realizar o acesso ao kibana no endereço `http://localhost:5601` e, em seguida, clicar no item "DashBoards", link " [Metricbeat Docker] Overview ", que deverá resultar na seguinte visualização:



5.7.1 Salvando os trabalhos

Após a realização das atividades, salve o resultado do trabalho no github, através dos seguintes comandos (a partir da pasta onde os trabalhos se encontram):

```
# Adicionar os arquivos atuais ao repositório
git add .
# Realizar o 'Commit' das mudanças no repositório local.
git commit -m "Aula 4 - Exercicio 1"
# Criar uma etiqueta para esta aula.
git tag -a aula4_1 -m "Aula 4 - Exercicio 1"
# Enviar as mudanças para o repositório remoto.
git push -u origin master aula4_1
```