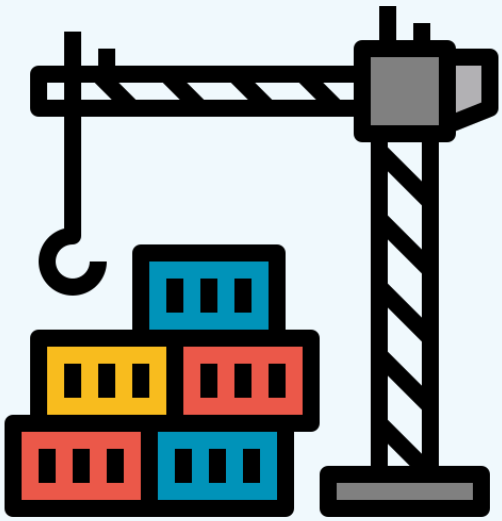
4LINUX

OPEN SOFTWARE SPECIALISTS



Container Fundamentals

João Paulo Serafim / joaopauloserafim@gmail.com

Sumário

1

Introdução

A TI nunca parou de crescer, os sistemas e infraestruturas nunca deixaram de ser desenvolvidos e melhorados, e hoje em torno de 50% da população mundial tem acesso a internet. Apesar de ser um número um pouco triste pensando pelo lado social, pois a internet é a grande biblioteca do nosso planeta, 50% da população mundial aponta para algo em torno de 3.5 bilhões de usuários.

Os números acima indicam que o uso da internet, de softwares, sites e etc, mesmo se estagnassem, ainda representaria uma quantidade imensa de usuários, que utilizam serviços online ou em suas empresas das mais diversas formas.

Além do usuários temos as comunicações entre os mais diversos serviços que acontecem automaticamente por todo o lado. Pode ser desde o sistema que controla a entrada e saída de usuários da padaria do bairro e as envia para um banco na nuvem ou as milhões de transações processadas pelas empresas de cartão de crédito.

O que tudo isso significa?

Significa que precisamos de software, softwares melhores, mais fáceis de usar, de desenvolver, mais performáticos e, principalmente, mais fáceis de provisionar.

É interessante que nossas máquinas pessoais compartilhem o maior número possíveis de bibliotecas, tenham um software compilado especificamente para nossas arquiteturas de processador e todas as outras coisas exclusivas que até o mais simples dos computadores pessoais desfruta.

Mas e em uma grande infraestrutura que roda milhares de aplicações, cada qual com suas

dependências, bibliotecas e exigências? Uma simples atualização, se apresentasse problemas, poderia gerar prejuízos incalculáveis. Aliás, incalculáveis é uma forma simplista de dizer, afinal aqueles que gerenciam as empresas sabem exatamente qual o preço das horas de um determinado setor parado, ou mesmo da empresa toda. Parece besteira? Em 2012 a Knight Capital perdeu 440 milhões em 45 minutos quando um software de negociações se comportou de forma inesperada, neste caso o software não parou, pelo contrário, funcionou a todo o vapor, mas acredito que deu para pegar a ideia. Voltando ao problemas das dependências, bibliotecas e exigências percebemos o tamanho do problema em criar aplicações estáveis o suficiente para resistirem ao tempo e ao stress do uso contínuo, o desafio é maior ainda quando precisamos atualizá-las, e muito maior quando precisamos escalar essas aplicações para várias réplicas de forma a antederem a demanda que cresceu por qualquer motivo, como por exemplo, uma black-friday.

É um pesadelo, não é mesmo? Eu não sei vocês mas me lembro das noites mal dormidas, dos copos de café e coração acelerado.

Parte deste pesadelo agora é passado, ou pelo menos mais fácil de resolver. Com os contêineres podemos deixar todas as dependências das aplicações dentro deles mesmos, de forma isolada, sem interferir com o sistema ou com as outras aplicações. Além disso, podemos movê-los de um lado para o outro, entre os ambientes de teste, homologação e produção, facilitando os testes e garantindo que o objeto que transita durante este processo é exatamente o mesmo que entrará em produção, sem alterações, sem ajustes. Podemos até mesmo criar os ambientes de forma praticamente instantânea!

Parece fantasia, certo? Eu sei, mas toda essa possibilidade já existe a um tempinho. Ela se moldou através do tempo, das tecnologias, e evoluiu para o que hoje conhecemos como "container".

Sobre as Tecnologias

Hoje existem várias ferramentas para se criar contêineres no Linux, a mais famosa ainda é o Docker e é o foco deste curso. Não se preocupe em precisar aprender todas, elas fazem praticamente a mesma coisa e os conceitos teóricos a respeito dos contêineres continuam o mesmo.

São como as ferramentas para criar máquinas virtuais, QEMU (com ou sem KVM), VirtualBox, VMWare ou HyperV.

Isso significa que o que você aprender aqui servirá para as outras ferramentas, as vezes nem a parte prática precisa ser ajustada pois os comandos são os mesmos.

Apenas por questão de curiosidade, as outras ferramentas equivalentes ao Docker são: LXD e podman.

Entretanto, existem outras categorias de ferramentas como o containerd, katacontainer e cri-o mas elas estão atreladas a um tipo de arquitetura mais complexa, geralmente distribuída e possuem um propósito diferente.

Não abordaremos estas tecnologias e suas diferenças em relação ao Docker neste curso pois isso traria elementos que não são necessários neste momento e na maioria das vezes são transparentes para seus usuários.

2

Contêiner

Neste primeiro capítulo abordaremos a teoria a respeito dos contêineres, suas diferenças em relação a infraestrutura que tínhamos até então, algumas ferramentas de maior destaque no mercado, suas vantagens e valores para o negócio além do papel dos orquestradores, os gerenciadores de cluster de contêineres. Não se preocupe, provavelmente você já ouviu falar de um deles, o Kubernetes.

O que é um contêiner

Contêiner é um pacote de uma aplicação que funciona de forma isolada do sistema operacional, com todas as suas dependências dentro de si. Uma aplicação autocontida que se comporta da mesma forma independente do ambiente em que está.

Bom, pelo menos é isso o que eu sempre digo para os meus alunos.

Mas para reformar minha definição vamos ver o que alguns dos grandes nomes do software open source também dizem:

- **SUSE**: Containers are software packages that provide an entire runtime environment: an application, plus its dependencies, system libraries, settings and other binaries, and the configuration files needed to run it. Containerized apps will always run the same, regardless of the environment. - *https://susedefines.suse.com/definition/containers/ - 08/2020*
  - **Tradução:** Contêineres são pacotes de software que providenciam o ambiente de execução completo: a aplicação, mais suas dependências, bibliotecas de sis-

tema, configurações e outros binários além dos arquivos de configuração necessários para seu funcionamento. Aplicações conteinerizadas funcionarão sempre da mesma forma, independente do ambiente.

- **RedHat**: Linux containers are technologies that allow you to package and isolate applications with their entire runtime environment—all of the files necessary to run. This makes it easy to move the contained application between environments (dev, test, production, etc.) while retaining full functionality. - *https://www.redhat.com/en/topics/containers - 08/2020*
  - **Tradução:** Contêineres Linux são tecnologias que nos permitem empacotar e isolar aplicações com seus ambientes de execução por completo somado a todos os arquivos necessários para funcionar. Desta forma facilitando a movimentação das aplicações conteinerizadas entre ambientes (desenvolvimento, teste, produção, etc.) enquanto mantêm o funcionamento completo.

A segunda definição citou a palavra "tecnologias", no plural. Isso me permite avançar um pouco mais na explicação e dizer que um contêiner não é uma única tecnologia, como o Docker que veremos mais adiante no curso, mas sim um conjunto de tecnologias presentes no kernel Linux que quando unidas nos permitem obter todas as vantagens que citamos e citaremos durante o curso.

Então, ao utilizar contêineres, o mínimo que teremos é uma aplicação isolada um pouco mais segura do que se estivesse avulsa e esquecida dentro do nosso sistema.

Tecnologias Envolvidas

Se um contêiner é o resultado de um conjunto de tecnologias do kernel do Linux, quais são estas tecnologias? Bem, vamos analisar quais tecnologias envolvidas existem e quando foram criadas até o ponto em que estamos:

- Em 1979, a versão 7 do Unix criou os fundamentos da tecnologia, ou do isolamento, o `chroot`. O chroot isola o processo somente no nível de sistema de arquivos, ele ainda tem acesso a rede original ou aos usuários, por exemplo.
- Em meados de 1980 um sistema operacional chamado Plan 9 passou a utilizar uma forma de combinar diretórios conhecida como **union mounting**.
- Em 1986 a Sun Microsystems introduziu a primeira implementação de um sistema de arquivos baseado em camadas com "copy on write", o **Transluced File Service** no SunOS 3.
- Nos anos 2000 o FreeBSD 4 apresentou os **jails**, a evolução do chroot. Neste caso o processo continua isolado em seu sistema de arquivos, mas o acesso a outros recursos como usuários, rede e até mesmo outras partes do sistema de arquivos da máquina podem ser liberados e acessados de forma virtualizada, como se a máquina estivesse particionada em vários segmentos. Aqui surgiu o termo conhecido como **virtualização**

em nível de sistema operacional.

- Em 2001 o jails foi melhorado com o **Linux VServer** adicionando um particionamento quase que total, vários sistemas Linux poderiam rodar na mesma máquina sem a camada de emulação, todos em cima de um único sistema operacional, utilizando o mesmo Kernel.
- Em 2002 surgem os **namespaces**, capaz de separar os recursos do Kernel do Linux de forma que cada processo dentro de um namespace diferente visualizasse um conjunto diferente de recursos, sendo assim, processos em um mesmo namespace veriam os mesmos recursos.
- Em 2005 o **OpenVZ** adicionou oficialmente as capacidades do Linux VServer ao Kernel do Linux.
- Em 2006 a tecnologia evoluiu mais ainda com o **cgroups** (control groups) que implementaram o isolamento e o limite a recursos como CPU e memória.
- Em 2008 apareceu o **LXC** (Linux Containers), que definiu como seriam os contêineres de um futuro não tão distante, funcionavam no Kernel de forma completa, sem nenhum tipo de patch. O LXC era tão estável e simples que as outras duas próximas tecnologias surgiram utilizando-o como núcleo.
- Em 2011 surge o **Wardem**, hoje o núcleo do Cloud Foundry. Neste ponto o interessante era criar uma ferramenta que não só fosse capaz de criar contêineres mas também gerenciá-los por conta própria.
- Em 2013 aparece o tão conhecido **Docker**, que no momento era baseado em LXC e libcontainers, com o passar do tempo as coisas mudaram um pouco.

Então todas essas tecnologias estão presentes no Docker? Essa é uma pergunta bastante pertinente, nem todas, além do mais muitas delas evoluíram. Diferentes ferramentas utilizam diferentes tecnologias, mas podemos citar três delas como as principais:

- **Namespaces:** É através dos namespaces que conseguimos o isolamento dos processos que rodam dentro dos contêineres, não somente dos processos em si mas também dos pontos de montagem como diretórios e dispositivos de disco, da rede e alguns outros.
- **Cgroups:** É uma abreviação para **control groups**, através do cgroups é possível limitar e isolar o acesso a recursos da máquina como processador, memória e tráfego de rede.
- **Union Mount:** São sistemas de arquivos baseados em camadas. Essas camadas podem ser compartilhadas entre diferentes contêineres, reduzindo o tamanho total da imagem em que o contêiner é baseado. Alguns exemplos são UnionFS, overlay2, devicemapper, btrfs, zfs e aufs.

Certo, eu sei o que estão pensando, provavelmente a próxima pergunta será:

Mas que imagem?

Imagem

São as imagens que tornam os contêineres fáceis de se transportar. As imagens são imutáveis, como uma foto, contêm todo o ambiente de execução de um contêiner, suas dependências e binários. Tudo isso junto em um único "pacote", e a este pacote damos o nome de imagem. Uma imagem, ou uma foto, é a definição de algo naquele determinado instante no tempo e não podemos mudá-la (bem, ao menos não deveríamos) e uma vez que tenhamos criado esta imagem seu comportamento será o mesmo independente do ambiente, facilitando o trabalho dos programadores e dos responsáveis pelos provisionamentos.

Nas ferramentas que citaremos ou apresentaremos no curso, é através das imagens que criamos os nossos contêineres.

Podemos fazer a analogia de que imagens de contêineres são exatamente como discos óticos ou arquivos do tipo ISO, por exemplo, de uma distribuição Linux. Uma vez que tenhamos aquela imagem de uma distribuição Linux podemos realizar quantas instalações quisermos, cada instalação será um sistema de um usuário diferente e portanto, com o tempo de uso, apresentarão suas diferenças.

Mas a grande novidade para os contêineres não são as imagens, mas sim imagens baseadas em camadas.

Camadas

Na maior parte das vezes os contêineres são baseados em imagens preexistentes, e estas imagens, por sua vez, podem ser baseadas em outras imagens e assim sucessivamente. Por exemplo, a imagem padrão do servidor web **Apache** é baseada em **Debian**. Se criarmos uma imagem de contêiner com um site nosso teríamos uma imagem baseada na imagem do **Apache** que por sua vez é baseada em **Debian**.

Para simplificar, vamos imaginar que a imagem citada acima possui três camadas: a camada do nosso site, a camada do Apache e a camada do Debian. Com nossa imagem pronta para ir para produção pedimos para o servidor **containers.example.com** baixar esta imagem. O nosso servidor então baixa as três camadas, a do site, a do Apache e a camada do Debian. Alguns dias depois criamos uma imagem de um novo site e pedimos para o servidor **containers.example.com** baixá-la. Desta vez o servidor reconhece que possui as camadas Apache e Debian e baixa apenas a camada da imagem que representa o novo site.

Desta forma baixamos ou enviamos somente as camadas que não temos, fazendo com que novas aplicações ou atualizações cheguem mais rápido de um lugar ao outro.

Parece fantástico, certo?! Eu sei que tudo isso soa um pouco mágico, mas veremos mais

detalhes sobre imagens quando começarmos a colocar a mão na massa.

Contêineres e Máquinas Virtuais

Muita gente compara contêineres com máquinas virtuais, e apesar dos contêineres não serem um substituto direto das máquinas de virtuais, ambas as tecnologias possuem conceitos similares e talvez comparando-os lado a lado facilite a compreensão.

Se "máquina virtual" lhe é um assunto novo, apenas imagine que uma máquina virtual é uma máquina física transformada em software e que pode funcionar dentro de outros sistemas operacionais.

Quando abordamos temos como virtualização ou mesmo contêineres, temos um conceito em comum, a máquina hospedeira. A máquina hospedeira é a máquina pela qual as máquinas virtuais e/ou os contêineres executam, já a máquina convidada é somente a máquina virtual dentro de uma hospedeira.

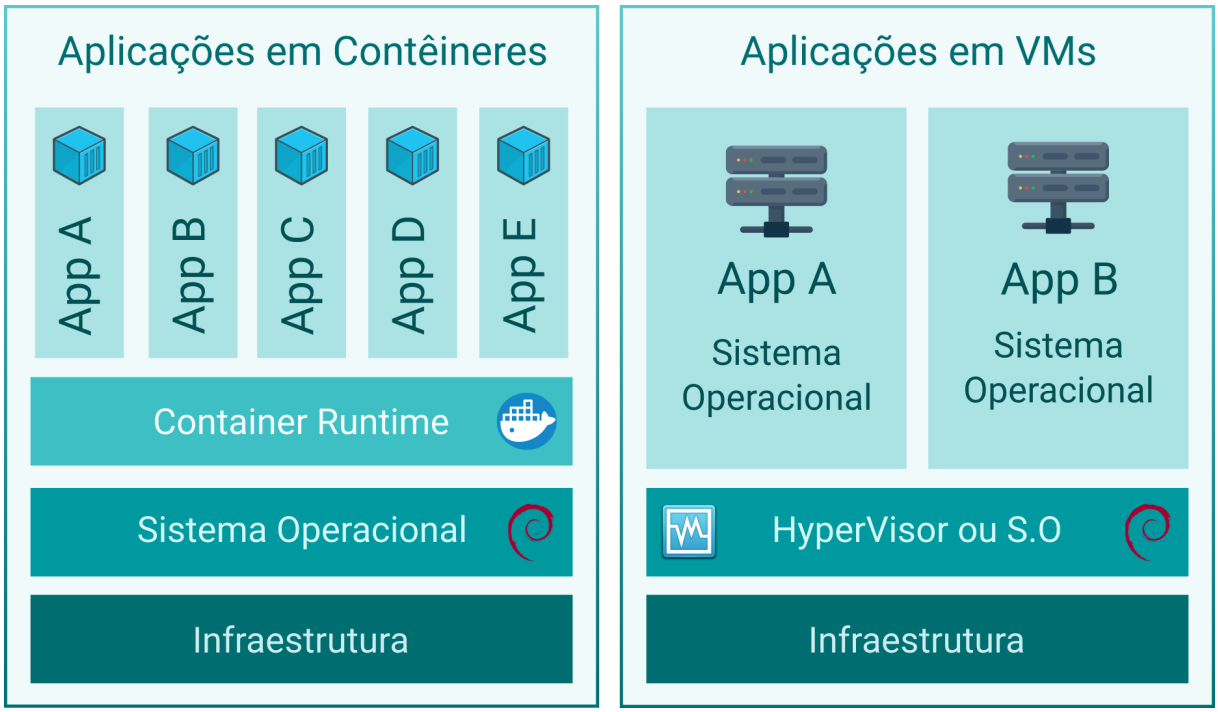


Fig. 2.1: Contêineres e Máquinas Virtuais

Primeira Camada

No nível mais baixo temos a infraestrutura, que pode ser nossa própria máquina ou um servidor.

Segunda Camada

Para máquinas que rodarão suas aplicações em contêineres temos apenas o sistema operacional, já para aqueles que rodarão máquinas virtuais temos a figura do **Hypervisor**, que pode ser o próprio sistema (XEN, KVM ou VMWare) ou uma aplicação dentro do sistema operacional (VirtualBox, KVM, Hyper-V).

Terceira Camada

Não colocamos uma terceira camada nas máquinas virtuais pois não há uma definição clara a respeito de Hypervisors do tipo 1 e do tipo 2, principalmente com o surgimento do KVM, mas isso não é assunto para este curso. Do lado dos contêineres aparece a figura do **Container Runtime**, que no nosso caso podemos entender como o **Docker** que veremos mais adiante, mas é interessante saber que existem outras ferramentas para criar contêineres no Linux.

Container runtimes são implementações de baixo nível, elas que de fato criam os contêineres no sistema operacional, geralmente utilizados por uma **Container Engine**, por exemplo, o Docker, RKT, LXD ou CRI-O.

Última Camada

Na última camada temos os nossos contêineres ou as nossas máquinas virtuais, cada qual com suas devidas aplicações. As máquinas virtuais são sistemas completos, com todos os seus processos, drivers e kernel, quando iniciam verificam se todo o hardware está funcional, fazem o boot do sistema operacional e então começam a rodar as aplicações. Já os contêineres, por sua vez, apenas iniciam a aplicação, compartilhando o kernel com o sistema hospedeiro.

Um pouco confuso? Bom, o aspecto mais básico que podemos notar é que as máquinas virtuais são sistemas operacionais completos somados a nossa aplicação e suas dependências funcionando na máquina hospedeira, enquanto os contêineres são praticamente apenas nossa aplicação e suas dependências, utilizando o máximo possível de "coisas" da máquina hospedeira.

O fim das Máquinas Virtuais?

Não, bem longe disso.

Cada caso tem sua vantagem, explicaremos alguns casos a seguir do ponto de vista do Linux:

Quando utilizar VMs

- Quando o sistema operacional a ser executado não é um Linux, por exemplo um Unix, ou um Windows.
- Quando procura-se um nível de persistência de dados maior do que os próprios dados da aplicação.
- Quando a arquitetura da aplicação for diferente da arquitetura da máquina hospedeira, por exemplo amd64 vs arm64, ou mesmo x86.
- Quando a aplicação for antiga (legada) e sua forma de trabalho é completamente monolítica (sem sessão externa, banco local, etc).

Quando utilizar contêineres

- Quando construirmos aplicações "voltadas para cloud", ou seja, podem trabalhar em qualquer lugar.
- Quando estamos criando microsserviços.
- Quando queremos aplicar práticas DevOps ou de CI/CD de forma mais agressiva.
- Quando o projeto é escalável e pode se espalhar em uma infraestrutura que compartilha o mesmo sistema operacional.

Resumindo, os contêineres não vieram para substituir as VMs em todos os casos, é preciso ter cuidado. Mas, na teoria, onde cabe uma VM, cabe mais do que um contêiner. Pequenos componentes, leves e fáceis de movimentar, isso me lembra do próximo assunto...

Por que o símbolo contêiner?

Com o que vimos até agora é fácil entender o nome contêiner como sendo um recipiente para nossa aplicação. Porém podemos aprofundar um pouco mais e talvez essa analogia sirva para fixar os pontos fortes da adoção de contêineres.

Para isso precisamos voltar para século XVIII, antes mesmo das ferrovias, no surgimento das primeiras utilizações de um processo conhecido como **transporte intermodal de carga**, que consiste em um recipiente capaz de transportar algo por meios variados de transporte. Nesta época, meados de 1780, a Inglaterra utilizava o que podemos chamar de primeiros contêineres para transportar carvão através de um canal conhecido como Bridgewater Canal. Esses contêineres de carvão eram feitos de madeira, conhecidos como caixas soltas, logo foram também utilizados nas estradas, puxados por cavalos. Em 1830 estes mesmos contêineres foram utilizados em ferrovias e em 1841, Isambard Kingdom Brunel, um engenheiro civil importante na revolução industrial, apresentou os contêineres de ferro para transportar o carvão. Em torno de 1900 os primeiros contêineres cobertos apareceram e a partir de então, entre 1910 e 1930 passaram a se padronizar. A partir desta data os contêineres passaram a ser carregados dos caminhões diretamente nos vagões e vice-versa. E em meados de 1950,

Malcom McLean, considerado o desenvolvedor do transporte intermodal moderno de carga, ou o pai da "conteinerização", revolucionou o transporte e o comércio internacional adaptando navios petroleiros para carregar contêineres, sendo o primeiro conhecido como SS Ideal X. A forma de carregar o navio era bastante simplificada e diminuiu o tempo de carga, daí em diante as coisas são mais ou menos como já conhecemos.

Estes contêineres de aço são transportados através de estradas, ferrovias, navios e aviões sem modificações em sua estrutura, o transporte se modifica, mas o contêiner não.

Eu sei, muita história, mas é engraçado não é mesmo? Olhando do futuro daquele passado, ou do nosso presente, tudo isso parece muito óbvio, padronizar para facilitar o transporte e manuseio, e por consequência, economizar investimento e tempo. Isso não lhe soa familiar?

Resumo

Nossa! Este foi um capítulo e tanto!

Agora que já sabemos o que é um contêiner e suas singularidades, acrescentamos uma definição: um contêiner é uma virtualização em nível do sistema operacional. Posso dizer isso com tranquilidade, pois nós já sabemos que isso não tem relação nenhuma com máquinas virtuais.

Neste capítulo abordamos o tema *contêiner* como um todo, uma parte técnica e teórica além de um pouco de história.

- Vimos o que são contêineres, pequenos pacotes de software com todas as dependências dentro de si, leves e fáceis de mover entre os mais variados tipos de infraestrutura.
- Aprendemos que um contêiner não é uma ferramenta, mas sim um conjunto de tecnologias do kernel do Linux.
- Descobrimos que os contêineres que utilizaremos são baseados em imagens e estas imagens possuem camadas que podem ser compartilhadas entre si.
- Também comparamos os contêineres com as máquinas virtuais e chegamos a conclusão de que no mínimo possuem um tamanho bem menor.
- Voltamos um pouco no tempo para entender o porque o símbolo do contêiner e pudemos perceber que um formato único que possa ser transportado no oceano, nas estradas e nas ferrovias é muito mais fácil de lidar e econômico financeiramente.

Já que falamos em economia, estamos pronto para o próximo tópico.

3

Mercado

Talvez você esteja procurando trocar de emprego, ou trocar de carreira, como muitos dos nossos alunos, mas não tem muita segurança em relação a isso, ou talvez você esteja tentando convencer sua empresa a utilizar contêineres e o pouco do aspecto técnico que vimos não lhe convenceu. Sendo assim vamos apresentar outros argumentos, assim como alguns números e indicadores a respeito da utilização de contêineres em relação a infraestrutura "clássica", tanto para o mercado de trabalho como para os negócios.

Antes de abrir este capítulo, apenas para efeito de curiosidade, muitas empresas estavam desenvolvendo algum tipo de tecnologia de virtualização em nível de sistema operacional, entre elas podemos citar a Google e a Sun, ou seja, já faz um tempinho que estão procurando alternativas para a virtualização como conhecemos.

Mercado de Trabalho

Aqui pela própria 4Linux, temos uma visão muito boa em relação ao que o mercado de trabalho está procurando, ou seja, quais são as procuras dos alunos e quais treinamentos as empresas tem comprado, e nos últimos anos as demandas dos cursos relacionados a contêineres cresceram bastante.

O Stackoverflow em 2019 fez uma pesquisa com 90 mil programadores e constatou que o Docker é a terceira plataforma de desenvolvimento mais utilizada pelos desenvolvedores, assim como Linux, Docker e Kubernetes ocupam as três primeiras posições entre as plataformas que os desenvolvedores preferem.

Uma outra pesquisa, da Diamanti, uma empresa que procura facilitar a adoção de Kubernetes em nuvem híbrida, indica que 47% dos líderes de TI entrevistados planejam implementar contêineres em um ambiente em produção, enquanto outros 12% afirmaram já utilizá-los desta forma. A pesquisa ainda indica que dois quintos das empresas planejam substituir máquinas virtuais por contêineres.

A Datadog, uma empresa focada em monitoramento, fez uma pesquisa sobre a crescente adoção de contêineres entre seus clientes, mostrando que os números tendem a crescer com o passar do tempo.

Não é preciso ir muito longe, se você possui um perfil no LinkedIn pode ir até as vagas de TI que lhe interessa e muito provavelmente você verá estas três palavras:

- Docker
- Kubernetes
- Containers

Contêineres são sem dúvida uma das melhores opções para se conhecer e aprofundar, sendo assim não precisa se preocupar, você escolheu o curso certo.

Padrão

Sim a tecnologia voa e por vezes estamos estudando algo que eventualmente pode ser descontinuado ou superado por alguma outra ferramenta mais nova. O que garante que não utilizaremos algo que possa ser apenas um "fogo de palha"?

É uma outra questão bastante pertinente. Os contêineres existem a bastante tempo, mas somente de 2013 para cá é que eles realmente se espalharam e praticamente tomaram conta do mercado. Apesar desse caldeirão não parar de ferver os contêineres foram padronizados por um órgão não governamental conhecido como OCI, Open Container Initiative. A OCI padronizou a definição dos **containers runtimes**, suas APIs, estruturas internas e formatos de imagem. Isso garante que não ficaremos presos a um determinado fornecedor além de tornar os contêineres completamente portáveis de uma plataforma para a outra. Descontente com a solução "X"? Sem problemas, podemos utilizar a "Y", a "Z" ou mesmo a "D".

Se desenvolvemos todas as nossas aplicações em máquinas virtuais utilizando VMWare teremos um pouco de trabalho para convertê-las para KVM, isso não acontecerá caso utilizemos contêineres.

Bimodal IT

Segundo o **Gartner**, um modelo de negócio bimodal é aquele capaz de gerenciar dois estilos de trabalho separados mas coerentes, um focado em previsibilidade, manutenção de processos e ambientes legados para o mundo digital, e outro focado em exploração, ou o que podemos chamar de inovação e experimentação, com o objetivo de solucionar novos problemas, geralmente trabalhando na zona da incerteza. Para TI podemos resumir como os ambientes legados, hierárquicos e tradicionais e os ambientes novos, cloud e metodologias ágeis.

Os contêineres criaram, ou facilitaram drasticamente, a conexão necessária para esses dois modelos de governança dentro da TI, e eu explicarei por quê.

Com a utilização de contêineres podemos transferir as aplicações legadas para novos ambientes sem comprometer a segurança ou o funcionamento da aplicação, pois internamente, na prática, ela estará no mesmo ambiente que sempre esteve. Ao transferir nossas aplicações para contêineres, na teoria, automaticamente estamos a um passo para utilizar novos modelos de infraestrutura baseadas em IaC (Infrastructure as Code), isso significa que além de poder mover minha aplicação de forma simplificada entre um número qualquer de máquinas, ainda posso replicar essa infraestrutura quantas vezes eu quiser e geralmente em minutos, ou até segundos.

Documentação sempre foi um problema nas metodologias ágeis certo? Pois eu lhe digo que ao construir nossas imagens, utilizamos um arquivo de texto, e para definir nossa infraestrutura também utilizamos algum tipo de arquivo de texto, ambos completamente legíveis. Na pior das hipóteses temos pelo menos um esboço de documentação completamente reproduzível.

Podemos replicar ambientes completos em segundos, facilitando qualquer tipo de teste. Ganhamos consistência, produtividade, replicabilidade, segurança e agilidade, muita agilidade, tudo isso porque estamos utilizando contêineres.

Microsserviços

Temos um capítulo a respeito de microsserviços, mas sim, microsserviços também são uma vantagem para o negócio. Os microsserviços permitem que apenas os pedaços mais exigentes da nossa aplicação cresçam em número de réplicas atendendo as demandas de forma dinâmica, nos custando apenas o tempo em que foram ativados e utilizados, eliminando o custo quando forem desativados.

Separando a aplicação em pequenos serviços e utilizando contêineres, deixamos nossa aplicação otimizada para a esteira de produção, ou pipeline. A reconstrução exige apenas que o serviço em questão, ou os serviços envolvidos sejam reconstruídos, ao contrário da aplicação por completo. Isso se reflete na quantidade de "deploys" que podemos fazer por dia.

Uma aplicação com pedaços facilmente atualizáveis! Isso é perfeito para as metodologias ágeis que focam em tarefas específicas e também para a regra de negócio que para acompanhar o mercado precisa se modificar com muito mais frequência, geralmente mais do que o planejado.

SLA e SLO

SLA significa Service Level Agreement, um acordo sobre o nível de serviço estabelecido entre duas partes, por exemplo, tempo de resposta a chamados e incidentes, latência da rede entre outras coisas, uma espécie de direitos e deveres. Já SLO mensura o que foi especificado na SLA.

Mas o que um contêiner pode fazer em relação a isso? Bom, um contêiner não mas um orquestrador de contêineres sim!

Para o SLA os orquestradores de contêineres são ferramentas capazes de fornecer "self healing", alta disponibilidade, tolerância a falhas e "auto scaling". A palavra "self healing" pode ser traduzida como autocura, e significa que o orquestrador sabe que a aplicação apresentou problemas em uma determinada máquina e agora provisionará uma nova réplica em outro lugar, ou mesmo, detectou que uma atualização causou problemas na aplicação e não espalhará essa atualização entre as demais réplicas. Já "auto scaling" significa que o orquestrador detectará que a aplicação não está dando conta das requisições e então uma nova réplica será criada automaticamente balanceando a carga entre as réplicas existentes.

Para o SLO os orquestradores já estão prontos para serem monitorados, fornecendo métricas que podem ser guardados de forma histórica e consultadas mais tarde para comparação ou outros cálculos preditivos.

É eu sei que parece muito bom para ser verdade, eu também duvidaria se estivesse aí do outro lado, mas provavelmente todos já conhecemos um orquestrador que está fazendo um barulho no mercado e foi adotado por praticamente todas as clouds, inclusive as menores, e o seu nome é Kubernetes. Não trabalharemos com orquestradores neste curso, falaremos um pouco mais a respeito mais adiante, mas caso esteja interessado, aqui na 4Linux temos cursos que abordam a utilização do Kubernetes e do Docker Swarm, um outro orquestrador bastante simples.

Números

Segundo a MarketsandMarkets o mercado global do ecossistema de contêineres girava em torno de 1.2 bilhões de dólares em 2018, com uma projeção de 4.98 bilhões de dólares em 2023, e seu crescimento pode ser atribuído a modernização em massa de aplicações críticas para o negócio, além de fintechs e outras pequenas e médias empresas estarem adotando esse

tipo de tecnologia. Os maiores nomes do mercado investindo em tecnologias desse tipo são IBM, AWS, Microsoft, Google e VMWare.

Uma outra pesquisa focada em software gerenciadores de contêineres da Gartner aponta que a receita crescerá de 465.8 milhões de dólares em 2020 para 944 milhões em 2024, e ainda citam que esta é a primeira pesquisa da Gartner a respeito de contêineres em resposta ao crescimento da importância desta tecnologia. A pesquisa ainda aponta que até 2022 75% das organizações ao redor do mundo rodarão aplicações conteinerizadas em produção, e em 2024 15% das aplicações consideradas como "enterprise". A pesquisa ainda cita que contêineres podem ser o combustível para um ecossistema aberto, similar ao que acontece com Linux.

Um ecossistema aberto significa menor custo com licenças de software e liberdade para trocar de fornecedor, conforme citamos no tópico a respeito dos padrões da Open Container Initiative.

Ainda segundo um case da própria VMWare, um banco nacional da América do Norte que adotou tecnologia de contêineres teve seus tempos de entrega de aplicações reduzidos de 12 meses para algo entre 3 e 6 meses.

A Forrester fez um relatório a respeito do impacto econômico da adoção das tecnologias da IBM e RedHat, e como resumo os clientes chegaram a economizar até 50% no custo com infraestrutura, conseguiram realocar de 30% a 90% da mão de obra de administração de infraestrutura, aceleraram até 66% o ciclo de desenvolvimento e chegaram a aumentar a frequência dos lançamentos das versões em até 10x. Este não é um estudo somente sobre contêineres mas é possível notar no relatório que a utilização do Kubernetes e Openshift (que também é baseado em Kubernetes) é citada em praticamente todos pontos analisados.

E para finalizar eu ainda lhe digo que a utilização de contêineres aumenta o seu ROI (Return of Investment), ou retorno de investimento, e reduzem o TCO (Total Cost of Ownership), ou o custo total de propriedade em comparação as máquinas virtuais. As máquinas virtuais geralmente são utilizadas para garantir "multitenancy", que é a capacidade de várias pessoas ou equipes utilizarem um ambiente compartilhado sem prejudicarem umas as outras, e facilitar sua movimentação entre servidores. Para permitir escalabilidade, as máquinas virtuais geralmente são alocadas em máquinas físicas com um pouco mais de recursos do que o necessário, esta sobra é repassada para o cliente final ou inutilizada na nossa infraestrutura "on premise". Em contrapartida os contêineres também permitem "multitenancy", são muito mais fáceis de mover e possuem uma densidade muito maior do que as máquinas virtuais, significando que podemos colocar mais aplicações no mesmo hardware. Ambos são escaláveis, mas os contêineres são muito mais rápidos, mais leves e mais dinâmicos.

Orquestradores

Chegamos então ao último tópico do primeiro módulo, orquestradores.

Citamos a palavra "orquestradores" mais de uma vez e está na hora de dizer o que é um orquestrador. Segundo o Dicionário Priberam da Língua Portuguesa, orquestrador é aquele que orquestra, e orquestrar é compor as diferentes partes de uma peça de música ou mesmo elaborar um plano. Ambas as definições servem muito bem para nós, podemos imaginar os diferentes contêineres como cada categoria de instrumento em uma música e o trabalho minucioso de colocar tudo isso em harmonia, ou mesmo ou a execução de um plano para que tudo funcione como deveria.

Assim são os orquestradores de contêineres, ferramentas capazes de gerenciar e monitorar os contêineres com a capacidade de movê-los de um lugar para o outro caso algo errado aconteça, balancear a carga entre as réplicas existentes, fornecer uma espécie de "auto discovery" (descoberta automática) para novos serviços e muitas outras coisas a mais. O orquestrador tem a capacidade de atualizar a sua aplicação de forma cíclica, interrompendo a atualização das outras réplicas caso uma das unidades atualizadas apresente problemas.

O principal orquestrador do mercado chama-se Kubernetes, desenvolvido pela Google e hoje um projeto aberto com milhares de contribuidores. Um segundo orquestrador, geralmente utilizado em projetos menores é o Swarm e ainda podemos citar o Nomad um orquestrador com propósitos mais genéricos da Hashicorp.

Resumo

Este capítulo foi exclusivamente voltado para análise de mercado, do ponto de vista daqueles que procuram um novo emprego, trocar de área ou se especializar, e também do ponto de vista daqueles que gerenciam as empresas e as equipes, e tem dúvidas em relação ao rumo que as coisas vão tomar.

- Analisamos um panorama nacional sobre a adoção de contêineres pela visão da própria 4Linux além de um panorama global.
- Analisamos o porquê adotar contêineres do ponto de vista do negócio, ganhando economia e agilidade sem sacrificar a estabilidade.
- E ao final apresentamos uma introdução sobre orquestradores e algumas opções do mercado.

O próximo tema tem muito a ver com a forma de desenvolver e como estão ficando as infraestruturas baseadas em contêineres.

4

Microsserviços

Microsserviço é um tipo de arquitetura que organiza a aplicação em uma coleção de serviços fracamente acoplados. O que?!

Certo, vamos desacelerar, porque aposto que nem todos que estão acompanhando são programadores, certo?

Arquitetura Monolítica

Muitos dos sistemas desenvolvidos são peças grandes e únicas de software, todo o código é desenvolvido em uma única linguagem de programação. É como se todo o software fosse um grande pacote. Esse tipo de arquitetura é muito comum e quase sempre mais simples de conceber. A este tipo damos o nome de "molito", ou dizemos que a arquitetura é "monolítica".

O problema deste tipo de aplicação é justamente sua estrutura monolítica. A menor alteração, mesmo que insignificante, exige o provisionamento da aplicação por completo.

A solução então era separar a aplicação em pedaços isolados menores. Com pedaços menores poderíamos criar e espalhar réplicas das partes que mais exigiam das máquinas, dividindo a carga entre as réplicas, e também facilitaríamos o provisionamento de novas versões, já que ao atualizar um pedaço isolado, o restante da aplicação não sofreria.

Arquitetura Orientada a Serviços

Uma abordagem para separar as aplicações monolíticas em serviços independentes apareceu antes dos anos 2000, e cresceu bastante em 2003 quando uma forma de comunicação entre processos chamada SOAP (Simple Object Access Protocol) ganhou sua versão 1.2 especificada pela W3C (World Wide Web Consortium). De lá para cá, o SOA (Service-oriented architecture) tem sido utilizado para o desenvolvimento de grandes arquiteturas, principalmente em aplicações que tratam de quantidades massivas de requisições. Nesta época as aplicações (ou serviços) conversavam diretamente entre si e não havia um padrão de comunicação, o que dificultava alterações e novas implementações. Mais ou menos nesses anos surge a figura do ESB (Enterprise Service Bus), uma espécie de tradutor comum para solucionar o problema entre todos os serviços.

Com o ESB as aplicações passaram a conversar de forma padrão, o ESB se encarregava de enviar as mensagens de um lado para o outro.

Parece promissor, certo? Mas o ESB trouxe um problema, o próprio ESB, que passou a ser não somente um SPOF - Single Point of Failure - como também um possível gargalo entre as aplicações e suas mensagens. Existem diversas abordagens para evitar que isso aconteça, mas uma das abordagens, que acabou se tornando uma forma de implementação é justamente o que conhecemos como **microsserviços**.

Arquitetura de Microsserviços

Após analisar tudo isso que foi dito, percebemos então que os microsserviços são então uma implementação específica do SOA em que os serviços conversam diretamente entre si - sem a figura do ESB - através de um protocolo e formato de dados padrão. Se você pensou em APIs REST está acompanhando o raciocínio!

Se os serviços podem se comunicar entre si, são mais tolerantes a falhas, pois podem haver várias réplicas de um mesmo serviço. Se a comunicação é um formato padrão, os serviços podem ser desenvolvidos em tecnologias diferentes. Cada pedaço pode ser provisionado individualmente, isoladamente, sem afetar a aplicação como um todo.

Os contêineres facilitaram drasticamente a adoção deste tipo de arquitetura, e os motivos eu espero que vocês consigam perceber com o passar do curso.

Resumo

Este capítulo abordou especificamente o tema "microsserviços", voltamos um pouquinho no tempo para poder remover todas as dúvidas atuais e descobrimos que nada mais são do que uma implementação específica do SOA, com forte utilização de APIs REST para a comunicação entre as partes.

As discussões sobre o que é ou não um microsserviço vão além do escopo deste curso, e algumas implementações mais extremas parecem ser inviáveis devido ao excesso de requisitos e extrema complexidade. Independente disso, adotar algumas das suas práticas podem facilitar o desenvolvimento em alguns casos, principalmente em novos projetos.

5

O Sysadmin, o Programador e o DBA

O título deste capítulo daria um bom nome para um livro de reflexões, mas a intenção é responder duas perguntas bastante comuns:

- Sou desenvolvedor, preciso aprender Docker? Eu não quero mexer com infraestrutura.
- Sou sysadmin, preciso aprender Docker? Eu não quero programar.
- Sou DBA, preciso aprender Docker? Ouvi dizer que não se roda bancos em contêineres.

A resposta para as três perguntas é: Sim! Todos precisam aprender, não necessariamente Docker, mas o que são contêineres, como criá-los e testá-los. O ideal é ambos saibam o que o outro sabe, pelo menos um pouco.

Existem, entretanto, algumas diferenças de atuação e conhecimento necessário entre os tipos de profissionais.

Sysadmin

Não importa se você se considera um SRE, trabalha com Middleware ou tem DevOps no nome do seu cargo, quando as máquinas deixam de responder por `SYN Flood` ou a aplicação para por conta do `OMM Killer` no final do dia somos todos sysadmins tentando resolver problemas.

O papel do sysadmin, ou o time de operações, é trabalhar com os orquestradores, como Kubernetes ou Swarm, e também trabalhar com as máquinas que possuem contêineres soltos,

que apesar de não ser o cenário ideal, é bastante comum. Isso significa que saber criar as imagens não é essencial, essencial é conseguir levantar ambientes de testes, rodar contêineres locais ou em um orquestrador, entender como estes contêineres se comunicam e como os dados são persistidos em volumes.

Além do mais, é papel dos administradores de sistema aplicar alguma forma de observabilidade nas aplicações para poder auxiliar o time de desenvolvimento em relação a melhores práticas relacionadas a comunicação, limites de memória/CPU além de possíveis alternativas para contornar instabilidades.

Programador

Não importa se você se considera um Arquiteto, se trabalha com diagramas e modelagem, ou faz os testes, quando a aplicação parar de funcionar alguém chegará até você nos horários em que menos gostaria, e então seremos todos programadores tentando corrigir um bug.

O papel do programador, ou do time de desenvolvimento, é criar imagens e testá-las. Isso inclui praticamente todos os comandos e possibilidades disponíveis no Docker. A maioria dos testes mais básicos acontecem localmente, sendo assim é essencial criar imagens, levantar ambientes de testes e rodar contêineres locais. Também é preciso entender a respeito de sessões e volumes distribuídos, bem como configurações externas a aplicação e variáveis de ambiente. Um pouquinho de conhecimento em orquestradores fará bem, cedo ou tarde será necessário.

DBA

DBA? Mas eu achei que você estava brincando!

Sim, os DBAs também estão fadados a trabalhar com contêineres, apesar de ser um pouco menos comum, nos últimos anos as comunidades do MySQL, MongoDB e PostgreSQL criaram formas de automatizar e gerenciar os bancos dentro do Kubernetes através do que chamamos de Operators, controlando-os de uma maneira mais inteligente e facilitando tarefas como backups e replicação.

Então um DBA precisa entender como levantar ambientes de testes, rodar contêineres locais e aprender o funcionamento dos orquestradores e estas novas possibilidades de administração.

Resumo

Independente da função, as obrigações se cruzam, por esse motivo o DevOps ganhou força, afinal as equipes agora tem mais áreas em comum do que antes.

Para os três casos, uma coisa é certa, precisam aprender como funcionam os contêineres e como criá-los. É isso que começaremos a estudar nos próximos capítulos.

6

Docker - Teoria

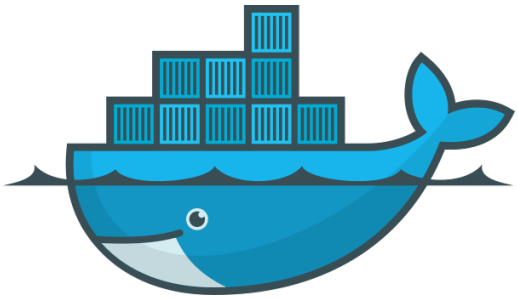


Fig. 6.1: Docker

Neste segundo capítulo vamos conhecer a ferramenta simbolizada por essa baleia azul sorridente, o Docker. Veremos um pouco a respeito de sua história e superficialmente como funcionam os isolamentos dentro do contêiner e as imagens que criam estes contêineres e suas camadas.

O que é

Docker é uma plataforma de código aberto, escrita em Go, que utiliza virtualização em nível do sistema operacional para construir, provisionar e gerenciar aplicações em um formato conhecido como contêineres. De forma simplista, Docker é uma ferramenta para criar contêineres.

Foi desenvolvido em 2013 por uma empresa chamada **dotCloud**, hoje conhecida como **Docker Inc.**.

A adoção do Docker no mercado foi meteórica, apesar de já existirem outras tecnologias

similares como o **LXC**, que na época fazia parte de seu núcleo, o Docker uniu a facilidade em manipular contêineres, que já existia de certa forma na época, com a simplicidade de criar, transferir, provisionar e compartilhar imagens. No mesmo ano de lançamento a RedHat anunciou a colaboração com o projeto e no ano seguinte a Microsoft anunciou a integração do Docker engine no Windows Server. Uma análise do LinkedIn mostra que as menções sobre Docker cresceram 160% em 2016.

Segundo o próprio site da Docker Inc. o Docker foi feito por desenvolvedores para desenvolvedores, então podemos chegar a conclusão de que o Docker é uma ferramenta de desenvolvimento! Mas isso é assunto para outros cursos...

Como Funciona

Conforme foi dito nos capítulos anteriores o Docker utiliza de algumas tecnologias presentes no kernel do Linux como namespaces, cgroups e union filesystems para criar contêineres através de imagens preexistentes.

Essas imagens são imutáveis, podem possuir diversas camadas e essas camadas podem ser compartilhadas entre si. Através de uma única imagem é possível criar um número qualquer de contêineres. Dentro destas imagens há um pequeno sistema operacional que compartilha o kernel com a máquina hospedeira, além da aplicação, binários, bibliotecas e qualquer outro tipo de arquivo especificado durante a criação desta imagem.

Quando um contêiner é criado, internamente seus processos ficam isolados do sistema operacional através dos namespaces, isso significa que os processos dos contêineres podem ser vistos da máquina hospedeira, mas os processos da máquina hospedeira não podem ser vistos pelo contêiner. Além do mais, o contêiner "acha" que seu processo é o principal, ou seja, para o contêiner, seu processo principal é o PID número 1. Caso esse processo principal de PID 1 venha a terminar o contêiner encerra-se automaticamente.

Contêiner

```
/usr/local/apache2 # ps -ef
PID   USER     TIME  COMMAND
  1 root      0:00 httpd -DFOREGROUND
  7 daemon    0:00 httpd -DFOREGROUND
  8 daemon    0:00 httpd -DFOREGROUND
  9 daemon    0:00 httpd -DFOREGROUND
 91 root      0:00 sh
 98 root      0:00 ps -ef
```

Máquina Hospedeira

```
hector@red-ryzen:~> ps -ef | grep -E 'httpd|\ssh|PID'
UID        PID TIME CMD
root     10030 0:00 httpd -DFOREGROUND
daemon   10079 0:00 httpd -DFOREGROUND
daemon   10080 0:00 httpd -DFOREGROUND
daemon   10081 0:00 httpd -DFOREGROUND
root     10213 0:00 sh
hector   13186 0:00 grep --color=auto -E httpd|PID
```

Fig. 6.2: Processos Isolados

O mesmo funciona para a interface de rede, por padrão um contêiner criado pelo Docker acredita que é o "localhost" e possui sua própria interface de rede física. Mas esta interface

pode ser vista da máquina hospedeira como uma interface virtual começando com o prefixo **veth**.

Contêiner

```
root@d27ee074fc69:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Máquina Hospedeira

```
hector@red-ryzen:~> ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: wlp28s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether c0:25:e9:2c:f6:37 brd ff:ff:ff:ff:ff:ff
    inet 192.168.15.191/24 brd 192.168.15.255 scope global wlp28s0
       valid_lft forever preferred_lft forever
9: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:7f:7a:ab:29 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
15: vethd29de13@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    link/ether 0a:d7:13:7a:76:2a brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Fig. 6.3: Rede Isolada

O isolamento também acontece no sistema de arquivos do contêiner que por padrão é baseado somente em sua imagem. A raiz de seu disco é a raiz da imagem, os processos não conseguem chegar até a raiz do sistema de arquivos da máquina hospedeira, tão pouco "sabem" que exista algo além da imagem. Dissemos que essas imagens são imutáveis, mas ao iniciar um contêiner, uma pequena camada acima da imagem, conhecida como "writable layer", é criada permitindo que alterações do tipo COW (Copy on Write), ou escreva enquanto copia, modifiquem sua própria visão do sistema de arquivos. É preciso atentar-se ao fato de que a "writable layer" é temporária, caso o contêiner seja removido, os dados desta camada também serão.

As imagens por sua vez podem ter uma ou mais camadas, mas, independente disto, para a aplicação todas essas camadas mais a "writable layer" é uma coisa só, completamente transparente, inclusive para nós. O nome disso é Union Mount, mas também podemos encontrar como "union filesystem".

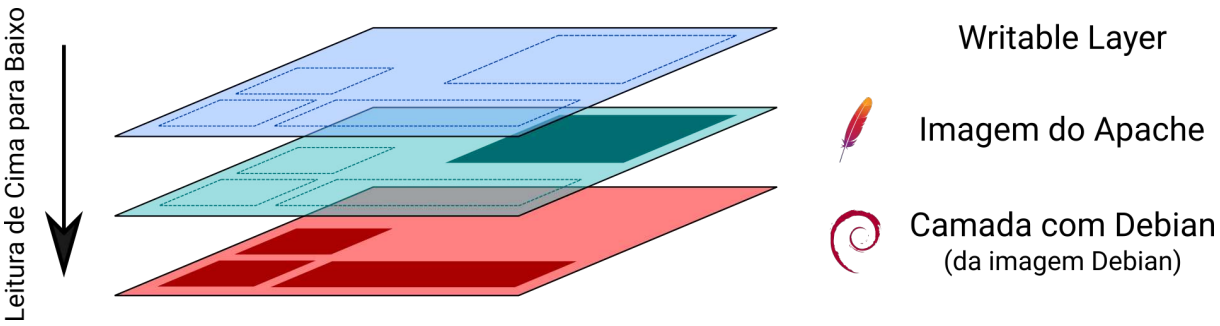


Fig. 6.4: Camadas Unidas

Na imagem a cima fica mais fácil de entender como as camadas se relacionam entre si. Imagine cada camada como um vidro translúcido capaz de exibir o que está embaixo desde que nada tenha sido escrito por cima. Se tivéssemos uma caneta, o único vidro que poderíamos escrever seria o superior, ou seja, a "writable layer". A vantagem está no fato de que através de uma única imagem, podemos criar um número indeterminado de contêineres. Cada contêiner seria inicialmente uma cópia fiel da imagem e ao fazer alterações em seu sistema de arquivos essas alterações ficariam todas na "writable layer".

As imagens de contêineres que utilizaremos durante o curso são as imagens presentes no repositório público da própria Docker, o `https://hub.docker.com/` ou Docker Hub. Um repositório de imagens é conhecido como **registry**

Acho que isso é suficiente como uma visão superficial de como os contêineres funcionam. Podemos dizer então que um contêiner é igual a um humano: acredita ser peça principal e conhece pouco sobre o todo a sua volta.

Resumo

Neste capítulo conhecemos o Docker e os principais aspectos teóricos a respeito de seus contêineres, como o isolamento entre os processos, a rede e o sistema de arquivos dos contêineres que por sua vez é baseado em uma imagem somente leitura, muito parecido com um disco ótico.

Também abordamos um pouco sobre como as camadas das imagens funcionam e que no "topo" destas camadas fica uma camada especial conhecida como "writable layer". Essas camadas são anexadas umas as outras e o resultado final é um único sistema de arquivos transparente para as aplicações dentro do contêiner e também para nós.

7

Docker - Primeiros Passos

Daqui em diante começa a parte prática. Conheceremos os comandos mais básicos do Docker, mas peço que tenha calma, inicialmente começaremos devagar e somente no próximo capítulo que realmente veremos suas grandes vantagens.

Prepare seus dedos e jogue o seu mouse fora pois vamos utilizar, e muito, o terminal, ou como já ouvi algumas pessoas dizerem: a tela preta.

Ah sim! Não se preocupe se você está no Linux, no Windows ou no Mac, só precisaremos de um navegador!

Preparando o Ambiente

Tudo o que você precisa para continuar a parte prática deste curso é de acesso a internet - e provavelmente você já tem, pois está acompanhando o curso - seu navegador e uma conta em `https://hub.docker.com/`.

Com a conta ativada, utilizaremos uma simples, mas fantástica, ferramenta que pode ser acessada em `https://labs.play-with-docker.com/`:

Alguns pontos a observar:

- Pode ser que seu navegador tenha problemas para acessar a interface, sendo assim teste com outros navegadores.
- Caso o site esteja fora do ar, você pode tentar utilizar o playground do Katacoda.

- Caso tenha Linux ou máquinas virtuais com Linux, você pode utilizá-las no lugar do "play with docker".

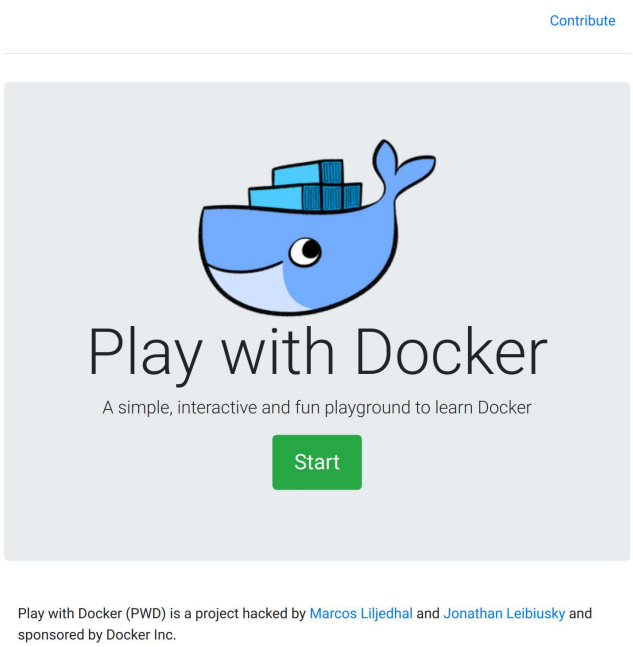


Fig. 7.1: Play With Docker

Faça o login clicando no botão central e então clique em **Start**.

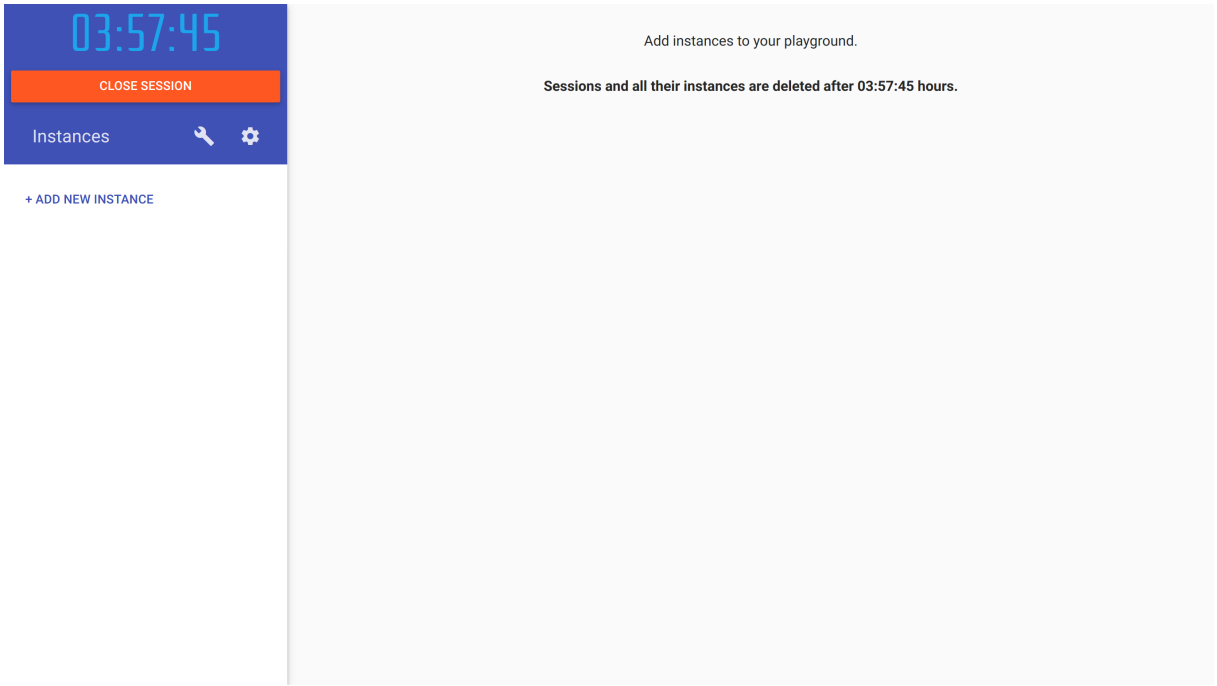


Fig. 7.2: Play With Docker - Interface

Esta será nossa bancada de trabalho! Tudo o que precisamos fazer é clicar em "+ ADD NEW INSTANCE" e pronto! Um terminal do Linux prontinho!

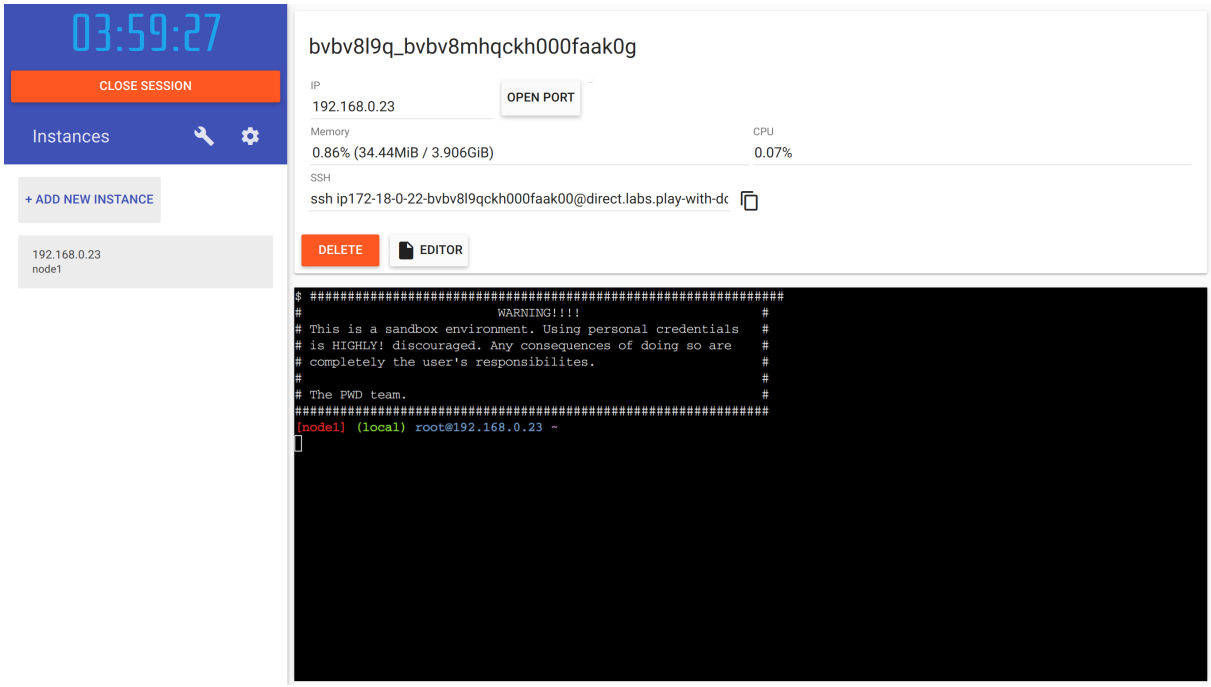


Fig. 7.3: Play With Docker - Instância

Cada sessão do "Play With Docker" dura 4 horas, sendo assim, caso este tempo expire, você precisará criar outra sessão nova. Por conta disso, tome cuidado com os arquivos que criar e caso queira salve os comandos do histórico em um arquivo separado.

Estamos prontos, de agora em diante ficaremos nessa "tela preta" a maior parte do tempo, você pode jogar até o seu mouse fora se quiser.

Sobre o Terminal

A primeira vista o terminal parece uma ferramenta arcaica, mas eu imagino que você já tenha parado para pensar: Porque através dos anos o terminal nunca deixou de ser usado apesar da interface gráfica ficar cada vez melhor?

A resposta é: facilidade.

É mais fácil desenvolver uma aplicação que responda aos comandos e parâmetros do terminal, é mais rápido executar e configurar aplicações que rodam no terminal, desde que conheça os comandos e parâmetros. Mesmo que não conheça, é possível acessá-los através dos parâmetros de ajuda, como por exemplo:

```
ls --help
```

Testando o Docker

Todos os comandos do Docker são executados pelo comando `docker`, isso é bastante sugestivo. Veja que é possível obter informações a respeito da versão do Docker através do seguinte comando:

```
docker --version
```

A saída deverá ser semelhante a seguinte:

```
Docker version 20.10.0, build 7287ab3
```

Primeiro Container

Nada melhor do que testar o docker iniciando um contêiner. Para estes casos existe uma imagem de contêiner feita especialmente para isso, essa imagem chama-se **hello-world**.

O comando `docker` possui algumas formas abreviadas de executar alguns comandos, mas neste curso utilizaremos apenas as formas longas pois são mais fáceis de entender apensar observando o comando. Por exemplo, para executar ações relacionadas aos contêineres executamos o comando `docker container`, já para gerenciar imagens utilizamos o comando `docker image`.

O subcomando `run` inicia um contêiner, e seu argumento é justamente o nome da imagem que utilizaremos. Não precisa se preocupar com as imagens por enquanto, o comando `run` cuidará disso para nós. Vamos iniciar o nosso primeiro contêiner:

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

A saída deste comando é bastante interessante e informativa:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:1a523af650137b8accdaed439c17d684df61ee4d74feac151b5b337bd29e7eec
Status: Downloaded newer image for hello-world:latest

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.
    (amd64)
 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 ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

A tradução explicará melhor do que qualquer trecho que possamos escrever, comentários estarão seguidos do símbolo "->":

```
Não foi possível encontrar a imagem 'hello-world:latest' localmente
latest: Baixando de library/hello-world
0e03bdcc26d7: Download completo
Digest: sha256:1a523af650137b8accdaed439c17d684df61ee4d74feac151b5b337bd29e7eec
Status: Baixado uma nova imagem do docker para hello-world:latest

Olá do Docker!
Essa mensagem mostra que sua instalação parece funcionar corretamente.

Para gerar essa mensagem o Docker executou os seguintes passos:
 1. O Docker client entrou em contato com o Docker daemon. -> (processo do Linux)
 2. O Docker Daemon baixou a imagem "hello-world" do Docker Hub. -> (repositório)
    (amd64) -> arquitetura de processador dos binários da imagem
 3. O Docker daemon criou um novo container a partir daquela imagem que roda
    um executável que produz a saída que você está lendo agora.
 4. O Docker daemon enviou a saída para o cliente do Docker, que a enviou
    para o seu terminal.

Tente algo mais ambicioso, você pode rodar um contêiner do Ubuntu com:
 $ docker run -it ubuntu bash

Compartilhe imagens, automatize rotinas, e mais com um Docker ID gratuito:
 https://hub.docker.com/

Para mais exemplos e ideias, visite:
 https://docs.docker.com/get-started/
```

Como eu disse, a imagem foi baixada "automágicamente". O contêiner agiu como um humano observado pela ótica do tempo de existência da terra, em instantes ele:

- Apareceu do nada;
- Fez o que foi programado para fazer e;
- Despareceu, morreu.

Mesmo este contêiner sendo extremamente efêmero e simples, todos os princípios foram váli-

dos:

- Foi baseado em uma imagem;
- Possuía uma "writable layer";
- Era isolado do restante do sistema;

Estes princípios se aplicam a qualquer imagem, independente da aplicação que foi executada aquilo que explicarmos para este contêiner servirá para todos os demais, sendo assim, vamos manter as coisas da forma mais simples possível.

Sabemos que o contêiner veio de uma imagem que foi baixada do Docker Hub, mais precisamente esta aqui. Mas para onde ele foi? E, porque ele se foi?

Estas perguntas serão respondidas no próximo tópico.

Listando os Contêineres

Para onde ele foi?

Os contêineres podem ser visualizados com o comando `ls`:

```
1 docker container ls
```

Porém, como nosso contêiner morreu, ou parou, precisamos especificar que queremos visualizar todos os contêineres, inclusive os parados, e fazemos isso através do parâmetro `--all`:

```
1 docker container ls --all
```

A saída será semelhante a seguinte:

```
1 CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS
      PORTS     NAMES
2 22a6c5e33966   hello-world   "/hello"   19 minutes ago   Exited (0) 19 minutes ago
                   gifted_benz
```

Todo contêiner do Docker:

- Possui um ID aleatório;
- É baseado em uma imagem;
- Tem um único comando principal;
- Tem um nome que quando não especificado é a junção de um adjetivo com o nome de um cientista, ambos aleatórios.

Os demais campos como CREATED e STATUS indicam quando o contêiner foi criado e qual seu estado atual, respectivamente. Já o campo PORTS indica quais portas este contêiner está expondo e/ou escutando.

Agora podemos responder a segunda pergunta:

Porque ele se foi?

O campo COMMAND é talvez o mais interessante, este campo indica qual é o comando principal do contêiner, que quando finalizado faz com que o contêiner termine. O comando neste caso é apenas um executável que gera uma saída de texto, aquela que vimos no terminal. Independente do comando, um simples script, um shell, um gerenciador de banco de dados, um webserver ou o Firefox, quando este comando terminar o contêiner é "desligado", logo, para que o contêiner fique "em pé" é preciso que o comando principal seja duradouro.

O ideal é que cada contêiner tenha uma única função, ou comando, se preferir. Quando esta função acaba o contêiner morre. Podemos dizer que, talvez assim como o ser humano, o contêiner tem um objetivo de vida e quando terminado ele está livre para deixar este mundo, ou a memória neste caso.

Criando a Primeira Aplicação

Os contêineres facilitaram o provisionamento de aplicações por causa das imagens e de seu isolamento. Seja lá onde criaram estas imagens, elas devem funcionar neste nosso ambiente. Nada melhor do que rodar um contêiner de uma aplicação conhecida para testar essa afirmação.

Vamos criar um contêiner que consiga ficar em pé, diferente do **hello-world**, e para isso vamos procurar por apache no Docker Hub. Procurou?! Certo, apareceram várias, mas sabemos que a imagem do webserver chama-se **httpd**.

Antes de prosseguir, uma nota:

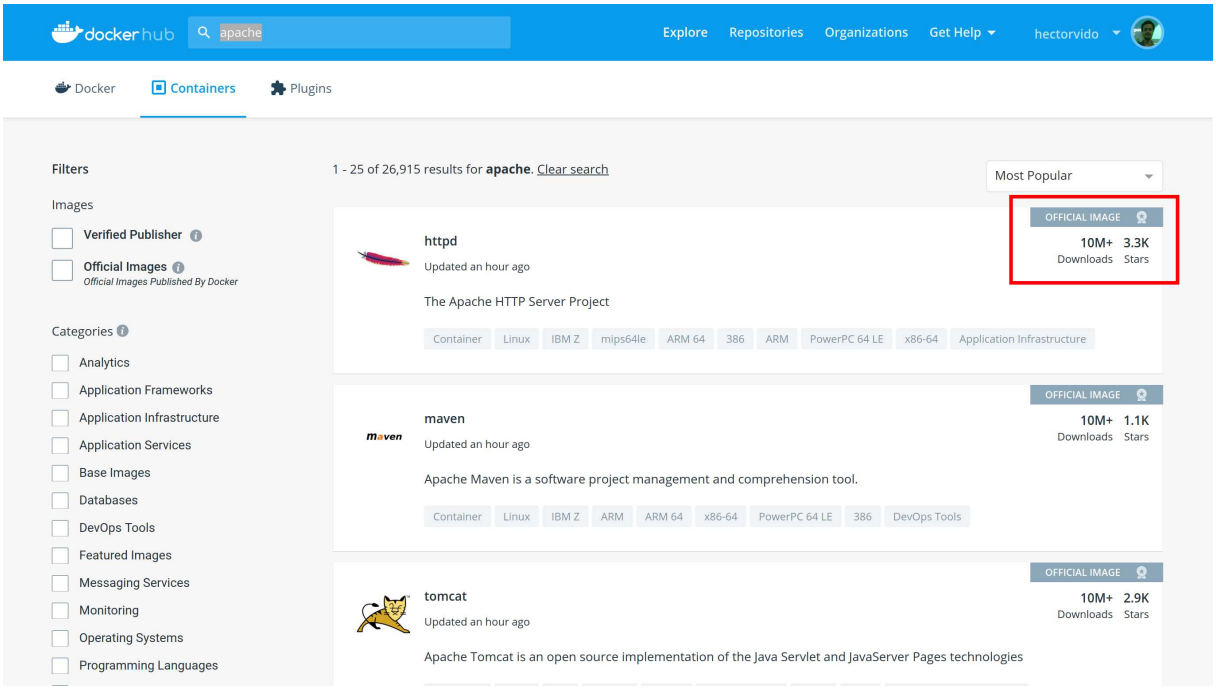


Fig. 7.4: Docker Hub - Imagens Confiáveis

Sempre procure as imagens que tenham o selo de imagens oficiais, pois estas são mais confiáveis. Caso não encontre utilize as imagens com maior quantidade de estrelas e downloads. Lembre-se que o Docker Hub é um repositório privado e pode-se encontrar de tudo lá, de tudo mesmo.

O comando `container run` é talvez aquele que tenha a maior quantidade de parâmetros e para este contêiner em específico vamos presenteá-lo com um nome, o chamaremos de **apache** e para isso utilizaremos o parâmetro `--name`:

```
1 | docker container run --name apache httpd
```

> Veremos muito mais parâmetros ainda, mas lembre-se que, como via de regra, o último parâmetro é o nome da imagem.

Nossa, agora ele baixou o que pareceu serem várias camadas! E é isso mesmo, as camadas estão aparecendo, mas não precisamos nos preocupar com isso agora, nem nunca praticamente.

Uma coisa curiosa aconteceu, o terminal parou. Parece que travou, mas não travou, estamos "presos" ao processo principal do contêiner, que no final das contas é o próprio processo do webserver apache, isso que estamos vendo no terminal é a saída de logs, ou seja informações a respeito do próprio apache e dos acessos que está recebendo. Como ninguém está acessando, nada novo está aparecendo.

Para sair deste processo utilizamos a sequência `CTRL + C` no teclado, infelizmente isso enviará

um sinal ao processo do **apache** que por sua vez terminará, finalizando o contêiner, mas não há outra opção tão simples quanto esta, sendo assim, finalize-o.

Agora sabemos que nosso contêiner, mesmo parado, se chama **apache**, podemos listá-lo com `docker container ls -a`, o equivalente a `--all`. Sendo assim, vamos aprender um novo comando, o comando para iniciar contêineres parados `start`:

```
1 |docker container start apache
```

> **Observação:** Pode-se utilizar o nome ou o id do contêiner nos casos em que precisamos especificá-los.

Pronto, desta vez o contêiner não travou o nosso terminal, e podemos vê-lo rodando com o comando `docker container ls`.

Não é preciso dizer que para parar um contêiner em funcionamento podemos utilizar o subcomando `stop`, certo?

Interagindo com Contêineres

Os contêineres são sistemas completos, podemos interagir diretamente com os contêineres através de `stop` ou `start` e mais uma outra porção de comandos. Mas também é possível interagir com as aplicações dentro dos contêineres, inclusive acessá-los como se fossem máquinas virtuais. Vamos testar algumas destas possibilidades.

Já sabemos listar os contêineres e agora podemos removê-los para não deixar as coisas bagunçadas. O id do meu contêiner baseado na imagem **hello-world** é `22a6c5e33966`, sendo assim podemos removê-lo com o subcomando `rm`:

```
1 |docker container rm 22a6c5e33966
```

> **Observação:** Para remover um contêiner que esteja rodando é preciso especificar o parâmetro `--force`, ou `-f`, ex: `docker container rm --force apache`.

Lembra que dissemos que o Docker coloca cada contêiner em um rede virtual? Essa rede tem a máscara `172.17.0.0/16`, para quem não conhece essa expressão, isso significa que rede tem o prefixo `172.17.0` e vai do `172.17.0.2` até `172.17.0.254`, sendo que `172.17.0.1` é a própria máquina, ou seja, onde nosso terminal está rodando no momento.

Podemos extrair uma série de informações a respeito de um contêiner com o subcomando `inspect`, vamos analisar o **apache**:

```
docker container inspect apache
```

É muita informação!


```
[
    {
        "Id": "8654a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551",
        "Created": "2020-12-15T00:59:28.879674446Z",
        "Path": "httpd-foreground",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 27406,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-12-15T01:11:57.369819689Z",
            "FinishedAt": "2020-12-15T01:07:00.032059922Z"
        },
        "Image": "sha256:
             dd85cdbb99877b73f0de2053f225af590ab188d79469eebdb23ec2d26d0d10e8",
        "ResolvConfPath": "/var/lib/docker/containers/8654
             a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/8654
             a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551/hostname",
        "HostsPath": "/var/lib/docker/containers/8654
             a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551/hosts",
        "LogPath": "/var/lib/docker/containers/8654
             a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551/8654
             a4c08ba95ad9447b269d2496f198b4c57c673999eee5f1e7f11ba1660551-json.log",
        "Name": "/apache",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "docker-default",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
            "AutoRemove": false,
            "VolumeDriver": "",
            "VolumesFrom": null,
            "CapAdd": null,
            "CapDrop": null,
            "CgroupnsMode": "host",
            "Dns": [],
```

```
            "DnsOptions": [],
            "DnsSearch": [],
            "ExtraHosts": null,
            "GroupAdd": null,
            "IpcMode": "private",
            "Cgroup": "",
            "Links": null,
            "OomScoreAdj": 0,
            "PidMode": "",
            "Privileged": false,
            "PublishAllPorts": false,
            "ReadonlyRootfs": false,
            "SecurityOpt": null,
            "UTSMode": "",
            "UsernsMode": "",
            "ShmSize": 67108864,
            "Runtime": "runc",
            "ConsoleSize": [
                0,
                0
            ],
            "Isolation": "",
            "CpuShares": 0,
            "Memory": 0,
            "NanoCpus": 0,
            "CgroupParent": "",
            "BlkioWeight": 0,
            "BlkioWeightDevice": [],
            "BlkioDeviceReadBps": null,
            "BlkioDeviceWriteBps": null,
            "BlkioDeviceReadIOps": null,
            "BlkioDeviceWriteIOps": null,
            "CpuPeriod": 0,
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DeviceRequests": null,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
            "OomKillDisable": false,
            "PidsLimit": null,
            "Ulimits": null,
            "CpuCount": 0,
            "CpuPercent": 0,
            "IOMaximumIOps": 0,
            "IOMaximumBandwidth": 0,
            "MaskedPaths": [
                "/proc/asound",
                "/proc/acpi",
                "/proc/kcore",
                "/proc/keys",
                "/proc/latency_stats",
                "/proc/timer_list",
                "/proc/timer_stats",
                "/proc/sched_debug",
                "/proc/scsi",
                "/sys/firmware"
            ],
            "ReadonlyPaths": [
                "/proc/bus",
```

```
120                 "/proc/fs",
121                 "/proc/irq",
122                 "/proc/sys",
123                 "/proc/sysrq-trigger"
124             ]
125         },
126         "GraphDriver": {
127             "Data": {
128                 "LowerDir": "/var/lib/docker/overlay2/5235
                        dbf5b78e142c79e343a104c9764b00eeb7f6606501a00c2014a8a91d3a78-init/
                        diff:/var/lib/docker/overlay2/96886286
                        f4a8ed20833bb0b7235245f328110fd1e75af39f9b5fb27c312f318a/diff:/var/
                        lib/docker/overlay2/2
                        ea849052ad1a3bdbe775366f6da5906accee5195c6cdab4f02be4b4fcdad155/
                        diff:/var/lib/docker/overlay2/
                        c89eb4f5c4b201fd49974a9869ff8558bd69e9fd3f82315091f04d417375510b/
                        diff:/var/lib/docker/overlay2/
                        fff4e24ee0282c79e4b2a32fe06835bb6179ca2e555994fa5257a6c0c04a78ce/
                        diff:/var/lib/docker/overlay2/
                        b89d931ee445ee5f60d8588ea95595aa2853990e01990be77f6006f0ef3099b2/
                        diff",
129                 "MergedDir": "/var/lib/docker/overlay2/5235
                        dbf5b78e142c79e343a104c9764b00eeb7f6606501a00c2014a8a91d3a78/merged
                        ",
130                 "UpperDir": "/var/lib/docker/overlay2/5235
                        dbf5b78e142c79e343a104c9764b00eeb7f6606501a00c2014a8a91d3a78/diff",
131                 "WorkDir": "/var/lib/docker/overlay2/5235
                        dbf5b78e142c79e343a104c9764b00eeb7f6606501a00c2014a8a91d3a78/work"
132             },
133             "Name": "overlay2"
134         },
135         "Mounts": [],
136         "Config": {
137             "Hostname": "8654a4c08ba9",
138             "Domainname": "",
139             "User": "",
140             "AttachStdin": false,
141             "AttachStdout": true,
142             "AttachStderr": true,
143             "ExposedPorts": {
144                 "80/tcp": {}
145             },
146             "Tty": false,
147             "OpenStdin": false,
148             "StdinOnce": false,
149             "Env": [
150                 "PATH=/usr/local/apache2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
                        usr/bin:/sbin:/bin",
151                 "HTTPD_PREFIX=/usr/local/apache2",
152                 "HTTPD_VERSION=2.4.46",
153                 "HTTPD_SHA256=740
                        eddf6e1c641992b22359cabc66e6325868c3c5e2e3f98faf349b61ecf41ea",
154                 "HTTPD_PATCHES="
155             ],
156             "Cmd": [
157                 "httpd-foreground"
158             ],
159             "Image": "httpd",
160             "Volumes": null,
161             "WorkingDir": "/usr/local/apache2",
162             "Entrypoint": null,
163             "OnBuild": null,
164             "Labels": {},
165             "StopSignal": "SIGWINCH"
166         },
167         "NetworkSettings": {
168             "Bridge": "",
```

```
            "SandboxID": "1328
                 ab4a7e13e14689977076870ea209126a78a85041788e597b29c346f09fec",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "80/tcp": null
            },
            "SandboxKey": "/var/run/docker/netns/1328ab4a7e13",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID": "1
                 aa19c3caa272109209069e1f48262b86395c6140c04973446e4a93e8b561e93",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "74320
                         eb65b01c04385bd81241ca05d546bbd560745cd52f607dff16e38228736",
                    "EndpointID": "1
                         aa19c3caa272109209069e1f48262b86395c6140c04973446e4a93e8b561e93
                         ",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

Existem muitos detalhes nessa saída, mas vamos focar em um dos pontos interessantes: o endereço IP do contêiner. Veja que a última sessão diz respeito a "rede" e podemos ver que o "IPAddress" de nosso contêiner é `172.17.0.2`.

Vamos acessar esse endereço através do terminal com o comando `curl`, capaz de fazer chamadas HTTP como se fosse um navegador:

```
curl 172.17.0.2
```

```
<html><body><h1>It works!</h1></body></html>
```

O retorno é uma página em HTML, não é nenhuma página glamorosa mas é a resposta do processo do apache dentro do nosso contêiner, está funcionando! Criamos um webserver com

único comando!

Vamos entrar no contêiner, para isso utilizaremos um comando novo, o `exec` e precisaremos de dois parâmetros, o `--tty` e o `--interactive`. O subcomando `exec` executa um comando dentro do contêiner, se executarmos um shell como `bash`,`sh` ou `ksh` podemos "entrar" dentro do contêiner, porém precisamos informar ao Docker que queremos "conectar" nosso input (nosso teclado) ao input do shell do contêiner com `--interactive`, além também de criar um "pseudo terminal" para que tudo isso funcione com `--tty`. O último parâmetro é o comando que queremos executar, neste caso o `bash`:

```
1 docker container exec --tty --interactive apache bash
```

As "letras" do nosso terminal mudarão, e ficarão parecidas com o seguinte:

```
1 root@8654a4c08ba9:/usr/local/apache2#
```

Estamos dentro do contêiner! Aqui de dentro podemos executar comandos do sistema em questão, que é um Debian, teste:

```
1 apt-get update          # atualiza lista de pacotes
2 apt-get install -y nano # instala o editor de texto nano
3 dpkg -l | grep glib     # lista todos os pacotes filtrando por "libc"
```

Podemos listar aquele arquivo HTML que visualizamos ao utilizar o `curl` e inclusive alterar o seu conteúdo:

```
1 ls htdocs                         # lista os arquivos
2 echo '4Linux' > htdocs/index.html # substitui o conteúdo por 4Linux
```

Saia do contêiner digitando `exit` e teste novamente o webserver com `curl`:

```
1 curl 172.17.0.2
```

O retorno agora será:

```
1 4Linux
```

Este sistema que roda no terminal do navegador é um Linux chamado Alpine, se tentar rodar os comandos que rodamos no Debian todos falharão pois o Alpine não possui os mesmos binários que o Debian, isso só foi possível pois estávamos dentro do contêiner.

Isso demonstra que um contêiner é capaz de "rodar" uma outra distribuição Linux (ou um

outro sistema) dentro de um sistema, muito parecido com uma máquina virtual. Neste caso estamos utilizando um contêiner com apache e Debian mas poderia ser qualquer aplicação em qualquer distribuição Linux, inclusive versões bem mais antigas.

Caso de uso - Aplicações Legadas

Algumas aplicações legadas utilizam um antigo servidor Debian 8 com o MySQL 5.5 como banco de dados. Por questões de segurança estão atualizando todas as máquinas da infraestrutura para Debian 10, o problema é que o Debian 10 suporta oficialmente o MariaDB 10.3 e o MySQL $>=$ 5.6 através da Oracle. Não houve tempo para testar como as aplicações se comportariam com uma versão nova do MySQL e para agravar a situação a aplicação é crítica para o negócio. A solução é utilizar um contêiner baseado em Debian 8 mantendo a versão do MySQL 5.5 dentro da máquina com Debian 10.

Imagens e Tags

Imagens são como um disco de um sistema somente leitura nos os quais os contêineres se baseiam para criarem seus próprios sistemas de arquivo, e as tags são variações existentes deste disco, ou da aplicação dentro do disco, como por exemplo uma versão anterior do sistema ou mesmo um outro sistema. Esta separação parece estranha, mas logo veremos como faz todo o sentido.

Vamos rodar um contêiner baseado em **alpine**, mas ao invés de utilizarmos o comando `run` diretamente, vamos antes apenas baixar a imagem com o subcomando `image` e o parâmetro `pull`:

```
1 |docker image pull alpine
```

Baixe também a imagem do Debian:

```
1 |docker image pull debian
```

Este comando apenas baixa a imagem, e a deixa pronta para ser utilizada, isto é útil para analisá-la antes de criar um contêiner. Podemos visualizar as imagens através do parâmetro `ls`:

```
1 |docker image ls
```

```
1 |REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
```

```
2 httpd         latest    dd85cdbb9987   3 days ago      138MB
3 debian        latest    6d6b00c22231   4 days ago      114MB
4 alpine        latest    b14afc6dfb98   4 days ago      5.57MB
5 hello-world   latest    bf756fb1ae65   11 months ago   13.3kB
```

Veja que a menor imagem, de longe, é a **hello-world**. Já o Alpine em relação ao Debian possui praticamente 5% de seu tamanho! Para remover imagens utilizamos o subcomando `rm` seguido do nome da imagem:

```
1 docker image rm hello-world
```

> **Atenção:** Não é possível remover uma imagem caso um contêiner faça referência a ela, parado ou não.

Todas as imagens do Docker possuem um **tag**, uma tag é uma espécie de identificador extra e geralmente indica uma versão da imagem em questão ou até mesmo uma alternativa a "imagem principal". As imagens com tag ficam no formato `nome:tag`.

Quando nenhuma tag é especificada o Docker assume que a tag é **latest**. Como exemplo, podemos baixar uma imagem do Debian bem menor, com o mínimo possível instalado:

```
1 docker image pull debian:buster-slim
```

Talvez tenha acontecido um evento curioso, e digo talvez porque pode ser que não ocorra. Durante o download desta imagem pode ser que uma mensagem dizendo `6ec7b7d162b2: Already exists`, ou algo semelhante, apareça no terminal. Isso aconteceu porque a imagem **httpd** é baseada em `debian:buster-slim`, sendo assim já tínhamos baixado essa camada anteriormente, mesmo sem saber. Isso significa que podemos economizar transferência de dados utilizando camadas, o Docker saberá automaticamente se já temos uma camada ou não. Tudo o que o Docker fez foi "expor" aquela camada como uma imagem completa.

Veja agora com `docker image ls` que existem duas imagens do Debian, mas uma é menor do que a outra e possuem tags diferentes:

```
1 REPOSITORY   TAG           IMAGE ID       CREATED      SIZE
2 httpd        latest        dd85cdbb9987   3 days ago   138MB
3 debian       buster-slim   4a9cd57610d6   4 days ago   69.2MB
4 debian       latest        6d6b00c22231   4 days ago   114MB
5 alpine       latest        b14afc6dfb98   4 days ago   5.57MB
```

> **Observação:** As tags são completamente arbitrárias e variam drasticamente de imagem para imagem, sempre consulte o Docker Hub para saber quais tags foram criadas.

Existe uma imagem do **httpd** baseada em alpine, baixe-a e veja a diferença em relação as

demais:

```
1 docker image pull httpd:alpine
```

Liste as imagens com `docker images ls`:

```
1 REPOSITORY   TAG           IMAGE ID       CREATED      SIZE
2 httpd        alpine        7c3185ece933   3 days ago   55.5MB
3 httpd        latest        dd85cdbb9987   3 days ago   138MB
4 debian       buster-slim   4a9cd57610d6   4 days ago   69.2MB
5 debian       latest        6d6b00c22231   4 days ago   114MB
6 alpine       latest        b14afc6dfb98   4 days ago   5.57MB
```

A diferença é bem grande, 55MB contra 138MB!

Caso de uso - Custo de Tráfego

As transferências da cloud para a internet devem ser evitadas ao máximo a você possui um ambiente multicloud com Kubernetes que sempre força o download das imagens, mesmo que já existam, diminuir as imagens é uma ótima ideia. Imagens menores são melhores para transferir de um lado para o outro, principalmente quando estamos trabalhando com orquestradores como Kubernetes ou Docker Swarm e ferramentas de entrega e integração contínua como o Jenkins. Além do mais, quanto menor o "footprint" de uma aplicação menores os riscos de segurança, pois há menos software instalado que possa ser corrompido. Apesar do Alpine ser extremente pequeno e bastante utilizado algumas tarefas são relativamente difíceis de obter sucesso como por exemplo a compilação de algum software já que sua biblioteca de C é bastante diferente da padrão na maioria das distribuição GNU/Linux.

Imagens Base

Essa nomenclatura não existe oficialmente, mas entenda contêineres base como contêineres criados através de imagens base de sistemas como Debian, openSUSE e Alpine. Estes contêineres podem ser traiçoeiros para quem está começando mas muito úteis para realizar os mais diversos testes.

Eu disse que elas são traiçoeiras pois o seguindo comando não retorna absolutamente nada, e o contêiner aparece morto na listagem:

```
1 docker container run alpine
2 docker container ls
```

Lembra do "comando principal"? Quando uma imagem é criada, um comando principal é

definido. É daí que veio o comando que iniciou o webserver Apache, afinal nós não dissemos mais nada ao contêiner. Acontece que para os contêineres dessas imagens base o comando principal é um **shell** e para isso precisamos utilizar a mesma lógica do `exec` quando entramos dentro do contêiner. Só que agora, utilizamos os mesmos parâmetros no subcomando `run`:

```
1 |docker container run --tty --interactive alpine
```

Pronto, o contêiner foi criado e já caímos dentro dele. Se digitarmos `exit` mataremos o contêiner pois fecharemos o processo principal, que é o shell, para sair de um processo destes sem finalizá-lo podemos executar a sequência de "escape" do Docker: `CTRL + P + Q`.

As vezes não queremos iniciar e nos atachar ao processo principal de um contêiner, na verdade não queremos que isso aconteça na grande maioria das vezes, para isso podemos utilizar o parâmetro `--detach`:

```
1 |docker container run --detach --tty --interactive alpine
```

Apesar do comando ser exatamente o mesmo que o anterior, cada comando `run` inicia um novo contêiner. Isso significa que temos ao menos dois contêineres da imagem **alpine** rodando ao mesmo tempo.

> **Observação:** Os parâmetros `--detach`, `--tty` e `--interactive` podem ser substituídos por `-d`, `-t` e `-i` respectivamente. Frequentemente são dispostos todos juntos da seguinte forma: `-dti`. A ordem não tem relevância.

Caso de uso - Testes

É preciso testar uma aplicação em uma versão antiga do OpenSUSE, a versão 42. A opção mais comum seria baixar a ISO desta versão na internet instalar em uma máquina virtual, porém, com Docker é possível baixar essa imagem em segundos, testar tudo o que for necessário e depois descartar o contêiner. Um outro caso muito útil, que ainda serve no exemplo acima é a criação de imagens. Veremos como criar imagens mais adiante mas por vezes não saberemos todos os comandos e pacotes de cor e será preciso testá-los antes de escrevê-los no arquivo que define a imagem.

Resumo

Apesar de termos abordado apenas o essencial, são muitos detalhes.

Preparamos o nosso ambiente no Play With Docker e desde então trabalhamos exclusivamente

no terminal.

Criamos o primeiro contêiner com o comando `docker container run` e observamos seu funcionamento, a escrita de um texto na tela. Apesar de não termos baixado nenhuma imagem, o `run` cuidou disso para nós, e pudemos observar que ao terminar seu comando principal o contêiner morre. Descobrimos que para listar os contêineres utilizamos o comando `docker container ls` e navegamos pelo Docker Hub procurando por uma imagem do Apache que baixamos ao executar o `run`, isso nos deixou preso ao processo do apache o que nos fez aprender os comandos `docker container start` e seu antagonista, o `stop`.

Removemos contêineres parados com `docker container rm` e obtivemos muitas informações através do comando `docker container inspect`, dentre elas o endereço IP do contêiner do apache, que utilizamos para acessar o webserver com `curl`.

Interagimos com o "shell" do contêiner através do comando `docker container exec` pedindo para iniciar um `bash` lá dentro e para isso foi necessário utilizar dois parâmetros interessantes, o `--tty` e o `--interactive`, ou `-ti` para simplificar.

Vimos que uma mesma imagem pode ter outras variantes através das **tags**, baixamos estas imagens com o comando `docker image pull` e as listamos com `docker image ls`. Foi interessante descobrir que a imagem do apache, assim como muitas outras aplicações, possui uma versão baseada em **alpine** que é bem menor que a versão padrão de tag **latest**. Neste momento também pudemos ver que as vezes o Docker não baixa algumas imagens porque elas já existem em nosso sistema como camadas, ele simplesmente as deixa disponível, economizando tempo.

Por último vimos a respeito das "imagens base", imagens de distribuições Linux puras, muito úteis para teste. Estas imagens precisa dos parâmetros `-ti` no comando `docker container run` pois por padrão iniciam um shell, sem o `-t` o processo morreria instantaneamente e sem o `-i` o shell não interpretaria nenhum comando.

O último parâmetro visto foi o `--detach`, ou `-d`, que envia o contêiner que acabamos de criar para "background", desatachando nosso terminal do processo principal do contêiner.

8

Docker - Caminhando Sozinho

Agora chegamos no momento de nos aprofundar um pouco mais naquilo que podemos fazer com o Docker e começar a rodar alguns contêineres mais complexos e também facilitar a forma de alterar os dados dentro de seus sistemas de arquivos.

Volumes

Se os dados de um determinado diretório devem persistir independente do contêiner, por exemplo, os arquivos de dados de um banco de dados, ou imagens de um site, você precisa colocá-los em um volume.

Quando iniciamos um contêiner, temos a opção de atachar um volume, este volume pode ser um diretório da sua máquina, um NFS, um iSCSI ou mesmo um disco fornecido por uma cloud. Os volumes são a única forma de persistir os dados de um contêiner quando estes foram destruídos ou substituídos. Além do mais, na maioria das vezes é possível compartilhar um volume entre vários contêineres tornando a tarefa de distribuir a aplicação muito mais simples. Por exemplo, ao criar um site os arquivos do site poderiam ficar em um volume compartilhado entre vários contêineres que foram criados para atender a um número maior de conexões simultâneas. Outra vantagem está no caso em que a imagem do contêiner precisasse ser atualizada, os dados continuariam todos dentro do volume.

Nós trabalharemos apenas com os volumes locais, ou seja, nosso volume será um simples diretório na máquina.

Para esta parte continuaremos com o contêiner do **apache**. Sendo assim, remova-o:

```
1 |docker container rm -f apache
```

A imagem **httpd**, independente da sua variante, guarda os arquivos do webserver (conhecido como "document root") em `/usr/local/apache2/htdocs/`, podemos encontrar essa informação em algum lugar da documentação da imagem em httpd. A ideia é criar um volume com um site pronto para o contêiner nos apresentar.

Vamos "clonar" através do git um site em um diretório de nome `html`:

```
1 |git clone https://github.com/4linux/4542-site.git html
```

Este é um site exemplo utilizado em nosso curso a respeito de OpenShift, uma ferramenta capaz de gerenciar vários contêineres e o ciclo de vida das aplicações que desenvolvemos, se quiser saber mais a respeito, recomendo o nosso próprio curso!

Agora precisamos dizer ao docker para montar esse volume dentro do contêiner, e fazemos isso através do parâmetro `--volume`:

```
1 |docker container run --detach --name apache --volume /root/html:/usr/local/apache2/
  |     htdocs/ httpd:alpine
```

> **Observação:** Geralmente os volumes são abreviados como `-v`.

Quando utilizamos diretórios como volumes, precisamos especificar o diretório pelo seu caminho absoluto, ou seja, desde o início, por isso especificamos `/root/html`, o mesmo vale para o lugar em que o diretório será montado dentro do contêiner. Veja que utilizamos a imagem **httpd** de tag **alpine**.

Vamos descobrir o IP do contêiner inspecionando o contêiner apache e filtrando pelo trecho "IPAddress":

```
1 |docker container inspect apache | grep IPAddress
```

Neste caso, meu contêiner tem o ip `172.17.0.2`, vamos consultar o webserver com o `curl`:

```
1 |curl 172.17.0.2
```

Eita! Desta vez veio muito mais conteúdo do que o simples "It Works!" que vimos anteriormente, o site parece estar funcionando, mas testar as coisas dessa forma não é muito legal, e para isso vamos pedir para o Docker abrir uma porta na máquina e redirecionar todo o tráfego para o contêiner. Destrua o contêiner do apache:

```
1 docker container rm -f apache
```

Expondo o Contêiner

Para expor um contêiner, ou a porta de um contêiner, e tornar os nossos testes mais simples - e mais realistas - precisamos especificar um parâmetro extra para o docker, e o parâmetro que expõe uma porta é o `--publish`, ou simplesmente `-p` seguido da porta da máquina e da porta do contêiner separadas pelo sinal de ":" (dois pontos):

```
1 docker container run --detach --name apache --volume /root/html:/usr/local/apache2/
      htdocs/ --publish 80:80 httpd:alpine
```

Neste caso abrimos a porta 80 na máquina e redirecionamos o tráfego para a 80 do próprio contêiner. Veja que a interface do "Play With Docker" agora possui a porta "80" ao lado de um botão chamado "OPEN PORT":

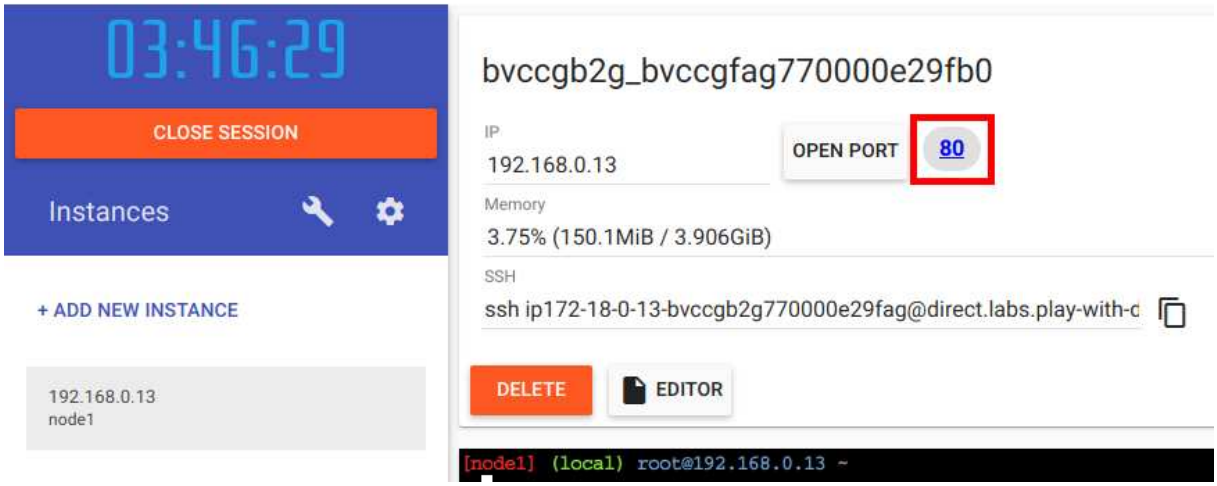


Fig. 8.1: Play With Docker - Porta Exposta

Clique nesta porta e você verá um site a respeito do OKD, a versão livre e de código aberto do Openshift. Fica muito mais fácil e interessante testar as aplicações podendo visualizá-las no navegador não é mesmo?!

Agora que temos um diretório e sabemos que podemos utilizá-lo como volume, vamos criar um segundo contêiner, desta vez baseado em **nginx** (versão **alpine**) que exibira o mesmo site, mas em uma porta diferente:

```
1 docker container run --detach --name nginx --volume /root/html:/usr/share/nginx/html --
      publish 81:80 nginx:alpine
```

O comando é muito semelhante ao anterior, trocamos o **nome**, o **diretório** de montagem, a **porta** da máquina e a **imagem**. Veja que o site é exatamente o mesmo, basta clicar na nova porta que apareceu na interface.

Se estivéssemos em nossa máquina, poderíamos consultar o endereço `localhost` ou `localhost:81` para acessar ambos os contêineres. A porta 80 é a padrão, por isso não precisamos especificá-la. Ainda assim é possível fazer isso através do `curl`, já que "estamos dentro da máquina, e não precisamos mais nos preocupar com o endereço IP do contêiner:

```
curl localhost
curl localhost:81
```

A saída é imensa, mas podemos ver que uma determinada porta é servida pelo **apache** e a outra pelo **nginx** através do parâmetro `-I` do `curl`:

```
curl -I localhost:81
```

```
HTTP/1.1 200 OK
Server: nginx/1.19.5
Date: Tue, 15 Dec 2020 14:53:58 GMT
Content-Type: text/html
Content-Length: 27601
Last-Modified: Tue, 15 Dec 2020 14:29:59 GMT
Connection: keep-alive
ETag: "5fd8c867-6bd1"
Accept-Ranges: bytes
```

A informação do webserver em questão está na linha **Server**.

O mais interessante é que podemos fazer alterações nos arquivos do diretório que virou um volume e essas alterações se refletirão automaticamente nos contêineres. É possível abrir o arquivo com o `vim` e editá-los, ou instalar o `nano` com `apk add nano` e editá-los, mas vamos fazer essa edição diretamente por um comando chamado `sed`, exemplo:

```
sed -Ei 's,O</b>rigin,4</b>Linux,' html/index.html
sed -Ei 's,K</b>ubernetes,O</b>pen,' html/index.html
sed -Ei 's,D</b>istribution!,S</b>ource,' html/index.html
```

Veja que agora, em ambos os endereços e portas o conteúdo do site se modificou pois a fonte é a mesma, um diretório.

Caso de uso - Versões Conflitantes

Uma aplicação web em Python 2 está em fase de migração para Python 3 e testes precisam ser feitos em ambas as versões. As duas versões das aplicações devem funcionar simultaneamente

durante um tempo. Nada melhor do que um volume e dois contêineres, cada qual com uma versão do Python. Desta forma é possível fazer ajustes em "tempo real" no diretório do volume e testar o comportamento em portas ou endereços diferentes.

Variáveis de Ambiente

Variáveis de ambiente são informações disponíveis "no ambiente", e quando falamos ambiente estamos nos referindo ao sistema. São informações que as aplicações podem acessar, no nosso caso seria como a temperatura do dia, o dia atual, o mês... esse tipo de informação que todo mundo tem acesso. No Linux já existem algumas variáveis de ambientes, podemos visualizá-las com o comando `env`. Do ponto de vista da aplicação, estas variáveis são muito úteis pois podem conter o usuário de acesso a um determinado banco de dados, o endereço de algum servidor de e-mail, tokens de validação e etc.

Existem muitas variáveis, e podemos acessar seus valores individualmente com `echo` seguido do nome da variável precedido por um `$` (cifrão):

```
1 echo $HOME
2 echo $PATH
3 echo $SHELL
```

Para criar uma variável no shell simplesmente digite seu nome seguido de um sinal de `=` e o valor:

```
1 CURSO=containers
2 echo $CURSO
```

Podemos enviar nossas próprias variáveis de ambiente para dentro de um contêiner durante a sua criação. As vezes os contêineres precisam destas variáveis para se ajustarem antes de ficarem prontos, um exemplo destes são os contêineres baseados em imagens de bancos de dados, por exemplo o MariaDB.

Olhando a documentação da imagem, notamos 4 variáveis interessantes:

- `MYSQL_ROOT_PASSWORD` - a senha do administrador do banco.
- `MYSQL_USER` - O nome de um usuário mais limitado que acessará uma base específica.
- `MYSQL_PASSWORD` - A senha do usuário citado acima.
- `MYSQL_DATABASE` - A base de dados do usuário citado acima.

Para especificar variáveis de ambiente durante a criação de um contêiner utilizamos o parâmetro `--env`, ou `-e` seguido do nome da variável, um sinal de igual e seu valor. Fazemos isso para cada variável que declaramos.

Vamos iniciar o contêiner do MariaDB com a senha do root como `Abc123_`, o usuário `container`, a senha `4linux` e a base `container`:

```
docker container run -d --name mariadb -e MYSQL_ROOT_PASSWORD=Abc123_ -e MYSQL_USER=
    container -e MYSQL_PASSWORD=4linux -e MYSQL_DATABASE=container mariadb
```

O processo do MariaDB se iniciará e se configurará através das variáveis que informamos. Por curiosidade, vamos aprender um novo comando, o `container logs`, para visualizar os logs do contêiner:

```
docker container logs mariadb
```

A saída é bem extensa, mas se olhar com atenção verá que o banco de dados está se preparando e leva um tempo até ficar pronto. A obtenção das variáveis é feita logo nas primeiras linhas onde temos a palavra `Entrypoint`, mas o processo por trás está escondido. O `entrypoint` é um conceito bastante interessante das imagens do Docker mas não abordaremos isso neste curso, basta saber que o `entrypoint` possibilita ao contêiner se preparar antes do comando principal entrar em ação.

Vamos executar alguns comandos no banco para verificar seu funcionamento:

```
docker container exec -ti mariadb bash
```

```
mysql -u container -p4linux container # conectando no banco e na base container
```

```
SHOW DATABASES;
CREATE TABLE teste (id INT AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(255));
INSERT INTO teste (nome) VALUES ('4Linux');
SELECT * FROM teste;
```

> **Observação:** Saia digitando `exit` ou pressionando `CTRL + D`.

Criando o contêiner desta forma não teremos os dados do banco persistidos caso o contêiner seja destruído, neste caso vamos destruir o contêiner e recriá-lo com um volume. Como estamos trabalhando com um banco de dados, seria interessante persistir os dados do banco em algum lugar mas sem precisar nos preocupar com o diretório em questão. Para estes casos vamos criar um volume gerenciado pelo próprio docker, para isso basta não utilizar um diretório do lado esquerdo do valor do parâmetro `--volume`, ou `-v`.

Segundo a própria documentação da imagem, os dados do MySQL são salvos em `/var/lib/mysql`:

```
1 docker container rm -f mariadb
2 docker container run -d --name mariadb -e MYSQL_ROOT_PASSWORD=Abc123_ -e MYSQL_USER=
      container -e MYSQL_PASSWORD=4linux -e MYSQL_DATABASE=container -v dados:/var/lib/
      mysql mariadb
```

Veja que o nosso volume não fez referência a nenhum diretório na nossa máquina, tudo o que temos é o nome **dados**. Estes volumes podem ser visualizados com o comando `docker volume`:

```
1 docker volume ls
```

Este volume ainda assim é do tipo local, a diferença é que o Docker ficará encarregado de cuidar dele. É possível criar volumes mais complexos (NFS, iSCSI, etc) com o comando `docker volume create` mais as suas opções, mas não abordaremos este assunto neste curso. A nossa única vantagem neste caso é não deixar o diretório de dados do MariaDB solto pela máquina correndo o risco de ser corrompido ou removido sem querer.

Sobre as Imagens

Os contêineres facilitaram o provisionamento de aplicações como webservers, bancos de dados, servidores de cache e etc. Porém nestes casos, geralmente não chegamos a criar estas imagens. Infraestruturas baseadas em contêineres possuem muitos contêineres deste tipo, mas também rodam outras aplicações desenvolvidas pelas próprias equipes, customizadas, feitas sob medida através das noites, forjadas nas canecas de café. Estas aplicações precisam de imagens. Uma das grandes vantagens inegáveis dos contêineres e principalmente do Docker está na etapa de desenvolvimento.

O objetivo agora é criar uma imagem de uma aplicação que se comunique com o banco de dados, mas para isso precisamos testar a aplicação localmente.

Essa aplicação, quando iniciar, criará as tabelas no banco e exibirá uma página em HTML listando alguns nomes. Sempre que acessarmos a aplicação, ela fará uma inserção de um registro aleatório no banco. A aplicação será desenvolvida em python utilizando um framework chamado flask e utilizará variáveis de ambiente para saber onde o banco de dados está e quais os dados de acesso. Não se preocupe, o código da aplicação está pronto.

Primeiro Passo - Testar o Código

Para não confundirmos o alho com bugalho, fazermos uma salada, ou ainda se preferir, "nos perdermos em um mato sem cachorro", vamos inicialmente baixar e testar a aplicação para então, se tudo ocorrer como o planejado, construir a imagem. É fundamental que tenhamos ciência do processo de criação de aplicações e das linguagens utilizadas, mesmo que de forma superficial, para nos indicar como deveremos construir as nossas imagens e que tipo de com-

portamento poderemos esperar. Mesmo que você não seja programador, conhecer em qual linguagem uma determinada aplicação foi desenvolvida ajuda a filtrar o que se devemos buscar na internet nos momentos de instabilidade, momentos estes que precedem o caos.

Primeiro vamos instalar o gerenciador de pacotes do python no "Play With Docker". Este terminal roda em um Linux baseado em alpine, sendo assim o gerenciador de pacotes chama-se `apk`:

```
1 apk add py3-pip
```

> **Atenção:** O nome dos pacotes mudam de distribuição para distribuição, no Debian o pacote do pip se chama `python3-pip`.

Em seguida, vamos clonar o repositório que possui a aplicação:

```
1 git clone https://github.com/4linux/containers-apps.git
```

Este repositório possui vários exemplos de aplicações, neste caso vamos entrar no diretório `python` e instalar as dependências que estão listadas em um arquivo chamado `requirements.txt`:

```
1 cd containers-apps/python
2 pip3 install -r requirements.txt
```

Agora que instalamos as dependências podemos iniciar a aplicação. Vamos rodar o comando e mais abaixo explicaremos o que são todos estes parâmetros:

```
1 DB_HOST=172.17.0.2 DB_USER=container DB_PASS=4linux DB_NAME=container flask run -h
      0.0.0.0
```

Veja que no exemplo acima, após acessarmos o diretório em questão, nós definimos variáveis de ambiente antes da execução do comando que inicia a aplicação, esta é apenas uma das formas de utilizar estas variáveis. Por falar em variáveis de ambiente, estas variáveis são familiares certo? Isso mesmo, definimos variáveis parecidas para iniciar o banco, aqui, apesar dos valores serem os mesmos, os nomes das variáveis são diferentes. Isso é muito comum, cada aplicação utiliza a sua própria nomenclatura. Independente disso, estas variáveis são essenciais para o funcionamento da aplicação, sem elas, a aplicação não saberá onde o banco está e tão pouco os dados de acesso.

O último parâmetro `-h 0.0.0.0` indica ao python que queremos expor essa aplicação em todas as interfaces da máquina, ou seja, precisamos fazer isso para acessar a aplicação no nosso navegador.

Agora vá até a interface do "Play With Docker" e clique em "OPEN PORT", digite 5000 no pop-up que aparecer e em seguida clique em "OK", uma nova aba abrirá e a aplicação aparecerá, tome cuidado pois o navegador provavelmente bloqueará essa janela, libere-a nas configurações.

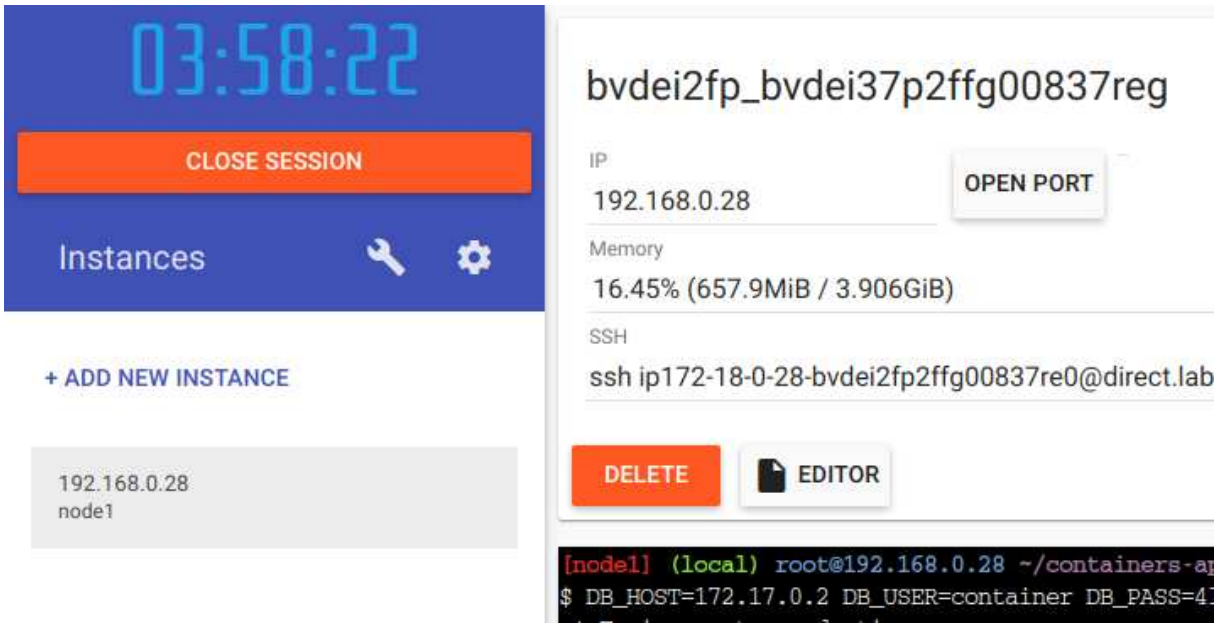


Fig. 8.2: Play With Docker - Porta Exposta

A aplicação consiste em uma simples lista e a cada vez que a página é atualizada, um novo registro é inserido no banco de dados e exibido na página. Apesar de extremamente simples já estamos nos comunicando com um banco de dados dentro de um contêiner! Agora vamos criar a imagem.

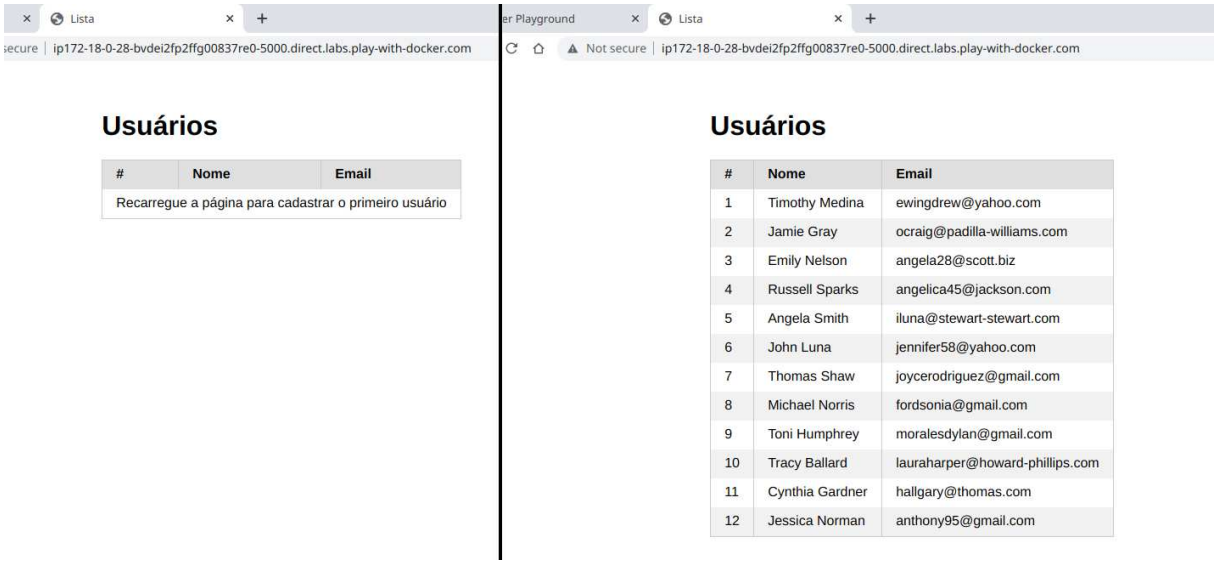


Fig. 8.3: Play With Docker - Porta Exposta

Segundo Passo - Criar a Imagem

A etapa mais importante de um contêiner do Docker é a criação da imagem, pois através dela poderemos criar infinitos contêineres desta nossa aplicação, além do mais teremos a garantia de que os contêineres funcionarão em qualquer outro sistema com uma tecnologia semelhante ao Docker, ou seja, com um "container runtime" compatível. Nossa tarefa agora é empacotar tudo e criar uma imagem com todas as dependências da aplicação dentro si, para complexo, e é, mas o Docker facilitou tudo para nós.

Testados toda a aplicação, ou pelo menos quase tudo? Chegou a hora de criar a imagem, esta imagem poderá gerar infinitos contêineres com a nossa aplicação!

Existem algumas formas de criar imagens no Docker, mas a principal e a mais efetiva é criando um arquivo de texto chamado `Dockerfile` capaz de descrever passo a passo como será a imagem. Por ser um arquivo de texto, é extremamente fácil reproduzi-lo, copiá-lo e principalmente fazer o controle de versão.

Ao invés de criar arquivos simples para explicar cada opção de um Dockerfile, criaremos o arquivo completo. Para isso, precisamos conhecer primeiro como é um `Dockerfile`:

```
1  FROM alpine
2
3  EXPOSE 5000
4
5  RUN apk add py3-pip
6
7  COPY . /opt/app
8
9  RUN pip3 install -r /opt/app/requirements.txt
10
11 WORKDIR /opt/app
12
13 CMD flask run --host 0.0.0.0
```

Veja que existem algumas palavras no início de cada linha, por exemplo `FROM` e `RUN`, explicaremos cada diretiva:

- `FROM` - Define qual será a imagem em que basearemos a nossa, ex: debian, httpd
- `EXPOSE` - Informa que esta imagem utiliza a porta 5000, é bastante útil para efeitos de documentação.
- `RUN` - Roda um comando durante a criação da imagem, geralmente instalamos coisas, preparando o sistema de arquivos para a aplicação.
- `COPY` - Copia arquivos e diretórios da máquina para dentro da imagem. Neste caso copiamos "." (ponto, o diretório atual) para `/opt/app`, se o diretório não existir, ele será criado.
- `WORKDIR` - Configura o diretório "principal" da imagem, a partir de sua definição, todos os demais comandos executarão a partir deste diretório.
- `CMD` - O principal comando da imagem, ou seja, quando iniciarmos o contêiner sem

nenhum parâmetro após o nome da imagem este será o comando principal.

Existem muitas outras opções, mas estas são o suficiente para criarmos um bom `Dockerfile`, caso queira conhecer as outras opções você pode acessar a documentação oficial `https://docs.docker.com/engine/reference/builder/`.

Vá para o diretório da aplicação que acabamos de clonar e crie o `Dockerfile`:

```
cd ~/containers-apps/python
apk add nano # ou utilize o vim já existente
nano Dockerfile
```

> **Observação:** O nano é um editor de texto inicialmente mais simples de utilizar do que o `vim`, sinta-se a vontade para utilizar o `vim`, mas se você não o conhece, lembre-se que a pergunta mais visitada no Stack Overflow é "como sair do vim".

Ao executar `nano Dockerfile` ele abrirá o arquivo para edição, e caso o arquivo não exista ele será criado. Linux é muito prático!

Copie e cole o conteúdo do `Dockerfile` no editor com a sequência `CTRL + SHIFT + V` e salve com a combinação `CTRL + O`, em seguida aperte `ENTER` para confirmar o nome do arquivo e saia na sequência com `CTRL + X`. O arquivo está pronto, antes de inciar a construção execute um comando chamado `tree` e veja se a estrutura de diretório está igual a seguinte:

```
.├──
 Dockerfile├──
 README.md├──
 app.py├──
 db.py├──
 requirements.txt├──
 static│└──
    pure-min.css├──
 templates│└──
    index.html└──
 venv
```

> **Atenção:** Lembre-se que estamos no diretório `~/containers-apps/python`, podemos visualizá-lo no terminal em cima de onde digitamos os comandos, ou através do comando `pwd`.

O comando que instruí o Docker a criar uma imagem é o `docker image build`, seguido do parâmetro `--tag` indicando o "nome" da imagem e o diretório onde estão os arquivos que queremos copiar para dentro da imagem. Sendo assim, do mesmo diretório em que está o nosso Dockerfile devemos executar o seguinte comando:

```
docker image build --tag flask .
```

A imagem está pronta! Podemos listá-la com `docker image ls` e é possível notar que ficou com aproximadamente **74MB**:

```
docker image ls flask
```

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
flask        latest    d0e5d3456deb   5 minutes ago   74.2MB
```

É possível diminuí-la mais ainda, mas não vamos focar nisso, afinal **74MB** já é um tamanho minúsculo perto das coisas que existem por aí. Agora vamos rodar o contêiner, lembre-se de especificar as variáveis de ambiente e tomar cuidado com o endereço IP do contêiner do banco:

```
docker container run -d -e DB_HOST=172.17.0.2 -e DB_USER=container -e DB_PASS=4linux -e
     DB_NAME=container -p 5000:5000 --name flask flask
```

Basta clicar na porta `5000` que apareceu na interface do "Play With Docker" que poderemos acessar a aplicação, desta vez, dentro do contêiner! Estamos prontos para enviar essa imagem para o registry do Docker Hub.

Caso de uso - Na minha Máquina Funciona

Uma frase bastante conhecida entre os desenvolvedores e administradores de sistema. Uma nova versão do sistema foi feita pelos desenvolvedores e uma nova biblioteca precisou ser utilizada ou atualizada. Por diversas questões como, prazo, pressão ou falha na gerência, essa nossa versão foi enviada sem o devido aviso a respeito dessa biblioteca. No ambiente de homologação a aplicação apresentou problemas e não pode ser provisionada em produção. Um desgaste entre as equipes aconteceu até que pudessem notar o pequeno problema que passou despercebido. Um contêiner evita esse tipo de situação, entregando todas as dependências, alterações e qualquer outro detalhe em um pacote único e imutável.

Enviando Imagens para o Docker Hub

Uma das grandes vantagens do Docker frente as outras ferramentas da época em que surgiu é a facilidade com que podemos distribuir, ou entregar, estas imagens de contêineres que criamos.

Nossa imagem está pronta, sendo assim o próximo passo é torná-la disponível de uma forma que nós mesmos, nossa equipe, ou qualquer outra pessoa possa baixá-la exatamente como

baixamos as nossas imagens até então, com um simples `docker image pull`. A solução é enviá-la para um repositório de imagens, ou seja, um registry, desta forma teremos uma réplica da imagem a um comando de distância. Existem vários registries, o mais famoso é o Docker Hub pois é público, nasceu juntamente com o Docker e faz parte de todo esse ecossistema interessante, e é sem dúvida um dos grandes facilitadores da entrega de software.

Nas empresas é comum utilizar um registry privado, não somente um que exija usuário e senha para baixar e enviar as imagens, mas um registry que sem meios especiais não pode ser acessado por fora da empresa. Este registry pode ser fornecido pela própria cloud, como um serviço, ou mesmo através de alguns softwares como Portus, Harbor ou Nexus.

No início do curso criamos uma conta no Docker Hub para poder utilizar o "Play With Docker", se por algum motivo você ainda não tem a sua conta pode criar uma em `https://hub.docker.com/signup`.

Com o nosso usuário e senha em mãos, vamos nos logar através do comando `docker login`. Por padrão, o comando se conectará diretamente ao Docker Hub, mas poderíamos especificar outro registry se assim desejássemos:

```
1 docker login
```

Ao preencher os dados de acesso, a saída no terminal será semelhante a seguinte:

```
1 Login with your Docker ID to push and pull images from Docker Hub. If you don't have a
      Docker ID, head over to https://hub.docker.com to create one.
2 Username: hectorvido
3 Password:
4 WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
5 Configure a credential helper to remove this warning. See
6 https://docs.docker.com/engine/reference/commandline/login/#credentials-store
7
8 Login Succeeded
```

Para enviar uma imagem para um registry o nome da imagem precisa refletir o endereço, o nosso usuário, o nome da imagem e sua tag, que por padrão é a latest. Até então temos utilizado as imagens pelos seus nomes abreviados, por exemplo `debian`, `alpine`, mas tomando a imagem `debian` de tag `buster-slim` como base, seu "nome" completo é:

```
1 docker.io/library/debian:buster-slim
```

Já a imagem `alpine` é:

```
1 docker.io/library/alpine:latest
```

Isso significa que no caso das imagens baixadas pelo Docker, a parte `docker.io`, `library` assim

como a tag `latest` são opcionais. Tente baixar as imagens citadas acima e verá que funcionará da mesma forma.

Talvez você ainda não tenha notado, mas algumas imagens do Docker Hub não são compostas de um nome único, possuem "dois nomes", como, por exemplo, uma das imagens oficiais do openSUSE que se chama opensuse/leap ao invés de opensuse, que por sinal existe, mas está descontinuada segundo a própria documentação. O que é o primeiro nome?

O primeiro nome da imagem, ou `opensuse` neste caso, assim como `library` no caso anterior, no Docker Hub referem-se ao usuário responsável por aquelas imagens. Se tentarmos baixar a imagem do `opensuse` não conseguiremos pois este repositório está vazio:

```
1 |docker image pull opensuse
```

```
1 |Using default tag: latest
2 |Error response from daemon: manifest for opensuse:latest not found: manifest unknown:
  |     manifest unknown
```

Porém, se tentarmos baixar a versão estável (conhecida como `leap`) do "usuário" `opensuse`, o comando funcionará:

```
1 |docker image pull opensuse/leap
```

Podemos eventualmente especificar uma tag:

```
1 |docker image pull opensuse/leap:15.2
```

Ou até mesmo a especificação completa do nome da imagem:

```
1 |docker image pull docker.io/opensuse/leap:15.2
```

Certo... mas nosso objetivo era subir a nossa imagem. Tudo isso foi para mostrar que, no mínimo, a nossa imagem precisa ter como prefixo o nome do nosso usuário. Meu usuário é `hectorvido` sendo assim a imagem `flask` precisa se chamar `hectorvido/flask`. O comando para fazer isso no docker é o `docker image tag` e podemos executá-lo da seguinte forma:

```
1 |docker image tag flask hectorvido/flask
```

Feito isso, e com o comando `docker login` executado anteriormente, podemos enviar nossa imagem para o mundo:

```
1 |docker image push hectorvido/flask
```

Por curiosidade, se tentarmos enviar apenas a imagem chamada `flask` teremos um erro não muito detalhado para os desatentos:

```
1 |docker image push flask
```

```
1 |Using default tag: latest
2 |The push refers to repository [docker.io/library/flask]
3 |1aacd637ab5e: Preparing
4 |294cae6186d7: Preparing
5 |565db6091650: Preparing
6 |777b2c648970: Preparing
7 |
8 |denied: requested access to the resource is denied
```

Veja que o comando tentou enviar a imagem para `docker.io/library/flask`. Esta imagem não existe (pelo menos até o momento em que este curso foi gravado) mas o problema realmente está no fato de não termos nos autenticado com o usuário `library`, se quer temos estas credenciais.

Agora que nossa imagem está disponível na rede mundial de computadores podemos iniciar uma nova instância ou destruir e recriar a instância atual e baixá-la como qualquer outra imagem do Docker Hub, aliás qualquer pessoa que você conhece poderá baixar sua imagem:

```
1 |docker image pull hectorvido/flask
```

O Docker Hub fornece a possibilidade de impedir o download das imagens de forma pública e podemos fazer isso com uma única imagem sem custo algum, acima dessa quantia será necessário contratar o serviço, mas isto está fora do escopo deste curso.

Definindo uma Infraestrutura

Quando temos muitos contêineres, muitos parâmetros, muitas variáveis de ambiente, volumes e todas as outras coisinhas que fazem parte da nossa infraestrutura, decorar tudo isso para escrever nos "`docker container run`" não é uma tarefa fácil. Provavelmente alguém pode pensar que poderíamos escrever estes comandos em um script do próprio shell e executá-los sempre que desejássemos. Não é uma má ideia, mas repararam quantas vezes eu pedi para prestarem atenção no endereço IP do banco? Detalhes, são os detalhes esquecidos que causam as maiores catástrofes, pois são difíceis de se enxergar.

Não é o nosso foco agora, mas também é possível criar uma outra rede através do Docker e

dar um endereço IP fixo para cada container, mas isso é mais um detalhe na grande pilha de detalhes prontos para causar caos e desordem.

Existe uma solução para estes nossos problemas, e o nome desta solução é Docker Compose. Através do Docker Compose podemos criar um arquivo de texto, chamado "compose file", capaz de especificar e simplificar muitos destes detalhes. O compose file é escrito em YAML. Para manter as coisas da forma mais sucinta possível, podemos dizer que o YAML é uma alternativa ao XML, JSON ou até mesmo ao INI files com suas próprias vantagens e desvantagens.

O primeiro detalhe (mais uma vez os detalhes) está em sua indentação, no YAML a indentação é obrigatória e indica qual elemento pertence, é filho, ou está dentro, de outro. Infelizmente a indentação dos arquivos YAML pode ser realizada apenas através de espaços. Uma outra característica do YAML é separar as definições entre "chave" e "valor", unidas por um sinal de "dois-pontos". Sendo assim, um YAML que pudesse definir este curso poderia ser escrito como:

```
1 nome: Containers
2 empresa: 4Linux
3 duracao: 20 horas
4 formato: EAD
```

No YAML podemos ter chaves em que seus valores são outras chaves, cada qual com seu valor. Podemos imaginar como um outro documento YAML dentro de um destes valores, por exemplo, poderíamos adicionar o instrutor ao documento acima:

```
1 nome: Containers
2 empresa: 4Linux
3 duracao: 20 horas
4 formato: EAD
5 instrutor:
6   nome: Hector Vido Silva
7   email: hector@example.org
```

Tanto a chave `nome` como a chave `email` possuem seus próprios valores, mas ambas estão dentro da chave instrutor.

A próxima característica do YAML são as listas, ou arrays, que definem uma série de elementos em uma ordem específica. Por exemplo, poderíamos adicionar os capítulos deste curso em uma lista do mesmo objeto:

```
1 nome: Containers
2 empresa: 4Linux
3 duracao: 20 horas
4 formato: EAD
5 instrutor:
6   nome: Hector Vido Silva
7   email: hector@example.org
```

```
capitulos:
- 'Introdução'
- 'Containers'
- 'Docker'
```

A chave `capitulos` é uma lista com três itens. Estes itens poderiam ser outros objetos, ou documentos, mas não precisaremos avançar até este ponto.

Estamos prontos para criar o nosso "compose file", o conteúdo dele será o seguinte:

```
version: '3.0'

services:
  aplicacao:
    image: flask # ou hectorvido/flask
    environment:
    - 'DB_HOST=banco'
    - 'DB_USER=container'
    - 'DB_PASS=4linux'
    - 'DB_NAME=container'
    ports:
    - '5000:5000'
  banco:
    image: mariadb
    environment:
    - 'MYSQL_ROOT_PASSWORD=Abc123_'
    - 'MYSQL_USER=container'
    - 'MYSQL_PASSWORD=4linux'
    - 'MYSQL_DATABASE=container'
    volumes:
    - 'data:/var/lib/mysql'

volumes:
  data: {} # indica chave de valor vazio
```

Como de costume, vamos explicar cada sessão:

- `version` - Indica a versão do compose file, versões mais recentes suportam opções adicionar, principalmente para o Docker Swarm, no nosso caso `3.0` é suficiente. Esta versão também pode ser compatível ou não com a versão do docker conforme pode ser consultado aqui.
- `services` - Inicia a definição dos contêineres.
- `aplicacao` - Define o primeiro contêiner, este nome é utilizado para sua própria referência dentro do compose file.
- `image` - Indica em qual imagem se basear para criar o contêiner.
- `environment` - Define as variáveis de ambiente para o contêiner, exatamente como no `--env`. Perceba a referência ao **nome** `banco` ao invés do endereço IP do contêiner do banco de dados. Esta é uma vantagem bastante cômoda do docker compose.
- `ports` - Define quais portas abrir, exatamente como no `--publish`.
- `volumes` - Existem duas chaves volumes, uma delas é **global**, ou seja, está na extrema esquerda e serve para definir um volume. Como vários contêineres podem utilizar um volume, é necessário que sejam declarados desta forma. Já o `volumes` dentro do `banco` é exatamente como o `--volume` que já conhecemos.

Por padrão o nome do compose file é `docker-compose.yml`, crie este arquivo utilizando o nano:

```
1 apk add nano # caso não tenha sido instalado
2 nano docker-compose.yml
```

Copie e cole o conteúdo do `docker-compose.yml` no editor com a sequência `CTRL + SHIFT + V` e salve com a combinação `CTRL + O`, em seguida aperte `ENTER` para confirmar o nome do arquivo e saia na sequência com `CTRL + X`.

Chegou a hora de rodar os contêineres. Por garantia, vamos limpar todos os contêineres que estão rodando na máquina com o seguinte comando:

```
1 docker container rm -f $(docker container ls --quiet -a)
```

Mais um novo elemento, ou um novo "detalhe". Já conhecemos o comando `container rm` mas talvez você não conheça o subshell, ou seja `$()`. Um comando dentro de um subshell é executado antes do comando mais externo e o trecho `docker container ls --quiet -a` lista todos os contêineres, por causa do `-a`, porém exibe apenas seus ids, por causa do `--quiet`, ou `-q`. Isso faz com que todos os ids fiquem na frente do parâmetro `-f` e consequentemente sejam removidos pelo comando externo.

Um subshell deve ser lido exatamente como a operação matemática `(1 + 1)* 2` que apesar da multiplicação ter prioridade, os parênteses alteram essa prioridade para a expressão interna, resultando em `4`, ao contrário de `3`.

Imagine que temos os contêineres de ids `aa9b7cdbd58a`, `07db888b6d40`, `98c842272311`. O resultado do comando acima seria o seguinte:

```
1 docker container rm -f aa9b7cdbd58a 07db888b6d40 98c842272311
```

Agora que limpamos todos os contêineres, podemos iniciar o docker compose. O comando a seguir deve ser executado do mesmo diretório em que o `docker-compose.yml` está:

```
1 docker-compose up
```

Os logs dos contêineres aparecerão na tela, separados por cores:

Ao acessar a porta 5000, clicando na interface do "Play With Docker", acessamos a aplicação exatamente da mesma forma que fizemos anteriormente. Por debaixo dos panos, tudo o que o docker compose faz é criar os contêineres para nós e conectá-los de uma forma conveniente. Pare o docker compose pressionando `CTRL + C`.

É possível listar os contêineres exatamente como antes, nada novo aconteceu, os processo é

exatamente o mesmo que fizemos manualmente, mas agora está automatizado e bem mais fácil de reproduzir. Para iniciar o docker compose e desatachar o terminal, basta utilizar a opção `-d`:

```
1 |docker-compose up -d
```

Agora o terminal foi liberado. O docker compose possui uma série de comandos para trabalhar com os contêineres definidos no arquivo, por exemplo, podemos listá-los da seguinte forma:

```
1 |docker-compose ps
```

Podemos visualizar os logs de um determinado contêiner, especificando o nome definido no arquivo:

```
1 |docker-compose logs -f aplicacao
```

O comando `exec`, funciona da mesma forma, porém é um pouco mais simples:

```
1 |docker-compose exec aplicacao sh
```

Apesar desta nova forma de interação, ainda é possível listar os contêineres e trabalhar diretamente com seus nomes reais ou ids, o compose apenas facilita ligeiramente esta tarefa.

Qualquer alteração no `docker-compose.yml` pode ser feita enquanto o mesmo está em execução, e para aplicar as alterações basta executar um `up -d` novamente.

Para desligar os contêineres especificados no arquivo utilizamos a opção `down`:

```
1 |docker-compose down
```

Observe que além dos contêineres, uma rede também foi removida. Esta rede é quem nos permitiu fazer com que o contêiner `aplicacao` enxergasse o `banco` pelo seu nome e nos poupou do trabalho de descobrir o seu endereço ip.

A principal utilidade de um compose file é definir como é a infraestrutura que uma determinada aplicação precisa ou faz parte. Como exemplo podemos citar o aplicativo de comunicação RocketChat que fornece um compose file para subir um ambiente rapidamente, ou mesmo o software para gerência de logs Graylog e seu compose file.

Os exemplos são inúmeros, mas graças ao Docker, os contêineres, as imagens, os registries e o Docker Compose, podemos subir estas infraestruturas como forma de estudo com apenas um único comando.

Caso de Uso: Infraestrutura como Código

Definir a infraestrutura em um documento de texto facilmente reproduzível e fiel ao formato final é principal caminho da IaC (Infra as Code). O compose file nos permite definir exatamente como a infraestrutura é e se comunica. Outras ferramentas possuem seus próprios arquivos em formato semelhante como, por exemplo, o Kubernetes e o Terraform. Ter a infraestrutura codificada em um formato simples também serve como documentação mínima para a equipe ou novos membros.

Resumo

Chegamos ao final do capítulo e desta vez nos aprofundamos um pouco mais nos conceitos e práticas do Docker.

Iniciamos o capítulo aprendendo sobre volumes, e que devemos utilizá-los para persistir os dados devido a natureza efêmera dos contêineres. Para adicionar um volume a um contêiner fazemos isso durante o `run` e com o parâmetro `--volume`, ou `-v`. Utilizamos dois tipos de volumes, um deles era um diretório com arquivos HTML e outro um tipo de volume gerenciado pelo próprio Docker para guardar os arquivos do banco. Utilizamos volumes como diretórios sempre que vamos modificar os arquivos com frequência, por ser mais cômodo do que acessar o contêiner.

Aprendemos também a redirecionar uma porta da máquina para o contêiner, facilitando drasticamente a conexão e nossos testes, pois assim não precisamos nos preocupar com o endereço do contêiner. Mas tome cuidado, não é boa prática expor contêineres de bancos de dados, ou servidores de cache pois desta forma eles se tornam extremamente vulneráveis.

Conhecemos as variáveis de ambiente e seus casos de uso, vimos que o contêiner do **MariaDB** nos obrigava a especificar algumas variáveis e depois, mais a frente, utilizamos estas variáveis na nossa aplicação.

Clonamos, ou baixamos, uma repositório com uma aplicação em Python para se conectar a este banco e antes de mais nada, instalamos suas dependências e testamos seu código para não encontrar surpresas durante a criação da imagem.

Uma vez que a aplicação se comportou como o esperado, criamos nossa primeira imagem com um arquivo chamado `Dockerfile`. O `Dockerfile` contém as instruções necessárias para o Docker construir a nossa imagem com todas as dependências da aplicação. Algumas instruções importantes são `FROM`, `RUN`, `COPY` e `CMD`. Ao término de sua edição, criamos finalmente a imagem com o comando `docker image build`.

Com a imagem pronta, nos logamos no Docker Hub através de `docker login`, descobrimos que o nome das imagens do docker contém o endereço do registry, o usuário, o nome da imagem e sua

tag, por exemplo `docker.io/library/alpine:latest`. Ao final enviamos nossa imagem renomeada com o nome do nosso usuário através do comando `docker image tag` para o Docker Hub com `docker image push`.

Na última parte conhecemos uma ferramenta para facilitar a criação e definição de ambientes em contêineres, o Docker Compose, juntamente com o formato de seus arquivos, o YAML. Definimos nossa infraestrutura anterior que estava um pouco confusa pela quantidade de parâmetros em um arquivo simples de fácil compreensão. Iniciamos os dois contêineres definidos no compose file através de um único comando, o `docker-compose up`.

9

Final

Chegamos ao final do material, e acredito que agora você tenha uma visão melhor a respeito dos contêineres e as vantagens que eles trazem tanto para os administradores como para os desenvolvedores. Se sua dúvida era mais relacionada a parte de negócios ou mercado de trabalho, também vimos informações mais do que suficiente para convencê-los de que esta tecnologia não é passageira. De qualquer forma, independente da ferramenta, o conceito é o mais importante e permanece o mesmo.

Na primeira parte do curso descobrimos que um contêiner é um apanhado de tecnologias que foram utilizadas em conjunto e o Docker foi um dos facilitadores. Também mencionamos que os contêineres facilitaram a implementação de microsserviços.

Abordamos o porquê do símbolo do contêiner, padronização, e seguimos com vantagens para o negócio e mercado de trabalho.

Analisamos o Docker de um ponto de vista mais teórico, mostrando a rede e os processos isolados, além das muitas camadas que uma imagem pode ter, e então passamos a parte prática. Apesar de bastante coisa, uma vez que temos conhecimento de que as imagens geralmente são baixadas do Docker Hub, boa parte das coisas que fizemos pode ser resumida neste comando:

```
1 docker container run -d --name banco -e MYSQL_ROOT_PASSWORD=4linux -v dados:/var/lib/
      mysql -p 3306:3306 mariadb
```

Aí em cima temos o download da imagem e a criação do contêiner, utilizando de variáveis de ambiente, volumes e expondo o serviço em uma determinada porta. Mais importante do que usar imagens é criar as nossas próprias imagens, e foi o que aprendemos a fazer através de um

arquivo chamado `Dockerfile` e o comando `docker image build`.

Na última parta conhecemos o `docker compose` e pudemos definir nossos contêineres em um arquivo do tipo YAML, o que simplificou bastante nosso trabalho e nos apresentou a um tema bastante interessante, a infraestrutura como código.

E agora?

Agora que aprendemos o começo, temos outros caminhos para seguir. Não se iluda pois o Docker não é somente isso, o que apresentamos é apenas a superfície de um assunto muito maior, ou de uma ferramenta muito maior, sendo assim continue estudando para pegar os detalhes e entender como melhorar tudo o que foi feito aqui e muito mais, afinal, vários detalhes foram deixados de lado e muitos passos foram executados de forma mais longa para melhor compreensão.

O próximo caminho mais interessante talvez seja conhecer um orquestrador como Kubernetes ou Swarm, particularmente eu lhe recomendo o Swarm pois é possível trabalhar com o "compose file" que criamos neste curso, além dos comandos serem muito similares e não precisarmos de mais nada além do próprio Docker.

Com os orquestradores nossas configurações ficam no ambiente, e duas figuras novas aparecem, o `config` e o `secret`, que apesar de terem nomes diferentes no Kubernetes e no Swarm, servem para o mesmo propósito.

Uma outra figura importante são os volumes distribuídos, e um bom e velho NFS é mais do que suficiente para entender estes conceitos, seria interessante utilizá-lo juntamente ao orquestrador. Já que estamos falando em cluster, não podemos deixar de mencionar os load balancers, afinal, com mais de uma máquina, seria interessante criar um único ponto de acesso, e para isso você pode utilizar muitas ferramentas, como por exemplo o HAProxy ou mesmo o Traefik.

Por fim, também recomendo que procure alternativas ao Docker como Podman e LXD, isso provavelmente lhe mostrará vantagens e desvantagens destas ferramentas em relação ao Docker.

Abordamos estes conceitos em nossa trilha de contêineres, com outros 4 cursos. Um se aprofunda no Docker e os outros três abordam o Kubernetes e suas variantes bastante interessantes.

Espero que tenha gostado, e nos vemos no próximo curso.

Referências

Imagem de capa: Nor Phai - flaticon http://www.flaticon.com