LEANDRO COSTA

DOCKER UM GUIA RÁPIDO

O MÍNIMO QUE VOCÊ PRECISA SABER PARA COMEÇAR

+ 25 PERGUNTAS SOBRE DOCKER MAIS COMUNS EM ENTREVISTAS

GRATIS

ISENÇÃO DE RESPONSABILIDADE

Todas as informações contidas nesse livro são provenientes e aprendizado pessoais com Docker e gestão de contêineres ao longo de vários anos. Embora eu tenha me esforçado ao máximo para garantir a precisão e a mais alta qualidade dessas informações e acredite que todas as técnicas e métodos aqui ensinados sejam altamente efetivos para qualquer pessoa desde que implementados corretamente, não existe qualquer garantia de qualquer resultado, e eu não me responsabilizo pela implementação do leitor. Sua situação e/ou condição particular pode não se adequar perfeitamente aos métodos e técnicas ensinados neste livro. Sendo assim você deverá utilizar e ajustar as informações deste livro de acordo com sua situação e necessidades.

Todos os nomes de marcas, produtos e serviços mencionados neste livro são propriedades de seus respectivos donos e são usados somente como referência. Além disso em nenhum momento neste livro há a intenção de difamar, desrespeitar, insultar, humilhar ou menosprezar você leitor ou qualquer outra pessoa, cargo ou instituição. Caso qualquer escrito seja interpretado dessa maneira, eu gostaria de deixar claro que não houve intenção nenhuma de minha parte em fazer isso. Caso você acredite que alguma parte deste livro seja de alguma forma desrespeitosa ou indevida e deva ser removida ou alterada, você pode entrar em contato diretamente com minha equipe de suporte utilizando o e-mail ***leandro@erudio.com.br***.

DIREITOS AUTORAIS

SOBRE O AUTOR

Eu sou Leandro Costa, **analista de sistemas** e **desenvolvedor**, bacharel em Sistemas de Informação com pós-graduação em Engenharia de Software. Trabalho a **mais de 10 anos como analista, desenvolvedor** back-end e sou um entusiasta apaixonado pela área. Em todos esses anos eu errei bastante e aprendi muito com esses erros. Por isso sei exatamente o que você precisa aprender e com o que não deve perder seu tempo **para ter uma carreira bem-sucedida na área de tecnologia**.

Comecei minha carreira com manutenção de computadores, impressoras e monitores - arriscava fuçar em tudo que tinha um *chip* dentro. Comecei a programar em Delphi e logo depois iniciei a faculdade onde me dediquei à aprender Java e GNU/Linux. Como desenvolvedor eu iniciei trabalhando com Delphi, depois Java com JSF e Spring, Groovy, depois AngularJS e me apaixonei por JavaScript. Atualmente trabalho com .NET Core e C# e estou gostando bastante principalmente depois que a plataforma virou Open Source. Já trabalhei em diferentes tamanhos de projetos nacionais e internacionais, em setores como varejo, farmacêutico, atacadista, marketing, governo, ERP e fintechs. Participei de projetos Agile, *"fake Agile"*, Cascata, RUP e claro me deparei com muito *Extremme go Horse* e confesso que tive que fazer algumas gambiarras ao longo da carreira.

Nos diferentes projetos em que trabalhei utilizei diversas tecnologias como Java, Spring, Hibernate, JSF, C#, Entity Framework, ASP.NET, Groovy, Grails, JavaScript, AngularJS, React JS, Docker, Docker Compose, Kubernetes, MySQL, Postgree, SQL Server, Oracle dentre outras.

Em 2012 tive o prazer de criar dois cursos gratuitos de Groovy e JSF no Youtube, são eles:

- Curso De JavaServer Faces (JSF) Do Zero À Nuvem

- Curso Básico de Grails

Recentemente **publiquei dois livros sobre engenharia de software** em que detalho a evolução dos modelos de gestão do desenvolvimento de software. Entretanto os frameworks ágeis ganharam uma atenção especial principalmente o eXtreme Programming e o Scrum. Você pode conferir o resultado na Amazon, procurando pelos títulos:

- **Engenharia de Software Essencial: Um livro rápido com foco em Agile**

- **Do Waterfall ao Scrum: Um guia para uma transição segura**

Atualmente tenho como objetivo ajudar você a conseguir o trabalho/estágio que você deseja ou dar um up na sua carreira. Colaborando no aprendizado de pessoas que buscam conhecimento através de treinamentos eficientes e práticos na área de desenvolvimento, Agile, TDD, Integração Contínua, Docker, Clean Code e API's (SOA), especialmente RESTfull API's. Sou apaixonado por transmitir conhecimentos e contribuir para que as pessoas se desenvolvam e alcancem o melhor de si.

O objetivo aqui é te ensinar o que você precisa saber e sem enrolação, do básico ao avançado. Quando você aprende mais, você se destaca. E quando você se destaca, você começa a crescer na sua carreira.Animado para começar? Então vamos lá!

Sumário

Introdução

Você já se perguntou por que algumas pessoas conseguem evoluir muito rápido na área de tecnologia e outras passam anos e mal aprendem a fazer um *Hello World*?

E por que algumas dessas pessoas além de evoluírem muito rápido ainda conseguem ser muito mais produtivas que outras?

Inclusive, você já percebeu que certas coisas você conseguiu aprender rapidamente e outras nem tanto ou você simplesmente desistiu e nunca aprendeu?

A verdade é que essas pessoas que "dão mais certo" não tem mais sorte que as demais. Isso acontece por que elas sacaram, mais rapidamente que as outras, que existem alguns "atalhos para a evolução". E se você está pensando que eu faço parte desse grupo de pessoas abençoadas está enganado. Eu apanhei bastante até chegar aqui e posso dizer que descobri esses "atalhos para a evolução", mas foi bem demorado.

A boa notícia é que eu mapeei todos os atalhos para encurtar a sua jornada evolutiva com Docker e nesse livro eu vou te passar tudo o que você precisa aprender para começar, quase que de presente. Assim, ao contrário de mim você não irá desperdiçar meses ou mesmo anos estudando coisas que não serão usadas no mundo real. São coisas simples que passam até desapercebido, mas que se você implementar vão fazer bastante diferença no resultado final.

Eu chamo esses atalhos de "Docker um guia Rápido: O Mínimo que você Precisa Saber para Começar". São os tópicos centrais do Docker que eu fui descobrindo ao longo dos anos e que facilitam e muito o meu trabalho. Quando se fala de Docker existe uma "tonelada" de informações que eu compactei em alguns pontos centrais que além de facilitar minha vida, garantir maior qualidade no meu trabalho e em menos tempo. É exatamente isso que eu vou te passar neste e-book.

São atalhos que se usados de forma correta irão capacitá-lo para passar em entrevistas sobre Docker e aumentar o seu salário em bem pouco tempo. Se você sonha em se tornar um ***Ninja Docker*** ou encara os contêineres apenas um hobby essas dicas serão fundamentais pra sua evolução.

Os capítulos foram ordenados para te permitir saber o que fazer em cada fase. Te ensinando desde o que é o Docker e como funciona sua arquitetura, passando pela instalação e as primeiras interações com o Docker no terminal. Depois aprenderemos tudo de Dockerfile, como construir nossas próprias Docker images e executar contêineres a partir delas. Aproveite cada um dos tópicos e tenha certeza de que está caminhando para o sucesso. Bom então chega de conversa e vamos logo conhecer o Docker.

1. Introdução aos Contêineres

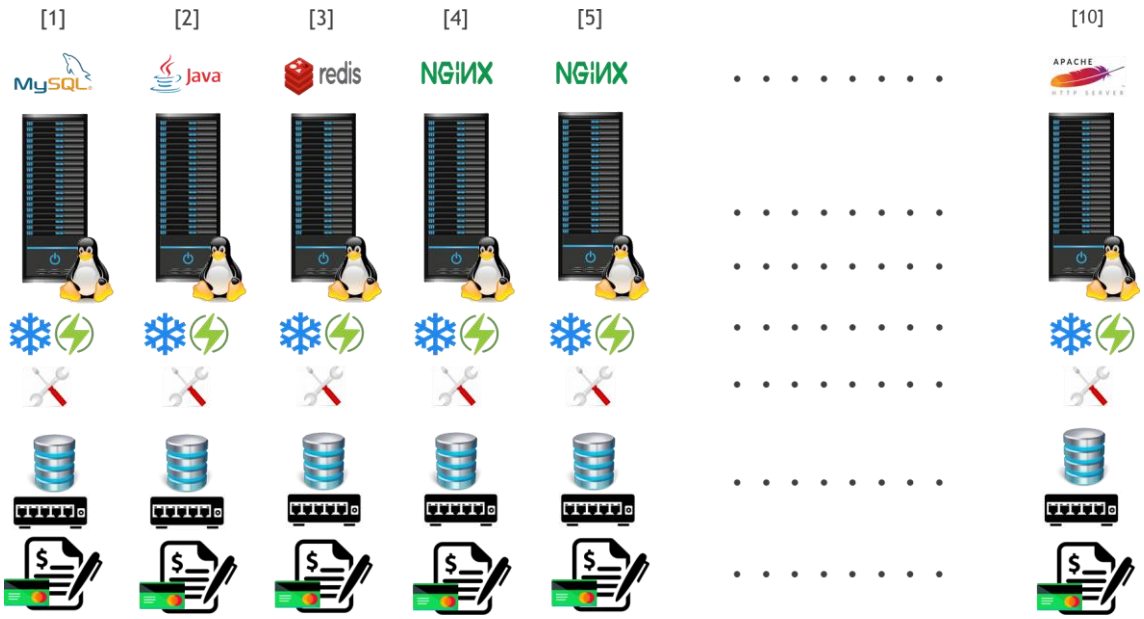
Docker pode parecer complexo à princípio especialmente se você não tem muita familiaridade com o Linux, mas à medida que formos progredindo vou me aprofundar em cada um dos pontos pra que você aprenda de verdade. Aprenderemos o que são contêineres e suas diferenças em relação às máquinas virtuais. O que é o Docker e como ele tem evoluído. A compatibilidade entre contêineres no Windows e no Linux e o que eu acredito que deve acontecer nos próximos anos com a plataforma Docker.

1.1. O Surgimento das Máquinas Virtuais

Antes de entendermos o que é Docker vamos começar com uma breve, mas absolutamente vital, introdução aos contêineres. Afinal de contas, não adianta ter pressa em aprender sobre contêineres, sem ter a certeza de que vantagens eles realmente agregam. Para fazer isso, acho que antes precisamos dar uma analisada em como fazíamos as coisas antes do Docker e das máquinas virtuais.

Antes de tudo precisamos enfatizar que ***tem tudo a ver com aplicações***. Antigamente, era bastante comum configurarmos servidores e aplicações em uma proporção de um para um. Obviamente manter as coisas no mainframe organizadas era muito difícil e claro aconteciam muitos erros de configuração.

Figura 1 – A Implantação de Aplicações Tradicionais



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Conforme mostra a Figura 1, para cada aplicação que precisávamos implantar, precisávamos comprar e configurar um servidor dedicado para isso. Se fossem 10 aplicações, precisávamos de 10 servidores, portanto 10 pedidos de compra, 10 instalações do sistema operacional, se o sistema operacional fosse licenciado então eram necessárias 10 licenças do sistema operacional, 10 posições para os hacks, 10 serviços redundantes de energia e resfriamento, 10 infraestruturas de rede e armazenamento provavelmente redundantes também. E tudo isso incluindo cabeamento, switches, firewall e mais um monte de outras coisas. Se a organização começasse a crescer a tendência era que isso se tornasse cada vez mais complexo. Isso se tornava ainda mais evidente a cada implantação de uma nova aplicação que claro exigia mais uma *"tonelada"* de infraestrutura.

Isso não era nada bom para agilidade. Se precisássemos subir uma nova aplicação. Seriam 10 semanas enquanto nos preparávamos para a mudança. Obtínhamos aprovação financeira, fazíamos o pedido, preparávamos o servidor, readequávamos a rede e finalmente instalávamos a aplicação. Sem dúvida todos temos que concordar que eram tempos difíceis, mas na verdade existia um problema ainda pior do que esse.

Além de levar uma eternidade até disponibilizar uma aplicação em produção, havia outro problema profundamente desagradável que era o desperdício maciço de recursos. Para, cada um dos servidores que nós compramos e configuramos para cada uma das nossas aplicações. Bom em noventa e nove por cento das vezes, nós só os utilizávamos uma pequena fração de sua capacidade. Por isso, era comum o servidor estar em execução operando com menos de 10% de sua capacidade. Às vezes, estávamos falando de dois a três por cento de utilização média para um servidor físico e como você sabe essas coisas não são baratas.

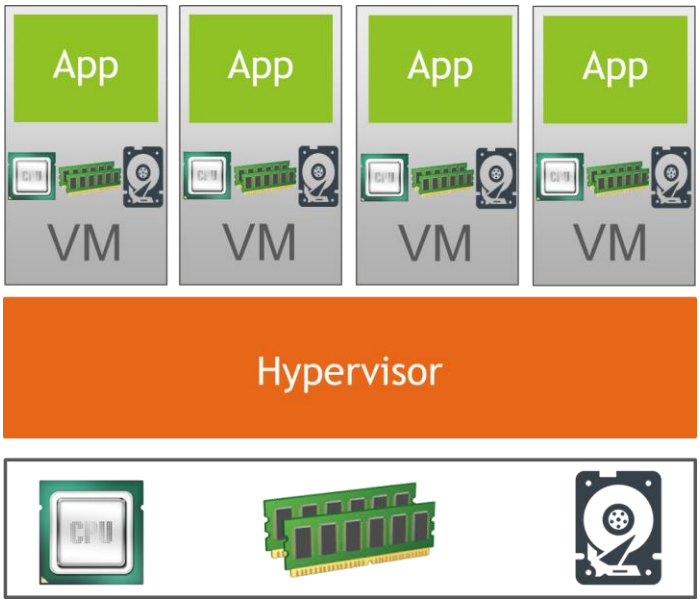
A conclusão óbvia é que isso era um tremendo desperdício. Ainda mais se considerarmos que os servidores dedicados custam uma tonelada em investimentos financeiros e operacionais, executando quase sempre abaixo da capacidade. Claro que isso não poderia ser sustentável para sempre. Algo claramente tinha que ser feito e algo foi feito esse algo é a virtualização.

Eu sei que a virtualização pode significar muitas coisas para muitas pessoas. Então conforme mostra a **Figura 2**, no nosso caso, estamos falando sobre a virtualização de hypervisor, onde tomamos um servidor físico. Na verdade, vamos chamar de máquina física. E dividimos em várias máquinas virtuais. E cada máquina virtual se comporta exatamente como se fosse uma máquina física. Isso significa que agora podemos executar várias aplicações em uma única máquina física. Uma aplicação por máquina virtual.

Então, como um simples exemplo rápido, vamos supor que temos 10 aplicações e cada uma consome menos de 10% da CPU, da RAM e do espaço em disco da nossa máquina física. Poderíamos conseguir executar todas as 10

aplicações em uma única máquina física com cada aplicação em uma máquina virtual. Quão poderoso isso é? Quase da noite para o dia passamos de um modelo de uma aplicação para um servidor físico, em que a utilização era escandalosamente baixa, para um modelo de várias aplicações por servidor, onde a utilização do servidor era geralmente superior a 50% e em alguns casos se aproximava de 80% e 90%.

Figura 2 – A Virtualização de Hypervisor



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

É desnecessário dizer que isso foi uma verdadeira revolução e um modelo muito melhor comparado com o anterior, mas na verdade, está longe de ser perfeito. Claro que foi uma melhoria massiva quando comparada com o modo como as coisas eram feitas anteriormente, mas não é uma solução perfeita.

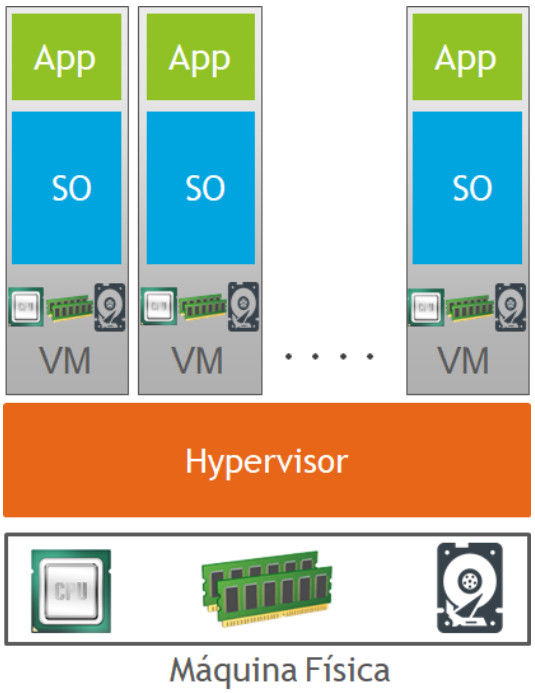
1.2. O Lado Ruim das Máquinas Virtuais

O modelo de máquina virtual era insanamente melhor que o modelo anterior. Aquele em que todas as aplicações precisavam de sua própria máquina física. Na verdade, era muito melhor até certo ponto, mas de todo modo no início nós adorávamos essa revolucionária mudança. As máquinas virtuais eram usadas exageradamente. Como aquelas vodcas vagabundas em festas de faculdade. Na manhã seguinte bate aquela ressaca quase mortal e percebemos que não foi uma boa idéia. Do mesmo modo aos poucos fomos começando a perceber que o modelo máquinas virtuais na verdade estava se tornando um monstro feio e desagradável com o qual era bem difícil conviver. Vamos apenas retroceder um pouco ao fato de que isso Tem tudo a ver com aplicações. Então isso significa que é tão legal quanto os sistemas operacionais.

Na teoria de sistemas operacionais aprendemos que na realidade o sistema operacional só existe para facilitar a execução da aplicação ou, dito de outra forma. Se pudéssemos criar aplicações sem a necessidade de um sistema operacional, nós certamente faríamos. Nós ficaríamos livres e assumiríamos o comando, mas antes veja o modelo das máquinas virtuais. **Tem tudo a ver com sistemas operacionais** certo? Bom, mas, **tem tudo a ver com aplicações também**. Esse modelo é fundamentalmente uma aplicação por máquina virtual em que cada máquina virtual requer um sistema operacional completo.

Na prática o que realmente fizemos com o modelo de máquinas virtuais foi cortar o excesso de camadas físicas. Agora podíamos executar 10 aplicações em uma única máquina física e isso era magico, mas ainda precisávamos de 10 sistemas operacionais para essas 10 aplicações. Logo não é tão mágico quanto nos pareceu à princípio. Para chegarmos a essa conclusão basta nos lembrarmos de que cada sistema operacional traz consigo um monte de custos adicionais.

Figura 3 – A Virtualização de Hypervisor e o Desperdício de Recursos



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Como mostra a **Figura 3** todo sistema operacional consome também CPU, memória RAM e claro espaço em disco. O modelo de máquinas virtuais não faz nada para ajudar com isso e além do mais não podemos nos esquecer que alguns sistemas operacionais precisam de uma licença por instalação. Você certamente já percebeu o tamanho da enrascada. Vou ser um pouco mais claro *"mais sistemas operacionais não significam mais valor de negócio"*. Não existem empresas por aí se gabando sobre o enorme número de horas de

instalações de sistemas operacionais que eles fazem todos os meses. Na verdade, se existe algo a se vangloriar certamente será o oposto disso.

Eu posso estar parecendo um pouco extremista, pois o modelo de máquinas virtuais é melhor que o modelo físico. É mais eficiente e é bem seguro. Cada máquina virtual ou sistema operacional é um limite de segurança. Entretanto as despesas gerais do modelo simplesmente não podem ser negligenciadas em uma avaliação séria e imparcial. Cada instância do sistema operacional é um desperdício grosseiro de recursos. A evolução e a seleção natural não permitem que isso persista, não importa quão grande e quão rica seja a organização.

Realmente tem tudo a ver com aplicações! E tudo o que uma aplicação precisa para executar é um ambiente isolado, seguro e com serviços mínimos do Sistema Operacional e provavelmente, alguns controles de qualidade de serviço. A máquina virtual nos fornece exatamente isso, mas os custos são simplesmente altos demais e quando os custos ficam muito altos as pessoas buscam por alternativas.

Você poderá demorar um pouco para entender isso, especialmente se você trabalha com máquinas virtuais. As máquinas virtuais são literalmente o elefante na sala. Com tudo isso em mente, creio que fosse inevitável que algo mais eficiente viesse e substituísse as máquinas virtuais. A *"economia da natureza"* e da evolução parecem ditar que as soluções mais eficientes substituem as soluções menos eficientes. E esse próximo passo na evolução da execução de aplicações e que atualmente é a solução mais eficiente é o contêiner.

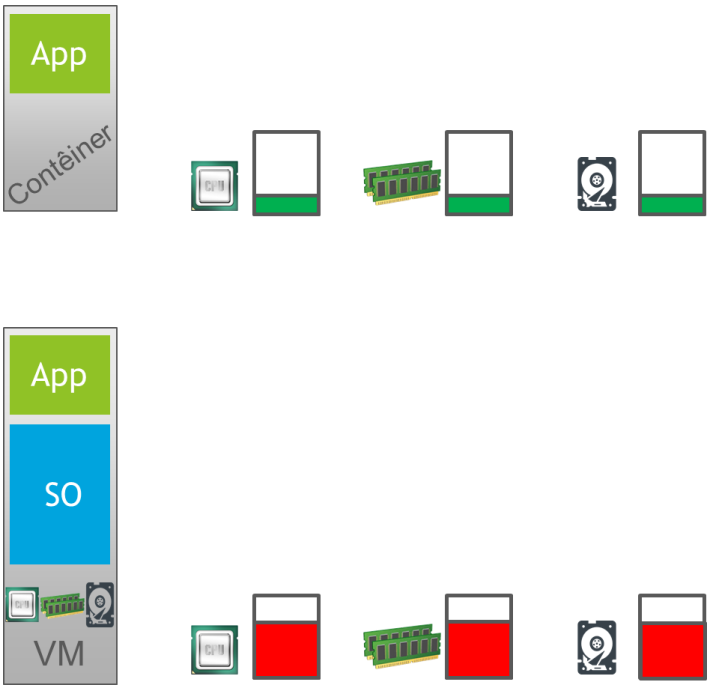
1.3. O que são Contêineres

Em uma abordagem superficial, podemos dizer que os contêineres são um pouco similares às máquinas virtuais. Principalmente se levarmos em conta que eles são um ambiente de execução de aplicações. Então, nós executamos aplicações dentro deles. Na prática o verdadeiro ponto de partida ao comparar contêineres a máquinas virtuais é que os contêineres são muito mais leves do que as máquinas virtuais. Conforme mostra a **Figura 4**, cada contêiner consome menos CPU, menos memória RAM e menos espaço em disco do que uma máquina virtual. Além disso eles fornecem um ambiente de execução isolado e seguro para implantarmos nossas aplicações. Ou seja, tem alguma semelhança com as máquinas virtuais, mas são muito mais leves.

Agora vamos ver com mais detalhes como os contêineres Linux funcionam. Vamos começar pela máquina física, temos CPU, memória RAM e armazenamento em disco. Claro algum sistema operacional instalado. Como o Docker foi concebido originalmente no Linux, então, para nós, o sistema

operacional é o Linux. Como instalamos o Linux no hardware, o kernel Linux controla e gerencia o hardware sob ele.

Figura 4 – Contêineres são Muito mais Leves do que as Máquinas Virtuais



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

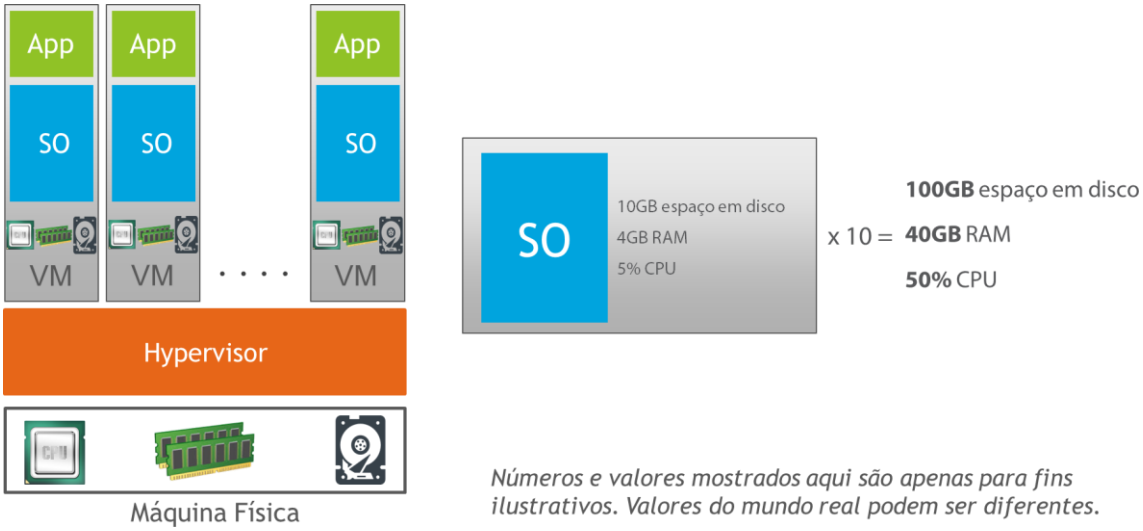
Em um mundo sem contêineres, nós instalamos nossas aplicações ou serviços sobre o Linux. Em uma estrutura que chamamos de espaço do usuário ou *user space*. E é assim que as coisas se pareciam antes da invenção do contêiner ou da máquina virtual. Na prática o que os contêineres nos permitem é criar várias instâncias isoladas do user space. Basicamente podemos dizer que cada instância isolada do espaço do usuário é um contêiner. Logo, dentro de cada contêiner, podemos instalar uma aplicação ou um serviço. Simples.

Em outra terminologia, que podemos ouvir se referem aos contêineres por virtualização de contêiner ou virtualização no nível do sistema operacional. Normalmente utilizam essa abordagem para diferenciá-lo da virtualização do hypervisor. De qualquer forma, na minha opinião podemos dizer que os contêineres são conceitualmente similares às máquinas virtuais, mas possuem uma enorme vantagem por serem mais leves que as máquinas virtuais.

E como isso funciona? Conforme ilustrado pela **Figura 5** no modelo de máquinas virtuais dentro de cada máquina virtual precisamos instalar um sistema com uma versão completa de um sistema operacional como o Linux ou o Windows. Então, criaremos 10 máquinas virtuais no nosso servidor físico. Serão 10 instalações completas do Linux ou do Windows. Digamos que cada um desses 10 sistemas operacionais requer 10 gigas de espaço em disco, 4 gigas de memória RAM e 5 por cento de CPU. Bem, isso nos custará um espaço de disco de 100 gigabytes. Uma vez que estamos assumindo que no nosso

exemplo são 10 máquinas virtuais. Então, 100 gigabytes de espaço em disco, 40 gigas de RAM e 50% de CPU. E tudo isso, antes mesmo de instalarmos uma única aplicação.

Figura 5 – O Modelo de Máquinas Virtuais



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Bem, os contêineres são fundamentalmente diferentes. Cada contêiner é um sistema operacional superleve, nada parecido com o sistema operacional completo que temos dentro de cada máquina virtual. Por exemplo, cada contêiner compartilha um único kernel Linux em comum, portanto, apenas uma única instância de um kernel da máquina hospedeira. Mesmo que tenhamos vários contêineres rodando o resultado é positivo. Cada contêiner consome menos memória RAM, menos CPU e menos espaço em disco do que uma máquina virtual. Além disso os contêineres também são mais rápidos e possuem maior portabilidade que as máquinas virtuais. Talvez isso soe bom demais para ser verdade. Mas acredite nós ainda estamos fazendo uma avaliação superficial.

1.4. Contêineres Vistos de Dentro

Antes de continuarmos vamos nos aprofundar um pouco no Linux, mas apenas o suficiente para entendermos como os contêineres foram construídos e como operam. Pra começo de conversa acredito que preciso responder duas perguntas bem rápidas. Que na verdade já pode estar martelando aí na sua cabeça. *Por que precisamos de contêineres? Por que não instalamos cinco aplicações diferentes em uma mesma instalação normal do Linux ou do Windows?* Bem a resposta é mais complexa, mas eu vou dar um único exemplo que ilustra muito bem um dos motivos. Sistemas operacionais como o Linux e

o Windows simplesmente não são muito bons em isolar múltiplas aplicações e impedi-las de se sobreporem umas às outras. Como ilustrado na **Figura 6**, imagine duas aplicações instaladas no mesmo sistema operacional ambas usam um arquivo de biblioteca compartilhado. Mas cada uma dessas aplicações precisa de uma versão diferente desse arquivo. Logicamente isso é um desastre esperando para acontecer. Nós simplesmente não temos uma maneira segura de fazer algo tão simples quanto ter duas versões diferentes do mesmo arquivo no mesmo sistema. Existem outras razões também, mas esse é um ótimo exemplo de algo que os contêineres absolutamente resolveram para nós.

Figura 6 – O Compartilhamento de Bibliotecas

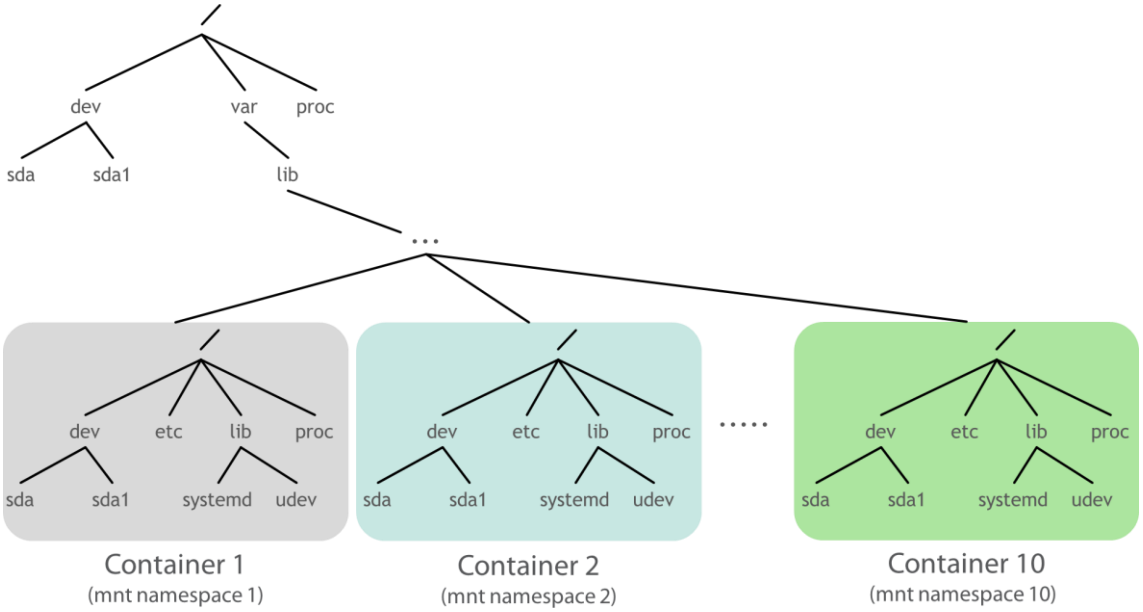
Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Voltando aos contêineres. Dizíamos que um contêiner é uma instância isolada do user space. Portanto, um sistema com 10 contêineres possui efetivamente 10 instâncias isoladas e independentes do user space. Isso significa que podemos instalar com segurança 10 aplicações nesse sistema. Bem, vamos ver como esses contêineres ou instâncias isoladas do user space são criados. Em primeiro lugar, precisamos ser capazes de criar instâncias isoladas de coisas como sistemas de arquivos, árvores de processo, definições de rede, coisas assim.

Como ilustrado pela **Figura 7**, em uma visão isolada do sistema de arquivos podemos ver que um contêiner tem seu próprio sistema de arquivos seu próprio etc, dev, proc, var etc. Sim 10 contêineres significam 10 visualizações isoladas independentes do sistema de arquivos. Uma aplicação em execução dentro do contêiner pode incluir, remover, modificar arquivos em qualquer lugar dentro da sua *"visão"* do sistema de arquivos, sem afetar em

nada as aplicações em outros contêineres. Deixando um pouco mais claro, cada contêiner tem sua própria visão do sistema de arquivos para que ele possa manipulá-lo sem afetar nenhum outro contêiner.

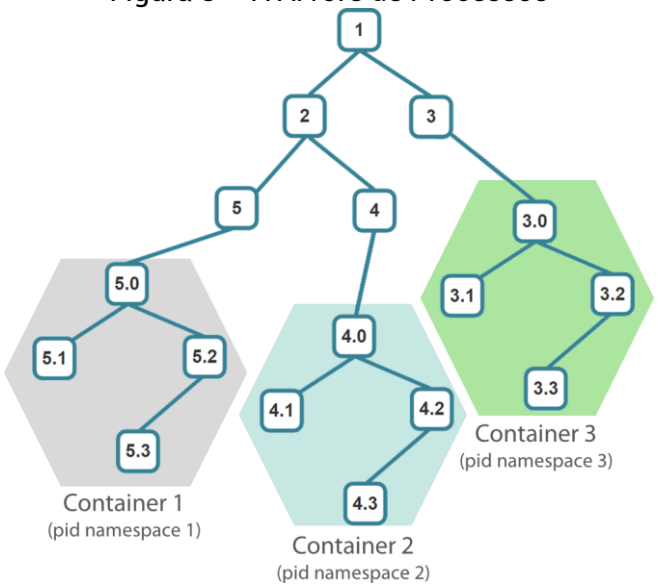
Figura 7 – O Sistemas de Arquivos



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

O mesmo vale para a visão independente da árvore de processos ilustrada pela Figura 8. Cada Contêiner possui seu próprio PID 0 assim como seu próprio System D. Então, cada contêiner possui a sua própria hierarquia de processos independente e isolada. De novo 10 contêineres, 10 árvores de processos isoladas e independentes. Isto significa que um processo dentro de um contêiner não pode enviar um sinal para um processo dentro de outro contêiner. É o isolamento.

Figura 8 – A Árvore de Processos



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Sim, e adivinhe. O mesmo vale para a rede. Cada contêiner obtém sua própria visão ou sua própria stack de rede, de modo que seus próprios endereços IP e portas alcancem sua tabela root.

Mas aí vem aquelas perguntas: *como construímos essas coisas? Como nós realmente realizamos esse isolamento?* Bem, esse tipo de isolamento é conseguido através de um recurso do *kernel* chamado *namespaces*. Os *namespaces* do *kernel* nos permitem fazer exatamente o que acabei de explicar. O sistema operacional particiona a árvore de processos em namespaces e atribui cada uma dessas partições a um contêiner específico. E claro uma partição é isolada e só pode ser acessada pelo Contêiner pai. Há uma única visão da árvore de processos do host, mas cada uma dessas partições é atribuída a um contêiner específico. Os contêineres funcionam como *"paredes"* que impedem que os processos de dentro do contêiner conheçam ou sejam capazes de acessar processos fora do contêiner.

Os namespaces de kernel do Linux nos permitem particionar vários aspectos do sistema, incluindo, mas não se limitando, a árvore de processos, às configurações de rede, os pontos de montagem e claro o namespaces do usuário. Na verdade, o namespace de usuário nos permite ter contas de usuário com privilégio de root dentro de um contêiner, mas não fora dele. Podemos dizer que o namespace é um componente vital para a construção de bons contêineres. Outro recurso de kernel usado pelos contêineres são os *C Groups* ou grupos de controle.

Eu ia dizer que eles são menos importantes ou mais legais do que os namespaces, mas isso não é verdade. Pelo menos não é verdade em um sistema de contêineres decente. Em uma visão superficial, não irei me aprofundar muito por que isso diz mais respeito a Linux que Docker, basicamente os C Groups agrupam recursos e aplicam limites como regras de qualidade de serviço. No caso dos contêineres, nós mapeamos contêineres para C Groups em um mapeamento de um para um. Na prática, um C Group equivale a um contêiner. Eles nos permitem estabelecer limites para a memória, o CPU, *block IO* dentre outras coisas que o contêiner tem acesso. Os C Group são realmente flexíveis. Se precisarmos aumentar a quantidade de recursos do sistema que um contêiner pode usar basta ajustar os limites do C Group. Honestamente, eles são potencialmente mais flexíveis do que uma CPU ou memória RAM virtual executando sobre um hypervisor.

Podemos flexibilizar a utilização de recursos para mais e para menos em um C Group de forma realmente fácil. Além disso obviamente, eles nos ajudam a garantir que nenhum contêiner possa rodar de forma descontrolada em um sistema e impactar qualquer outro contêiner. Estritamente falando, os C Groups não são um requisito complicado para os contêineres. Mas se você planeja executar vários contêineres em um sistema como todos nós fazemos. E especialmente se esses contêineres tiverem cargas de trabalho pesadas, então

você realmente vai querer um sistema de contêiner que suporte *C Groups* como o *Docker*.

Uma última feature do kernel que merece menção *Capabilities*. Novamente em uma visão superficial *Capabilities* nos fornece um controle refinado sobre quais privilégios um usuário ou processo obtém. Então, ao invés da abordagem tudo ou nada de conceder acesso de root ou acesso de não root, onde root é todo poderoso e a não root é praticamente desprovido de quase todos os privilégios. Bem, Capabilities obtém os privilégios do usuário root e os dividem em privilégios menores, e então podemos atribuir esses privilégios menores, quando precisarmos deles.

Claro que isso é uma visão superficial. Suponha que tenhamos um processo que precisa fazer o *bind* de uma porta em um contêiner, mas além disso, não precisa de nenhum outro privilégio especial. Em vez de conceder permissão de root e dar a ele muito mais privilégio do que o necessário, podemos conceder-lhe a permissão *CAP_NET_BIND_SERVICE* que na prática lhe permite fazer apenas isso. Isso certamente é muito mais seguro do que conceder permissão de root. Capabilities são importantes do ponto de vista da segurança. O Docker obviamente suporta Capabilities e de fato, opera uma abordagem de White list para as Capabilities. Todas as Capabilities são negadas por padrão, exceto aquelas explicitamente listados na White list.

Esses são os fundamentos básicos de como os contêineres operam "por baixo dos panos". Como você percebeu a maior parte do que dissemos até agora está relacionado aos contêineres em geral.

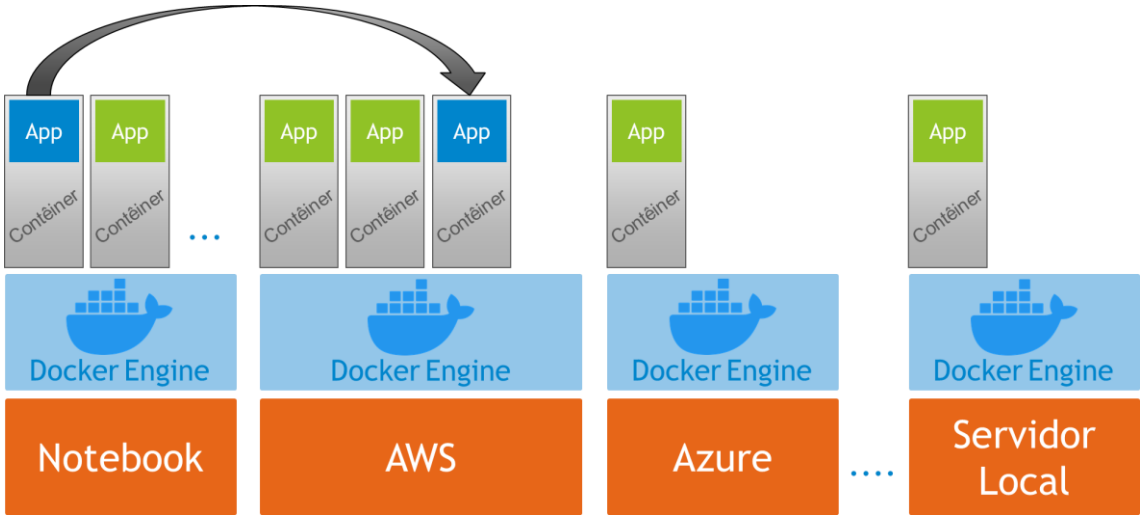
1.5. O Docker

Agora estamos prontos para finalmente começar a falar sobre o Docker. O Docker é tanto uma empresa quanto uma tecnologia ou uma tecnologia/plataforma. No lado tecnológico podemos dizer que o Docker tem crescido muito ultimamente. Mas em essência o núcleo Docker é o Docker contêiner Runtime. Se pensarmos em tudo o que vimos anteriormente sobre contêineres como genéricos e como uma espécie de *framework*, o Docker é uma implementação real de uma tecnologia de contêineres. Ele reúne kernel namespaces, C Groups, capabilities todas essas coisas em um único produto.

Eu não tenho muita certeza se esta é uma boa analogia, mas de todo modo eu vou *"forçar um pouco a barra"* e tentar. O Docker é para os contêineres o que a Red Hat, o CentOS ou o Ubuntu é para o kernel do Linux. É uma implementação pronta e empacotada para a qual você pode obter suporte. A idéia com essa analogia é "desenhar" um quadro geral de modo que todos consigam ligar os pontos.

Uma das belezas do Docker é que ele fornece um ambiente de execução uniforme e padrão. Entenda ambiente de execução, como um ambiente com todas as coisas que uma aplicação precisa para executar como dispositivos, de sistemas de arquivos, privilégios, variáveis, funções e API's. Como ilustrado na Figura 9 o Docker fornece um ambiente de execução padrão, o que significa que os desenvolvedores podem codificar aplicações em contêineres Docker em seus notebooks e, literalmente, subi-los e baixá-los diretamente nos contêineres Docker. E isso pode ser aplicado por exemplo na Amazon AWS, Microsoft Azure ou mesmo em um servidor local. Literalmente, não importa onde, desde que no ambiente de destino esteja rodando o Docker Runtime ou o Docker daemon.

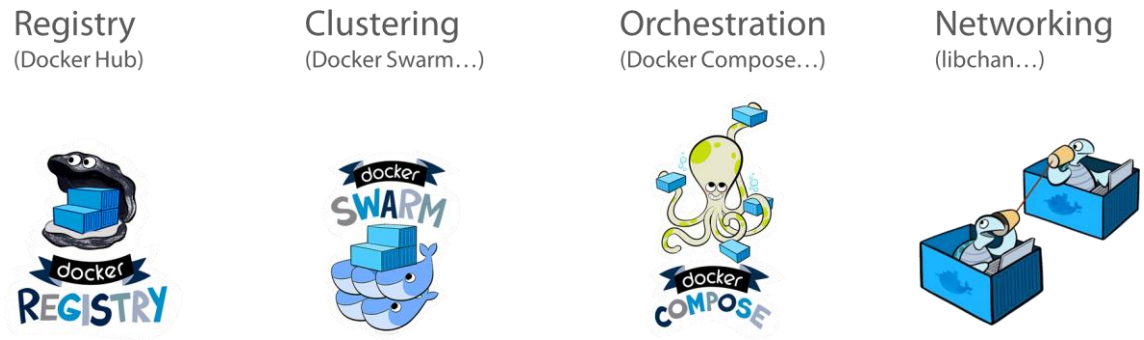
Figura 9 – O Docker e o Ambiente de Execução Padrão



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Honestamente é ridículo de tão fácil nunca ouve nada parecido antes. Forçando mais uma vez as analogias podemos dizer que é muito parecido com a criação de uma aplicação para Android. A aplicação irá funcionar em qualquer dispositivo, telefone ou tablet cujo sistema operacional é o Android. Esse tipo de portabilidade de aplicações é incrivelmente legal para o desenvolvimento de aplicações.

Figura 10 – O Docker e o Ambiente de Execução Padrão



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

O Docker é open source e está crescendo e se tornando bem mais do que apenas o Docker contêiner Runtime está se tornando uma plataforma. Conforme podemos ver na Figura 10 temos um formato padrão das *Docker images*, o Docker contêiner Runtime que já mencionamos, o Docker registry, clusterização, service Discovery, overlay networking, um monte de coisas legais. Além de estar evoluindo como uma tecnologia o Docker também é uma empresa. Anteriormente conhecida como Dot Cloud, agora renomeada para Docker Ink.

A Dot Cloud nasceu como uma plataforma as a service, portanto. Essa pode ser parte da razão pela qual a base de código e produto Docker estarem evoluindo como uma plataforma e não simplesmente como o Docker contêiner Runtime. De todo modo a tecnologia que hoje conhecemos como Docker era um projeto interno na Dot Cloud, liderado por um cara chamado Solomon Hikes (Figura 11). Solomon Hikes é como se fosse o Linus Torvalds do Docker, sim, mais uma analogia meio tosca. O Docker foi escrito na linguagem Go do Google e está licenciado como uma tecnologia open source sob a licença da Apache 2.

Figura 11 – Solomon Hikes o Criador do Docker

Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Daqui em diante, iremos nos concentrar no Docker como uma tecnologia. Entraremos em contato com os principais componentes do Docker como o Docker contêiner runtime, o *client*, o *daemon* além disso exploraremos todos os detalhes de como as imagens e os contêineres funcionam.

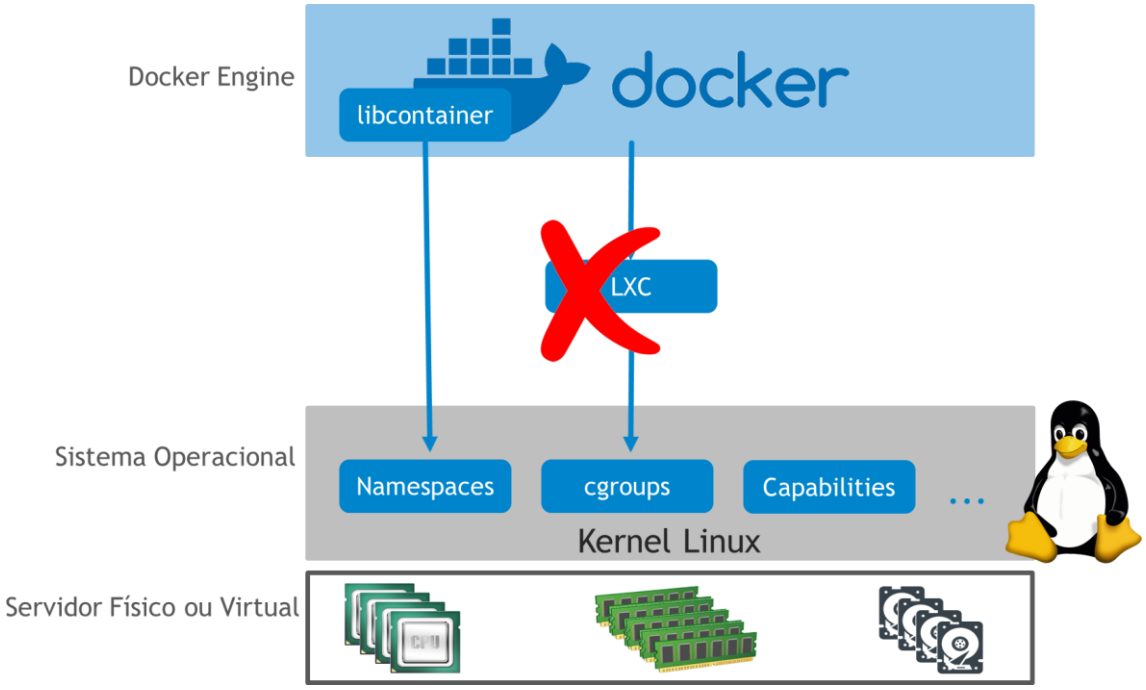
Vale mencionar brevemente também, porque sei que é uma fonte de confusão para muitas pessoas. *Qual é a diferença entre o Docker e o LXC?* Originalmente o Docker usava o LXC como seu driver de execução padrão, como uma interface para manipular os recursos do kernel que mencionamos anteriormente *namespaces*, *C Groups* e *capabilities*. O pessoal do Docker costumava confiar e fomentar o desenvolvimento do LXC, mas essa abordagem começou a causar alguns problemas que afetavam a evolução do Docker. Pois isso significava que os usuários tinham que se certificar de que a versão do LXC em sua distribuição Linux específica estivesse em dia e funcionasse

corretamente com o Docker. Isso não era necessariamente um problema, mas era algo que poderia eventualmente te pregar uma peça e claro os desenvolvedores do Docker não gostavam dessa idéia.

Um segundo problema, só que dessa esse ainda maior era que o pessoal do Docker não controlava o desenvolvimento do LXC e isso é um grande problema. Resumindo, confiar em algo fora do seu controle como um componente central da sua solução. Bem todos nós temos que concordar que isso não é uma boa idéia. Então os caras do Docker decidiram escrever e abrir o código de um novo driver de execução chamado *libcontainer*.

Conforme ilustrado pela Figura 12 na prática o *libcontainer* é um substituto o LXC no que diz respeito ao Docker. Portanto, o libcontainer fornece acesso direto aos importantes recursos de kernel. A grande diferença agora é que o libcontainer é diretamente controlado pelo pessoal do Docker. Enfim eles têm um maior controle das coisas. O que significa que eles podem ajustar o que precisam mais rapidamente e inclusive podem empacota-lo junto com o Docker daemon. E isso trouxe um potencial enorme para o Docker uma vez que o *libcontainer* possibilitou que o Docker se torne *cross plataforma*. Isso significa dizer que não apenas executa em sistemas operacionais Unix, mas também no Windows. Quer você ame ou odeie a Microsoft isso dá um enorme potencial ao Docker como uma plataforma e um ecossistema.

Figura 12 – O Docker e a Substituição do LXC pelo Libcontainer



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

De todo modo o libcontainer é agora o driver de execução padrão nos sistemas Docker. No entanto, se você realmente quiser, é possível substituir o

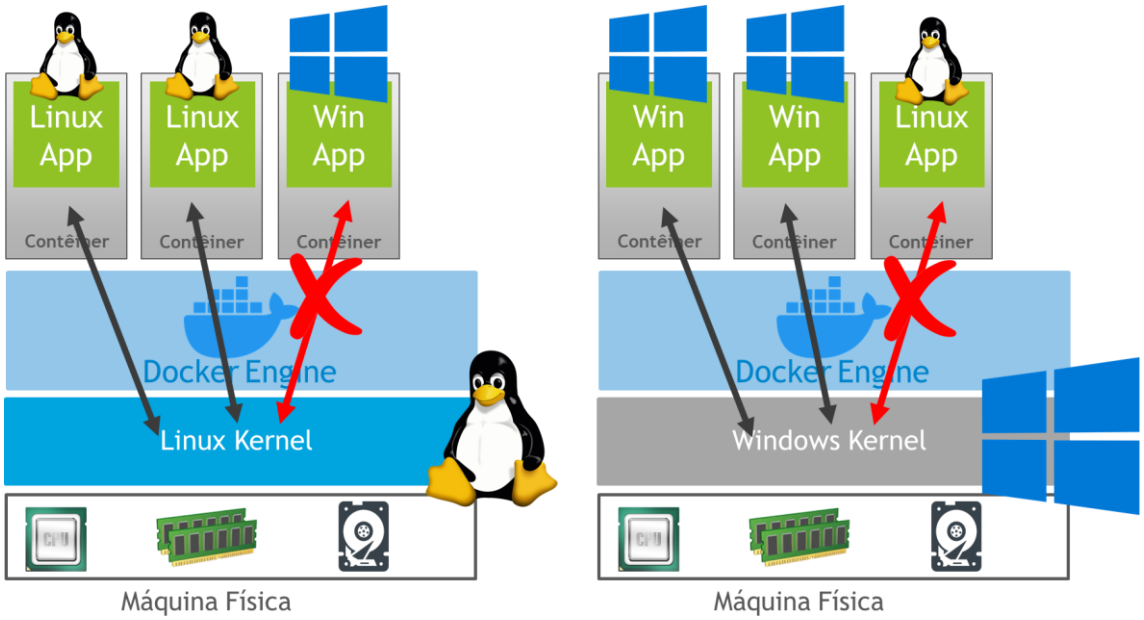
libcontainer e forçar o uso do LXC, mas honestamente essa não é uma prática recomendada atualmente.

1.6. O Docker no Windows e a Compatibilidade no Windows e no Linux

Apesar de nesse livro focarmos na implementação do docker para Linux é importante que você saiba de alguns detalhes do Docker no Windows. O Primeiro suporte ao Docker no Windows se deu pelo Docker Toolbox, hoje obsoleto e substituído pelo Docker Desktop. Ambos, rodavam sobre máquinas virtuais com suporte via Virtual Box e Hyper-V, respectivamente.

Conforme já vimos o Docker fornece um ambiente de execução isolado e padronizado, e os desenvolvedores podem escrever código em contêineres Docker em seus notebooks e então pegar esse código e executá-lo sem nenhuma modificação em uma máquina virtual na Amazon AWS ou na Microsoft Azure ou mesmo servidor Linux interno, desde que o ambiente de destino tenha o Docker contêiner Runtime. Diante disso você deve estar pensando: *"será que um contêiner Windows roda no Linux"?* A resposta para essa pergunta é: *"ainda não"*. Isso acontece por quê cada contêiner compartilha um kernel comum com o kernel do sistema operacional hospedeiro.

Figura 13 – A Compatibilidade entre Contêineres e Hosts



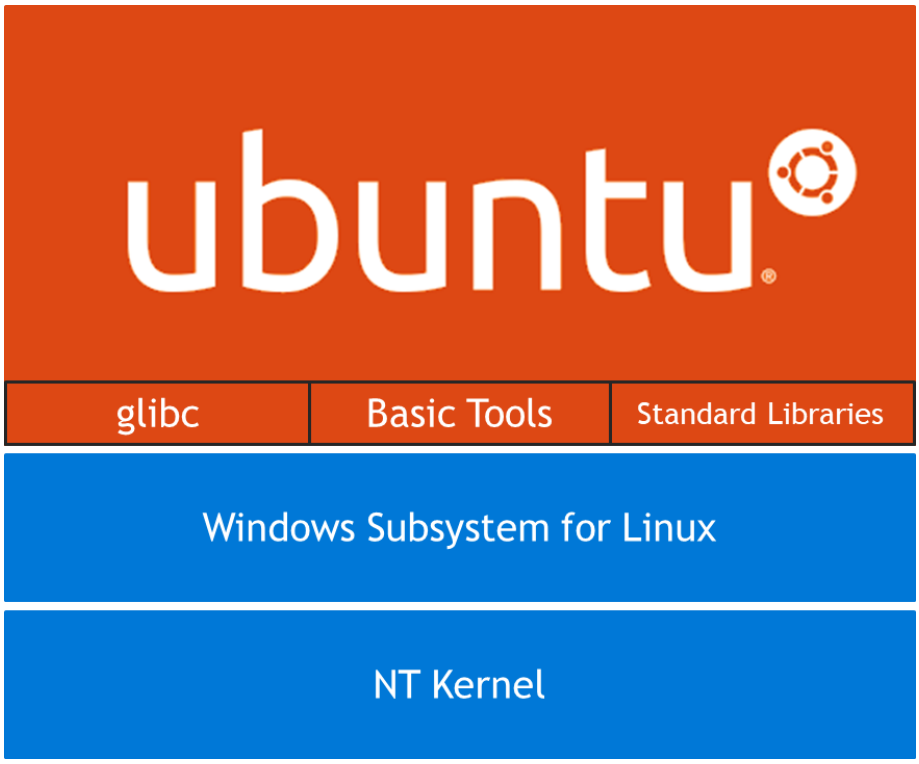
Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Hoje quando você instala o Docker no Windows você pode escolher se ele irá usar o kernel do Windows ou o kernel do Linux. Isso é possível graças ao Hypervisor que possibilita a compatibilidade com o Linux. Portanto conforme a Figura 13, um contêiner em execução em uma máquina host Linux enxerga e usa o kernel Linux da máquina hospedeira Linux. E isso significa que as

aplicações do Windows desenvolvidos em contêineres do Windows só serão executadas em hospedeiros Windows e o mesmo vale para os contêineres do Linux. Exceto se o Docker no Windows estiver executando sobre o Hypervisor em modo de compatibilidade.

Só pra reforçar. Sua aplicação com Docker engine Windows só irá executar no Windows. Se a sua Docker engine for Linux ou estiver em modo compatibilidade com o Linux então ela será executada nas duas plataformas. Pouco a pouco essa incompatibilidade tende a desaparecer ou ser relativizada visto que desde 2016 a Microsoft juntamente com a Canonical vem desenvolvendo o Windows Subystem for Linux ou WSL. Essa tecnologia permite rodar um Kernel Linux direto no Windows, sem recursos de virtualização. E isso irá impactar diretamente como os contêineres Docker rodarão no Windows 10.

Figura 14 – A Arquitetura do WSL2



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Conforme a Figura 14 no começo o WSL oferecia uma camada de compatibilidade para gerar binários executáveis do Linux “nativamente” no Windows 10. Fornecendo uma interface de núcleo compatível com o kernel Linux, ela usava bibliotecas do Kernel Windows (ou seja, sem nenhum código Linux). Mesmo assim, era impossível, por exemplo, executar a Docker engine e o Kubernetes diretamente no WSL. Pra resolver isso, a equipe de desenvolvimento do Docker implementou uma solução alternativa usando o

Hyper-V e o LinuxKit para obter a perfeita integração. Essa é a solução utilizada atualmente e é conhecida como Docker Desktop.

Em junho de 2019, foi lançado WSL2 na versão Preview Build do programa Windows Insider. Ao invés de usar a emulação, o WSL2 usa realmente um kernel Linux rodando dentro de uma VM leve. Embora o WSL2 use tecnologia de virtualização, ele não é uma máquina virtual. Mas, "por baixo do capô", será gerenciado e executado como tal sem intervenção direta do usuário. Para a equipe de desenvolvimento do Docker, essa abordagem é arquitetonicamente muito próxima do que é feito com o LinuxKit e o Hyper-V hoje, com o benefício adicional de ser mais leve e melhor integrado ao Windows. Para eles, o WSL2 vai facilitar a interoperabilidade do Docker com o Windows, deixar o desenvolvimento e a execução de Contêineres Docker mais rápidos.

Atualmente o WSL2 ainda está em preview, mas no futuro a equipe de desenvolvimento do Docker já planeja substituir a VM do Hyper-V, suportado no Docker Desktop, por um pacote de integração que fornecerá os recursos já existentes, mas com suporte nativo ao WSL2. Este pacote de integração conterá os componentes necessários para executar o Docker e o Kubernetes, bem como as ferramentas client usadas para interagir com esses componentes no WSL2.

1.7. O Futuro do Docker e dos Contêineres

Esse é o momento de tentarmos prever o futuro do Docker. Creio que inicialmente, muitas pessoas irão rodar contêineres Docker encima de máquinas virtuais. Faz sentido em ambientes de estudo, mas acredite isso não acontece apenas em ambientes de estudo. Serviços de nuvem, como a Amazon Web Services ou a Microsoft Azure, estão inicialmente disponibilizando contêineres executando sobre máquinas virtuais. No caso da Amazon o Docker executa sobre instancias EC2. Na minha opinião no futuro quase certamente as pessoas irão querer cortar custos com máquinas virtuais.

E certamente irão executar o Linux no hardware do servidor e, em seguida, apenas colocará os contêineres executando sobre o Linux. O mesmo vale para servidores Windows. Desse modo, ganhamos em eficiência e cortamos toda a gordura que era exigida pelas máquinas virtuais. Eu também creio que poderemos ver fabricantes de chips como a Intel, a AMD e outros desenvolvendo processadores especiais para oferecer uma melhor assistência de hardware para contêineres. Pense em tecnologias como Intel VT-X e VT-D. Uma vez que havia demanda de mercado suficiente, os fabricantes de processadores chegaram ao mercado com essas assistências de hardware que agregaram valor ao mundo das máquinas virtuais. Nesse sentido, é totalmente concebível que o mesmo possa acontecer em ambientes de contêineres.

Poderemos ver extensões em nível de chip que melhorem o desempenho e a segurança dos contêineres.

O Docker e os contêineres em geral também tendem a funcionar melhor e, na verdade, eles realmente estimulam as arquiteturas de microsserviços. Pra ser honesto, nós já estamos vendo essa tendência no dia a dia. Essas são arquiteturas de aplicações em que as aplicações são projetadas como um conjunto de serviços menores que operam coletivamente para formar a aplicação. Cada um desses Serviços de Componentes menores é executado como um único processo dentro de seu próprio contêiner. Esses contêineres e processos comunicam entre si para formar a aplicação geral que engloba cada componente de serviço em execução em seu próprio contêiner. Cada um desses componentes de serviço pode ser atualizado individualmente, independentemente de todos os outros componentes da aplicação em geral.

Esse tipo de projeto de aplicação parece ser definitivamente o futuro para aplicações em escala web, em geral são arquitetonicamente superiores às aplicações monolíticas em que a atualização de qualquer componente é um grande trabalho que geralmente envolve muitos riscos e um tempo considerável de inatividade. Por isso, eu acredito que veremos mais e mais arquiteturas de aplicações de microsserviços e o Docker e os contêineres são perfeitos para esse tipo de arquitetura de aplicações da próxima geração.

Mencionamos brevemente sobre o armazenamento em cluster, service Discovery e orquestração de contêineres. Esse tipo de tecnologia já existe e está em franca expansão e assim como os microsserviços se encaixa como uma luva no mundo Docker. No momento em que esse livro está sendo escrito, essas tecnologias já estão quase ou tão maduros quanto o próprio Docker e a médio prazo contribuirão ainda mais para o sucesso do Docker. Talvez eu venha a fazer cursos adicionais sobre algumas dessas tecnologias em breve.

Um último ponto que eu acho que deve se popularizar são os *Docker Registries* que como veremos são similares ao Github só que para *Docker images*. Inicialmente só existia o Docker Hub depois outras empresas como a Amazon AWS começaram a disponibilizar serviços com o mesmo propósito como é o caso do Amazon ECR. Acredito que acontecerá um fenômeno similar ao que aconteceu após o surgimento do GIT em que apareceu o Github, o Bitbucket, o Gitlab e tantos outros.

Essa é a nossa breve e superficial introdução ao Docker e aos contêineres e seus conceitos principais. A idéia aqui é te preparar para o restante do livro, onde veremos todas essas coisas em ação e detalharemos tudo de forma mais aprofundada.

1.8. Considerações Finais

Esse foi um capítulo essencialmente teórico e nele aprendemos como era antes do Docker e como funcionam as máquinas virtuais e a diferença destas para os contêineres. Conhecemos todos os detalhes de como funciona cada um dos componentes do Docker. Aprendemos como é a *"anatomia do Docker"* e como ele usa os recursos do sistema operacional para isolar e prover um ambiente seguro para nossas aplicações. Em seguida aprendemos como o Docker surgiu, quem o criou dentre outras coisas. Pra fechar falamos sobre a compatibilidade entre contêineres no Windows e no Linux e como ele deve evoluir nos próximos anos. É fundamental ter todos esses conceitos em nossas mentes antes de partirmos para a prática e executarmos comandos no terminal.

2. Instalando o Docker

Atualmente o processo de instalação do Docker é bem simples, mas nem sempre foi assim. Concebido inicialmente no ambiente Linux ele passou a ser gradualmente portado para Windows e Mac. A princípio através do hoje obsoleto Docker Toolbox e hoje via Docker Desktop. Hoje independentemente da plataforma é possível executar o Docker praticamente sem problemas em qualquer plataforma.

2.1. Instalando o Docker no Windows

O processo de instalação no Windows é bem simples basicamente um Next > Next > Next, a única coisa mais complexa é que você precisará habilitar a virtualização na BIOS da sua máquina e isso claro vai variar de uma máquina pra outra. Você pode conferir o guia de instalação completo no link https://docs.docker.com/docker-for-windows/install/. Clicando na imagem abaixo você acessa a um vídeo disponível no meu canal no Youtube com o passo a passo.

2.2. Instalando o Docker no Linux (Ubuntu)

O processo de instalação no Ubuntu também é bem simples assim como no Windows você precisará habilitar a virtualização na BIOS da sua máquina. Dependendo da versão da distribuição precisará remover versões preexistentes do Docker ou do Podman em seguida é só seguir o passo a passo do guia de instalação disponível no link https://docs.docker.com/engine/install/ubuntu/. Nesse link você acessa o guia para todas as principais Distros além do Ubuntu. Assim como pro ambiente Windows clicando na imagem abaixo você acessa a um vídeo disponível no meu canal no Youtube com o passo a passo.

2.3. Validando se o Docker está Funcionando Corretamente

Se você tiver executado o passo a passo então agora é só conferir se o Docker está funcionando corretamente. Para isso podemos executar o comando **docker -v**, conforme o Console 1 isso irá exibir sua versão do Docker e o build corrente.

Console 1 – Saida do comando *docker -v*

```
$ docker -v
Docker version 20.10.6, build 370c289
```

Fonte: AUTOR, 2021

Além desse comando existe o comando **docker info** que podemos ver no Console 2 e dá ainda mais detalhes sobre o nosso ambiente Docker como contêineres em execução, *docker images*, driver de armazenamento, versão do kernel e muito mais.

Console 2 – Saida do comando *docker info*

```
$ docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Build with BuildKit (Docker Inc., v0.5.1-docker)
  compose: Docker Compose (Docker Inc., 2.0.0-beta.1)
  scan: Docker Scan (Docker Inc., v0.8.0)

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 20.10.6
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local
logentries splunk syslog
 Swarm: inactive
 Runtimes: runc io.containerd.runc.v2 io.containerd.runtime.v1.linux
 Default Runtime: runc
 Init Binary: docker-init
```

```
 containerd version: 05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc version: 12644e614e25b05da6fd08a38ffa0cfe1903fdec
 init version: de40ad0
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 5.10.25-linuxkit
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 2
 Total Memory: 1.941GiB
 Name: docker-desktop
 ID: VYTJ:YVBL:SG7Q:MCQH:AIAO:EO5J:7HG6:BSVS:733G:3QYK:MIGM:MPHK
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
```

Fonte: AUTOR, 2021

2.4. Executando o primeiro contêiner

Claro que o nosso teste não estaria completo se não executássemos um contêiner. Para isso ainda no console digite o comando **docker run hello-world** isso irá baixar uma *Docker image* do Docker Hub e iniciar um contêiner bem simples a partir dela. Esse contêiner irá inicializar e imprimir no console uma mensagem validando o ambiente e te encorajando a seguir em frente.

Console 3 – Saida do comando *docker build*

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pulling fs layer
b8dfde127a29: Verifying Checksum
b8dfde127a29: Download complete
b8dfde127a29: Pull complete
Digest:
sha256:9f6ad537c5132bcce57f7a0a20e317228d382c3cd61edae14650eec68b2b345c
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/
```

Fonte: AUTOR, 2021

Se for exibida uma tela similar ao Console 3 significa que o seu ambiente Docker está plenamente operável e você pode começar a fazer coisas mais ousadas.

2.5. Considerações Finais

Esse foi um capítulo bem enxuto e nesse ponto o seu ambiente Docker está pronto. No próximo capítulo aprenderemos quais são os principais componentes do Docker. Se você for mais apressado e quiser ver o Docker em ação enquanto aprende todos os conceitos teóricos clique na imagem ao lado e confira o meu minicurso

Docker para Iniciantes: Minicurso Gratuito Completo!

3. Os Principais Componentes do Docker

Nesse capítulo iremos conhecer todos os principais componentes de um ambiente Docker padrão e não clusterizado. Que é exatamente o que normalmente instalamos nas nossas máquinas. Por mais que à primeira vista, o que veremos aqui, lhe pareça complexo ainda é moderadamente superficial. Uma visão mais aprofundada será feita no próximo capítulo, nesse vamos aprender sobre *Docker engine*, *Docker Images*, Contêineres e também registries e repositórios. Evidentemente falarei de cada uma dessas coisas conforme for se tornando necessário. A idéia geral nesse capítulo é entender como cada uma das peças se encaixam.

3.1. Uma Visão de Alto Nível

Antes de analisarmos diretamente cada componente vamos dar uma olhada para como é o panorama geral do Docker. Vou usar uma analogia que está longe de ser perfeita, mas é a que eu aprendi quando conheci o Docker e se você já fez algum curso à respeito certamente já viu essa analogia. Quando se fala em Docker ou em contêineres em geral a analogia mais usada é comparar as coisas com um porto. E nesta nossa analogia o Docker engine também conhecido como Docker daemon seria o nosso porto. As Imagens são como os *cargo manifests* [1]que é similar a uma nota fiscal e é usado para despachar um contêiner.

No mundo da tecnologia da informação os contêineres são um ambiente isolado no qual podemos executar às nossas aplicações. Como vocês provavelmente já perceberam é análogo ao contêiner físico usado para transportar diversos tipos de cargas. Observando todo esse cenário eu não conheço nenhuma analogia para registries, mas não se preocupe. Como eu disse essa não é uma analogia perfeita. Mas o ponto é que todos esses componentes trabalham juntos para disponibilizar a nossa aplicação dentro de um contêiner. E isso é feito exatamente como o estaleiro de expedição de um porto faz através do *cargo manifest* só que no nosso caso o Docker usa as imagens.

Esses componentes trabalham em conjunto para garantir que a exportação e importação global continue funcionando. O objetivo pelo qual o contêiner é utilizado é para otimizar a exportação e importação. Antes da padronização e do surgimento dos contêineres o processo de exportação e

---

[1] Mais detalhes sobre o que é *cargo manifest* em https://www.marineinsight.com/maritime-law/what-is-a-cargo-manifest-in-shipping/

importação era um verdadeiro pesadelo. Tudo era muito mais caro, arriscado e restrito a um número bem menor de portos. Os navios simplesmente passavam semanas no porto aguardando o lento processo de carga e descarga manual. Além disso os diferentes tipos de cargas muitas vezes representavam risco de vida aos trabalhadores para mover a carga para dentro e para fora dos navios.

Certamente um profissional de segurança do trabalho moderno, não poderia nem sonhar em passar perto de um ambiente desses. Isso sem falar nos impactos à saúde sofridos por quem fazia esse doloroso trabalho. Frequentemente alguns desses profissionais despencavam de grandes alturas, se ferindo ou morrendo no trabalho. E tudo isso sem mencionar, é claro, que as cargas levavam semanas ou anos para ir de um porto A a um porto B. Enfim se você trabalhasse com importação e exportação a sua visão certamente não era nada interessante. Logo, todo o comércio global era de alto custo e de alto risco. Produtos frágeis ou perecíveis poderiam se quebrar ou perder durante o transporte. E é claro os atrasos eram bastante comuns.

Qualquer um olhando pra esse cenário diria que algo precisava ser feito para tirar o comércio global dessa situação. Na tecnologia da informação antes do surgimento dos contêineres a implantação de aplicações em um ambiente de produção também consumia muito tempo e os riscos não eram pequenos. Sem a contêinerização, o tempo necessário para entregar uma aplicação em produção não raramente era no mínimo absurdo. Além disso havia um sério risco de que a aplicação não funcionasse em produção da mesma forma que no ambiente de teste e de desenvolvimento. O clássico comentário "*na minha máquina funciona*" era muito comum. E claro isso não era bom para os negócios. Assim, como no transporte global, algo tinha que ser feito.

Agora que entendemos como as exportações e importações globais funcionam sabemos que os contêineres padronizaram o envio de cargas a nível mundial. Atualmente, nada entra a bordo de um navio, a menos que esteja dentro de um contêiner padronizado. Aos poucos os portos se tornaram verdadeiras maravilhas modernas onde eficiência e segurança reinam absolutas. Comparando com como as coisas costumavam a ser feitas podemos dizer que literalmente que o comércio global sofreu uma verdadeira revolução. Hoje os custos de frete são praticamente insignificantes e o risco de remessa foi removido da equação e tudo acontece mais rápido.

E no mundo das aplicações? Adivinha. O mesmo vale para a implantação de aplicações em produção através de Contêineres Docker. Você codifica uma aplicação em um Contêiner Docker na sua máquina ou no seu notebook e depois a executa inalterada em um Contêiner Docker em qualquer plataforma com suporte ao Docker. Isso permite que as aplicações sejam implantadas em produção de forma incrivelmente rápida e claro as falhas e erros são corrigidas muito mais rapidamente.

O Docker tornou-se literalmente em um divisor de águas para a Tecnologia da Informação. Agora vou dar um *spoiler*, espero que você não fique chateado com ele, a segurança segue mais importante do que nunca. Assim como é fácil ocultar mercadorias maliciosas ou roubadas dentro de um contêiner o mesmo se aplica a Contêineres Docker. Contêineres assim como qualquer outro software podem conter código nocivo e malicioso dentro deles. Tivemos um exemplo recente disso quando descobriram algumas imagens que vinham com código para mineração de criptomoedas de brinde.

Portanto, seja o que estiver fazendo, confie nos contêineres que você executa ou confie nos desenvolvedores do código que será executado em seus contêineres. Por mais que sejam contêineres, eles não são uma desculpa para executar códigos não confiáveis ou não testados em produção.

3.2. O Docker engine

O *Docker engine* também é conhecido como *docker Daemon* ou *runtime*. Esse é o cara que baixamos e instalamos em nossas máquinas. É o programa Docker que instalamos na máquina *host*. Ele é o responsável por prover um ambiente docker e possibilitar o acesso a todos os serviços do Docker. Voltando para a nossa analogia com o porto, o Docker engine é o porto em si. Ou seja, ele é toda a infraestrutura necessária para importar e exportar mercadorias.

Figura 15 – A Infreastrutura do Porto

Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Conforme podemos ver na Figura 15 estamos falando de atracadouros, guindastes de descarga, guindastes de movimentação de contêineres, terminal

ferroviário, pistas para caminhões. Todos os portos atuais possuem essas coisas que constituem o coração do processo de exportação e importação modernos.

No mundo do Docker estamos falando de infraestrutura de aplicações e dependências de runtime, como *stacks* de redes, sistemas de arquivos, rotas, hierarquias de processos, variáveis de acesso ao kernel e sua alocação de recursos. Todas essas coisas. Nos dois cenários a padronização é a chave. Tudo se parece e se comporta exatamente da mesma maneira de uma instalação do Docker para outra. Voltando à analogia do porto, essas seriam as dimensões padrão dos atracadouros. Os guindastes de carga e descarga são padronizados em todos os portos, mesma capacidade, mesmo alcance, mesmo tamanho, mesmos mecanismos de controle simplesmente tudo é padronizado. O mesmo é válido para guindastes sobre rodas e guindastes de descarga todos são padronizados mesma capacidade de carga, mesmos mecanismos de controle, mesmos procedimentos operacionais. Da mesma forma os guindastes sobre trilhos e as empilhadeiras de contêineres também seguem um padrão.

Absolutamente tudo é padronizado e essa padronização traz coisas surpreendentes. Poderíamos mover um administrador de um porto para outro ou um gerente de um pátio de contêineres para outro sem nenhum treinamento e sem familiarização visual com o ambiente. O único pré-requisito é que os dois portos tenham sido construídos e sejam operados com as mesmas especificações mencionadas. E claro analogamente o mesmo se aplica às aplicações escritas em contêineres Docker tudo funcionará exatamente da mesma maneira durante o tempo de execução desde que seja fornecido ambiente com o Docker.

Literalmente, não importa qual seja a plataforma, pode ser o notebook ou a máquina do desenvolvedor, um servidor interno ou em instâncias na nuvem. Contanto que o Contêiner Runtime seja o mesmo, você pode simplesmente executar sua aplicação nele, sem necessidade de nenhuma alteração. Isso é o mais legal do Docker.

Atualmente o Docker se tornou a ferramenta de contêinerização padrão. Nós podemos desenvolver nossas aplicações em contêineres não importa onde, pode ser no nosso notebook nós podemos genuinamente "transportá-lo" para qualquer outro ambiente sem nenhum problema. Desde que o ambiente de destino também esteja executando o Docker Contêiner Runtime a nossa aplicação, funcionará.

Conforme eu já comentei antes é muito similar à desenvolver aplicações para smartphones Android. Nós escrevemos uma aplicação e ele funciona em qualquer dispositivo, celular ou tablet cujo sistema operacional seja o Android. Obviamente, sabemos que existem restrições de versões mínimas suportadas para Android e similares, e claro o mesmo vale para o Docker.

Esse é o *Docker engine* ou *Docker daemon*, como você preferir chamar. Em poucas palavras: é o *"porto"* padronizado ou o ambiente de runtime padronizado que tem sempre o mesmo comportamento, independentemente da plataforma em que está sendo executado. Ele torna a portabilidade da aplicação incrivelmente simples. Bom essa é a idéia geral por trás do Docker engine.

3.3. As Docker Images

Voltando brevemente à nossa analogia, podemos dizer que as imagens, são um pouco similares aos *cargo manifests*. Em uma visão superficial podemos dizer que eles contêm uma lista de tudo o que está em um contêiner além de instruções sobre como montá-lo. As imagens como veremos se parecem muito com isso. No mundo das *Docker images* sempre que queremos inicializar um contêiner, precisamos de uma imagem.

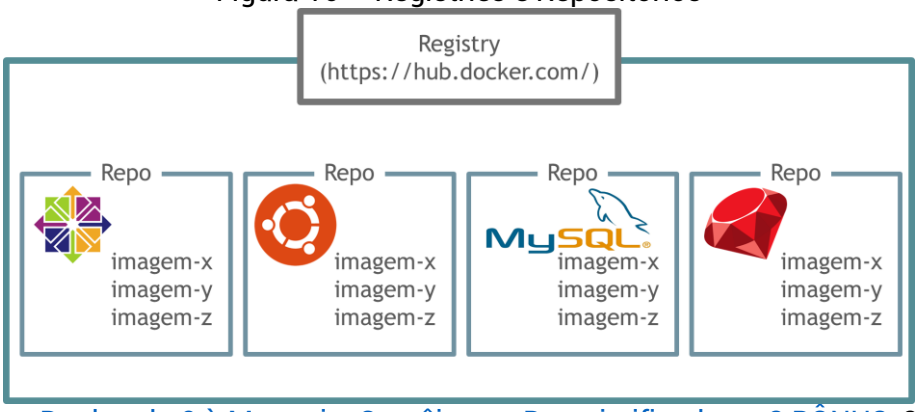
3.4. Os Contêineres Docker

Os contêineres são inicializados a partir de imagens e, como tal, podemos considerá-los como se fossem instâncias em execução de uma imagem. Na verdade, podemos pensar em imagens como se fossem o nosso *build-time* e os contêineres são as imagens em tempo de execução ou runtime.

3.5. Registries e o Docker Hub

Conforme se pode ver na Figura 16 por padrão todas as imagens que usamos são baixadas de repositórios e os repositórios ficam dentro dos registries. O registrie público padrão do Docker é o Docker Hub. Dentro do Docker Hub, existem vários repositórios. Lá encontramos os repositórios oficiais do CentOS, do Ubuntu, do MySQL, do Ruby enfim todas as principais tecnologias.

Figura 16 – Registries e Repositórios



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Como esses são os repositórios oficiais, eles são confiáveis. Eles são mantidos de forma conjunta entre a ***Docker Ink*** e a empresa responsável pela distribuição ou pelo software. Portanto, o repositório do ***Ubuntu*** é mantido em conjunto pela ***Docker Ink*** e pela ***Canonical***. Essa parceria entre eles nos garante que estamos recebendo os releases oficiais em que podemos confiar. Em cada um desses repositórios temos as diferentes versões das imagens no caso do Ubuntu será algo como 20.04, 20.10, 21.04, de fato, todas as imagens suportadas.

3.6. Considerações Finais

Nesse capítulo nós aprendemos sobre o Docker engine. Vimos todos os detalhes de como ele fornece a infraestrutura básica necessária, na máquina hospedeira, para permitir a execução de contêineres. Em outras palavras como ele nos dá acesso às features do kernel. E não importa em que plataforma o executemos pode ser no nosso notebook, em instâncias na nuvem como Azure ou AWS, em um servidor interno, obteremos sempre o mesmo comportamento. Como vimos isso é incrível e torna os nossos contêineres altamente portáteis.

Em seguida, aprendemos sobre as imagens e como elas são a definição estática necessária à inicialização de um contêiner. Elas contêm todos os dados e metadados necessários para iniciar a execução de um contêiner, isso significa que na prática inicializamos todos os nossos contêineres a partir de imagens.

Se executarmos um contêiner a partir de uma imagem baseada no Ubuntu, adivinhem. Teremos um contêiner executando um sistema operacional com os recursos mínimos do Ubuntu. Lembre-se que na prática o Docker usa apenas um minúsculo núcleo do Sistema operacional com no máximo 20 megabytes.

Além disso aprendemos também que as imagens são marcadas para que possamos ter várias versões do mesmo sistema operacional em um único

repositório. Então, teremos diferentes versões do Ubuntu como a 20.04, 20.10, 21.04 enfim diversas versões da mesma imagem.

Aprendemos ainda sobre os contêineres. Vimos que se as imagens são construídas em tempo de compilação, os contêineres são o nosso tempo de execução. Na prática um contêiner é uma instância em execução de uma imagem e é uma máquina Linux. Podemos iniciá-los, pará-los, atachá-los enfim várias coisas comuns aos sistemas operacionais.

Fechamos o capítulo com um overview sobre os registries e os repositórios e fizemos uma rápida introdução ao Docker Hub. Analisamos como funcionam os repositórios open source oficiais e como obter imagens estáveis e confiáveis. O desenvolvimento dessas imagens é uma espécie de parceria entre a equipe do Docker e os funcionários da empresa responsável pela imagem.

Dar uma fuçada em imagens de outras pessoas pode ser interessante, mas tome um cuidado extra, pra ter certeza de que estará executando um código seguro e confiável nos seus contêineres. Recentemente apareceu na mídia casos de imagens com código malicioso usado pra minerar criptomoedas. Enfim é um tipo de software como qualquer outro e sempre que você está mexendo com código não oficial precisa tomar cuidado com esse tipo de risco.

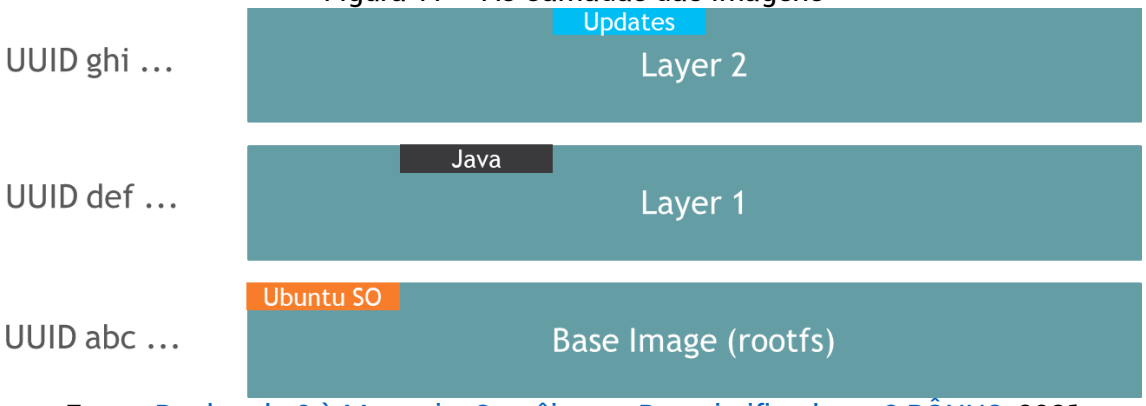
4. Conhecendo Imagens e Contêineres

Agora que já sabemos o básico é hora de aprofundar um pouco mais em imagens e contêineres. Veremos como elas funcionam, as camadas das imagens, os contêineres e muito mais.

4.1. As Camadas das Imagens

Nós conhecemos o básico sobre as imagens. Elas são estáticas e contêm as definições que nossos contêineres terão quando forem iniciados. Acho que mencionei brevemente sobre as imagens terem "camadas empilhadas" então agora é a hora de vermos como isso funciona. Conforme a terminologia sugere as imagens são estruturadas em camadas empilhadas. Na prática uma imagem é formada por uma pilha de outras imagens sobrepostas e interconectadas.

Figura 17 – As Camadas das Imagens



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Na Figura 17, temos três camadas, a camada zero na parte inferior e a camada dois na parte superior. Na terminologia do Docker, às vezes nos referimos a essas camadas como imagens. E isso é uma das coisas meio chatas da terminologia do Docker. Temos três imagens empilhadas umas sobre as outras e juntas elas formam uma única imagem. Bem devemos dizer uma imagem ou três imagens?

Na verdade, acho que poderíamos dizer que temos uma única imagem composta por três imagens em camadas. É meio chato e pode causar confusão, mas de qualquer forma, é assim que funciona.

Na parte inferior nós temos nossa imagem de base, ou melhor a *base image*, é nela que está o nosso sistema de arquivos raiz ou *root file system*. Todos os arquivos e diretórios necessários ao funcionamento de nossos contêineres foram reduzidos ao mínimo. Portanto, se estiver em um contêiner

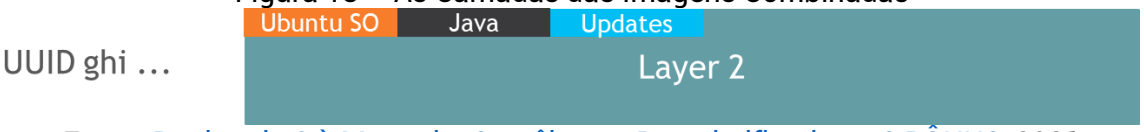
baseado no Ubuntu, todos os arquivos e diretórios necessários para fazer com que o contêiner se comporte exatamente igual a uma máquina com o sistema operacional Ubuntu.

É acima dessa camada que instalamos nossas aplicações como o Java por exemplo. Por fim se for necessário se instalarmos alguns patches ou atualizações então teremos uma terceira camada formando uma imagem com três camadas. A primeira com o sistema operacional base, a segunda para o Java e a última para configurações e atualizações.

Você provavelmente pode estar se perguntando por que três imagens umas sobre as outras. *Por que não apenas uma imagem com tudo dentro dela?* Bom imagine o seguinte cenário escolhemos o Ubuntu 20.10 como *base image* no repositório do Ubuntu no DockerHub. Imagine que fizemos isso seis meses atrás e gostamos do resultado. Então instalamos o Java e usamos por seis meses essa imagem para rodar todos os nossos contêineres Java.

Com o passar do tempo, surgiu a necessidade de algumas customizações e nós ajustamos o contêiner e aplicamos alguns patches e coisas do tipo. Então, obviamente criamos a terceira camada. Conforme podemos ver na Figura 18 acabamos com uma imagem de três camadas. Essa é a nossa *Docker image* com patches e atualizações que usamos para basear todos os nossos contêineres Java. E é assim que as coisas acontecem com bastante frequência. Certamente, existem maneiras de acabarmos com imagens em camadas como essa.

Figura 18 – As Camadas das Imagens Combinadas



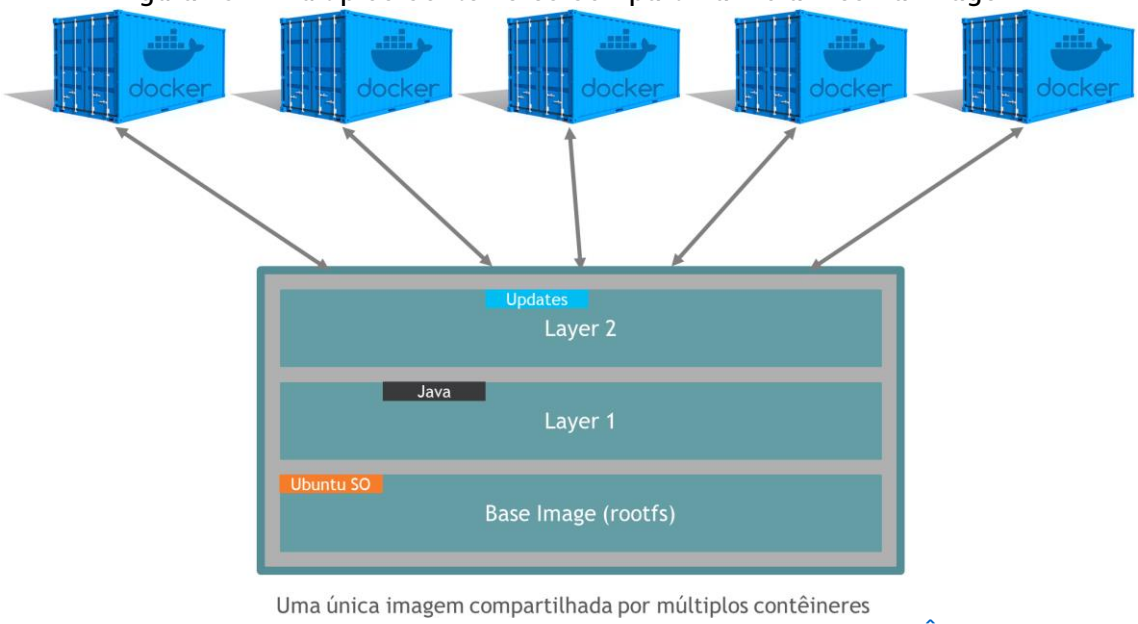
Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Mas saiba que essa abordagem de imagens em camadas é absolutamente poderosa para o Gerenciamento de Conflitos. Ela nos permite manter uma *base image* original limpa. Sabe, aquela imagem primitiva do Ubuntu feita sob medida pelos caras da Canonical especialmente para contêineres Docker. Sempre que necessário basta apenas, adicionarmos facilmente camadas superfinas com aplicações adicionais e configurações. Isso elimina a necessidade de precisarmos abrir a imagem original do Ubuntu. Como isso você sabe que pode confiar na sua *base image* do Ubuntu que vemos na base das nossas três camadas.

Além do mais conforme podemos ver na Figura 19 a *Docker image* pode ser compartilhada por quantos contêineres quisermos no nosso Docker host. Isso é realmente bastante eficiente para lidarmos com espaço em disco e cache além de facilitar e muito na gestão de conflitos. Como eu disse, não é necessário abrir uma única imagem monolítica e inserir novas alterações nem ninguém

para fazer essas alterações. Sempre que necessário é só criar uma nova camada.

Figura 19 – Múltiplos Contêineres Compartilhando a Mesma Imagem



Uma única imagem compartilhada por múltiplos contêineres

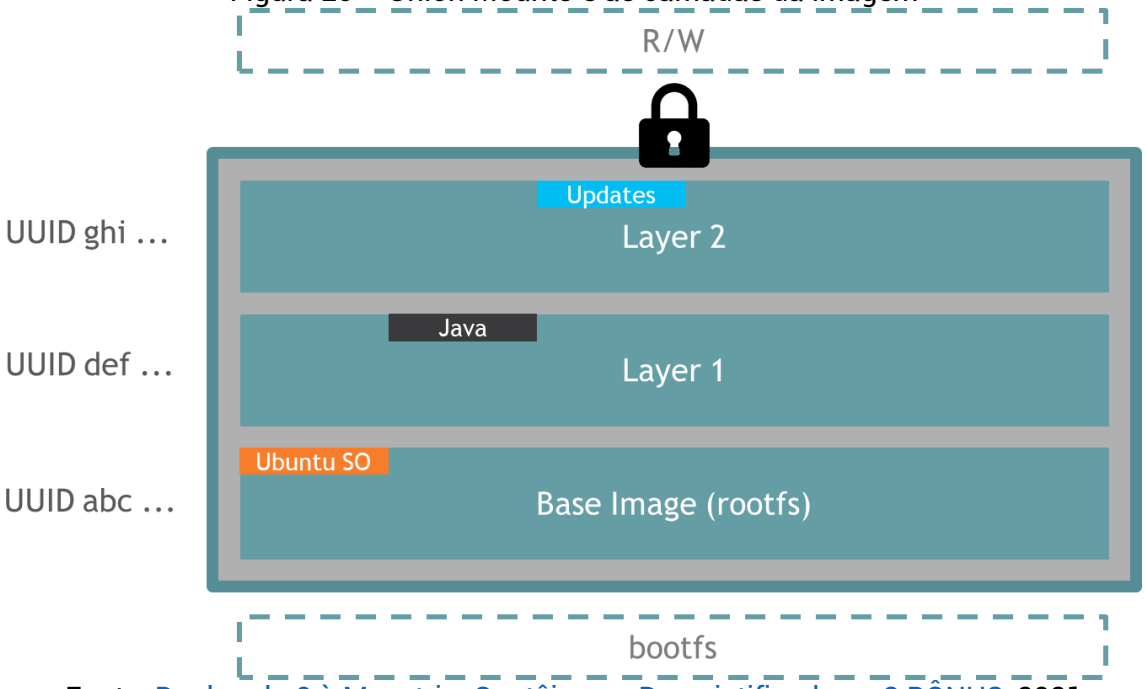
Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

4.2. Union Mounts

Conforme podemos ver na Figura 20 cada camada ou imagem recebe seu próprio *unique ID*. Esses ID's são armazenados na *Docker image* junto aos metadados que informam ao Docker como criar o contêiner em runtime. No nosso exemplo os metadados diriam ao Docker para colocar a imagem de ID ABC na primeira camada, depois na segunda camada é inserida a imagem DEF por fim na última camada é adicionada a imagem GHI. Por fim, todas as camadas são combinadas em uma única visualização. Como resultado temos uma única visualização combinada de todas as camadas, com os dados nas camadas superiores, ocultando os dados nas camadas inferiores.

Vale mencionar que caso ocorra algum conflito, digamos que você faça alterações em ***/etc/resolv.conf*** na camada superior. E isso sobrescreve o que está definido na camada base então temos um conflito. O Docker lida com isso considerando sempre o que está definido na camada superior. Lembre-se disso as camadas superiores sempre "vencem" quando há conflitos. Por baixo dos panos o Docker usa toda a "mágica" provida pela tecnologia de Union Mounts.

Figura 20 – *Union Mounts* e as camadas da imagem



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Na prática *Union Mounts* permite montar vários sistemas de arquivos uns sobre os outros se comportando como se fosse um único sistema de arquivos. Assim, com Union Mounts, ao invés de montar um sistema de arquivos em diretórios como o */root* ou */home* é possível montar vários sistemas de arquivos um sobre o outro, mas combinando todas as camadas em uma única visualização. Union Mounts dá à aplicação, ao sistema operacional e a nós a aparência de um único sistema de arquivos normal.

O Docker lida com Union Mounts da seguinte maneira. Ele monta todas as camadas da imagem como somente leitura. E então, adiciona uma nova camada no topo. Na prática quando iniciamos um contêiner somente a camada superior permite gravação. Falaremos um pouco mais sobre essa camada quando formos falar sobre os contêineres.

Lembre-se de que dissemos que nossa base image na parte inferior tem nosso *rootfs* e nosso sistema de arquivos raiz. Bem, na verdade, algumas vezes pode existir uma outra camada abaixo. Quando nós iniciamos um contêiner, na verdade existe um pequeno *bootfs* abaixo do *rootfs*. Mas o ciclo de vida do *bootfs* é muito curto e deixa de existir depois que o contêiner é iniciado. E para ser sincero como administradores ou desenvolvedores, nós nunca precisamos lidar com isso. Mas vale a pena saber desses detalhes. Por esse motivo a terminologia mais apropriada é dizer que está iniciando contêineres ao invés de bootar contêineres.

O processo de iniciar contêineres é quase o mesmo que inicializar uma VM Linux completa, como se não houvesse código BIOS ou código no nível da máquina que localizado no kernel. Na prática o Kernel já foi inicializado na

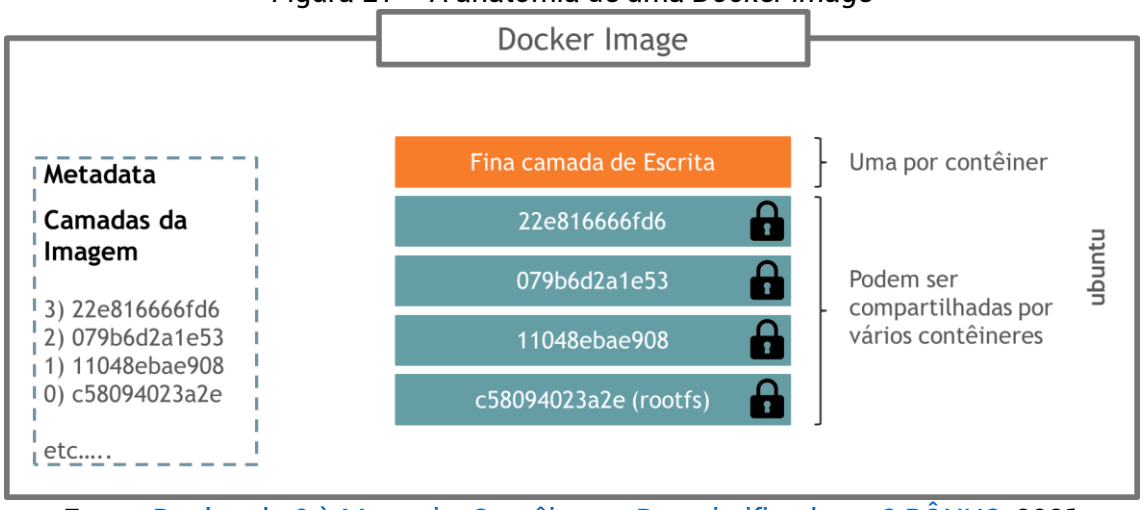
máquina hospedeira. Então nós compartilhamos o mesmo kernel comum entre todos os contêineres em execução.

Atualmente, o Docker usa Union Mounts e AUFS, a implementação de Union Mounts padrão usada em hosts Ubuntu executando o Docker. Conforme já discutimos o Docker pode gravar alterações no contêiner apenas a camada superior. Em tempo de execução as alterações são commitadas nessa camada superior por meio do comportamento de cópia em de gravação. Se precisarmos atualizar um arquivo que existe em uma das camadas inferiores, que são de somente leitura, primeiro precisamos copiá-lo para a camada superior e, em seguida, armazenamos as alterações feitas na mesma camada.

4.3. A Camada de Escrita dos Contêineres

Dissemos antes que as imagens são construídas em tempo de build e que os contêineres são construídos em tempo de execução. Quero reforçar que essa é uma distinção importante e que devemos mantê-la em mente. Com isso em mente, também aprendemos que as imagens são usadas para inicializar os contêineres. Logo, os contêineres são algo como instâncias de imagens em tempo de execução.

Figura 21 – A anatomia de uma *Docker image*



Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Quando inicializamos contêineres com o comando ***docker run***, o Docker engine lê a imagem e qualquer metadado, conforme visto na Figura 21 e, em seguida, cria o contêiner empilhando as diferentes camadas da imagem conforme as instruções nos metadados. E então todos os contêineres obtêm a sua própria camada de escrita no topo e todas as camadas abaixo são somente leitura. E é nessa camada superior, que todas as alterações em um contêiner são feitas, instalação e atualização de aplicações, escrita de novos arquivos, resolução de conflitos, como endereços IP, ou qualquer outra coisa.

Absolutamente qualquer mudança é escrito na camada superior. E, como os contêineres são *statefull*, ou seja, mantém seu estado, as alterações no estado do contêiner são gravadas nessa camada. Vale destacar que essa fina camada está inicialmente vazia. Então, esse espaço é consumido sempre que fazemos alterações em nosso contêiner. Portanto, em um contêiner que sofre uma grande quantidade de alterações, essa camada irá crescer rapidamente e consumirá mais espaço. Já em um contêiner que não grava muitos dados, essa camada permanecerá bastante eficiente em termos de espaço.

Uma coisa interessante que vale a pena mencionar é que o *rootfs* de um contêiner nunca se torna realmente gravável. Em um cenário normal de inicialização do Linux, o *rootfs* é montado inicialmente como somente leitura, e depois, no processo de inicialização, é montado novamente. No mundo dos contêineres as coisas são um pouco diferentes e isso nunca acontece. Lembre-se de que nosso *rootfs* está na camada inferior da nossa *Docker image* e, portanto, só pode ser acessada somente para leitura. Mas graças à tecnologia de Union Mount o sistema grava na camada superior como se fosse um sistema de arquivos normal.

Lembre-se de que, embora os contêineres sejam ambientes totalmente isolados para a execução de aplicações e processos Linux, existem algumas diferenças sutis entre eles e os sistemas operacionais completos. Claro que você já aprendeu isso né? Todos os contêineres em uma máquina compartilham o mesmo kernel do sistema operacional hospedeiro.

O processo de inicialização também tem algumas diferenças sutis, mas para nossas aplicações, nada disso importa. No que diz respeito às nossas aplicações, eles têm um ambiente de runtime Linux no qual podem executar. Da nossa perspectiva, de desenvolvedor ou administrador, cada contêiner é um ambiente de tempo de execução totalmente em isolado onde nossas aplicações executam e é isso que mais importa.

4.4. Considerações Finais

Nesse capítulo iniciamos aprendendo sobre as camadas das imagens. Vimos que na prática as *Docker images* são compostas por várias imagens em camadas. E aprendemos que tudo isso é implementado para ser totalmente transparente e parecer e se comportar como um sistema de arquivos normal através de Union Mounts. Esse recurso permite que vários sistemas de arquivos sejam montados uns sobre os outros ou sobrepostos. Mas, como eu disse, se comporta exatamente como um sistema de arquivos comum.

Posteriormente, mudamos nosso foco das imagens para os contêineres, que na prática são instâncias de imagens em tempo de execução. Dissemos que o Docker Engine lê os metadados em uma *Docker image* e empilha as

camadas da imagem de acordo com as instruções nos metadados. Em seguida, explicamos que uma das principais características dos contêineres é a fina camada de escrita que é sobreposta nas camadas da imagem. Conforme vimos é nessa fina camada na parte superior que todas as alterações de runtime de um contêiner são gravadas.

5. Conhecendo o Dockerfile

Nesse capítulo a gente vai se aprofundar um pouco mais no Docker e vamos aprender como criar as nossas próprias imagens usando um arquivo chamado Dockerfile. Podemos criar novas *Docker images* usando o comando do **docker commit** e isso funciona muito bem. Mas definitivamente a maneira mais flexível e poderosa de criar *Docker images* é através de um arquivo de Dockerfile. Apesar de ser bastante flexível e poderoso é muito simples de se trabalhar com Dockerfiles. E se você é como eu certamente acha que quanto mais simples for melhor.

5.1. Conhecendo o Dockerfile

Vamos começar a nos familiarizar com o Dockerfile. Antes de mais nada se escreve Dockerfile com a primeira letra maiúscula e as demais minúsculas. Não pode ser Dockerfile com todas maiúsculas ou todas minúsculas. Se não for exatamente assim, não vai funcionar. É um arquivo comum de texto plano sem nenhuma extensão. É um arquivo de texto simples legível por humanos e tem sua própria sintaxe.

Conforme se pode ver na Figura 22 sua sintaxe é bastante simples e você não terá nenhuma dificuldade em entendê-la. É nele que são definidas às instruções que determinarão como sua imagem será construída. Essas instruções são lidas uma de cada vez, de cima para baixo, da esquerda para a direita. Elas são estruturadas em propriedades chave e valor onde a instrução é a chave e é escrita em maiúsculas, o valor vem após o espaço e são usados para definirmos um ou mais valores para a instrução.

Figura 22 – Um exemplo de Dockerfile

```
# Hello World with Ubuntu Based Image

FROM ubuntu:18.04
MAINTAINER leandro@email.com
RUN apt-get update
CMD ["echo", "Hello World"]
```

Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Teoricamente o Dockerfile pode ser colocado em qualquer lugar desejado. Mas na verdade, precisamos escolher com muito cuidado o lugar mais apropriado de acordo com a arquitetura da nossa aplicação. Explicando

um pouco melhor. A principal coisa que devemos considerar ao escolher onde colocar nosso Dockerfile é que, quando criamos uma imagem a partir dele, qualquer arquivo e diretório no mesmo local ou mais abaixo na árvore do sistema de arquivos, qualquer coisa nos subdiretórios é incluída na compilação. Portanto, definitivamente nós não queremos colocar nosso Dockerfile no diretório ROOT, a menos é claro que desejemos incluir todos os arquivos do nosso sistema em nossa compilação, o que certamente nunca será o caso. Além disso para otimizar e excluir arquivos da compilação podemos usar um *.dockerignore*.

Então resumindo o que é o Dockerfile. Letra D da primeira letra maiúscula as demais minúsculas, texto plano sem formatação contendo todas as instruções necessárias para construir a nossa imagem. Após criarmos o Dockerfile, executamos o comando *docker build* para criar uma nova imagem a partir das instruções no arquivo. Parece complexo, mas na verdade é muito simples.

5.2. As Principais Instruções do Dockerfile

Agora vamos conhecer as principais instruções usadas em Dockerfiles[2]:

- **FROM**: Define a partir de qual base image será construída a nova imagem. Raramente uma imagem será construída sem uma base image e nesse caso o valor é FROM *scratch*, ou seja, do zero;
  Ex: `FROM node:14.15.0`

- **MAINTAINER**: Instrução opcional atualmente depreciada, é usada para informarmos o nome do mantenedor da imagem. Pessoalmente eu não recomendo usar pois cada instrução gera uma nova camada na imagem o que não é recomendado pela Docker;
  Ex: `MAINTAINER leandro@email.com`
  Prefira: Não usar

- **LABEL**: Adiciona metadados a uma imagem, informações adicionais que servirão para identificar versão, tipo de licença, ou host. Assim como MAINTAINER o Docker não recomenda usar muitas instruções LABEL eu pessoalmente não uso. Acesse o DockerHub e tente encontrar uma *docker image* oficial que use, normalmente eles preferem usar comentários que nos Dockerfiles iniciam após uma #;
  Ex: `LABEL 'Alguma informação'`

---

[2] Mais detalhes sobre cada uma das instruções do Dockerfile em https://docs.docker.com/engine/reference/builder/

Prefira: `# Alguma informação`

- **RUN**: Executa comandos em *buildtime*/Compilação e são usados para instalar pacotes ou aplicações e executar diversos tipos de instruções Linux. Podem haver várias instruções RUN em um Dockerfile, mas cada instrução RUN adiciona uma nova camada à imagem. Para minimizar o número de camadas use dois && entre cada propriedade;

  Ex: `RUN mkdir /docker-entrypoint-initdb.d`
  Ex com múltiplas instruções concatenadas: `RUN apt-get update && apt-get install netcat -y && chmod +x ./wait-for.sh ./startup.sh && npm install –production`

- **CMD**: Define comandos a serem executados em runtime, ou seja, assim que o contêiner iniciar. Equivalente aos parâmetros passados no comando *docker run <parâmetros> <comando>*. Só pode haver um por Dockerfile e seu valor pode ser sobrescrito pelos parâmetros passados no comando docker run;

  Ex: `CMD ["npm", "start"]`

- **EXPOSE**: Define uma ou mais portas na(s) qual(is) o container será acessível após a inicialização. Entretanto o EXPOSE no Dockerfile não expõe nem faz o *bind* com a porta do host permitindo assim o acesso externo. Isso precisa ser feito durante a inicialização do contêiner com o parâmetro `"-p {PORTA_NO_HOST}:{PORTA_NO_CONTAINER}"` ou através de um arquivo de configurações do Docker Compose;

  Ex: `EXPOSE 80`
  `EXPOSE 443`

- **ENV**: Instrução que cria e atribui um valor para uma variável de ambiente dentro do contêiner.

  Ex: `ENV MYSQL_VERSION 8.0.25-1debian10`

- **COPY <src> <dest>**: Copia arquivos ou diretórios locais **<src>** para dentro do sistema de arquivos do contêiner **<dest>**. Se não precisar adicionar recursos provenientes de URL's e/ou compactados prefira sempre a instrução **COPY.**

  Ex: `COPY ./RestWithASPNETUdemy/db/migrations/ /home/database/`

- **ADD <src> <dest>**: Similar ao COPY essa instrução copia arquivos locais ou disponibilizados através de uma URL **<src>** para dentro do sistema de arquivos do contêiner no caminho **<dest>**.
  - **ADD** permite que **<src>** seja uma URL;
  - Se o parâmetro **<src>** de **ADD** for um arquivo em um formato de compactação reconhecido, ele será descompactado no sistema de arquivos do contêiner.

  Exemplo de uso da instrução ADD para instalar um pacote:

  ```
  ADD https://example.com/big.tar.xz /usr/src/things/
  RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
  RUN make -C /usr/src/things all
  ```

  Como o tamanho da imagem é importante, não é recomendável usar o ADD para baixar pacotes de URL's remotas; prefira usar o curl ou wget em vez disso. Dessa forma, você pode excluir os arquivos de que não precisa mais depois de terem sido extraídos e adiciona menos camadas à sua imagem. Em situações como essa a Docker sugere usar a instrução RUN para minimizar a quantidade de camadas:

  ```
  RUN mkdir -p /usr/src/things \
      && curl -SL https://example.com/big.tar.xz \
      | tar -xJC /usr/src/things \
      && make -C /usr/src/things all
  ```

- **ENTRYPOINT**: Define comandos a serem executados em runtime, ou seja, assim que o contêiner iniciar, diferentemente do CMD, o ENTRYPOINT não é sobrescrito, isso quer dizer que este comando sempre será executado.

  Ex: `ENTRYPOINT ["dotnet", "RestWithASPNETUdemy.dll"]`

- **VOLUME**: Mapeia um diretório do host tornando-o acessível ao container;

  Ex:
  ```
  VOLUME /usr/data
  VOLUME /var/www/html
  ```

- **USER**: Define com qual usuário serão executadas as instruções durante a execução do contêiner. Jamais usei essa instrução e sinceramente não vejo necessidade para adotá-la;

  Ex: `USER leandro`

- **WORKDIR**: Define qual será o diretório no contêiner para onde serão copiados os arquivos, criadas novas pastas e inicializada a aplicação;

  Ex: `WORKDIR /app`

- **ONBUILD**: Define instruções que podem ser executadas em *runtime* quando determinada ação for executada, funciona de modo similar à uma trigger.

  Ex: `ONBUILD ADD . /app/src`

  ```
  ONBUILD RUN /usr/local/bin/python-build --dir /app/src
  ```

5.3. Criando o Primeiro Dockerfile

Visto que conhecemos cada uma das principais instruções e descobrimos para que servem chegou o momento de criarmos o nosso primeiro Dockerfile. Vamos começar com um Hello World assim como faríamos se estivéssemos aprendendo qualquer linguagem de programação. Clique na Figura 23 para ver Solomon Hykes na PyCon 2013, apresentando o docker ao mundo pela primeira vez. Se o criador do Docker começou com um Hello Word então pra dar sorte nós faremos o mesmo.

Figura 23 – Na PyCon 2013, Solomon Hykes mostra o docker ao mundo

Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

Crie um diretório na sua máquina e dentro dele crie um arquivo de texto chamado Dockerfile sem extensão. Como vimos precisa respeitar a

capitalização deve ser nomeado exatamente Dockerfile. Se for por exemplo DOCKERFILE, dockerfile ou DoCKERFILE não irá funcionar. Abra esse arquivo no seu editor de texto plano favorito e vamos definir as instruções.

Conforme o Código 1 vamos começar com um comentário dizendo o que a nossa *Docker image* fará para isso use uma # e defina a sua descrição. Depois vamos adicionar a nossa primeira instrução FROM (ela sempre será a primeira instrução) definindo qual será a nossa *base image* no nosso exemplo o ubuntu 20.04. Em seguida vamos adicionar o e-mail do mantenedor da imagem, conforme vimos isso é opcional e atualmente prefere-se usar a instrução LABEL. Honestamente em aplicações reais eu não uso nenhuma delas. Em seguida vamos executar uma instrução RUN que no nosso caso irá executar um *apt-get update*. Como você pode imaginar não precisamos dela pra fazer um simples Hello World, mas escolhi adicionar para mostra-la em ação. Por fim adicionaremos a instrução CMD que irá imprimir uma mensagem de *Hello World*.

Código 1 – Nosso primeiro Dockerfile

```
# Hello World with Ubuntu Based Image
FROM ubuntu:20.04
MAINTAINER leandro@email.com
RUN apt-get update
CMD ["echo", "Hello World"]
```

Fonte: Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS, 2021

5.4. Criando a Primeira Docker image

Agora que o Dockerfile já foi criado e adicionamos cada uma das instruções chegou o momento de criar a nossa primeira *Docker image*. Para isso abra o terminal no mesmo diretório em que está o *Dockerfile*. E digite o comando:

```
docker build -t hello-docker .
```

O comando `docker build` diz ao Docker que ele deve ler o Dockerfile, processar as instruções e construir uma *Docker image*. O parâmetro -t é usado pra definirmos o nome da imagem e a tag no nosso caso `hello-docker` e optamos por não definir uma tag. O sinal de "." é usado pra dizer ao Docker em relação ao contexto onde está o Dockerfile. Se o Dockerfile estivesse dentro de um subdiretório chamado por exemplo de app o comando seria:

```
docker build -t hello-docker ./app
```

Perfeito agora que sabemos pra que serve cada um dos parâmetros vamos teclar ENTER:

Console 4 – Saida do comando *docker build*

```
$ docker build -t hello-docker .
#1 [internal] load build definition from Dockerfile
#1 sha256:0ff803470d4e1807f35cb29d42cc8ba2df513fb62852eff50db28d3036560ccf
#1 transferring dockerfile: 170B done
#1 DONE 0.1s

#2 [internal] load .dockerignore
#2 sha256:7d2836a75b7f5fd8945ffcb83d040db58996c8fd310edfbbd687812660edd1ce
#2 transferring context: 2B done
#2 DONE 0.1s

#3 [internal] load metadata for docker.io/library/ubuntu:18.04
#3 sha256:ae46bbb1b755529d0da663ca0256a22acd7c9fe21844946c149800baa67c4e4b
#3 ...

#4 [auth] library/ubuntu:pull token for registry-1.docker.io
#4 sha256:c5d27e8d952c2455abffbdfaa3330d292823513bf905962459a53e13920a5ba0
#4 DONE 0.0s

#3 [internal] load metadata for docker.io/library/ubuntu:18.04
#3 sha256:ae46bbb1b755529d0da663ca0256a22acd7c9fe21844946c149800baa67c4e4b
#3 DONE 12.9s

#5 [1/2] FROM docker.io/library/ubuntu:18.04@sha256:139b3846cee2e63de9ced83cee7023a2d95763ee2573e5b0ab6dea9dfbd4db8f
#5 sha256:8f723e04c031a656785227517f3ab11b81b38a8bcc5a66d4ec502b56eed45e47
#5 resolve docker.io/library/ubuntu:18.04@sha256:139b3846cee2e63de9ced83cee7023a2d95763ee2573e5b0ab6dea9dfbd4db8f 0.0s done
#5 sha256:139b3846cee2e63de9ced83cee7023a2d95763ee2573e5b0ab6dea9dfbd4db8f 1.41kB / 1.41kB done
#5 sha256:ce1e17c0e0aa9db95cf19fb6ba297eb2a52b9ba71768f32a74ab39213c416600 529B / 529B done
#5 sha256:7d0d8fa372249c6a1de9868ad62af9a14aaae2a2b17da867d8fad099a637fd0f 1.46kB / 1.46kB done
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 0B / 26.70MB 0.1s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 3.15MB / 26.70MB 0.7s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 7.34MB / 26.70MB 1.0s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 9.44MB / 26.70MB 1.2s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 11.53MB / 26.70MB 1.4s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 13.63MB / 26.70MB 1.5s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 15.73MB / 26.70MB 1.7s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 19.92MB / 26.70MB 2.1s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 22.02MB / 26.70MB 2.3s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 24.12MB / 26.70MB 2.5s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 26.70MB / 26.70MB 2.7s
#5 sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 26.70MB / 26.70MB 2.7s done
#5 extracting sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd
#5 extracting sha256:25fa05cd42bd8fabb25d2a6f3f8c9f7ab34637903d00fd2ed1c1d0fa980427dd 1.1s done
#5 DONE 4.0s

#6 [2/2] RUN apt-get update
#6 sha256:da302e7e87a8bd16e5b84e4daeb652018df14fdc6d7079ddb8aa062dd714ced9
#6 0.882 Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
#6 1.106 Get:2 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [2220 kB]
#6 1.796 Get:3 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
#6 2.315 Get:4 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
#6 2.760 Get:5 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
#6 2.983 Get:6 http://archive.ubuntu.com/ubuntu bionic/universe amd64 Packages [11.3 MB]
#6 6.659 Get:7 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 Packages [24.7 kB]
#6 6.663 Get:8 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [1418 kB]
#6 11.02 Get:9 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packages [473 kB]
#6 44.21 Get:10 http://archive.ubuntu.com/ubuntu bionic/restricted amd64 Packages [13.5 kB]
#6 44.21 Get:11 http://archive.ubuntu.com/ubuntu bionic/main amd64 Packages [1344 kB]
```

```
#6 49.66 Get:12 http://archive.ubuntu.com/ubuntu bionic/multiverse amd64 Packages [186 kB]
#6 49.72 Get:13 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages [33.5 kB]
#6 49.72 Get:14 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [2653 kB]
#6 59.51 Get:15 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [2188 kB]
#6 68.26 Get:16 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages [505 kB]
#6 69.10 Get:17 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [11.4 kB]
#6 69.15 Get:18 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [11.3 kB]
#6 69.16 Fetched 22.9 MB in 1min 9s (332 kB/s)
#6 69.16 Reading package lists...
#6 DONE 70.5s

#7 exporting to image
#7 sha256:e8c613e07b0b7ff33893b694f7759a10d42e180f2b4dc349fb57dc6b71dcab00
#7 exporting layers
#7 exporting layers 0.2s done
#7 writing image sha256:ed24f76ccb2d7e117c28e4fd5290b15bc981f6c6def4be4a87ac21b7c8f65464 done
#7 naming to docker.io/library/hello-docker done
#7 DONE 0.2s
```

Fonte: AUTOR, 2021

Se você teve uma saída no log algo similar ao Console 4 significa que tudo está dentro do esperado. Se estiver diferente revise os passos e tente novamente.

5.5. Conferindo se a Docker Image foi Realmente Criada

Agora é o momento de conferirmos se a *Docker image* foi realmente gerada pra isso digite o comando:

`docker images` ou `docker image ls`

Os dois commandos possuem o mesmo comportamento e servem para listar as *Docker images* na nossa máquina. Se tudo estiver correto você verá algo similar ao Console 5:

Console 5 – Saida do comando *docker build*

```
$ docker image ls
REPOSITORY    TAG     IMAGE ID      CREATED         SIZE
hello-docker  latest  ed24f76ccb2d  12 minutes ago  100MB
```

Fonte: AUTOR, 2021

5.6. Executando a Docker Image que Criamos

Com a *Docker image* construída podemos executar o nosso primeiro contêiner customizado. Pra isso digite o comando:

```
docker run hello-docker
```

Se tudo estiver sido feito corretamente isso irá iniciar o contêiner e imprimir Hello World no console conforme podemos ver no Console 6. Como o único processo executado pelo contêiner é imprimir um Hello World assim que ele faz isso ele é encerrado.

Console 6 – Saida do comando *docker build*

```
$ docker run hello-docker
Hello World
```

Fonte: AUTOR, 2021

5.7. Considerações Finais

Nesse capítulo iniciamos aprendendo sobre o que é o Dockerfile. Conhecemos as principais instruções do Dockerfile, exemplos de uso e quando usar cada uma. Vimos como criar o nosso próprio Dockerfile e como criar *Docker images* a partir deles. O nosso exemplo é bem simples mas a base para dockerizações mais complexas é essencialmente a mesma.

Posteriormente, usamos a *Docker image* construída por nós mesmos para subirmos um contêiner. Com esses fundamentos você já é capaz de construir seus próprios *Dockerfiles* e *Docker images*. Evidentemente você precisará pesquisar e expandir esse conhecimento pouco a pouco.

6. Os principais comandos do Docker

Nesse capítulo a gente vai conhecer os principais comandos usados na gestão de contêineres, *Docker images*, volumes, redes e inspecionar o *Docker Daemon*. Podemos acessar a lista completa de comandos Docker apenas digitando docker e teclando ENTER no terminal.

6.1. Os Principais Comandos para Usados na Gestão de Contêineres

➔ docker run -p HOST_PORT:CONTAINER_PORT -d REGISTRY/IMAGE:TAG
- O comando mais conhecido do Docker e é usado para inicializar um contêiner;
- O parâmetro -p é opcional e é usado para definir quais portas HTTP serão expostas. À esquerda especificamos a porta no host (nossa máquina) e a direita no contêiner;
- O parâmetro -d é opcional e é usado para subir o contêiner em background e nos permitir continuar usando o console;
- Para imagens oficiais não precisamos passar o REGISTRY/ já para às nossas *Docker images* precisamos passar o nosso usuário no Docker Hub ou no registry em uso;
- Por fim o nome da imagem e a TAG que é opcional. Se não especificarmos uma TAG o Docker subirá um contêiner a partir da *Docker image* com a TAG *latest*;
- Existem outros parâmetros, mas esses são mais do que suficientes para começar.

➔ docker container ls -a → Lista os contêineres em execução. O parâmetro -a é opcional e pode ser usado para listar todos os contêineres inclusive os que estão parados;
- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker ps -a;
- Hoje podemos usar ainda o comando docker container ps -a além dessas duas sintaxes.

➔ docker container logs -f CONTAINER_ID → Acessa os logs de um contêiner. O parâmetro -f é opcional e pode ser usado para acompanhar os logs de modo **análogo ao tail** do Linux;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker logs -f CONTAINER_ID;
- Hoje podemos usar as duas sintaxes, mas eventualmente a Docker pode depreciar o segundo comando.

➔ docker container exec -it CONTAINER_ID bash → Acessa o terminal do container. O parâmetro -it é opcional e pode ser usado para conseguir acesso interativo com o shell do container;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker exec -it CONTAINER_ID bash;
- Hoje podemos usar as duas sintaxes;
- Para maior compatibilidade de comandos em ambiente Windows é interessante prefixar todo o comando com winpty;
  - winpty docker container exec -it CONTAINER_ID bash.

➔ docker container start | stop | pause | unpause | kill CONTAINER_ID → Comandos básicos para gestão de contêineres;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker start | stop | pause | unpause | kill CONTAINER_ID;
- Hoje podemos usar as duas sintaxes.

➔ docker container top CONTAINER_ID → Mostra o processo em execução no contêiner;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker top CONTAINER_ID;
- Hoje podemos usar as duas sintaxes.

➔ docker container inspect CONTAINER_ID → Inspeciona detalhes de um contêiner;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker inspect CONTAINER_ID;
- Hoje podemos usar as duas sintaxes.

➔ docker container rm CONTAINER_ID → Remove um contêiner;

- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker rm CONTAINER_ID;
- Caso o contêiner esteja em execução ele não será removido. Podemos adicionar o parâmetro -f para forçar a remoção mesmo se estiver em execução;
- Hoje podemos usar as duas sintaxes.

➔ docker container prune → Remove todos os contêineres parados;
- Não existia um comando equivalente em versões anteriores do Docker e normalmente improvisávamos com o comando:
  - docker rm $(docker ps -a -q) -f → A parte dentro dos parêntesis lista todos os contêineres. E a parte externa remove todos os contêineres retornados na lista interna. O -f pode ser usado, de forma opcional, para forçar a remoção, mesmo que o contêiner esteja sendo usado.

6.2. Os Principais Comandos para Usados na Gestão de Docker Images

➔ docker pull REGISTRY/IMAGE:TAG → baixa uma *Docker image* do Docker Hub.
- Para imagens oficiais não precisamos passar o REGISTRY/ já para às nossas *Docker images* precisamos passar o nosso usuário no Docker Hub ou no registry em uso;
- O comando docker run faz um docker pull de modo totalemnete transparente se ele não encontrar a imagem necessária à inicialização do contêiner no Host.

➔ docker image ls → Lista todas as *Docker images*;
- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker images;
- Hoje podemos usar as duas sintaxes.

➔ docker image rm IMAGE_ID → Remove uma *Docker image*;
- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker rmi IMAGE_ID;
- Hoje podemos usar as duas sintaxes.
- Caso algum contêiner esteja usando a *Docker image* ela não será removida. Podemos adicionar o parâmetro -f para forçar a remoção mesmo se estiver em execução;
  - Caso a remoção seja forçada qualquer contêiner relacionado também será removido.

➔ docker image prune → Remove todas as *Docker images* que não estão em uso;

- Não existia um comando equivalente em versões anteriores do Docker e normalmente improvisávamos com o comando:
  - docker rmi $(docker images -q) -f → A parte dentro dos parêntesis lista todas as imagens. E a parte externa remove todas as imagens retornadas na lista interna. O -f pode ser usado, de forma opcional, para forçar a remoção, mesmo que a imagem esteja sendo usada em algum contêiner em execução.

➔ docker image tag SRC_IMAGE:TAG DOCKER_HUB_USER/IMAGE_NAME:TAG → Cria uma tag a partir de uma *Docker image* existente.
- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker tag SRC_IMAGE:TAG DOCKER_HUB_USER/IMAGE_NAME:TAG;
- Hoje podemos usar as duas sintaxes pessoalmente eu prefiro a segunda opção.

➔ docker login docker.io → Autentica no DockerHub via console.

➔ docker image push DOCKER_HUB_USER/IMAGE_NAME:TAG → Envia uma imagem para o DockerHub;
- Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker push DOCKER_HUB_USER/IMAGE_NAME:TAG;
- Hoje podemos usar as duas sintaxes pessoalmente eu prefiro a segunda opção;
- Se ocorrer um erro de autenticação ao executar o comando docker login docker.io você não conseguirá executar esse comando;
- Firewalls, VPN's e antivírus podem te pregar boas peças nessa etapa.

6.3. Os Principais Comandos para Usados na Gestão de Volumes

➔ docker volume ls → Lista todos os volumes Docker;
- Não existia um comando equivalente em versões anteriores do Docker e normalmente improvisávamos com o comando:

        - docker inspect -f '{{ .Mounts }}' CONTAINER_ID → Na prática a gente filtrava por todos os pontos de montagem do contêiner e retornava a lista.

➔ docker volume prune → Remove todos os volumes *Docker* que não estão em uso;
    - Não existia um comando equivalente em versões anteriores do Docker.

6.4. Os Principais Comandos para Usados na Gestão de Redes

➔ docker network ls → Lista todas as redes *Docker*;
    - Não existia um comando equivalente em versões anteriores do Docker e normalmente improvisávamos com o comando:
        - docker inspect -f '{{ .NetworkSettings}}' CONTAINER_ID → Na prática a gente filtrava por todos as configurações de rede e retornava a lista. Não era perfeito mas atendia;

➔ docker network prune → Remove todas as redes *Docker* que não estão em uso;
    - Não existia um comando equivalente em versões anteriores do Docker.

6.5. Os Principais Comandos Usados na Inspeção do Docker Daemon

➔ docker events → nos permite inspecionar o que está acontecendo com o *Docker Daemon*. forçando um pouco a barra podemos dizer que é "um *debug*".

➔ docker stats → Detalhes sobre cada container métricas

➔ docker system df → Detalhes sobre containers, images e volumes

➔ docker system prune → Remove todas as redes *Docker* que não estão em uso;

- Não existia um comando equivalente em versões anteriores do Docker.

6.6. Considerações Finais

Bom esse capítulo foi bem diferente dos demais, iniciamos os comandos Docker para gestão de Contêineres, depois conhecemos os comandos usados na gestão de *Docker images*, passando pelos comandos de gestão de volumes e redes. Por fim vimos os comandos usados na inspeção do Docker Daemon. Claro que esses só alguns dos comandos *Docker* e com o tempo você precisará pesquisar e expandir essa lista pouco a pouco. Você pode conhecer todos os comandos disponíveis digitando docker e teclando ENTER no terminal. Se eu tiver esquecido algum comando importante sinta-se à vontade em me avisar.

7. As 25 Perguntas e Respostas sobre Docker mais Comuns em Entrevistas

Se você chegou até aqui é por que está buscando melhores oportunidades na sua carreira. Atualmente além de grandes empresas como Google, Amazon, VMware e Netflix em praticamente qualquer vaga o Docker é um requisito fundamental. Mais do que nunca esse, é o momento certo para aprender essa tecnologia e fazer carreira com ela. Se por acaso está se preparando para uma entrevista, abaixo estão as 25 principais perguntas sobre Docker mais Comuns em Entrevistas e que claro você precisa saber!

PERGUNTA N° 01: Explique o que é um contêiner Docker?

RESPOSTA: Um contêiner é a unidade básica de software que contém o código e todas as suas dependências, a fim de possibilitar a execução da aplicação sem problemas, de forma rápida e confiável de um ambiente Docker para outro. Um contêiner Docker pode ser criado usando uma *Docker image*. É um pacote executável do software, que contém tudo o que é necessário para executar uma aplicação, que são ferramentas do sistema, bibliotecas, código, runtime e configurações. Na prática um contêiner é uma instância em execução de uma imagem e é uma "máquina Linux". Podemos iniciá-los, pará-los, atachá-los enfim várias coisas comuns aos sistemas operacionais.

PERGUNTA N° 02: Explique os componentes da Arquitetura Docker?

RESPOSTA: Os componentes da arquitetura Docker são:

- O **Host:** é onde está instalado o *Docker Daemon*, também chamado de *Docker Engine*. Este por sua vez gerencia imagens e contêineres. Enquanto o Docker Daemon estabelece um link com o Registry, as *Docker images* atuam como metadados para as aplicações mantidas nos contêineres Docker.
- O **Client:** o *Docker Client* se comunica com o *Docker Daemon* e executa operações como gerenciar contêineres e imagens.
- O **Registry:** este componente Docker é usado para armazenar as Docker images. Docker Hub e GitHub Packages são Registries públicos, que podem ser utilizados por qualquer pessoa. Atualmente os principais serviços em nuvem oferecem serviços de registries privados como Amazon ECR, Azure Container Registry e Google Container Registry.

PERGUNTA N° 03: Explique como funciona um Docker Registry em detalhes?

RESPOSTA: O local onde todas as *Docker images* são armazenadas é conhecido como Docker Registry. O Docker Hub é um Registry público padrão para essas imagens. Outro Registry público é o GitHub Packages. O Docker Hub é o registry público mais popular, mantido por um grande número de

desenvolvedores, empresas e colaboradores individuais. Todas as *Docker images* oficiais são mantidas pela empresa responsável pela tecnologia e pelo Docker Hub.

PERGUNTA N° 04: Explique resumidamente o ciclo de vida de um Contêiner Docker.
RESPOSTA: O ciclo de vida de um Contêiner Docker é:

- Criação do container
- Execução do contêiner
- Pausa do contêiner
- Despausar o contêiner
- Iniciar o contêiner
- Parar o contêiner
- Reinicia o contêiner
- Matar o contêiner
- Destruir o contêiner

PERGUNTA N° 05: Cite alguns comandos importantes do Docker?

RESPOSTA: Abaixo estão alguns comandos importantes do Docker:

- docker build: para construir uma *Docker image* a partir de um *Dockerfile;*
- docker run: para um contêiner a partir de uma *Docker image*;
- docker stop : para parar suavemente um contêiner;
- docker kill: para matar abruptamente um contêiner;
- dockerd : para iniciar daemon Docker;
- docker commit: para criar uma nova imagem a partir de mudanças feitas no contêiner. É uma alternativa à construção de *Docker images* à partir de um contêiner.

PERGUNTA N° 06: O que são namespaces no Docker?

RESPOSTA: *Docker Namespaces* é uma tecnologia que fornece *workspaces* isolados conhecidos como contêineres. Depois que um contêiner é iniciado, um *namespace* é criado para o referido contêiner. Esse namespace fornecem uma camada de isolamento para esse contêiner. Cada container funciona em um namespace distinto, com seu acesso limitado ao namespace mencionado.

PERGUNTA N° 07: O que é *Docker Swarm*?

RESPOSTA: O *Docker Swarm* é uma ferramenta nativa usada para clusterizar e orquestrar contêineres Docker. Usando o *Docker Swarm*, os desenvolvedores e o supervisores de TI podem configurar e gerenciar facilmente vários *Docker nodes*. Atualmente a própria Docker começou a admitir que o *Kubernetes* evoluiu mais e é uma solução mais sólida e robusta que o *Docker Swarm*.

PERGUNTA N° 08: Como identificar o status de um contêiner Docker?

RESPOSTA: Para identificar o status de um contêiner Docker, deve-se executar o comando

- ➔ docker container ls -a (o -a é opcional e é usado para listar todos os contêineres inclusive os parados);
- ➔ Nas versões mais antigas do Docker a sintaxe desse comando era mais simples, apenas docker ps -a;
- ➔ Hoje podemos usar as duas sintaxes, mas eventualmente a Docker pode depreciar o segundo comando.

Este comando irá listar todos os contêineres Docker disponíveis no host com o respectivo status. A partir da lista, pode-se facilmente selecionar um contêiner pretendido para verificar seu status e se for o caso parar, reiniciar, pausar ou mesmo verificar mais detalhes como log através do CONTAINER ID ou nome.

PERGUNTA N° 09: O que são *Docker Images* e comando Docker Run?

RESPOSTA: Uma *Docker image* são um conjunto de arquivos e parâmetros que permitem a criação de contêineres que são instâncias de *Docker images* em execução. Em uma visão superficial podemos dizer que eles contêm uma lista de tudo o que está em um contêiner além de instruções sobre como montá-lo. No mundo das Docker images sempre que queremos inicializar um contêiner, precisamos de uma imagem. O comando *docker run* é usado para criar uma instância em execução de uma *Docker image* em resumo criar um container. Podemos subir um ou múltiplos contêineres a partir de uma *Docker image*.

PERGUNTA N° 10: Liste algumas funcionalidades e aplicações do Docker?

RESPOSTA: Abaixo estão algumas funcionalidades e aplicações da dockerização:

- Torna a configuração mais simples e fornece facilidade de configuração no nível da infraestrutura;
- Ao ajudar o desenvolvedor a se concentrar exclusivamente na lógica de negócios, isso reduz o tempo de desenvolvimento e aumenta a produtividade;
- Ele amplia a capacidade de *debug* e isso provê funcionalidades úteis;
- Permite o isolamento da aplicação;
- Otimiza a utilização de hardware através da contêinerização;
- Facilita a rápida implantação no nível de sistema operacional.

PERGUNTA N° 11: Quais são os principais Objetos Docker?

RESPOSTA: Imagens, serviços e contêineres Docker são denominados objetos Docker.

- **Imagens** : um modelo somente leitura com instruções para criar um contêiner do Docker;
- **Contêineres** : uma instância de uma imagem em execução;
- **Serviços** : permite o dimensionamento de contêineres em uma variedade de *Docker Daemons*, que funcionam juntos através do *Swarm*;
- Outros objetos Docker incluem redes e volumes.

PERGUNTA N° 12: O que é mais adequado para aplicações conteinerizadas, Stateless ou Stateful?

A: Opte por aplicações *Stateless* (sem Estado) ao invés de aplicações Stateful em contêineres Docker. Por exemplo, podemos criar um contêiner pra nossa aplicação e tornar os parâmetros de estado da aplicação configuráveis. Feito isto, podemos executar o mesmo contêiner com parâmetros diferentes em produção e em outros ambientes. Através de aplicações Stateless, podemos reutilizar a mesma imagem em diferentes cenários. Além disso, quando se trata de contêineres Docker, é mais fácil escalar uma aplicação Stateless que uma Stateful.

PERGUNTA N° 13: Explique o uso do Dockerfile?

RESPOSTA: Antes de mais nada se escreve Dockerfile com a primeira letra maiúscula e as demais minúsculas. É um arquivo comum de texto plano sem nenhuma extensão e tem sua própria sintaxe. É nele que são definidas às instruções que determinarão como uma *Docker image* será construída. Essas instruções são lidas uma de cada vez, de cima para baixo, da esquerda para a direita. Elas são estruturadas em propriedades chave e valor onde a instrução é a chave e é escrita em maiúsculas, o valor vem após o espaço e são usados para definirmos um ou mais valores para a instrução. As instruções do Dockerfile serão processadas pelo Docker possibilitando a construção de *Docker images*.

PERGUNTA N° 14: Quais tipos de redes disponíveis por padrão no Docker?

RESPOSTA: As redes disponíveis no Docker por padrão são:

- **bridge**: rede padrão na qual os contêineres se conectam se a rede não for especificada de outra forma;
- **none**: conecta-se a uma stack de rede específica do contêiner sem interface de rede;
- **host**: conecta-se à stack de rede do host.

PERGUNTA N° 15: Liste as etapas do processo de implantação para aplicações dockerizadas armazenadas em um repositório Git?

RESPOSTA: Embora o processo de implantação mude de acordo com a aplicação, um processo de implantação básico terá o seguinte:

- Dockerize sua aplicação com um *Dockerfile*;
- Gere uma *Docker image* através da execução do comando *docker build* no diretório em que se encontra o *Dockerfile*;
- Faça o teste da sua *Docker image*;
- Envie sua *Docker image* para o *Docker Registry*;
- Notifique o servidor de aplicação remoto para que faça o pull da sua *Docker image* no Registry e execute-a;
- Altere a(s) porta(s) no proxy HTTP;
- Pare o contêiner antigo.

PERGUNTA N° 16: Explique como o Docker é diferente de outras tecnologias de contêiner?

RESPOSTA: Docker é uma das tecnologias de contêiner mais recentes e emergiu como uma das mais populares. Construído na era da nuvem, o Docker vem com muitos recursos novos que estavam faltando nas tecnologias de contêiner mais antigas. Um dos melhores recursos do Docker é que ele pode ser executado em qualquer infraestrutura, seja no seu notebook ou na nuvem. Por meio do Docker, mais aplicações agora podem ser executadas nos mesmos servidores e também permite empacotarmos processos e programas. O Docker também tem um o Docker Registry que atua como um repositório para Docker images, que são fáceis de baixar e usar. Além disso, essas *Docker images* podem ser compartilhadas por múltiplos contêineres. O Docker também é muito bem documentado, o que o torna melhor do que outras tecnologias de contêiner.

PERGUNTA N° 17: Se você sair do Contêiner Docker, perderá seus dados?

RESPOSTA: Quando um Contêiner Docker é encerrado, nenhuma perda de dados ocorre, pois, todos os dados são gravados no disco pela aplicação com o único propósito de preservação. Este processo é repetido de forma consistente até e a menos que o contêiner seja inequivocamente excluído. Além disso, o sistema de arquivos do contêiner Docker persiste mesmo depois que o contêiner é interrompido.

PERGUNTA N° 18: Como o Docker é monitorado em produção?

RESPOSTA: Para monitorar o Docker em produção existem comandos como *docker stats* e *docker events* do Docker. Por meio desses comandos, é possível obter relatórios sobre estatísticas importantes. Quando executamos o comando *docker stats* seguido do ID do contêiner, ele retorna o uso de CPU e memória do contêiner. Ele é semelhante ao comando top do Linux. Por outro lado, o comando *docker events* lista os processos em andamento no *Docker Daemon*. Alguns desses eventos são attached, commit, rename, destroy, die e

muito mais. Existe ainda a opção de filtrar os eventos de acordo com nosso interesse.

PERGUNTA N° 19: Detalhe o workflow de uso do Docker?

RESPOSTA: Abaixo está uma breve explicação do workflow de uso do Docker:

- Como o Dockerfile é o código-fonte da imagem, tudo começa com ele;
- Depois de criado, o Dockerfile é usado para construir a *Docker image*. Na prática a *Docker image* é a versão compilada do Dockerfile;
- Essa *Docker image* é então distribuída através do Docker Registry, que é algo como um repositório de *Docker images*;
- Além disso, a *Docker image* pode ser usada para executar contêineres. Um contêiner, enquanto está em execução, é muito semelhante a uma VM sem o hypervisor.

PERGUNTA N° 20: Explique a diferença entre os comandos 'docker run' e 'docker create'?

RESPOSTA: A principal diferença entre o *'docker run'* e o *'docker create'* é que, se você usar o último, o contêiner será criado em um estado 'parado'. Além disso, o comando *'docker create'* pode ser usado para armazenar e gerar contêiner ID's para uso posterior. A melhor maneira de fazer isso é usar *'docker run'* passando o parâmetro '*--cidfile FILE_NAME*', pois executá-lo novamente não permitirá sobrescrever o arquivo.

PERGUNTA N° 21: O que é virtualização?

RESPOSTA: A virtualização, em seus primeiros dias, era conhecida como um método de divisão lógica de mainframes para permitir que várias aplicações fossem executadas simultaneamente. Mas com o passar do tempo e a indústria foi capaz de permitir que vários sistemas operacionais fossem executados simultaneamente em um único sistema baseado em x86, o significado de virtualização mudou consideravelmente.

O resultado final é? A virtualização permite que o usuário execute dois sistemas operacionais diferentes no mesmo hardware. Embora o sistema operacional principal seja o administrador, cada sistema operacional convidado passa por processos como inicialização, carregamento de kernel e muito mais. Também é perfeito para segurança, já que o sistema operacional convidado não pode ter acesso total ao sistema operacional host.

Existem basicamente três tipos de virtualização:

- Paravirtualização;
- Emulação;
- Virtualização baseada em contêiner.

PERGUNTA N° 22: Qual é a diferença entre um registry e um repositório?

RESPOSTA: O Docker Registry é um serviço de hospedagem e distribuição de *Docker images*, atualmente o Docker Hub é o registry padrão. Por outro lado, o Repositório Docker é uma coleção de Docker images relacionadas. Ou seja, elas possuem o mesmo nome, mas *tags* diferentes.

PERGUNTA N° 23: Podemos usar arquivos JSON ao invés de YAML no Docker Compose? Se sim, como?

RESPOSTA: Sim, podemos usar JSON ao invés de YAML como arquivo de configurações do Docker Compose. Para iniciarmos uma stack de contêineres no *Docker Compose* usando um arquivo JSON, precisamos especificar o nome do arquivo no comando de inicialização da *stack* no *Docker Compose*:

```
docker-compose -f docker-compose.json up
```

PERGUNTA N° 24: Explique as diferenças entre CMD e ENTRYPOINT em um Dockerfile?

RESPOSTA: Em um Dockerfile, as instruções CMD e ENTRYPOINT definem qual comando será executado durante a execução de um contêiner. Existem algumas regras no uso dos comandos CMD e ENTRYPOINT, tais como:

- O Dockerfile deve especificar pelo menos um comando do CMD ou ENTRYPOINT;
- Se o contêiner for usado como um executável a instrução ENTRYPOINT precisa ser definida;
- Ao executar o contêiner com um argumento alternativo, o CMD será sobrescrito.

PERGUNTA N° 25: Explique o que precisamos fazer para executar uma aplicação dentro de um contêiner Linux usando Docker

A: Abaixo estão as etapas sobre como executar uma aplicação dentro de um contêiner Linux usando Docker:

- Instale e execute o Docker;
- Baixe (*docker pull*) uma *base image* como a do Ubuntu (sistema operacional baseado em Linux) do Docker Hub;
- Dockerize a sua aplicação na *base image* do Ubuntu;
- Gere uma *Docker image* para a sua aplicação;
- Execute o contêiner usando a nova *Docker image*;
- Confira os contêineres em execução no *host*;
- Inicie ou pare o contêiner Docker;
- Acesse o contêiner Docker da sua aplicação;
- Se for o caso remova o contêiner e/ou imagem.

Esta foi a lista das 25 perguntas e respostas sobre Docker mais comuns em entrevistas. Espero que elas possam te ajudar de alguma forma a avançar na sua carreira. Além disso, confira os nossos cursos sobre **Docker:**

- **Docker do 0 à Maestria: Contêineres Desmistificados + 3 BÔNUS**
- **REST API's RESTFul do 0 à Azure com ASP.NET "Core" 5 e Docker**
- **REST API's RESTFul do 0 à AWS com Spring Boot 2.x e Docker**
- **Docker para Amazon AWS Implante Apps Java e .NET com Travis CI**
- **Microservices do 0 à GCP com Spring Boot, Kubernetes e Docker**

Conclusão

Esses são os atalhos que eu chamo de "Docker um guia Rápido: O Mínimo que você Precisa Saber para Começar" que eu uso e que, até então, só havia ensinado em meus treinamentos. Mas agora você também pode colocar em prática!

Antes de mais nada meus parabéns e muito obrigado por terminar a leitura desse livro. Ele foi feito com muita atenção e muito carinho. Esse projeto começou aproximadamente a mais de três anos e primeiramente eu quero te dar os parabéns. Porque existem três tipos de pessoas, que encontramos ao longo da vida. Aquelas que já tem sucesso e já alcançaram o que queriam. Aquelas que não alcançaram e acham que não vão alcançar. E o terceiro tipo de pessoas que são as que ainda não tem o que quer, mas estão tentando buscar aprender o que falta para chegar lá; e você faz parte desse terceiro tipo de pessoa.

A grande vantagem é que as pessoas que não têm sucesso e que ainda não estão aonde querem, tendem ultrapassar aquelas que já tem. Porque é natural do ser humano estar constantemente querendo aprender mais e isso dá o gás que precisamos para avançar. Então parabéns para você, você já se destaca de mais de 80 por cento da população.

Isso soa como um clichê, mas é uma realidade, eu recebo depoimentos quase toda semana de alunos que não sabiam nada de Docker e outras tecnologias que estão na crista da onda e que deram verdadeiros saltos na carreira ou mesmo conseguiram o primeiro emprego em TI. Muitas dessas pessoas conseguiram mais resultados que aquelas que já atuavam na área. E sabe por que? Por que elas não perderam tempo estudando coisas inúteis e principalmente por que não aprenderam fazer as coisas de forma errada.

Sabe aquele cara que só de ver a falha no log já sabe de cara onde está o problema, consegue ser bem, mais ágil que outros programadores enfim, esse tipo de cara. Então, todas essas pessoas começaram atrás dele e começaram a estudar e se dedicaram em melhorar enquanto profissionais conseguiram resultados muito mais avançados.

Então a primeira boa notícia é que você já tem um diferencial muito importante. E por incrível que pareça, você tem uma vantagem, você ainda não aprendeu a fazer as coisas do jeito errado. É mais ou menos igual a um campeonato de futebol ou uma corrida. Nem sempre quem larga em último lugar, nem sempre quem está no final da tabela é o que chega por último. Às vezes é ele quem ganha o campeonato.

O que faz diferença é a sua dedicação e a busca por corrigir constantemente seus erros e melhorar. Então você é esse tipo de pessoa,

parabéns por estar se dedicando, por ter optado em querer melhorar e por querer crescer e se desenvolver.

Não sei se você percebeu ou não, mas os capítulos foram ordenados para que você evolua da forma correta com Docker. São os pontos centrais, mas que te levam nessa direção. Então aproveite esse conteúdo, aplique e veja os resultados.

Um grande abraço e muito sucesso!

Meus Cursos

Docker do 0 à Maestria: Contêineres Desmistificados + BÔNUS

Domine containers! Aprenda Docker do 0 absoluto na teoria e na prática! Integre o Docker à apps Java, .NET, Node JS e +

Leandro da Costa Gonçalves

4,7 ★★★★½ (1.259)

12,5 horas no total • 161 aulas • Todos os níveis

Classificação mais alta

Microservices do 0 à GCP com Spring Boot Kubernetes e Docker

Domine microsserviços com Spring Boot, Spring Cloud, Docker, Kubernetes e Google Cloud Platform

Leandro da Costa Gonçalves

4,6 ★★★★½ (18)

13,5 horas no total • 166 aulas • Intermediário

Classificação mais alta

REST API's RESTFul do 0 à Azure com ASP.NET Core 5 e Docker

Desenvolva uma API REST do zero absoluto atendendo todos os níveis de maturidade RESTful e implante na Azure + React JS

Leandro da Costa Gonçalves

4,6 ★★★★½ (1.221)

18,5 horas no total • 235 aulas • Todos os níveis

Classificação mais alta

REST API's RESTFul do 0 à AWS com Spring Boot 2.x e Docker

Desenvolva uma API REST do zero absoluto atendendo todos os níveis de maturidade RESTful e implante na AWS + React JS

Leandro da Costa Gonçalves

4,6 ★★★★½ (985)

18,5 horas no total • 230 aulas • Todos os níveis

Classificação mais alta

Docker para Amazon AWS Implante Apps Java e .NET + Travis CI

Saiba obter o máximo do Docker na Amazon AWS seja em aplicações Java ou .NET

Leandro da Costa Gonçalves

4,4 ★★★★½ (225)

4 horas no total • 66 aulas • Intermediário

Classificação mais alta

Agile desmistificado com Scrum, XP, Kanban, Spotify e Trello

Aprenda à aplicar Scrum|Kanban|eXtreme Programming e Spotify aos seus projetos maximizando a produtividade e a qualidade

Leandro da Costa Gonçalves

4,4 ★★★★½ (778)

14 horas no total • 226 aulas • Todos os níveis

Classificação mais alta

Trello: Gestão Otimizada de Equipes e Projetos Pessoais

Domine o Trello e otimize a gestão de suas Equipes e/ou Projetos Pessoais. Não basta ser completo tem que ser efetivo!

Leandro da Costa Gonçalves

4,7 ★★★★½ (662)

3 horas no total • 56 aulas • Iniciante

Classificação mais alta

Spotify Engineering Culture Desmistificado

Aprenda à aplicar Scrum e Agile em larga escala com o "Modelo Spotify" maximizando a produtividade e a qualidade

Leandro da Costa Gonçalves

4,4 ★★★★½ (86)

3 horas no total • 47 aulas • Intermediário

Do Waterfall ao Scrum: Acerte na Mudança do Modelo de Gestão

Aprenda a diferença entre as metodologias tradicionais como Waterfall e Agile e como fazer uma gestão de mudanças segura

Leandro da Costa Gonçalves

4,2 ★★★★☆ (125)

2 horas no total • 28 aulas • Intermediário

Career Hacking: Atalhos para o sucesso em TI

Saiba como se preparar para o mercado de Tecnologia da Informação saindo do 0 absoluto até a busca por vagas no exterior

Leandro da Costa Gonçalves

4,7 ★★★★½ (25)

2 horas no total • 49 aulas • Todos os níveis

Classificação mais alta

REST API's RESTFul from 0 to AWS with Spring Boot and Docker

Learn how to develop a REST API from absolute 0 by meeting all RESTful maturity levels and deploy on AWS using Travis CI

Leandro da Costa Gonçalves

4,2 ★★★★☆ (191)

14,5 total hours • 180 classes • All levels

Docker to Amazon AWS Deploy Java & .NET Apps with Travis CI

Build a Continuous Integration and Deploy pipeline with Docker, Travis CI and Amazon AWS for Java and .NET applications

Leandro da Costa Gonçalves

4,5 ★★★★½ (29)

4,5 total hours • 67 classes • Intermediate

Highest rated

Para ir Além

Ah, se você quiser ter acesso a conteúdo gratuitos que eu pessoalmente produzo, eu tenho duas dicas para você:

Inscreva-se no meu canal no Youtube

https://www.youtube.com/c/ErudioTraining

E acesse o nosso site


https://www.erudio.com.br/blog/