Como containers funcionam

Antes de darmos o pontapé inicial, precisamos entender o benefício da utilização de containers em diferentes aplicações.

Muitos sistemas atualmente são compostos por diversas aplicações executadas simultaneamente. Diversos problemas podem ocorrer quando muitas aplicações precisam se comunicar. Porém, antes de pensarmos em comunicação de aplicações, é preciso visualizar outras questões importantes para nosso trabalho, como **isolamento de contextos e versionamento de aplicações**.

Vamos partir do nosso problema inicial. Os sistemas atualmente têm diversas aplicações e ferramentas que interagem entre si para compor o todo desse sistema. E é mais ou menos esse caso que vamos visualizar nesse curso e entender porque os containers vão nos ajudar nessas situações. Mas antes vamos partir bem do básico da ideia de porque vamos chegar nessa solução.

[00:22] Nós teremos uma aplicação Nginx, que vai servir como um *load balancer* do nosso sistema. Temos uma aplicação Java e também uma aplicação C# rodando com o .NET.

[00:33] Isso é para ilustrar a nossa situação. E como queremos que todas essas aplicações estejam em execução para que o nosso sistema como um todo esteja ok, nós precisamos de uma máquina para que essas aplicações e, consequentemente, o nosso sistema estejam rodando.

[00:48] A nossa aplicação C# está rodando na porta 80, ela precisa da porta 80 para executar. A nossa aplicação Java também, assim como o Nginx. O C# estamos utilizando na versão 9, o Java na versão 17 e o Nginx na versão 1.17.0.

[01:05] E quais possíveis problemas podem ocorrer nessa situação? Se formos observar cada uma dessas aplicações e ferramentas, podemos acabar tendo um conflito de portas, porque as três aplicações desse cenário dependem da porta 80 para executar o fluxo que elas precisam.

[01:24] Além disso, como é que podemos alterar as versões de maneira prática? Será que se eu simplesmente fizer um *downgrade* ou um *upgrade* da versão do C#, atualizando meu .NET, eu vou quebrar alguma coisa? Será que eu preciso desinstalar para instalar uma nova?

[01:37] A mesma coisa para o Java e para o Nginx: será que eu vou conseguir atualizar de maneira prática?

[01:45] E outra questão é como teremos um controle de recursos de memória e de CPU para essas aplicações? Eu quero que, por exemplo, a minha aplicação C# precise de 100 milicores de CPU e 200 MB de memória para funcionar. Como é que eu posso definir isso de maneira fácil?

[02:00] A mesma coisa para as minhas outras aplicações. Como eu posso fazer essa definição de consumo de recursos desse sistema de maneira prática? É uma questão que precisamos levantar.

[02:09] E por fim, juntando todos esses problemas de uma vez só, como é que podemos fazer o processo de manutenção dessas aplicações? Como conseguiremos mudar a versão? Como conseguiremos ter esse controle sobre as portas da nossa aplicação, gerenciar os recursos e manter isso no longo prazo?

[02:28] Então uma solução que poderíamos pensar inicialmente e que seria bem simples, mas ao mesmo tempo problemática, e nós veremos o porquê, seria simplesmente comprarmos uma máquina para cada aplicação.

[02:40] Teremos uma máquina para nossa aplicação .NET, uma máquina para nossa aplicação Java e uma máquina para nossa aplicação Nginx.

[02:48] Assim nós temos o problema resolvido do conflito de portas. Cada máquina teria sua respectiva porta 80.

[02:55] Conseguiríamos controlar os recursos de maneira mais fácil, porque não teria essa dependência das aplicações entre si.

[03:01] E também faríamos o controle do versionamento um pouco mais facilmente, porque não teria diversas aplicações num mesmo sistema.

[03:10] Qual o problema disso? A questão é que o nosso dinheiro vai para o ralo, porque se temos uma máquina para cada aplicação, vamos pensar que tem sistemas com milhares ou milhões de aplicações executadas ao mesmo tempo. Então imagine ter milhares ou milhões de máquinas para que o seu sistema esteja operante. O dinheiro vai embora.

[03:31] Então existe uma solução já bem difundida, ela não foi criada agora, já é até razoavelmente antiga, que nos ajuda nesse tipo de problema, que são as máquinas virtuais, onde teremos nosso hardware bem definido, um sistema operacional, seja Windows, Linux, Mac e afins.

[03:51] E teremos também uma camada de *hypervisor*, que vai fazer um meio de campo entre virtualizar um novo sistema operacional, que pode ser um Windows, um Linux, um Mac, rodando dentro de outro sistema. Mas graças a essa camada de *hypervisor* nós teremos uma camada de isolamento desse sistema operacional original.

[04:10] E conseguiremos instalar nossas dependências e aplicações de maneira isolada, porque cada uma delas terá seu respectivo sistema operacional.

[04:19] É uma questão que já resolve esses problemas iniciais, mas a pergunta que fica nesse momento é: será que é realmente necessário fazer isso? Porque nós vamos querer executar nossas aplicações, como vimos, de maneira isolada uma da outra, ter um controle de recursos, ter um controle de versionamento bem definido.

[04:41] Então será que essa camada de *hypervisor* é realmente necessária? Será que precisamos sempre nessas situações virtualizar o sistema operacional? Pode ser que sim, pode ser que não. Mas o caso que veremos durante esse curso é a utilização de containers. O que um container é?

[04:58] Repara que se formos comparar os desenhos, não temos a camada de sistema operacional virtualizado e *hypervisor*, e sim diretamente a camada de container rodando no nosso sistema operacional, e mesmo assim essas aplicações estão isoladas entre si e também isoladas do nosso sistema operacional original.

[05:18] E é isso que vamos entender a partir de agora. Temos algumas perguntas que temos que responder de início: por que os containers são mais leves, além desse argumento visual que vimos agora?

[05:27] Como eles vão garantir esse isolamento? E também como eles vão funcionar sem instalar um sistema operacional? Porque no caso da máquina virtual, a nossa aplicação C# terá um sistema operacional para ela. A nossa aplicação Java também.

[05:39] Se olharmos dentro do nosso sistema de containers, a nossa aplicação C# está diretamente dentro do container. E qual sistema operacional ela vai usar? É Windows, é o Linux? Precisa instalar um sistema? Nós precisamos responder a essas perguntas, de onde vêm essas informações?

[05:57] E também como é que vai ficar a divisão de recursos entre essas aplicações que estarão a partir de agora containerizadas?

[06:02] Nós entenderemos a partir de agora, no próximo vídeo, como essas situações acontecem. Como o container será mais leve em questão de consumo de recursos, como ele vai garantir isolamento, como ele funciona sem instalar um sistema operacional, por assim dizer.

[06:18] Agora que já vimos como os containers nos ajudam, o que eles fazem, assim como as máquinas virtuais ajudam, vamos entender os diferenciais de como o container opera dentro do nosso sistema. Veremos isso na próxima aula. Te vejo lá.

No decorrer deste curso, você irá compreender como os containers resolvem estes problemas e como executar aplicações de maneira isolada entre si.

Vamos responder às perguntas: por que os containers são mais leves em relação a uma máquina virtual? O que muda no fim das contas?

[00:08] O container funciona da seguinte maneira: dentro do nosso sistema operacional nós temos diversos containers, diversas aplicações que estarão sendo executadas no nosso sistema operacional. Só que no fim das contas eles vão funcionar diretamente como processos dentro do nosso sistema.

[00:26] Enquanto uma máquina virtual terá toda aquela etapa de virtualização dos sistemas operacionais, dentro do nosso sistema original, os containers vão funcionar diretamente como processos dentro do nosso sistema.

[00:38] Então a grosso modo, conseguimos visualizar como o consumo de recursos, a carga para que ele consiga ser executado é um pouco menor. Porque eles serão processos, e não uma virtualização completa.

[00:53] Mas como o processo vai conseguir garantir isolamento, como é que vai funcionar sem instalar o sistema? Como vamos conseguir resolver e responder a essas perguntas?

[01:04] A outra questão é que quando os nossos containers estiverem em execução dentro do nosso sistema operacional, a fim de garantir o isolamento entre cada um deles e o nosso sistema operacional original, existe um conceito chamado *namespace*, que vai garantir que cada um desses containers tenha a capacidade de se isolar em determinados níveis.

[01:25] Mas que níveis são esses? Nós teremos os principais *namespaces*, que são os de PID, que garantem o isolamento a nível de processo dentro de cada um dos containers. Então um processo dentro de um container, que consequentemente é um processo dentro do nosso sistema operacional, vai estar isolado de todos os outros do nosso *host*, da nossa máquina original.

[01:46] Nós teremos o NET também, o *namespace* de rede, que vai garantir o isolamento entre uma interface de rede de cada um dos containers e também do nosso sistema operacional original.

[01:56] O de IPC, que vai ser de intercomunicação entre cada um dos processos da nossa máquina.

[02:01] O de MNT, que é da parte de *files system*, sistema de arquivos, montagem, volumes e afins. Também estará devidamente isolado.

[02:09] E o de UTS, que faz um compartilhamento, um isolamento ao mesmo tempo também, de *host* do nosso Kernel da máquina que está escutando o container.

[02:19] Esse último em específico, o UTS, ajuda a responder até a próxima pergunta, que é como o container dentro do nosso sistema operacional vai funcionar sem um sistema operacional?

[02:32] Porque na verdade graças ao *namespace* de UTS, se estivermos executando os nossos containers, por exemplo, em uma máquina que tem o Kernel Linux, cada um desses containers a princípio também vai ter um pedaço desse Kernel, só que devidamente isolado.

[02:49] Então conseguimos já responder a essa pergunta. Nós não precisamos necessariamente instalar um sistema operacional dentro de um container porque ele já vai ter, graças a esse *namespace* de UTS, também esse acesso ao Kernel do sistema original, só que isoladamente.

[03:06] E por fim, também na parte de gerenciamento de recursos, vamos dizer que queremos definir o que levantamos no problema anteriormente, de definir o consumo máximo de memória, de CPU e afins para cada um dos nossos containers.

[03:18] Existe um outro conceito que se chama “Cgroups”, que vai garantir que consigamos definir tanto de maneira automática quando de maneira manual como os consumos serão feitos para cada um desses containers dentro do nosso sistema operacional.

[03:33] Então no fim das contas, voltando às nossas perguntas originais, graças aos *namespaces* e aos Cgroups nós conseguimos garantir isolamento, conseguimos garantir que nosso container funcione sem instalar um sistema operacional dentro dele, e também conseguiremos ter um controle de gerenciamento de recursos, como memória de CPU.

[03:55] Sobre a parte de porque eles são mais leves, nós vimos que eles funcionam como processos diretamente do nosso sistema operacional, mas no decorrer desse curso vamos entender ainda mais porque eles conseguem ser tão mais práticos em relação à maquina virtual a nível de consumo de recurso e de tamanho de armazenamento no nosso sistema operacional.

[04:14] Nós respondemos as principais perguntas de quem está começando agora no mundo de containers sobre como vai garantir isolamento, como ele funciona sem instalar um sistema operacional e afins.

[04:25] Agora que entendemos como um container se diferencia de uma máquina virtual tradicional, veremos como instalar o Docker, inicialmente no Windows e depois também no Linux, que vai ser onde abordaremos as principais utilizações do Docker.

Instalando o Docker no Windows

Em relação à instalação do Docker no Windows, tem alguns pontos que são muito relevantes para se destacar.

[00:05] O primeiro deles, para já irmos adiantando, é que precisamos acessar a página da documentação oficial e clicar em “Docker Desktop for Windows”. Já tenho baixado, mas é só clicar no botão que vai começar a baixar.

[00:17] Nesse caso vou cancelar, porque como falei para vocês eu já tenho baixado. E quando baixarmos, basta clicar duas vezes no instalador que vai carregar e seguir o fluxo de instalação normalmente.

[00:29] Depois de um tempo, eu vou fazer o processo de instalação com vocês, ele vai abrir aquela tela de instalação e afins.

[00:38] Nesse momento ele mostra duas opções de instalar os componentes necessários para o Docker rodar com Windows em WSL 2, então isso é uma questão que precisamos responder; e adicionar um atalho na área de trabalho, que pode ser marcada sem nenhum problema.

[00:51] Então vamos deixar essas duas opções marcadas nesse momento. Eu vou clicar em “OK” para ele já ir instalando.

[00:57] Enquanto ele vai instalando precisamos destacar alguns pontos. O primeiro é que caso você seja um usuário de Docker ou esteja começando agora e esteja pensando em utilizar o Windows para isso, eu recomendo fortemente que você não faça a utilização do Docker no Windows, e sim utilize a ferramenta no Linux, que é um ambiente muito mais estável e pronto para gerenciar e utilizar essa plataforma toda de conteinerização.

[01:21] Um ponto que também já reforça isso que eu acabei de falar para vocês é que o Docker está fazendo uma atualização de termos de uso dele.

[01:29] O Docker Desktop, que é o que estamos utilizando agora no caso do Windows, é uma ferramenta que vai passar a ser paga a partir de 31 de janeiro de 2022, caso a sua empresa tenha mais de 250 funcionários ou tenha uma receita anual de mais de 10 milhões de dólares.

[01:47] Pode ser que não tenha, pode ser que tenha, mas já é um ponto que você deve ter atenção, dependendo da empresa em que você trabalha.

[01:54] O uso para estudo, a princípio, até o momento da gravação desse curso, continua liberado para caso você queira fazer seus experimentos e afins. Esse é o primeiro ponto que você precisa se atentar.

[02:07] O segundo é que você deve também ter atenção nesse cenário de WLS 2 *backend*. Nós vimos que aquela opção já estava marcada no momento da instalação do Docker.

[02:18] E o que significa aquele WSL 2? Ele nada mais é do que o Windows Subsystem for Linux. Nós teremos um subsistema Linux rodando dentro do nosso Windows.

[02:28] Então ele vai meio que virtualizar, a muito grosso modo, o sistema operacional Linux para que utilizemos o Docker nesse sistema operacional.

[02:37] Então caso você já tenha tanto o Windows 11 quanto o Windows 10 nas respectivas versões que estão aparecendo na tela, basta você habilitar o WSL 2 dentro do seu sistema operacional Windows, em conjunto com a documentação da Microsoft. É bem simples.

[02:53] Em conjunto com seu PowerShell também você consegue instalar o WSL. Tem toda a documentação, e caso você queira ver em português também, basta você alterar o idioma.

[03:06] A URL está com “en-us”, por praticidade eu vou mudar para “pt-br”, e a partir desse momento ele também mostra tudo em português para fazer a instalação do WSL caso você não tenha ele ativado ainda.

[03:20] Nesse momento, voltando para a instalação, ela já terminou. Vou clicar em “Close”. E como podemos ver se o Docker está funcionando?

[03:29] Primeiro precisamos executá-lo. Eu posso abrir o menu Iniciar, ou simplesmente abrir a área de trabalho podemos clicar no ícone de instalação do Docker.

[03:41] Basta clicar duas vezes e ele vai começar a carregar o Docker para a nossa máquina e executar tudo que é necessário para que o Docker funcione no ambiente Windows.

[03:52] E como podemos testar se o Docker realmente está funcionando agora? Antes de mais nada, outra coisa que eu tinha falado para vocês é que assim que você abrir ele vai mostrar essa parte de serviços que estão sendo alterados, sobre 250 funcionários e 10 milhões de dólares de receita, então você tem que aceitar os termos.

[04:14] Clicando que você aceita, o Docker vai começar a ser executado na sua máquina. Como você pode saber que o Docker está funcionando? Eu vou utilizar o PowerShell do próprio Windows e vou abrir. Podemos simplesmente fazer um teste.

[04:26] Como podemos ver algum comando tipo um “Olá, mundo”? Caso você já tenha feito algum curso de programação você entendeu exatamente o que eu estou falando. Como podemos ver se o Docker está rodando?

[04:36] Basta executarmos o comando `docker run`. Com isso vamos executar alguma coisa com Docker. E vamos colocar em seguida um `hello-world`, que é “olá, mundo”, um container clássico para vermos se o Docker está em funcionamento.

[04:50] Eu vou dar um “Enter”. Ele deu um erro, falando que não conseguiu encontrar uma imagem “hello-world” na versão “:latest” localmente. Ele meio que travou, e agora funcionou. Nós vamos entender o que isso quis dizer no decorrer do curso.

[05:07] Mas podemos ver que está funcionando, porque ele está mostrando “Hello from Docker”, um olá do Docker, e ele está falando nessa mensagem que nossa instalação funcionou corretamente.

[05:17] Então instalamos o Docker no Windows sem nenhum problema, no instalador fomos dando “Next” e tudo funcionou.

[05:23] Então a última observação antes de finalizar esse vídeo é: fique atento aos termos de uso do Docker com Windows, porque eles podem variar diversos termos, a partir de determinados momentos. Você tem que ficar atento quanto a isso para não ser cobrado posteriormente. Então tome cuidado, pode variar. Mas a instalação em si é essa.

[05:46] E não fique apreensivo ou apreensiva, nós vamos entender o que cada uma das coisas que apareceram na tela querem dizer, como esse *latest* e o *digest*. Nós não sabemos o que isso quer dizer ainda, mas vamos entender.

[05:56] Só que agora vamos fazer o processo de instalação do Docker no Linux, utilizando distros do Ubuntu. Faremos isso no próximo vídeo. Eu te vejo lá.

Instalando o Docker no Linux

O processo de instalação do Docker no Linux é bem simples e prático também, basta seguirmos a documentação.

[00:06] No caso do curso vamos utilizar o Docker no Ubuntu, que é uma das distros mais utilizadas dentro do ambiente Linux.

[00:14] Ao contrário do Windows, não precisamos nos preocupar com aquela questão da alteração dos termos de uso. Mas você precisa ficar atento a outra coisa, caso você esteja utilizando o Ubuntu, e eu recomendo que você esteja para ficar alinhado com o progresso do curso: o suporte versão 16.04, que é a penúltima LTS lançada até o momento, não está mais sendo dada. Então utilize no mínimo a 18.04, conforme a recomendação da própria documentação.

[00:39] Caso você esteja utilizando outros sistemas operacionais, ainda baseados em Linux, temos no menu lateral esquerdo o CentOS, o Debian, Fedora, RHEL, SLES e afins.

[00:48] Mas caso você, no fim das contas, esteja se preocupando e realmente queira, por algum motivo, utilizar o Docker com o Mac iOS, também tem o guia de instalação. Não recomendo que faça. Ainda tem muitas instabilidades, e o Docker roda com muita facilidade no Linux. O ambiente Linux é o ambiente inclusive recomendado para que você faça a utilização produtiva do seu ambiente Docker.

[01:16] Dados os recados, para instalar é bem simples. É só seguir a documentação, começando a partir do ponto em que vamos copiar o comando de atualizar os repositórios, os pacotes do nosso Ubuntu.

[01:29] Ele pede a senha de sudo, vou colocar a minha. Ele vai atualizar rapidamente e é só seguir copiando e colando os comandos. Cada um deles na instalação do pacote é necessário, tudo é necessário para que consigamos fazer.

[01:42] Eu já fiz a execução dele, vou refazer com vocês para vocês verem que está tudo funcionando corretamente.

[01:48] No meu caso ele está perguntando se eu quero sobrescrever, porque como eu acabei de falar, eu já fiz. Mas vou colocar que sim. No seu caso, se for a primeira vez, não vai aparecer isso, ele só vai executar sem nenhum problema.

[01:58] Agora nós vamos definir que queremos utilizar o Docker na versão estável. Ele tem a versão também "nightly" ou "test", que ainda é o próprio Docker, mas são diferentes versões a nível de teste e estabilidade da plataforma, da ferramenta.

[02:14] Nós queremos estabilidade, então vamos utilizar a versão estável. E por fim vamos efetuar o *update* dos nossos pacotes mais uma vez. E no fim das contas executar também o comando de instalação. Ele vai executar rapidamente.

[02:34] E assim que ele terminar de atualizar os pacotes do meu sistema, vamos executar o comando de instalação do Docker Community Edition, CLI do Docker e o containerd.io, que será responsável pela parte do funcionamento dos containers dentro do nosso sistema. Assim que ele terminar, o Docker estará 100% dentro da nossa máquina.

[02:58] Terminou de ler os pacotes, vou dar um "Ctrl + L" para limpar. E agora sim vou colar o último código e dar um "Enter". No meu caso já estava instalado.

[03:14] E agora, para validarmos isso, vamos executar o comando `docker run hello-world`, assim como nós fizemos no Windows.

[03:25] E o que aconteceu? Nós recebemos uma permissão negada. O que será que isso quer dizer? Por que isso aconteceu?

[03:34] Agora nós precisamos nos atentar ao seguinte: quando temos essa questão de permissão no Linux, geralmente se executarmos o mesmo comando com um `sudo` na frente, tudo vai funcionar. Vamos ver se é assim mesmo.

[03:48] Ele deu o "Hello from Docker", assim como deu no Windows. Então a princípio a instalação toda funcionou sem nenhum problema.

[03:56] Mas como fazemos agora para não ficar executando Docker com o sudo na frente? É uma coisa meio chata termos que lembrar de colocar o sudo toda hora.

[04:04] Existe uma maneira recomendada pela própria documentação de como podemos contornar esse problema, que é criar um grupo chamado Docker dentro do nosso sistema e colocarmos o nosso usuário, que no meu caso é Daniel, dentro desse grupo.

[04:17] Nós usamos o comando `sudo usermod` e passamos os parâmetros `-aG`, seguido de `docker`, que é o nome do grupo ao qual queremos adicionar o nosso usuário, e em seguida `$USER`, que será o nosso próprio usuário.

[04:32] Dessa maneira conseguimos executar o comando do Docker sem o sudo na frente. A única questão é que precisamos reiniciar a nossa máquina.

[04:42] Então vou mandar reiniciar a minha máquina, mas esse vídeo não vai parar. Parecia que eu ia cortar o vídeo no meio, mas não cortei. Enquanto a máquina vai reiniciando eu vou mostrar para vocês a questão na documentação de onde está isso que eu falei para vocês.

[05:01] Na parte de pós instalação no Linux ele tem algumas etapas interessantes, inclusive essa que eu mostrei para você, com o comando `sudo usermod -aG docker $USER`. Nesse momento ele está adicionando nosso usuário a esse grupo Docker.

[05:20] Ele fala no fim das contas que precisamos fazer o login novamente e tudo mais. No caso, se você estiver utilizando uma máquina virtual, será necessário reiniciar essa máquina para que as alterações tenham efeito.

[05:33] Como eu estou gravando originalmente no ambiente Windows, eu estou utilizando uma máquina virtual para que tudo fique mais fácil no processo de produção.

[05:42] Mas no fim das contas nós estamos no Linux. Já reiniciei a máquina, vou logar no meu usuário. E agora, quando ele terminar de carregar, nós vamos conseguir executar o mesmo comando que executamos previamente, que é o nosso `docker run hello-world`, só que agora sem o sudo. Vamos ver se vai funcionar mesmo.

[06:00] Ele está terminando de carregar a área de trabalho. Vou abrir o terminal e agora executar `docker run hello-world`, sem o sudo na frente. Agora está tudo funcionando. Nossa instalação está ok.

[06:18] Agora que já sabemos como instalar o Docker e fazer o teste para ver se ele realmente está funcionando, vamos começar a colocar a mão na massa e entender o que é esse `docker run` que executamos, o que aconteceu nesse momento.

[06:30] Nós precisamos entender sobre esse fluxo, o que o Docker fez e porque tivemos essa mensagem. Só que isso nós vamos descobrir nas próximas aulas. Eu te vejo lá e até mais.

Conhecendo o Docker Hub

Agora vamos entender o que o Docker fez com o `docker run`, como ele funcionou. Nós vamos começar a aprofundar essas questões do `docker run` e como ele identificou aquele “hello world”, como ele sabia o que deveria fazer naquele momento.

[00:17] Vamos voltar ao nosso terminal e executar mais uma vez o `docker run hello-world` do zero, para vermos o que acontece.

[00:26] Nesse momento o Docker fala que não encontrou essa imagem na nossa máquina, depois ele faz o download. Por fim ele finaliza e faz uma validação sha256, que vamos entender mais à frente o que significa.

[00:42] O ponto é que ele executa a saída do container que pedimos para executar, baseado naquele `hello-world` que nós passamos.

[00:51] Vamos limpar o terminal e agora vamos executar outro `docker run`. Só que ao invés de simplesmente colocar `hello-word`, que queremos descobrir de onde veio, vamos colocar, por exemplo, um nome maluco nessa sequência, sem sentido de caracteres e dar um “Enter”.

[01:09] A primeira parte foi igual, ele fala que não foi possível encontrar esse cara localmente. E depois ele dá um erro falando que ou tivemos acesso negado ou esse repositório que contém essa imagem não existe.

[01:21] Mas voltou essa questão, o que é imagem? Nós vamos responder também isso aos poucos, mas vai chegar a hora que vamos dar a resposta final.

[01:29] Mas precisamos entender que para que o nosso `docker run` funcione, ele precisa encontrar essa tal de imagem para que o container funcione. Nesse caso, quando passamos um nome maluco ele não encontrou. Mas com o `hello-world` ele encontrou.

[01:43] Então como é que o Docker sabe onde ele deve procurar e encontrar esses nomes para resolver e executar o nosso container?

[01:52] Ele até está dando uma dica para nós, de que existe um grande repositório de imagens que são essas receitas que vão executar os nossos containers, que é o chamado Docker Hub.

[02:05] Dentro do Docker Hub podemos encontrar diversas dessas imagens, que são esses parâmetros que estamos passando. Se fizermos `docker run --help`, além de todas as *flags*, ele mostra que o uso do `docker run` é `docker run`, seguido de algumas opções, que vamos entender quais são em breve; e o nome da imagem que queremos executar.

[02:28] Então "hello-world" é o nome de uma imagem. Aquele nome maluco que passamos também foi uma tentativa de um nome de imagem.

[02:35] Isso significa que se viermos no Docker Hub pesquisarmos, por exemplo, por "hello-world", ele vai achar nossa imagem. Temos como resultado o "hello-world", que tem o selo de imagem oficial e na descrição fala que é só uma imagem para exemplificar uma "dockerização" mínima, ou seja, executar um container com Docker para testarmos se está funcionando.

[02:58] A imagem ser oficial significa que ela foi feita e é mantida por um grupo confiável de pessoas que desenvolveram e têm reconhecimento da comunidade. Então ela recebe esse selo de oficial.

[03:12] Abaixo ele mostra as descrições, o que é essa imagem efetivamente, conta uma breve história sobre ela.

[03:19] Mas a questão é: se tentarmos pesquisar no Docker Hub a imagem maluca que criamos anteriormente, ele não encontra nada. Então é por isso que quando executamos o `docker run` com aquele nome maluco ele não encontrou. Mas o `hello-world` já existe.

[03:44] Então o que mais podemos fazer agora para começar a levantar algumas perguntas? Quais outras imagens são interessantes de conhecermos para darmos esse pontapé agora na parte do `docker run` e entender como isso vai funcionar?

[03:56] Existem diversas imagens que podem replicar, por exemplo, o conteúdo de um sistema operacional. Então por mais que um container não tenha necessidade de ter um sistema operacional instalado, conforme estávamos falando anteriormente, nós podemos ter um container que contém um sistema operacional, por exemplo, o Ubuntu.

[04:15] Então se pesquisarmos por Ubuntu no Docker Hub, existe uma imagem, oficial inclusive. Então ela foi criada e é mantida por um grupo de pessoas confiáveis. Quando essa imagem for executada, ela vai gerar um container baseado no Ubuntu.

[04:34] Então como podemos fazer isso agora? Vamos voltar para o terminal e dar um "Ctrl + C" para fechar e um "Ctrl + L".

[04:44] Se executarmos o comando `docker run ubuntu`, ele vai fazer exatamente isso: ele vai no Docker Hub, baixar essa imagem e executar esse container.

[04:56] Voltando no Docker Hub, se quisermos fazer em etapas, nós podemos primeiro baixar a imagem separadamente para executá-la depois.

[05:04] Então ao invés de `docker run ubuntu` podemos fazer `docker pull ubuntu`. Ele não vai dar o *output* de que não encontrou localmente, porque já estamos pedindo para fazer o download. Ele vai fazer o download, vai extrair e, por fim, vai fazer aquela verificação. Mas ele não vai executar, porque nós só pedimos para ele baixar a imagem.

[05:29] E agora se pedirmos para executar efetivamente, ele não vai fazer o download, porque nós já temos a imagem localmente. Então podemos simplesmente colocar agora `docker run ubuntu` para que o container execute.

[05:39] E o que vai acontecer agora? Nada. Por que nada aconteceu? Vamos na nossa apresentação mais uma vez para entender e levantar algumas dúvidas.

[05:52] No momento em que executamos o comando `docker run ubuntu`, nós esperávamos que subisse um container com o Ubuntu dentro dele e que alguma coisa executasse, como "Bem vindo ao Ubuntu", ou alguma coisa do tipo.

[06:04] Mas no fim das contas temos duas perguntas: por que o container não rodou? Ou será que ele roubou e nós não sabemos? Por que ele não exibiu nada para nós? São perguntas que precisamos responder.

[06:16] E também, o que o `docker run` fez por baixo dos panos agora? Já sabemos agora que ele foi no Docker Hub, buscou aquela de imagem, que é a receita para gerar um container, e executou.

[06:27] Mas como isso funciona efetivamente, para entendermos todo o fluxo e conseguirmos compreender o que estamos fazendo? Vamos responder isso no próximo vídeo. Eu te vejo lá.

O comando docker run

Já aprendemos que ao utilizar o Docker precisamos conhecer e saber para que servem alguns comandos para fazer ações quando trabalhamos com containers, sendo um deles o `docker run`.

Escolha a alternativa que mostra como a definição do comportamento do Docker ao executarmos o comando.

O comando `docker run` é responsável por executar um container em nosso host.

Alternativa correta! Através deste comando, o docker irá executar o container da maneira esperada.

Fluxo da criação de containers

Vamos entender o que aconteceu. Primeiro: por que o container não rodou? Vamos voltar ao nosso terminal mais uma vez.

[00:07] Vamos tentar mais uma vez por teimosia colocar `docker run ubuntu`. Nada aconteceu de novo. Mas vamos entender e aproveitar para conhecer outros comandos do Docker que vão nos auxiliar a entender se o container realmente executou ou não, e por que ele não exibiu nada.

[00:26] Um comando muito útil que vamos utilizar bastante é chamado `docker ps`. Ele vai exibir quais containers estão em execução no presente momento. Vou limpar a tela e fazer `docker ps`, e ele exibiu só o cabeçalho.

[00:40] No fim das contas significa que minha tabela de containers em execução está vazia. Então não tem nenhum container efetivamente em execução.

[00:48] Apenas a título de curiosidade, outra maneira um pouco mais semântica para utilizar o `docker ps`, é o comando `docker container ls`. É a mesma coisa, um pouco mais verboso, mas ao mesmo tempo mais semântico. É o mesmo comando que será executado, então a saída tinha que ser a mesma também.

[01:09] Nós já sabemos que nosso container não está em execução. Então como é que podemos ver todos os containers, inclusive os que já não estão mais em execução, para saber se o nosso `docker run` fez alguma coisa efetivamente ou não?

[01:21] Esse comando pode ser simplesmente o `docker ps -a`, ou o `docker container ls -a`, que é a mesma coisa. Vamos executar.

[01:30] Repara que ele tem nosso "hello-world", que executamos no caso há 9 minutos. Tem o Ubuntu que fizemos no vídeo anterior e o outro Ubuntu que é o que acabamos de fazer na base da teimosia.

[01:49] Vamos entender cada uma dessas colunas. A primeira é o container ID, que é um identificador. O ID, obviamente, tem que ser único para cada container que vai ser criado.

[01:58] Depois temos a imagem que foi usada como base para criação desse container. Temos `docker run ubuntu` duas vezes e `docker run hello-world`.

[02:06] Depois o comando que foi executado ao criar esse container. Para os comandos do Ubuntu foi executado um `bash`, para o `hello-world` foi executado um `/hello`. Em seguida temos o momento que foi criado, um deles foi criado há um minuto, outro há 4 e outro há 9.

[02:19] E em seguida temos o status, que está como *exited* para todos. Por isso que eles não foram exibidos no `docker ps` ou no `docker container ls`, só quando passamos a *flag* `-a`.

[02:29] Depois temos a questão das portas, que vamos entender em breve. E a coluna de *names*, que é simplesmente o nome que ele cria automaticamente para o container quando não especificamos um nome para ele. Por enquanto não estamos nos preocupando com isso, é só um preciosismo do Docker, que cria um nome aleatório para os containers.

[02:45] Mas vamos voltar e responder à pergunta: por que o container não está em execução? A resposta já está na tela inclusive. É por causa da linha de comando.

[02:57] No momento em que um container é executado a partir dessa imagem, está definido que quando o container subir e for executado ele vai executar o comando `bash`, no caso do Ubuntu. No caso da imagem do `hello-world` ele vai executar um `/hello`.

[03:14] No momento em que executamos mais uma vez o `docker run ubuntu`, ele vai subir o container, vai executar o comando `bash` e pronto. Quando ele executa o comando *bash* ele já fez o que tinha que fazer, já cumpriu o objetivo dele.

[03:32] Então voltando à nossa apresentação, no fim das contas o container foi executado. Ele subiu, executou o *bash*, fez o que tinha que ser feito, que era executar. E a partir desse momento não tinha mais nenhum processo segurando a existência desse container. Então por isso ele foi encerrado.

[03:48] Para que um container esteja em execução deve ter no mínimo um processo dentro dele para que o container fique vivo. Então se não houver processos em execução, o container não vai ficar em execução. E como só pedimos para ele executar um *bash*, ele abriu, fez o papel e fechou.

[04:06] Então essa é a primeira questão: no momento em que executamos o `run`, ele fez esse comando e fechou logo em seguida, porque nós não mantivemos nenhum processo em execução. Então vou limpar o terminal, e se executarmos o comando `docker run ubuntu` mais uma vez, o que podemos fazer?

[04:26] Vou passar um `docker run --help` para lembrarmos do que vimos, na verdade. Quando nós especificamos a imagem, nós podemos passar um comando para que esse container execute.

[04:41] Nós já sabemos que dentro do Ubuntu com um todo nós temos a linha de comando que nós podemos executar. Então eu quero que esse container, por exemplo, execute o comando de `docker run ubuntu sleep 1d`.

[04:56] Eu quero que o comando que o container execute quando esse Ubuntu subir seja um *sleep* de um dia. Então tecnicamente, quando esse container subir ele vai ter um processo de *sleep* travando ele por um dia. Será que vai funcionar? Vamos ver agora. Eu vou dar um "Enter".

[05:15] Ele travou o nosso terminal a princípio. Como ele travou nosso terminal, vamos abrir um novo e vamos executar `docker ps`.

[05:42] Nós temos um container ID, com a imagem Ubuntu. O comando é um `sleep 1d`. Como esse comando vai demorar um dia para ser executado, esse container vai ter um tempo de vida de um dia. E ele está rodando, o status dele contém "Up" há 27 segundos. A questão de portas, mais uma vez, vamos relevar por enquanto. E o nome dele é "funny_pike".

[06:02] Entendemos agora como conseguimos manter um container em execução e porque ele parou de executar, quando simplesmente não especificamos um comando que deveria ser executado. Porque não tinha nada travando a execução dele, agora tem, que é o nosso *sleep* de um dia que passamos que esse container deveria executar.

[06:21] E agora, a segunda pergunta que ainda temos que responder, voltando à nossa apresentação é: o que o `docker run` fez por baixo dos panos para que nosso container executasse daquela primeira vez?

[06:33] No momento em que tínhamos o nosso *host*, no caso o nosso Ubuntu, executando o comando `docker run ubuntu` ou `docker pull ubuntu`, ele simplesmente foi no Docker Hub e falou: "Eu preciso dessa imagem chamada Ubuntu".

[06:46] E o Docker Hub falou: "Eu tenho essa imagem e vou passá-la para você". Então fizemos o download dessa imagem e depois o que o nosso *host* fez, naquela parte que dizia "digest sha256", foi simplesmente uma validação para verificar a autenticidade da imagem, se realmente era a imagem que estávamos procurando.

[07:08] Então no momento em que fazemos o `docker run`, se não tivermos a imagem localmente nós a buscamos no Docker Hub, fazemos uma validação em cima de um *hash* e executamos nosso container, que como vimos no caso do Ubuntu, vai ter geralmente um comando padrão a ser executado. Se colocarmos mais uma vez o `-a` para comparar, veremos que esse comando será o `bash`.

[07:29] Então se simplesmente não tivermos cuidado, e vamos ver como lidar com isso no decorrer da criação dos nossos containers, nós simplesmente acabaremos criando containers zumbi. Eles vão subir e morrer logo em seguida, porque não teremos nenhum processo travando eles.

[07:43] Mas no caso de agora, vimos que se mantivermos no mínimo um processo, o nosso container vai ficar sempre em execução.

[07:51] Então por esse vídeo é isso. Entendemos agora o que mantém um container vivo, em execução versus o que mantém um container parado e o que o `docker run` faz para que consigamos executar efetivamente um container quando não possuímos uma imagem localmente.

Etapas do run

Quando queremos executar um container e usamos o comando `docker run`, ocorre uma série de etapas ordenadas até que a execução seja feita efetivamente.

Procura a imagem localmente -> Baixa a imagem caso não encontre localmente -> Valida o hash da imagem -> Executa o container.

Outros comandos importantes

Agora vamos entender mais alguns comandos que serão muito úteis para nós no decorrer desse curso para que consigamos desenvolver nosso conhecimento com o Docker.

[00:12] Até então nós temos a criação de containers, já vimos o que o `docker run` faz, como ele funciona; a questão do comando padrão, que a princípio é executado quando executamos uma imagem. E vimos como sobrescrever esse comando, no caso colocando um *sleep* de um dia.

[00:28] Mas o que mais é importante de saber? Tem uma coisa que ainda precisamos entender. Eu vou limpar a tela e fazer um `docker ps` novamente. Temos nosso container em execução com nosso *sleep* de um dia.

[00:48] Ele está em execução, nós mantivemos um terminal travado para isso, o que é um pouco chato. Mas o que acontece, por exemplo, se eu quiser fazer um fluxo um pouco mais utilizável, por assim dizer?

[01:01] Vamos fechar esse terminal que está travando o nosso. Se eu der um `docker ps` agora, ele vai manter o meu container em execução, porque eu fechei o outro.

[01:14] Mas se eu quiser parar a execução desse container, eu posso executar o comando `docker stop` e passar para ele o ID ou o nome do container que eu quero. Eu posso copiar o ID ou o nome do container que eu quero parar e colar logo após o comando `docker stop`.

[01:38] Ele vai demorar um tempo e vai parar a execução desse container. Então no momento em que damos um `docker ps` de novo, não teremos mais nenhum *output*. Não temos mais nenhum container em execução.

[01:53] E se eu quiser por algum motivo reexecutar o meu container que foi parado? Vou fazer `docker ps -a`.

[02:01] Eu posso pegar o ID do meu container e simplesmente fazer `docker start`, e passar o ID dele mais uma vez. Nesse momento ele vai voltar a executar o meu container.

[02:13] Se fizermos um `docker ps` de novo, ele está em execução. Ele foi criado há 11 minutos, mas o status dele é de execução só há 4 segundos.

[02:21] Então conseguimos parar e reexecutar os nossos containers com esses dois comandos, o `docker start` e o `docker stop`.

[02:32] Para que eu consiga a fazer inicialização a partir de um *start* é preciso que meu container esteja em estado de parada. E vice-versa: se eu quiser parar ele precisa estar no estado de execução.

[02:43] Vamos um pouco mais além. O meu container do Ubuntu que estamos usando para exemplificar está em execução. Mas eu como é que eu interajo com ele? Porque a princípio nós não estamos conseguindo fazer nada. Ele está em execução, mas não está servindo para nada.

[03:02] Então como é que eu posso interagir com esse container de maneira interativa para que eu consiga fazer alguma coisa?

[03:09] Existe um comando, também dentro desse universo todo do Docker, que é o `docker exec`. Eu posso simplesmente falar que eu posso executar algum comando dentro do meu container em modo interativo.

[03:23] Então para que seja em modo interativo eu vou adicionar `docker exec -it` ao comando. O I é de modo interativo, e o T é que eu quero acessar o tty, o terminal padrão desse container.

[03:37] E eu coloco também o ID do container que eu quero fazer isso, e em seguida eu posso executar algum comando.

[03:43] Qual comando eu posso usar para navegar dentro do meu Ubuntu, que estamos assumindo esse conhecimento de terminal? Eu posso executar simplesmente um `bash`.

[03:56] Ele alterou meu terminal, agora ele está com um usuário *root* nessa máquina, e se formos comparar é exatamente o ID do meu container.

[04:05] Então agora eu estou dentro do terminal do meu container. Então tudo que eu fizer agora, como vimos naquela questão dos *namespaces*, estará devidamente isolado.

[04:14] Se voltarmos à apresentação, por exemplo, nós conseguiríamos ver que na verdade tudo está devidamente isolado por conta dos *namespaces*. Nós temos processo isolado, rede isolada, a intercomunicação de processos, *file system*, o Kernel.

[04:33] Se voltarmos para o terminal e acessarmos, por exemplo, a *home* desse container e criarmos um arquivo `touch eu-sou-um-arquivo.txt`, o que vai acontecer? Eu vou dar um `ls` e vai aparecer o meu arquivo.

[04:52] Mas se eu vier na minha máquina e abrir minha *home* não aparece nada, porque são sistemas de arquivos isolados, graças ao *namespace* de MNT. É o *file system* que está completamente isolado.

[05:07] Nós podemos fazer diversas coisas agora nesse Ubuntu, porque ele vai estar devidamente isolado do nosso sistema. Nós poderíamos executar os comandos `apt`, com `apt-get update`, `apt-get install`, conforme nossa demanda, e vai estar tudo isolado do nosso sistema original.

[05:29] Vou parar essa execução do `apt update`, porque não estamos nos importando com isso e vou limpar a tela com “Ctrl + L”. Tudo que fizermos estará dentro desse terminal, desse container.

[05:40] No caso do Ubuntu, podemos até executar o comando `top`, e ele vai mostrar os processos que estão em execução, do nosso usuário *root*, que é exatamente nosso *sleep* que estava mantendo o nosso container em execução, e o nosso *bash*, que estamos escutando agora.

[05:55] Então nosso container está com dois processos neste momento no nosso usuário *root*.

[06:00] Vou parar novamente e dar um “Ctrl + L”. Vou dar um “Ctrl + D” para sairmos do container e um `docker ps` de novo. Ele ainda vai estar em execução porque o nosso *sleep* ainda está em execução.

[06:15] Eu vou dar um `docker stop` nesse meu container e reexecutar. E vou dar um `docker start` mais uma vez para vermos o que acontece.

[06:41] Mais uma vez vou executar aquele comando para acessarmos o terminal em modo interativo. E nesse momento, se eu voltar para minha *home* e der um `ls`, está aparecendo meu arquivo.

[06:53] Agora vou dar um `top`. Ele simplesmente resetou toda minha árvore de processos, ele deu um sinal de *SIGKILL*, por assim dizer, em toda minha árvore de processos, porque eu parei todos os processos e os recomecei. Então o tempo de execução deles foi zerado. Toda a contagem do processo foi reiniciada.

[07:13] Então essa é uma questão que quando executamos o `stop` acontece. Vou sair do terminal mais uma vez e executar outro comando como, por exemplo, `docker pause` e passar o ID do nosso container de novo. Com isso nós podemos também pausar o nosso container.

[07:34] Se fizermos `docker ps` ele vai aparecer ainda, mas no status de pausado, e não de parado. E se dermos um `docker ps` de novo, repara que a contagem do status dele continua.

[07:50] Se eu tentar acessar o container agora eu não vou conseguir, porque ele está pausado. E eu posso simplesmente despausar esse container com o comando `docker unpause`.

[07:59] E se eu tentar acessar agora eu vou conseguir. E se eu der um `top` vemos que nossa árvore de processos a princípio foi mantida, porque ele mantém todo o nosso fluxo de execução, assim como nossos arquivos. Mas essa é uma situação menos agressiva em relação ao *stop* que podemos fazer.

[08:20] Então nós vimos como podemos parar um container, dispará-lo, no caso, voltar a execução. Vimos como podemos pausar e despausar e a diferença entre cada um desses tipos de operação.

[08:35] E vimos também que devidamente os nossos containers estão isolados do nosso *host* original, graças à utilização do *namespace* NMT.

[08:48] Eu vou fazer um último detalhe para ficarmos com uma pulga atrás da orelha. Eu vou dar um `docker ps`. Mas agora eu vou pegar esse meu container ID e fazer um `docker stop` mais uma vez.

[09:05] E se eu quiser evitar que ele fique demorando 10 segundos para executar eu posso colocar a *flag* `-t=0`, antes do nome do meu container, para que não tenha nenhum tempo para parar, porque por padrão ele espera 10 segundos para nosso container parar de maneira saudável. Mas se quisermos nos apressar, podemos colocar o `-t=0`, que ele vai parar instantaneamente.

[09:31] E agora eu posso remover esse meu container com `docker rm`, que é o comando de remoção, passando o ID do meu container. Com isso ele vai remover.

[09:46] Se eu criar mais uma vez com o nosso `docker run Ubuntu sleep 1d` e manter um *sleep* de um dia, ele vai travar nosso terminal. E por fim, se eu abrir um novo e fizer um `docker exec -it 48aac971d7fb bash` neste nosso terminal que foi criado, nós acessamos o nosso terminal. Se entrarmos na nossa *home* com `ls` o nosso arquivo sumiu.

[10:30] Como o nosso container deixou de existir e ele estava completamente isolado dos outros containers e do nosso *host*, todo o conteúdo dele foi perdido.

[10:39] Então o container tem essa questão de ser efêmero nesse sentido. Os containers devem estar sempre prontos para deixar de ser executados, e nós devemos estar prontos para perder esses dados caso não configuremos nada em relação a isso. Então é uma questão que também veremos mais à frente, como lidar com a persistência de arquivos em container.

[10:58] E para finalizar, mais um comando interessante é o seguinte: ao invés de ter que fazer `docker exec` toda hora depois de dar um *sleep*, teria que ter uma maneira mais prática de já começarmos a executar nosso container e mantê-lo em execução sem precisar ficar dando *sleep*, sem precisar ficar roubando nesse sentido.

[11:18] Nós podemos simplesmente dar um `docker run ubuntu bash`, mas indo mais além, podemos falar que queremos executá-lo em modo interativo, assim como nosso exec: `docker run -it ubuntu bash`. E vamos simplesmente criar um container novo, e já estamos diretamente no terminal dele.

[11:39] Se abrirmos mais um terminal e dermos um `docker ps`, nós temos o que criamos com o *sleep* de um dia ainda há pouco, e o nosso *bash* que acabamos de criar há 10 segundos.

[11:52] Se fizermos qualquer coisa dentro dele, como `ls`, criar algum arquivo na *home*, dá certo. Mas no momento em que eu sair desse terminal ele vai matar o meu container, por aquele motivo que explicamos. Agora não há mais nenhum processo, antes tinha o nosso *bash*, agora não tem mais o nosso terminal que estava travando a execução de um processo para que o container se mantivesse em execução.

[12:20] Então a partir de agora, no momento em que finalizamos a execução do nosso único processo, o container morreu.

[12:27] Nós vimos agora alguns pontos que são muito importantes nessa parte de ciclo de vida dos containers, e entendemos que por eles serem devidamente isolados, nós não conseguimos manter as informações de alguma maneira. Mas vamos responder isso também em breve, fique tranquilo, fique tranquila.

[12:43] Essa aula foi para entendermos mais alguns comandos que são muito úteis e interessantes, e alguns parâmetros que podemos e devemos utilizar com o Docker. Eu te vejo na próxima e até mais.

Run vs Exec

Recentemente, vimos sobre o comando `docker run` e `docker exec`. Sabemos que ambos os comandos envolvem o fluxo de inicialização e execução de comandos em containers, porém em contextos diferentes.

Selecione a alternativa com a diferença do funcionamento entre esses comandos.

O `docker run` cria um novo container e o executa. O `docker exec` permite executar um comando em um container que já está em execução.

Alternativa correta! Esta é a grande diferença entre ambos.

Mapeando portas

Agora veremos um exemplo mais visual para vermos todo o fluxo, e veremos como interagir efetivamente com nosso container para ver uma saída, um *output* real, mais bonito, para realmente vermos como devemos interagir com um container.

[00:16] Ao invés de voltarmos a exe cutar `docker run` com Ubuntu e *hello-world*, vamos um pouco mais além. Vamos executar um exemplo prático de uma aplicação web cuja saída vamos conseguir visualizar através do nosso navegador.

[00:29] Que aplicação é essa? Se voltarmos no Docker Hub, existe um grupo de usuários, uma organização chamada “dockersamples”, que disponibiliza diversos tipos de aplicação a fim de exemplificar a utilização do Docker, um pouco mais bonitas e elegantes.

[00:48] Então repara que ao contrário da nossa imagem do Ubuntu, essa imagem do “dockersamples” não é oficial.

[00:56] E além de ela não ter aquele símbolo de verificado, como no Ubuntu, quando uma imagem não é feita por usuários reconhecidos pela comunidade ela segue o seguinte padrão: vai ter um nome do usuário ou organização barra o nome da imagem. Então conseguimos reconhecer que essa imagem não é oficial por conta desse padrão.

[01:17] Então é uma boa prática você sempre tentar manter a utilização de imagens oficiais dentro do seu projeto o máximo possível, mas nesse caso estamos utilizando essa imagem a fim de exemplificar e visualizar o Docker em execução com um container.

[01:34] Eu vou copiar mais uma vez o nome dessa imagem e vou fazer `docker run`, passando o `dockersamples/static-site`.

[01:44] Mas vimos quando executamos o Ubuntu que se simplesmente executarmos um `docker run` e o nosso container ficar em execução, assim como fizemos com o *sleep* naquele momento, ele vai travar o nosso terminal.

[01:55] Então se eu quiser executar esse comando e manter o comando em *background* no terminal, para que eu consiga manter o terminal em execução sem travar, eu posso passar a *flag* `-d`, de *detached*.

[02:07] Então ele vai executar o container baseado nessa imagem, mas não vai travar o terminal. Vamos ver o que vai acontecer. Ele vai efetuar o download das camadas necessárias para que esse container baseado nessa imagem "dockersample/static-site" execute.

[02:24] Nós já vimos esse fluxo, ele está indo no Docker Hub, validando todas as camadas que ele precisa para fazer a execução desse container, que é basicamente essa imagem que ele encontrou. E está fazendo a verificação depois de fazer todo o download e extração das camadas na nossa máquina.

[02:42] Então ele terminou a extração, fez o download, e repare que ele não travou nosso terminal, graças à *flag* `-d`. Mas agora se eu executar um `docker ps`, vamos ver o que vai acontecer.

[02:57] Nós temos nosso `docker ps`, nosso ID do container, nossa imagem baseada, o comando que ele manteve em execução; que ele foi criado há 22 segundos e que ele está em execução há 19 segundos.

[03:13] Então ao contrário do Ubuntu e do "hello-world", por exemplo, a imagem que foi usada para criação desse container definiu que o comando para o container ser executado trava o terminal.

[03:25] Então esse comando que começa com `/bin/sh -c`, é um comando que mantém um processo vivo dentro do terminal do nosso container.

[03:37] Então o container em si vai continuar em execução. Se fizermos `docker ps` várias vezes, a aplicação em si está em execução, graças ao comando padrão que foi executado quando o container subiu.

[03:47] E agora temos a nossa coluna de portas falando que a nossa aplicação está sendo executada na porta 80 e na 443, além do nosso nome, que é "friendly_lamarr".

[03:59] Então se nossa aplicação está sendo executada na porta 80, vamos abrir no nosso navegador o nosso "localhost:80". Não acessou. Por quê? Mais uma vez voltando à nossa apresentação, graças aos *namespaces*, nesse caso principalmente ao NET, nós temos um isolamento das interfaces de rede.

[04:20] Então a porta 80 do meu container não é exatamente uma porta que já está mapeada na minha máquina, no meu *host*. Eu não vou conseguir acessá-la diretamente assim, elas estão isoladas.

[04:30] Então vamos voltar ao nosso terminal para entender o que está acontecendo. Precisamos visualizar o seguinte: essa porta 80 é de uso do container, e é a porta 80 de dentro da interface de rede do container. Se eu quiser acessá-la de outras maneiras, nós precisamos expor essa porta de alguma maneira.

[04:53] Mas antes disso inclusive, precisamos fazer outra coisa. O nosso container já está em execução. Nós vamos parar e remover esse container de uma vez só, então podemos ser um pouco mais agressivos.

[05:10] Ao invés de fazer o `docker stop` e depois o `docker rm`, podemos copiar o ID e fazer `docker rm 1b6d75073457`, passar diretamente o ID e acrescentar `--force`. Assim ele vai parar e remover o container de uma vez só. Agora se eu fizer `docker ps`, não tem mais nenhum container em execução.

[05:30] Vamos executar agora mais uma vez o `docker run -d -P dockersamples/static-site`. Só que ao invés de só executarmos esse comando mais uma vez, agora vamos colocar a *flag* -P, com P maiúsculo.

[05:45] Vamos descobrir agora o que essa *flag* vai fazer. Ele vai executar o nosso container, sem travar mais uma vez, por conta do `-d`; e ele não fez o download porque ele já tinha o conteúdo na nossa máquina agora.

[05:56] E se fizermos um `docker ps`, ele está fazendo um mapeamento maluco na nossa coluna de portas, que está um pouco difícil de entender. Mas o resto é a mesma coisa.

[06:11] Vamos executar um comando agora `docker port b0e93e405db6` seguido do ID, que é um comando voltado para mostrar como está o mapeamento de portas de um container em relação ao *host*.

[06:29] Então vamos passar o container, e agora ele está falando que a porta 80 do meu container foi mapeada para a porta 49154 do meu *host*.

[06:40] Isso significa que se agora no meu navegador eu executar o "http://localhost:49154", nós conseguimos acessar o nosso container. O conteúdo do nosso container foi acessado, porque agora fizemos um mapeamento de uma porta interna do container para uma porta do nosso *host*.

[07:12] E no fim das contas nós poderíamos ter feito isso também de uma maneira mais bem definida. Vamos executar o nosso `docker rm`, passando o ID do nosso container com `--force`. Vamos executar um novo container.

[07:34] Como eu matei o container, se voltarmos para o navegador e dermos um "F5", não tem mais o container, não tem mais conteúdo.

[07:39] Mas se executarmos o `docker run -d -p dockersanples/static-site` mais uma vez, só que agora com a *flag* `-p` com P minúsculo, nós conseguimos fazer um mapeamento específico de uma porta do nosso *host*.

[07:51] Por exemplo, vamos mapear a porta 8080 do nosso *host*. Ela deve refletir em qual porta do nosso container?

[07:57] Nós vimos que por padrão ele expôs naquelas colunas de porta as portas 80 e 443. Então quero que a porta 8080 da minha máquina reflita na porta 80 do meu container.

[08:09] Ele vai executar. E agora, se voltarmos para o navegador e acessarmos "localhost:8080", conseguimos fazer esse mapeamento de uma porta agora específica do nosso *host* para o nosso container.

[08:23] Voltando na nossa apresentação mais uma vez, nós temos esse isolamento de rede, mas conseguimos fazer um mapeamento para que consigamos acessar o conteúdo do nosso container e vê-lo de maneira que consigamos validar o que está acontecendo.

[08:38] Fazemos isso para que não fiquemos às cegas do que está sendo executado dentro do nosso container e para que, no fim das contas, nós consigamos expor nossa aplicação também para que algum usuário consiga acessar.

[08:50] Nessa aula fizemos essa parte de voltar a execução e entender mais alguns atalhos, como a *flag* `-d`, por exemplo, que mantém nosso terminal destravado; e também fazer o mapeamento de portas entre o nosso *host* e o nosso container.

Visualizando mapeamentos

As flags `-p` e `-P` são úteis quando queremos fazer o mapeamento de portas entre nosso container e o nosso host. Existe um comando responsável pela visualização de como o mapeamento de portas de um container está sendo feito.

Selecione a alternativa que corresponde a este comando.

`docker port`

Alternativa correta! Este comando é responsável por exibir como o mapeamento de portas de um container está sendo feito.

Faça como eu fiz: Acessando portas externamente

Agora nossa proposta é executar um container e validar que, através da flag `-p`, é possível acessá-lo diretamente a partir do host. Lembrando que já sabemos como executar containers, porém, devido ao isolamento, ainda não conseguimos validar o funcionamento deles. Vamos fazer essa execução e validação?

Inicialmente, execute o comando `docker run -d dockersamples/static-site` para que o seu host baixe a imagem e execute o container em seguida.

A seguir, verifique se o seu container está em execução sem problemas através do comando `docker ps` ou `docker container ls`. Repare na coluna `PORTS` e veja que seu container está informando que possui uma aplicação que pode ser exposta tanto na porta 80 quanto na 443.

Obtendo esta informação, remova o container recém-criado com o comando `docker container rm <id-do-container> --force`.

Seu objetivo agora será criar um novo container fazendo o devido mapeamento de portas. Para isso, execute o comando `docker run -d -p 8080:80 dockersamples/static-site`. Repare que através da flag `-p`, estamos informando que a porta 8080 de nosso host irá refletir na porta 80 do container.

Por fim, acesse em seu navegador `localhost:8080` e veja a aplicação sendo carregada dentro de seu container.

Com a utilização da flag `-p`, é possível fazer uma ponte entre a aplicação dentro do container e o nosso host. Este cenário é muito interessante quando queremos testar o funcionamento e como uma aplicação está interagindo com outras.

Entendendo imagens

Chegou o grande momento de explicar o que são as imagens. Nós vamos entender efetivamente agora o que são as imagens, como elas funcionam, como elas viram containers e como criar nossas próprias imagens, porque não podemos depender só do trabalho alheio para desenvolver o nosso.

[00:15] Então chegou agora a hora de entender o que são as imagens. Por enquanto nós estamos aceitando que as imagens são uma receita para criar um container. Mas efetivamente como elas funcionam?

[00:28] Uma imagem nada mais é do que um conjunto de camadas, que na imagem estamos representando pelas cores vermelha, azul, rosa claro e cinza. É um conjunto de camadas, e quando juntamos essas camadas nós formamos imagens.

[00:45] E essas camadas em si são independentes, cada uma tem seu respectivo ID, por exemplo, cada uma tem o seu respectivo identificador.

[00:55] Então vamos voltar mais uma vez para o caso do dockersamples no nosso terminal para visualizarmos.

[01:04] Quando fazemos um `docker run` na nossa imagem, nós podemos ver agora as imagens que temos baixadas no nosso sistema através do comando `docker images`, ou `docker images ls`.

[01:19] Dando um `docker images` eu posso ver que tenho baixada a nossa imagem "dockersamples/static-site", com a *tag latest* e seu respectivo ID. E ela foi criada há 5 anos pelo grupo do dockersamples. E o tamanho dela é de 191 MB.

[01:39] Nós podemos ir um pouco mais além. Podemos dar o comando `docker inspect` em uma imagem. Vou fazer novamente o `docker images` para copiar o ID da imagem. E agora vou fazer `docker inspect`, passando o identificador do que nós queremos inspecionar.

[02:00] Nós temos diversas informações. Tem um conjunto muito grande de informação que podemos saber detalhadamente sobre determinado recurso dentro do nosso Docker.

[02:10] Temos o ID, qual é a *tag* do repositório, o *digest* que foi utilizado para validação da imagem, se tem alguma imagem que é um *parent*, uma imagem pai ou mãe, a data de criação, o container, container *config*. Então temos diversas informações acerca dos recursos que podemos ter dentro do Docker.

[02:36] Inclusive, no final nós conseguimos ver mais informações na parte de *layers*, que são as camadas. Mas podemos ir um pouco mais além. Tem um comando específico para ver quais são as camadas de uma imagem. Vou limpar o terminal.

[02:53] Temos o comando `docker history`. Vou fazer `docker images` para pegar mais uma vez o ID. E vamos fazer o comando `docker history`, passando esse ID. Nós temos a nossa imagem, que é a f589ccde7957, e ele mostra todas as camadas. Ela tem 13 camadas para essa imagem.

[03:28] E quando essas camadas são aglutinadas, empilhadas umas nas outras, elas formam essa imagem final que é "dockersamples/static-site".

[03:38] E caso alguma outra imagem venha a depender desses camadas, nós conseguimos reutilizá-las. Vamos entender isso daqui a pouco.

[03:45] Mas ele mostra detalhadamente qual é o tamanho de cada uma dessas camadas, a ordem de cada uma delas, o *command*, *workdir*, *copy*. Nós conseguimos ver também a data de criação. Conseguimos ver todas essas informações e entender o que está acontecendo.

[04:05] Mas agora que nós entendemos que uma imagem é um conjunto de camadas empilhadas para formar determinada regra de execução de um container, voltando à nossa apresentação, vamos ver como podemos visualizar um pouco melhor.

[04:22] Quando fizemos nosso `docker run` pela primeira vez ou simplesmente um `docker pull` para não executar o container, mas só baixar a imagem, nós fazemos o download das nossas imagens, das nossas camadas.

[04:36] Mas pode ser que, por exemplo, como eu falei para vocês agora, no nosso *host* nós já tenhamos algumas das camadas que queremos. Então no momento em que fizermos um `pull` ou um `run`, que vai fazer um `pull` consequentemente, nós vamos fazer simplesmente download só das camadas que necessitamos.

[04:52] Então o Docker é inteligente o suficiente para reutilizar essas camadas para compor novas imagens.

[04:58] Então conseguimos ter uma performance muito boa nesse sentido, já que não precisaremos ter informação duplicada ou triplicada, enfim. Porque nós conseguimos reutilizar as camadas em outras imagens. Isso por si só já é muito interessante. Então no fim das contas, conseguimos ter essa reutilização.

[05:17] Mas o que mais podemos ver na parte da criação de imagens? No fim das contas, quando temos a nossa imagem, ela é *read only*. Isso significa que não conseguimos modificar as camadas dessa imagem depois que ela foi criada.

[05:34] Então voltando ao nosso container, no momento em que nós temos essa imagem "dockersamples/static-site", ela é imutável, assim como, por exemplo, a imagem que temos do nosso Ubuntu.

[05:55] Vou fazer mais uma vez `docker run ubuntu –it bash` para executar de modo interativo o nosso *bash*. Eu já tinha apagado a imagem, e veremos como fazer isso aos poucos, mas vamos baixar a camada necessária para ter a nossa imagem de execução para que consigamos executar nosso container do Ubuntu. Ele baixou, extraiu e vai abrir o nosso terminal.

[06:26] Ele deu um erro porque eu coloquei na ordem errada. O correto é `docker run –it ubuntu bash`.

[06:37] Nós vimos que tínhamos criado na nossa *home*, por exemplo, um arquivo qualquer com o comando `touch um-arquivo-qualquer.txt`. Nós estamos meio que escrevendo dentro do container.

[06:47] Mas como estamos conseguindo fazer isso se a imagem que gera o nosso container é apenas para leitura, ou *read only*? Se ela é bloqueada para escrita como é que o container consegue escrever informação dentro dela?

[07:01] Porque no fim das contas, quando criamos o container, o container nada mais é do que uma imagem com uma camada adicional de *read-write*, de leitura e escrita.

[07:13] Então quando criamos um container nós criamos uma camada temporária em cima da imagem, onde conseguimos escrever informações. E no momento em que esse container é deletado, essa camada extra também é deletada.

[07:26] Por isso que quando fizemos aquele experimento anteriormente, a nossa informação dentro do container era perdida quando nosso container era apagado. Porque essa camada é temporária, bem fina e leve para que o container tenha um ambiente de execução muito leve e fácil de ser executado.

[07:43] E agora voltamos naquela primeira pergunta da primeira parte desse curso, que era por que os containers são tão leves?

[07:52] Além da parte de eles serem simplesmente processos dentro do nosso sistema, nós também podemos falar que quando um container entra em execução, nós estamos sempre reaproveitando a mesma imagem.

[08:08] Porque como a imagem é apenas de leitura, nós podemos ter um, dois, 100 ou 10 mil containers baseados na mesma imagem. A diferença é que cada um desses containers vai ter só uma camada diferente de um para o outro de *read-write*.

[08:23] E como essa camada é extremamente leve, a fim de manter essa performance, nós temos uma reutilização da imagem para múltiplos containers.

[08:32] Então no fim das contas o que acontece é que quando definimos um container ou outro baseado na mesma imagem também logo em seguida, o tamanho do container no fim das contas vai ser só o tamanho da camada de escrita que estamos gerando para ele, porque a imagem em si será reutilizada para cada um deles.

[08:52] E isso é muito legal. Nós veremos um experimento prático em breve, conforme formos avançando na criação e no fluxo das nossas imagens.

[08:59] Mas basicamente agora nós entendemos porque efetivamente o container é tão leve e otimizado. Porque ele consegue reaproveitar as camadas das imagens prévias que já temos.

[09:10] E quando criamos novos containers ele simplesmente só reutiliza as mesmas imagens e camadas, consequentemente, e utiliza a camada de *read-write* para utilizar de maneira mais performática o que ele já tem no ecossistema do Docker. Isso por si só é muito legal.

[09:28] E no fim das contas, basicamente é isso. Vamos entender ainda a partir de agora como definir um arquivo chamado "dockerfile" que vai nos ajudar a criar nossas próprias imagens, e no fim das contas como gerar os nossos containers através das imagens que vamos criar.

[09:46] Por esse vídeo é só. Agora desmistificamos efetivamente o que é uma imagem, como transformar uma imagem num container e porque o container é tão leve.

[09:56] E ainda vai ficar mais fácil de entender conforme formos avançando e criando a nossa própria imagem e vermos como são as etapas de criar uma camada, uma imagem, transformar em container. Mas isso nós faremos nos próximos vídeos. Eu te vejo lá e até mais.

Detalhes sobre imagens

Anteriormente vimos como as imagens se diferem de containers e como são compostas. Sabemos até então que imagens são "receitas" para gerar containers, porém, ainda precisamos fixar como imagens são construídas e gerenciadas pelo Docker.

Selecione as alternativas verdadeiras sobre a utilização de imagens.

Selecione 2 alternativas

Imagens são compostas por uma ou mais camadas.

Alternativa correta! As camadas são a menor unidade que compõem uma imagem.

Podemos visualizar as camadas de uma imagem através do comando `docker history`.

Alternativa correta! Este comando é responsável por exibir quais são as camadas de uma imagem.

Criando a primeira imagem

Agora vamos criar nossa primeira imagem. Vimos o que é uma imagem, como ela vira um container, qual a diferença de imagem para container.

[00:09] Mas precisamos entender agora como criar nossa imagem efetivamente para não dependermos 100% de imagens de outras pessoas diretamente. Nós vimos na nossa apresentação que precisamos no fim das contas seguir esse fluxo.

[00:23] Nós precisamos definir esse arquivo que é o Dockerfile. E a partir dele vamos criar nossa imagem. E imposta nossa imagem basta executar usando *run* para que o container seja gerado a partir da imagem.

[00:37] Voltando ao nosso projeto como um todo, nós temos uma aplicação Node. Não precisa se preocupar, nós não vamos entrar em nenhum detalhe específico de Node e de nenhuma linguagem de programação, você não precisa saber Node.

[00:53] Só estamos usando como exemplo para termos uma aplicação efetivamente para empacotar e transformar numa imagem e depois num container. Não precisa se preocupar quanto a isso.

[01:04] O que nós queremos no fim das contas é que quando formos executar nosso container e acessarmos via *host*, por exemplo, mapeando as portas, que tenhamos essa visualização, que é a nossa aplicação realmente em execução.

[01:15] Mas não queremos simplesmente clicar duas vezes no arquivo e abrir. Nós queremos um servidor que disponibiliza essa aplicação para nós. Então nós precisamos de alguma maneira colocar todo esse conteúdo dentro de uma imagem, instalar o Node, que será responsável por executar o servidor.

[01:32] E no fim das contas, quando nosso container executar, queremos que ele execute algum comando que mantenha esse servidor em execução.

[01:38] Então o que precisamos fazer de início é o seguinte: eu estou utilizando o Visual Studio Code para fazer a edição de texto nesse caso, mas você pode usar o da sua preferência.

[01:49] O que vou fazer é criar um novo arquivo dentro da nossa pasta do nosso exemplo, que estará disponível para você fazer o download. E dentro dessa pasta vou criar um arquivo chamado "Dockerfile", que é basicamente o arquivo que nós vamos criar. E vou dar um "Enter".

[02:14] Repara na parte superior esquerda que o VS Code tem esse charme de reconhecer que é um arquivo Dockerfile. E agora nós vamos simplesmente, dentro desse arquivo, definir como vai ser a criação da nossa imagem. O que nós queremos fazer?

[02:30] Nós queremos que dentro do nosso projeto como um todo nós tenhamos o Node para que consigamos rodar um servidor.

[02:46] Então se queremos usar o Node como base para nossa aplicação, nós podemos pegar emprestado do que já desenvolveram. Então vamos fazer o `pull` dessa imagem já existente para que possamos utilizar no nosso projeto, e a partir daí só fazer as nossas modificações para customizar o projeto à nossa maneira.

[03:09] No fim das contas nós precisamos do Node. Mas como nós colocamos o Node dentro da nossa imagem por padrão? Nós podemos colocar a princípio um Ubuntu, e dentro desse Ubuntu podemos instalar o Node e fazer toda a configuração necessária.

[03:26] Mas lembra que não precisamos ter necessariamente um sistema operacional dentro do nosso container. Nós podemos simplesmente utilizar alguma imagem que disponibilize o Node para nós, por exemplo, a própria imagem do Node no Docker Hub, que é uma imagem oficial, inclusive.

[03:43] Se olharmos a descrição como um todo, tem todas as versões do Node que nós podemos definir, como as versões 16, 17, 14, 12.

[03:53] Então o que nós podemos fazer nesse cenário? Nós podemos simplesmente falar que queremos pegar uma dessas versões do Node para que possamos executar o nosso projeto, usar essa imagem como base para a nossa. Então a partir da imagem que nós vamos definir nós vamos começar a usar a nossa.

[04:15] Como podemos pegar essa imagem emprestado? Por exemplo, queremos usar o Node na versão 14, então dentro do nosso "Dockerfile" eu quero pegar a partir do Node na versão 14. Como eu explicito a versão? Utilizando dois pontos e a versão que eu quero: `FROM Node:14`. Então a partir do Node na versão 14.

[04:40] Como eu sei que é a versão 14? Porque na documentação da imagem do Node ele está mostrando quais são as *tags* suportadas, inclusive a 14. E a partir do Node na versão 14, o que nós queremos fazer? Queremos colocar todo o nosso projeto, que são esses arquivos, menos o próprio "Dockerfile", dentro dessa imagem. Então nós queremos copiar esse conteúdo do nosso *host* para a nossa imagem.

[05:18] Para isso podemos simplesmente colocar `COPY`. Nós queremos copiar todo o conteúdo do nosso diretório atual. Em que diretório está o nosso "Dockerfile"? No diretório "exemplo.node".

[05:30] Então todo o conjunto do nosso diretório atual nós queremos copiar para algum diretório dentro do nosso container, por exemplo, para uma pasta chamada "/app-node", só para termos a distinção do que nós queremos fazer: `COPY . /app-node`.

[05:42] Então a partir desse momento nós estamos copiando esse conteúdo do diretório do nosso *host* para o diretório de dentro da nossa imagem que vai virar um container, chamada "/app-node".

[05:58] Nós queremos executar o comando `RUN npm install`. Só que esse comando terá que ser executado dentro do nosso diretório /app-node, para que consigamos instalar as dependências da nossa aplicação.

[06:11] Caso você não conheça Node, esse comando está sendo responsável só por instalar as dependências que o nosso projeto precisa em um projeto Node. Então basicamente estamos instalando as dependências e o Node está resolvendo isso automaticamente.

[06:24] E por fim nós queremos que o ponto de entrada do nosso container, ao executar esta imagem e começar a ter seu container devidamente em execução, seja startar a aplicação, então eu faço `ENTRYPOINT npm start`.

[06:40] E isso também tem que ser executado dentro desse diretório /app-node. Só que nós teríamos que ficar passando em todos os lugares o "/app-node". Será que podemos resolver isso de uma maneira mais simples?

[06:55] Eu quero que, por exemplo, esses comandos todos, por padrão, sejam executados no meu diretório que eu estou atualmente. E como é que eu defino o diretório que a imagem vai tratar como padrão? Qual será o meu diretório de trabalho, por assim dizer?

[07:10] Para isso eu tenho a instrução `WORKDIR`, e com ela podemos definir o nosso diretório padrão: `WORKDIR /app-node`.

[07:21] Inclusive, no nosso *COPY* eu posso fazer um *COPY* de ponto. Esse ponto é o nosso diretório atual dentro do nosso *host*, para ponto também, que vai ser o nosso diretório atual dentro da nossa imagem: `COPY . .`. E qual será nosso diretório atual? O nosso /app-node, que foi definido através do nosso *WORKDIR*.

[07:44] Então o que nós estamos fazendo? Nós estamos definindo que vamos utilizar imagem do Node na versão 14 como base para nossa imagem. E nós vamos definir o nosso diretório de trabalho padrão como sendo o /app-node.

[07:58] Vamos copiar do diretório atual, onde está o nosso "Dockerfile" do nosso *host*, que é a pasta "exemplo-node", para a pasta atual dentro da nossa imagem "/app-node" que foi definida dentro do nosso *WORKDIR*.

[08:14] E vamos executar o comando `npm install` enquanto a imagem estiver sendo criada. Esse comando será executado na etapa de criação da imagem. Então esse `npm install` será executado enquanto a imagem estiver sendo criada.

[08:26] E quando o container for executado a partir dessa imagem, o comando executado vai ser o `npm start`. Vamos dar um "Ctrl + S" agora, vamos ao nosso terminal e vamos acessar `cd Desktop/exemplo-node/`.

[08:45] Como é que a partir desse Dockerfile eu posso gerar uma imagem? Através do comando `docker build`.

[08:54] Também passamos o `-t` para podermos criar um nome, etiquetar nossa imagem. No caso vou colocar o meu nome, então `docker build -t danielartine/app-node:1`. Com o `:1` nós podemos explicitar qual é a versão que estamos criando. Eu vou colocar a versão 1.0 só para ficar mais bonito. Então `docker build ´t danielartine/app-node:1.0`.

[09:18] Em qual contexto tudo isso terá que ser executado? No contexto de diretório atual, ou seja, ponto, que é a referência ao diretório atual: `docker build -t danielartine/app-node:1.0 .`.

[09:26] Eu dou um “Enter”, e no Docker Hub, nesse exato momento, ele vai pegar a imagem do Node na versão 14 e baixar. Então ele vai pegar todo esse conteúdo para a nossa máquina, como já vimos que o Docker faz, e vai construir uma nova imagem utilizando essa como base.

[09:47] Então no momento em que isso terminar nós vamos executar o comando `docker history` para ver o que ele vai fazer no fim das contas e ver como essa imagem vai se comportar dentro do nosso sistema.

[09:58] Com o que precisamos nos preocupar agora? Fazendo um breve apanhado no nosso Dockerfile, ele está na etapa `FROM Node:14`, onde ele foi no Docker Hub e pegou a imagem do Node na versão 14. É isso que ele está fazendo agora.

[10:15] Mas um ponto interessante: você pode me perguntar como eu sei dessas instruções, como eu as deduzi. Essa é uma pergunta muito boa.

[10:22] Nós podemos entender tudo isso através da própria documentação do Docker, que eu vou mostrar para vocês. É uma documentação muito completa, na qual podemos e devemos sempre nos basear para seguir os nossos projetos e criação de imagens.

[10:38] Nós temos todas as principais instruções para a criação de uma imagem. Ele mostra os comandos e as principais instruções. Ele tem o *FROM*, que usamos agora há pouco; ele mostra como todas as sintaxes funcionam, como escapar caracteres.

[10:57] Temos o `ADD`, o `COPY`. Nós ainda não vimos algumas, mas veremos, por exemplo, o `ENV`, o `EXPOSE`. Já vimos o `FROM`. Com o `LABEL` podemos etiquetar e colocar algumas *labels* na nossa imagem; podemos definir algumas questões de volume, que ainda não sabemos o que é; o `WORKDIR`, que já vimos.

[11:15] Ele tem diversos exemplos que podemos ver como funcionam dentro da documentação e aplicar aos nossos projetos. Então vou deixar também o link da documentação nas atividades para vocês conseguirem ver como é que funciona.

[11:29] Mas ainda teremos outros exemplos de criação de imagens. Esse é só o primeiro para entendermos realmente como vai funcionar.

[11:35] Voltando para o terminal, ele está terminando de extrair nesse momento as últimas camadas dos downloads que ele fez da imagem do Node. Então vamos ver qual vai ser o resultado final.

[11:53] Ele está agora na etapa de `WORKDIR`, depois copiando os arquivos, executando o `npm install`. E por fim, no `ENTRYPOINT` ele definiu o `npm start`. Agora vou limpar o nosso terminal e vou fazer `docker images`. Olha o que temos agora.

[12:09] Temos “danielartine/app-node” na versão 1.0 e temos o ID dessa imagem. E se agora eu simplesmente fizer um `docker run` nessa imagem “danielartine/app-node” e fizer um mapeamento?

[12:27] Lembra que nós definimos a nossa aplicação dentro do container, mas ela é isolada? Então como eu posso agora saber em qual porta essa aplicação está rodando dentro do meu container?

[12:40] A princípio nós precisaríamos ser um pouco malandros. Se olharmos dentro do nosso “index.js”, veríamos que ela está sendo executada na porta 3000. Tem um breve problema que precisaremos resolver. Mas vamos colocar isso no terminal.

[12:55] Na nossa máquina nós vamos querer na porta 8080 de novo, mas nós queremos que a porta 8080 reflita na porta 3000, que é onde vimos que nossa aplicação vai ficar em execução dentro do nosso container: `docker run -p 8080:3000 danielartine/app-node`.

[13:13] Vou colocar também um `-d` para ficar em modo *detached*. E faltou especificar a versão, que é a 1.0, então `docker run -d -p 8080:3000 danielartine/app-node:1.0`.

[13:25] Na verdade ele deu um problema porque a porta 8080 já está em uso por causa dos nossos exemplos anteriores. Obviamente não podemos usar a mesma porta, então vou mudar para 8081.

[13:35] E agora se viermos no nosso navegador e tentarmos acessar o "http://localhost:8081", está tudo funcionando, conseguimos acessar a nossa aplicação agora de maneira conteinerizada.

[13:50] Então nós criamos nossa própria imagem e executamos um container a partir dela. Isso é muito legal.

[13:56] Mas ainda tem algumas questões que precisamos resolver. Nós deixamos em aberto aquela questão de como saber em qual porta nossa aplicação estava em execução, como sabemos como expor ou não expor.

[14:07] Tem algumas questões que precisamos deixar menos obscuras para o nosso Dockerfile, para que consigamos deixar tudo muito mais fácil de ser entendido.

[14:18] Mas vamos terminar esse vídeo por aqui. Nós criamos nossa primeira imagem, mas precisamos entender mais alguns detalhes acerca da criação de imagens. Mas por esse vídeo é só. Eu te vejo na próxima e até mais.

Já aprendemos que usamos Dockerfiles para criar nossas imagens quando queremos utilizar containers. Para isso, utilizamos algumas instruções.

Dentre as alternativas, escolha a que melhor define a utilização da instrução `FROM`.

A instrução `FROM` é usada para definirmos uma imagem como base para a nossa.

Alternativa correta! Desta maneira, podemos adicionar à nossa imagem conteúdos que utilizaremos de maneira mais prática.

Faça como eu fiz: Construindo uma imagem

Agora criaremos nosso primeiro Dockerfile para poder gerar nossa primeira imagem. Consequentemente, teremos nosso primeiro container próprio com o Docker. Durante as etapas anteriores, compreendemos a composição de imagens através de camadas e vimos a necessidade de criar imagens próprias a fim de ter um ambiente para executar nossas aplicações.

Você pode efetuar o download do projeto node através deste link.

Inicialmente, crie um diretório com um nome à escolha. Dentro dele, crie um arquivo chamado `Dockerfile` e extraia o conteúdo da pasta do zip neste mesmo diretório. Este será o arquivo usado para o Docker interpretar as instruções e construir a imagem final.

Com seu editor de texto favorito, edite o arquivo e adicione o seguinte conteúdo:

```
FROM node:14
WORKDIR /app-node
COPY . .
RUN npm install
ENTRYPOINT npm start
```

Com estas instruções acima, estamos informando ao Docker que queremos usar a imagem do `node` na versão 14 como base para nossa imagem. Definimos que nosso diretório padrão para executar os comandos dentro

do container será o `/app-node` e em seguida copiamos todo o conteúdo do diretório atual Dockerfile para o diretório `/app-node` dentro do container.

Por fim, em tempo de construção da imagem, executamos o comando `npm install` e definimos que, ao executar o container gerado por esta imagem, o comando executado será o `npm start`.

Ainda dentro do diretório do Dockerfile, através do terminal, execute o comando `docker build -t <seu-nome-de-usuario-do-docker-hub>/app-node:1.0 .`. Dessa forma, sua primeira imagem será criada. Confira através do comando `docker images` se sua imagem está sendo listada.

Saber como criar e definir imagens é de suma importância, pois elas são a base para o funcionamento de um futuro container.

Incrementando a imagem

Agora vamos voltar ao nosso projeto e entender algumas questões que ficaram pendentes e que poderíamos aperfeiçoar na criação do nosso Dockerfile, até mesmo em relação ao projeto.

[00:11] Primeiro vamos voltar ao nosso terminal e antes de mais nada, vamos entender um comando novo, que na verdade nem é tão novo, é a junção de dois comandos que já conhecemos. Vamos fazer `docker ps` e olhar que eu estou com vários containers em execução.

[00:30] Se eu quiser parar todos de uma vez, como eu faço? Eu posso colocar o comando `docker stop`, e eu posso simplesmente falar que eu quero parar todos os meus containers passando os IDs deles. Para fazer isso basta eu executar junto o comando `docker container ls`.

[00:49] Só que não tão rápido assim. Eu preciso falar que eu quero executar o comando `docker container ls` e usá-lo como entrada para o meu *stop*. Para isso eu uso `docker stop $(docker container ls)`.

[01:01] E no fim das contas também eu preciso falar que eu quero pegar só o ID, então eu coloco a *flag* `-q`, de *quiet*. Assim ele pega só o ID, `docker stop $(docker container ls -q)`.

[01:11] Vou dar um "Enter". Ele vai ter aqueles 10 segundos para parar os containers de maneira segura, por assim dizer. E ele vai parar todos os containers a partir desse momento.

[01:22] Mas enquanto ele vai parando, o que podemos observar no nosso "Dockerfile"? Nós construímos a nossa imagem, e no momento em que executamos, o que acontece?

[01:32] Vamos mais uma vez no terminal fazer um `docker run -p 8080:3000 -d danielartine/app-node:1.0`. Ele rodou. E agora vamos dar um `docker ps` mais uma vez.

[02:01] Repara que ele está mostrando para nós o mapeamento de portas que estamos fazendo graças à *flag* `-p`. Mas e se eu executar esse mesmo container sem fazer o `-p` e colocar só o `-d`? Vamos ver o que vai acontecer. Vamos dar um `docker ps` de novo.

[02:21] Repare que ele não está exibindo nada na coluna de portas. E sabemos que a nossa aplicação roda na porta 3000.

[02:28] Então como é que poderíamos documentar isso para que outras pessoas que fossem utilizar o nosso container posteriormente, baseado na nossa imagem, soubessem que a aplicação está exposta na porta 3000?

[02:40] Existe na verdade uma maneira que podemos fazer essa documentação para que fique explícito em qual porta a aplicação está sendo executada. Basta colocarmos a instrução `EXPOSE`.

[02:57] No nosso caso, colocamos `EXPOSE 3000`. Nesse momento estamos falando que a nossa aplicação estará exposta na porta 3000. Isso não é obrigatório, até porque já fizemos anteriormente e funcionou esse mapeamento.

[03:09] Mas agora vamos fazer o seguinte teste: vou salvar o arquivo e gerar uma nova imagem com o comando `docker build -t danielartine/app-node:1.1 .`. Coloquei a versão 1.1 e o ponto no final, que é o nosso diretório atual.

[03:31] Ele está fazendo todo o processo de *build*. E no momento em que eu executar agora mais uma vez o container sem fazer nenhum tipo de definição de porta, vamos ver o que vai acontecer. Se eu der um `docker ps`, repare que ele fala que a porta 3000 está exposta.

[03:55] Então qualquer pessoa que olhar agora vai saber que a porta 3000 tem alguma aplicação dentro daquele container executando na porta 3000 do container. Então não precisamos adivinhar, e fica muito mais fácil de fazer um possível mapeamento de portas a partir daí.

[04:12] Mas mais uma vez, vamos aperfeiçoar ainda mais a nossa imagem. Vamos entender o que está acontecendo.

[04:17] Nós deixamos exposta a porta 3000. Mas o que poderíamos fazer se a nossa aplicação estivesse em um cenário diferente? Porque vimos que no "index.js" estamos definindo que a porta 3000 é efetivamente a porta da nossa aplicação.

[04:32] E se quiséssemos fazer isso no momento da criação da nossa imagem, de maneira mais parametrizada, através de uma variável de ambiente?

[04:43] Nós podemos simplesmente usar só uma sintaxe muito específica do Node, que seria colocar um `process.env.PORT`. PORT é o nome da variável que estamos definindo. Com isso basicamente estamos atribuindo que queremos ler uma variável de ambiente chamada PORT.

[05:07] A partir desse momento podemos definir na nossa imagem o seguinte: nós vamos querer agora receber esse parâmetro definido na nossa imagem.

[05:18] Então nós podemos simplesmente definir que nós vamos ter um argumento, que será usado para definir essa variável de ambiente dentro do nosso container posteriormente, e que será a porta que queremos utilizar. Para isso podemos fazer, por exemplo, `ARG PORT=6000`.

[05:39] E depois vamos colocar o `EXPOSE` com essa porta. E como eu pego o valor desse argumento, dessa variável que estamos utilizando dentro da criação da nossa imagem? É só colocar `EXPOSE $PORT`.

[05:52] Mas tem um pequeno detalhe: esse `ARG` só funciona em tempo de criação, de *build* da nossa imagem.

[06:00] E se eu quiser passar isso efetivamente para dentro do meu container que será gerado? Ou seja, se eu quero que em algum momento essa variável possa ser lida dentro do meu container, eu preciso explicitar também um outro tipo de variável de ambiente para dentro do container, que é uma `ENV`.

[06:19] O `ARG` só é usado em tempo de *build* da imagem, e o `ENV` será usado dentro do container posteriormente. Então podemos colocar também `ENV PORT=$PORT`.

[06:32] Essa variável ambiente PORT que nós estamos definindo para dentro do container terá um valor previamente definido por essa variável $PORT.

[06:40] Nós podemos até ser um pouco mais semânticos e trocar para `ARG PORT_BUILD=6000` e `ENV PORT=$PORT_BUILD`, e também `EXPOSE $PORT_BUILD`.

[06:52] Agora vamos buildar essa imagem mais uma vez, agora na versão 1.2: `docker build -t danielartine/app-node:1.2 .`.

[07:05] Ele vai buildar todo o nosso processo. E agora eu vou fazer o `docker run` da nossa versão 1.2 sem definir nenhuma porta para ver se vai acontecer o que nós esperamos. E depois fazer um `docker ps`. Ele está mostrando a porta 6000.

[07:22] E agora nós já sabemos, olhando para nosso `docker ps`, que a porta que precisamos fazer algum mapeamento caso queiramos acessar esse container é a porta 6000.

[07:32] Então vamos fazer novamente nosso `docker run -p` da porta 9090 da nossa máquina, na porta 6000 do nosso container e em modo *detached*: `docker run -p 9090:6000 -d danielartine/app-node:1.2`. Vamos executar isso e voltar para nosso navegador, acessando o "localhost:9090". Conseguimos acessar nossa aplicação mesmo assim.

[08:07] Nós conseguimos definir variáveis de ambiente para dentro do nosso container. Ou seja, conseguimos fazer a leitura dessas variáveis e colocá-las dentro do container.

[08:16] Temos também as variáveis que são explícitas, específicas para a parte de construção da nossa imagem, que é o caso do `ARG`. O `ARG` é usado para construção da imagem e o `ENV` é usado posteriormente dentro do container.

[08:29] E conseguimos também utilizar a instrução de `EXPOSE`, para ser mais semântico e deixar claro para as pessoas que vão utilizar nosso container posteriormente que a aplicação que estará ali dentro está exposta em determinada porta.

[08:42] Então conseguimos deixar a nossa imagem ainda mais fácil de ser manuseada posteriormente, além de tornar mais parametrizável através das variáveis de ambiente.

[08:53] Por esse vídeo é só. Deixamos nossa imagem agora um pouco mais robusta. E vamos entender a partir de agora como fazer o *deploy* dessa imagem, como colocá-la no Docker Hub. Veremos isso no próximo vídeo. Eu te vejo lá e até mais.

ARG vs ENV

Vimos como podemos utilizar variáveis em nossas imagens e containers através das instruções `ARG` e `ENV`. Exemplificamos durante a aula o cenário de passar alguma informação para ser carregada dentro do container, por exemplo, qual será a porta que a aplicação usará para executar.

Assinale a opção que contém a principal diferença entre elas.

A instrução `ARG` carrega variáveis apenas no momento de build da imagem, enquanto a instrução `ENV` carrega variáveis que serão utilizadas no container.

Alternativa correta! Desta maneira, é possível diferenciar o que é necessário para a etapa de build e para a etapa de execução.

Subindo a imagem para o Docker Hub

Agora vamos fazer o *push* da nossa imagem para o Docker Hub. O primeiro passo é que você crie sua conta na parte direita da própria *home* do Docker Hub. Você define seu *username*, seu e-mail e sua senha, aceita os termos e marca o recaptcha. Depois é só clicar em "Sign Up" e confirmar sua conta por e-mail.

[00:20] Depois que você dizer isso, no canto superior direito tem a parte de “Sign In”. Você vai colocar o seu usuário, que no meu caso é “aluradocker”, mas você terá o seu. E também a senha que você usou no momento do cadastro.

[00:32] Quando você fizer o login ele vai carregar e na parte superior, na barra de navegação, tem a parte de repositórios. Se clicarmos, veremos que nós ainda não temos nada, porque nós simplesmente já criamos nossa imagem, mas ainda não a subimos para o Docker Hub.

[00:48] E como vamos fazer isso? Vamos voltar ao nosso terminal, e o primeiro passo que precisamos fazer é autenticar a nossa conta também no *Command Line Interface* do Docker, para que ele saiba que somos realmente quem nós somos.

[01:03] Então faço o comando `docker login -u aluradocker`. O `aluradocker` é o meu nome de usuário. Quando você der um “Enter” ele vai pedir a senha que você usou no momento do seu cadastro no Docker Hub. Ele não vai mostrar o número de caracteres por questão de segurança, mas ele vai digitar.

[01:22] Você vai dar um “Enter” e ele vai dar um *warning* de que sua senha foi armazenada de maneira não criptografada nesse caminho.

[01:30] Agora vou dar um “Ctrl + L” e um `docker images`. Agora nós queremos simplesmente colocar a imagem “danielartine/app-node” na versão 1.0 no Docker Hub.

[01:44] Nós precisamos usar um comando específico para isso. Ao invés de ser o `docker pull`, vai ser o `docker push`, seguido do nome da imagem e sua versão. Com isso ele vai saber que ele deve fazer esse *push* automaticamente para o Docker Hub.

[02:02] No momento em que dermos o “Enter”, ele vai preparar todas as camadas, mas repare que ele deu um acesso negado. Isso quer dizer que nós não temos permissão para colocar essa imagem “danielartine/app-node” na versão 1.0. Mas por quê?

[02:21] Vamos lembrar daquela questão, por exemplo, do “dockersamples/static-site”. Nós temos o nome do usuário ou da organização, barra e o nome da imagem.

[02:36] E qual é o meu nome de usuário efetivamente nessa conta? É “aluradocker”. Então não faz sentido eu ter permissão de fazer um *push* em nome de “danielartine/app-node”. Assim como não teria se eu tivesse uma imagem “dockersamples/alguma coisa”.

[02:51] Então eu vou voltar ao meu terminal e fazer `docker images` mais uma vez só para visualizar. Eu quero gerar uma cópia dessa imagem, mas com uma nova *tag*, um novo repositório na coluna que queremos gerar.

[03:07] Para fazer isso basta executarmos o comando `docker tag`. Então `docker tag danielartine/app-node:1.0`. E em seguida eu coloco a imagem que eu quero gerar, então fica: `docker tag danielartine/app-node:1.0 aluradocker/app-node:1.0`.

[03:25] No momento em que eu apertar o “Enter”, ele vai fazer sem nenhum *output*. E se eu der um `docker images` de novo, agora temos o “aluradocker/app-node” na versão 1.0. Inclusive com o mesmo ID, mas agora com um repositório diferente.

[03:41] Agora eu vou tentar efetuar mais uma vez o meu *push*, mas agora com a versão correta, que é o repositório de aluradocker: `docker push aluradocker/app-node:1.0`.

[03:52] Ele vai começar a fazer o *push*, só que agora efetivamente, sem dar nenhum problema de acesso, porque ele consegue, a partir desse momento, fazer o meu *push* de maneira bem mais segura, porque ele sabe

que eu sou realmente a pessoa que deveria ter essa permissão, baseado no nome de repositório que definimos para a nossa imagem.

[04:16] Enquanto ele vai fazendo o *push*, eu vou dar uma breve pausa no vídeo. Assim que ele terminar eu volto para seguirmos.

[04:22] Repara que agora o *push* terminou. Isso quer dizer que se agora formos ao Docker Hub na nossa parte de repositórios, vai aparecer a nossa imagem que acabamos de subir. Se clicarmos sobre nossa imagem, vemos a versão 1.0.

[04:44] Só que agora veremos uma coisa mais legal ainda. No terminal vamos dar um "Ctrl + L" e fazer o `docker images` mais uma vez. Nós temos a "danielartines/app-node" também na versão 1.2.

[04:57] Então se eu fizer um `docker tag` dessa imagem na versão 1.2 para "aluradocker/app-node", também na versão 1.2, eu vou fazer um *push* dessa nova versão.

[05:11] Só que qual vai ser a grande sacada? Ele vai fazer o processo de novo, só que agora ele sabe que diversas dessas camadas já existem no Docker Hub, e ele só faz o *push* das camadas necessárias.

[05:24] Então até o Docker Hub também consegue ser inteligente nessa parte. Ela já sabe que uma camada já está lá e ele consegue aproveitar para gerar uma imagem nova com as camadas já existentes. E isso é muito legal.

[05:36] Se voltarmos ao Docker Hub e atualizarmos a página da nossa imagem, além da nossa tag 1.0, nós também teremos a nossa 1.2, ambas com OS do Linux, e mostrando que o *pull* de uma foi feito há um minuto e ou outro há poucos segundos.

[05:54] Agora nós aprendemos a fazer o *push* das nossas imagens, gerando *tags* novas também localmente. E vimos que o Docker Hub é capaz de reutilizar as camadas já existentes também no nosso repositório remoto. E isso é muito legal.

[06:08] Então essa parte de criação de imagens nós finalizamos agora, e vamos começar um novo tópico. Só que isso nós veremos no próximo vídeo. Eu te vejo lá e até mais.

O problema de persistir dados

Antes de seguirmos no nosso fluxo original, vamos revisar os comandos importantes que já vimos.

[00:10] Vamos dar um `docker ps`. Não temos nenhum container em execução. Se fizermos um `docker ps -a`, temos vários containers parados.

[00:18] Vamos dar aquela revisada e remover todos os containers. Então vou fazer `docker container rm`. E vou usar como entrada a saída do meu comando `docker container rm $(docker container ls -aq)`.

[00:32] Na *flag* `-aq`, o q é para pegarmos somente os IDs e o `-a` é porque eu quero pegar todos os meus containers, inclusive os que estão parados. Se eu der um "Enter", ele vai remover. Vamos fazer um `docker images`. Nós temos algumas imagens.

[00:52] Agora o que vamos fazer é um `docker rmi $(docker image ls -aq)`. Com isso ele está removendo todas as imagens. Repara que ele deu um conflito.

[01:09] Ele está falando que algumas imagens estão sendo referenciadas por múltiplos repositórios, então precisamos forçar essa remoção. Vamos fazer o mesmo comando, mas passando o `docker rmi $(docker image ls -aq) --force` no final. Assim ele consegue agora remover essas imagens também.

[01:26] Então caso você não consiga, a *flag* `--force` vai dar essa força para nós. Agora se nós fizermos um `docker images`, não temos mais nada, foi tudo removido.

[01:37] Qual é a grande mágica que veremos durante essa etapa do curso? Vamos voltar às origens e dar um `docker run -it ubuntu`, porque eu quero iniciar um container do Ubuntu em modo interativo. Nós vamos entender o porquê.

[01:54] Vou dar um "Enter" para ele baixar a imagem rapidamente. Nesse caso, como a imagem é bem curta, vai ser rápido, não precisamos fazer nenhum corte nesse caso.

[02:06] Estou com meu container já praticamente em execução, ele está terminando de extrair. E eu vou abrir já outro terminal e fazer `docker ps`. E com isso ele mostra as informações desse container.

[02:25] Até então nada de novo. Mas existe uma *flag* bem interessante de vermos, que é a `-s`. Então se fizermos `docker ps -s` ou `docker container ls -s`, repara que surgiu uma coluna extra. Nessa coluna ele fala que o tamanho desse container é 0B, mas o tamanho virtual dele é 72.8 MB. O que isso quer dizer?

[02:53] Vamos voltar à nossa apresentação original falando sobre imagens, como elas funcionam e afins. Lembra que uma imagem, no fim das contas, é um conjunto de camadas que estão empilhadas uma em cima da outra? Nós conseguimos, aliás, ver essa informação com `docker history`.

[03:12] Se eu fizer um `docker history` na imagem do Ubuntu, por exemplo, vejo que ela é composta por duas camadas, uma de 0B e outra de 72.8MB.

[03:21] Então repara que o tamanho virtual do meu container é meio que o tamanho total da minha imagem. Isso faz total sentido, porque no fim das contas o container nada mais é do que a imagem com uma camada extra de *read-write*.

[03:43] No momento em que criamos esse container, até então não tem nenhum dado dentro dele além das informações originais da imagem, então o tamanho dele vai ser igual ao tamanho da imagem. Mas o tamanho mesmo dele efetivamente será 0B. O tamanho virtual dele vai ser só o tamanho da imagem no fim das contas.

[04:03] Vamos voltar ao nosso outro terminal e fazer algumas outras operações. Por exemplo, vamos fazer `apt-get update`. Vamos fazer algumas operações dentro do nosso container para ver o que vai começar a acontecer com aquele tamanho que estamos vendo.

[04:19] Ele vai atualizar o repositório, nós podemos criar alguns arquivos. Você também pode fazer os experimentos que você quiser. Eu só estou dando uma atualização no repositório.

[04:29] Se eu fizer agora `docker ps -s`, repara que o *size* dele já está em 16.2MB. Agora o tamanho virtual do meu container é o tamanho que era original da imagem que tínhamos com o `docker history` mais o tamanho das informações de dados que eu tenho dentro do meu container.

[04:56] Então essa informação na coluna de *size* nada mais é do que as informações que estão agora na nossa camada de *read-write*. São informações a princípio temporárias, porque lembra que essa camada é fina e temporária, só para informações que serão escritas dentro daquele container.

[05:14] E é por isso que se criarmos outros containers a partir da imagem do Ubuntu, teremos cada um com uma camada de *read-write* diferente e os containers terão o mesmo tamanho base no fim das contas, para que consigamos fazer essas operações. Mas cada um terá a sua camada de escrita separada.

[05:34] Então repara que se dermos um `docker ps -s` mais uma vez, o tamanho já vai estar praticamente o dobro, porque o `apt-get update` já foi executado provavelmente.

[05:42] Mas é aquela velha história: se eu voltar no meu container, sair dele e der um `docker ps`, ele já não está mais em execução. Mas agora eu vou criar um novo container com `docker run -it ubuntu bash`.

[05:59] Se eu voltar no meu outro terminal e fizer `docker ps -s`, teremos um novo, que é a mesma história: repara que o tamanho dele zerou, porque as camadas de *read-write* são isoladas umas das outras, cada container terá a sua.

[06:13] Se fizermos `docker ps -sa`, nós temos agora o antigo, que não está mais em execução, e o outro que ainda está em execução.

[06:22] Então agora precisamos entender como podemos persistir essas informações de alguma maneira para que containers que já foram removidos e talvez subam de novo com alguma informação mantenham esses dados.

[06:35] Porque vimos que essa camada não é persistente entre containers, ela também não é persistente caso eu remova esse container e suba um novo, eu não vou ter essa informação mais. Então como podemos persistir essas informações?

[06:49] A primeira informação que podemos seguir de uma maneira, além da documentação, é utilizar os volumes. Existem três tipos principais.

[06:58] Um deles é a parte do *bind mount*, que é uma maneira com que podemos fazer um *build* entre o *file system* do nosso sistema operacional e o sistema de arquivos do nosso container. Então teremos uma ponte entre essas duas partes, que vai persistir essa informação no nosso *host*.

[07:11] Temos o volume efetivamente, que será gerenciado pelo Docker. Mas vamos entender tudo isso em mais detalhes. E tem o tmpfs, que é temporário. Mas vamos entender a utilidade dele também.

[07:22] Agora que já entendemos o problema que precisamos enfrentar e como resolver ele, com essas três possíveis soluções, vamos terminar essa aula por aqui para ficar com essa parte mais instigada de entender com o que vamos resolver. Eu te vejo no próximo vídeo e até lá.

Utilizando bind mounts

Agora nós vamos entender como utilizar a primeira solução que o Docker disponibiliza para persistência de dados, que se dermos uma olhada na documentação, são os *bind mounts*.

[00:14] Ele vai fazer basicamente o *bind*, uma ligação entre um ponto de montagem do nosso sistema operacional e algum diretório dentro do container. Vamos entender agora como isso vai funcionar.

[00:27] Eu vou parar mais uma vez os containers com o comando de `docker rm $(docker container os -aq) --force`. E agora vamos executar mais uma vez aquele `docker run -it ubuntu`. E a questão agora é a seguinte: o que podemos definir no momento em que vamos executar um container?

[00:53] Eu posso informar que quando esse container for criado e executado eu quero que as informações persistidas em determinado diretório dentro dele sejam persistidas em algum diretório na minha máquina local mesmo, no meu *host*.

[01:06] Então além de definir os comandos básicos que já definimos e a *flag* de `-it` também, eu posso colocar uma *flag* de `-v`.

[01:16] Vamos ver o que essa *flag* faz. Eu vou criar uma nova pasta para esse caso específico. Já estou na minha *home*, vou criar um pasta com o comando `mkdir`, chamada `volume-docker`. Pode ser o nome que você quiser criar para essa pasta.

[01:40] E agora, voltando à nossa execução, eu quero que esse diretório que esteja na minha pasta "/home/daniel/volume-docker" corresponda a um determinado caminho dentro do meu container.

[01:59] Então eu coloco dois pontos e informo que, por exemplo, dentro do container eu terei um diretório chamado "/app", e tudo que for gravado dentro desse diretório será persistido nesse diretório no meu *host*, `docker run -it -v /home/daniel/volume-docker:/app Ubuntu bash`.

[02:13] Vamos ver como isso vai funcionar. Eu vou colocar ainda um *bash* no final e dar um "Enter". Estou dentro. Se eu der um `ls` agora, repara que ele tem esse diretório "/app". Agora vou acessar com `cd app/`. E vou criar um arquivo com `touch arquivo-qualquer.txt`. Vamos ver o que vai acontecer.

[02:43] Se eu simplesmente abrir o gerenciador de arquivos e entrar na pasta "volume-docker", está ali o meu "arquivo-qualquer.txt".

[02:52] Então eu consegui fazer essa definição de colocar um caminho dentro do meu diretório da minha máquina local para um diretório dentro do container e salvar esse arquivo.

[03:03] Mas como isso vai funcionar agora? Eu quero parar esse container. Vou dar um "Ctrl + D" e vou criar um novo container, definindo o mesmo *bind mount*, o mesmo caminho na minha máquina para esse diretório "/app".

[03:17] E vai ser um novo container, porque estamos dando um novo `run`, então sabemos que será uma nova camada de *read-write* para esse container: `docker run -it -v /home/daniel/volume-docker:/app ubuntu bash`. E vou dar um "Enter".

[03:27] Se agora eu der um `ls`, repare que eu vou continuar tendo minha pasta "/app". E se eu der um `cd app` e um `ls` em seguida, o meu "arquivo-qualquer.txt" ainda aparece.

[03:38] Então conseguimos agora persistir informações entre containers. Caso um container pare de funcionar de alguma maneira e queiramos persistir os dados que estejam lá, já conseguimos fazer isso agora, e de maneira a princípio prática.

[03:50] Nós só definimos um diretório dentro do nosso *host* para um diretório do nosso container e está tudo feito, 100% funcionando. É muito fácil e prático nesse sentido.

[03:59] Mas qual é a outra maneira que podemos fazer e que inclusive vem sendo mais recomendada pelo Docker para criação de volume?

[04:07] Mais uma vez, em conjunto com a documentação, repara que a maneira que estamos utilizando é com a *flag* `-v`.

[04:14] Mas vem sendo recomendado, por ser mais semântico, a utilização além da *flag* `-v`, a *flag* `--mount`. Vamos pegar um exemplo para ver o que ela faz.

[04:26] No terminal vou dar um "Ctrl + D" de novo e vou criar um terceiro container agora, só para vocês verem que eu não estou roubando, não estou com nenhum container execução.

[04:34] Vamos criar o terceiro container agora utilizando essa *flag* `--mount`. Como é que ela funciona? No nosso caso vamos fazer um *mount* do tipo *bind*, porque é um *bind mount*. E a nossa *source*, o diretório da nossa máquina, vai ser mais uma vez, no meu caso, `docker run -it --mount type=bind,source=/home/daniel/volume-docker,target=/app ubuntu bash`. E o meu *target* será um "/app" também dentro desse container.

[05:07] Vou dar um "Enter". Deu um erro, vou ver o que eu errei. Ficou o `-v` sobrando, nós não precisamos mais dele. Então vou tirar e dar um "Enter".

[05:28] Está dando erro de novo, foi porque eu escrevi meu nome errado, acontece. Vou arrumar, e agora sim.

[05:39] Isso é até uma coisa interessante: se o diretório que você estiver tentando utilizar não existir no seu *host*, a *flag* `--mount` vai falar que esse caminho não existe. Não foi ensaiado, mas é bom pegarmos esses erros, porque vamos pegando esses possíveis cenários.

[05:53] Agora dando um "Enter" tudo funcionou. Se eu der um `ls` tenho o meu app. E seu eu der `cd app/` e em seguida um `ls`, aparece o meu arquivo.

[06:03] Então tudo continua funcionando, utilizando tanto da maneira com a *flag* `--mount` quanto com a *flag* `-v`. Conseguimos agora persistir os dados entre os containers e também entre os próprios containers nas execuções seguintes para que os dados sejam persistidos caso seja necessário.

[06:20] Se tiver algum arquivo de configuração, que a sua aplicação precisa executar e coisas afins, nós conseguimos ter isso agora.

[06:26] Nós veremos alguns exemplos mais robustos posteriormente, mas até então vamos entender os conceitos primordiais que baseiam toda a utilização de volumes, *bind mount* e tmpfs.

[06:41] Mas vamos ficar com a pulga atrás da orelha no final dessa aula com o seguinte questionamento: será que é interessante o nosso container depender de um caminho dentro do nosso *host*?

[06:52] Pode ser que nós escrevamos como aconteceu agora, de um caminho não existir e dê algum problema; ou não tenhamos permissão para acessar; ou alguém simplesmente delete esse caminho localmente, porque ele estará no *host*.

[07:06] Então são N cenários que podem acontecer e que devemos nos preocupar. E para isso vamos conseguir utilizar uma solução ainda mais robusta e que é mais recomendada pelo Docker, que são os volumes em si, que serão gerenciados pelo próprio Daemon do Docker. Mas isso veremos no próximo vídeo. Eu te vejo lá e até mais.

Criando um bind

Anteriormente entendemos que os bind mounts são capazes de persistir dados de containers através de um vínculo criado com a estrutura de pastas do nosso host. Porém, ainda precisamos fixar como criar um.

Qual das alternativas abaixo contém a sintaxe correta para a criação de um bind mount?

```
docker run –mount type=bind,source=/home/diretorio,target=/app nginx
```

Alternativa correta! Com esta sintaxe, criaremos um bind mount para o container baseado na imagem do nginx.

Utilizando volumes

A segunda solução, que inclusive é a mais recomendada para se usar em ambientes produtivos e afins pelo Docker, é a utilização de volumes.

[00:08] Se olharmos a imagem que está presente na documentação, ele mostra que a utilização de volumes é uma área gerenciada pelo Docker dentro do seu *file system*.

[00:17] Então por mais que no fim das contas as nossas informações continuem dentro do nosso *host* original para ser persistidas, nós teremos uma área que o Docker vai gerenciar e é muito mais segura a nível de alguém mexer e fazer alguma loucura ali dentro, porque será gerenciada pelo próprio Docker.

[00:33] E como criamos um volume inicialmente? Vamos voltar no nosso terminal. Não estou com nenhum container em execução.

[00:41] Existe, dentre os diversos comandos que temos no Docker, um comando chamado `docker volume`. Podemos até dar um `docker volume ls`, para vermos que não temos nenhum criado. Nós podemos simplesmente criar. Eu posso dar um `docker volume create` e colocar em seguida nosso `meu-volume`.

[01:02] Quando eu der esse comando, ele vai criar, e se dermos um `docker volume ls`, ele mostra nosso volume, usando nosso driver local do nosso *host*, com o nome que demos.

[01:13] Mas onde é que ele está na nossa máquina? Como é que eu sei que ele vai gravar essas informações? Nós vamos chegar lá.

[01:19] Mas antes vamos fazer o mesmo experimento que fizemos anteriormente. Eu vou executar o `docker run` com a *flag* `-v`, como fizemos da maneira original. Só que ao invés de colocarmos o mesmo caminho de antes, eu não quero mais definir nenhum diretório dentro da minha máquina manualmente.

[01:40] Então eu vou simplesmente explicitar que eu quero utilizar o "meu-volume", que é o nome do meu volume, e ele será mapeado nesse meu diretório "/app" dentro do meu container, `docker run -it -v meu-volume:/app ubuntu bash`. Vou dar um "Enter" e um `ls`. Vou entrar no meu /app mais uma vez.

[02:00] Ele está vazio, porque repara que por mais que o /app seja igual, agora nós estamos utilizando um ponto dentro do nosso *host* diferente. Antes estávamos utilizando o diretório na nossa *home*, mas agora estamos usando um novo volume, que é esse "meu-volume" gerenciado pelo Docker.

[02:15] Mas se seguirmos a mesma receita, vamos criar um arquivo qualquer com `touch arquivo-qualquer.txt`. Vamos criar um novo container, seguindo o mesmo comando: `docker run -it -v meu-volume:/app ubuntu bash`. Se eu fizer agora um `cd app/` e um `ls`, temos agora o nosso arquivo qualquer.

[02:32] Então tudo continua funcionando. Mas a pergunta que precisamos responder agora é: onde está esse arquivo? Porque antes nós sabíamos que se viéssemos na nossa *home* e acessássemos a pasta "volume-docker" ele estaria dentro dela.

[02:47] Mas esse meu volume está aonde? Vamos descobrir esbanjando conhecimento para conseguirmos distribuir para outras pessoas também e mostrar o que sabemos, porque é importante sabermos disso.

[03:00] Eu vou entrar com o super usuário, fazendo `sudo su` e colocar minha senha. Agora existe um diretório onde o Docker está realmente na nossa máquina. Não o Docker em si, mas diversas informações que ele armazena na nossa máquina. Ele grava em "/var/lib/docker/".

[03:23] Se dermos um `ls`, nós temos diversas informações, como plug-ins, *buildkit*, imagens, *overlay*, e inclusive volumes. Se fizermos `cd volumes/`, nós encontramos o "meu-volume".

[03:37] Então se eu fizer um `cd meu-volume/`, ele vai ter um *hash* doido, dentro dessa pasta de data, e dentro dele estará o nosso arquivo qualquer.

[03:46] Então nós sabemos agora onde os nossos arquivos estão, porque agora por mais que eles estejam ainda dentro desse caminho, eles estão num lugar completamente gerenciado pelo Docker. Então conseguimos utilizar os comandos do Docker para gerenciar esse volume.

[04:02] Se sairmos do modo de super usuário e fizermos `docker volume` simplesmente sem passar nada, ele vai mostrar que conseguimos criar volumes, inspecioná-los, listá-los, remover os que não estão sendo usados, remover independentemente se estiver sendo usado ou não.

[04:18] Então conseguimos gerenciar esses volumes agora através da interface do Docker. Não ficamos 100% dependentes do nosso sistema de arquivos do nosso sistema operacional. E sim o Docker vai gerenciar isso para nós agora.

[04:31] Isso é muito interessante, porque agora não dependemos diretamente de uma estrutura de pastas específicas do nosso sistema operacional. Ele estará sempre nesse diretório de volumes.

[04:43] E por fim, para finalizar essa parte da criação de volumes, mais uma vez vamos olhar na documentação. Também temos a possibilidade de criar um volume com a *flag* `--mount`.

[04:54] E é até mais fácil nesse caso, porque por padrão ele já assume que o tipo que vamos criar é um volume. Então não precisamos colocar o tipo, como fizemos com o *bind*.

[05:04] Então vamos voltar ao terminal e executar o comando de criação de um container. Vamos colocar o `--mount`, sem o `type=bound`, porque não é mais necessário. E a *source* será o `meu-volume`, e o *target* continua para "/app", `docker run -it --mount source=meu-volume, target=/app ubuntu bash`. Se fizermos o `ls` dentro de /app, temos o nosso arquivo qualquer.

[05:31] E ainda tem uma graça dos volumes também que é o seguinte: vamos criar mais um container, mas agora vou colocar, por exemplo um `meu-novo-volume` em *source*, que é um volume que eu não tenho criado até então. Ele vai fazer isso automaticamente para nós.

[05:45] Se eu der um `ls app/` vai estar vazio, porque eu estou usando um volume novo. Mas se eu sair desse container agora e der um `docker volume ls`, vemos que ele criou já esse volume automaticamente.

[05:59] Então conseguimos ter esse controle também e o Docker vai nos ajudar nesse sentido. Então não precisamos se preocupar necessariamente em criar, porque como é gerenciado pelo Docker ele pode fazer isso por nós. E isso é muito interessante.

[06:13] Agora já vimos as duas principais formas de criar maneiras de persistir dados, tanto *bind mount* como com os volumes em si gerenciados pelo Docker. Mas ainda falta uma terceira forma, que eu vou colocar direto na URL da documentação, que são os tmpfs.

[06:35] Basicamente o nome já indica uma coisa bem peculiar, mas veremos isso no próximo vídeo. Eu te vejo lá e até mais.

Vantagens de volumes

Já aprendemos sobre a possibilidade de usar bind mounts e volumes para persistir dados no ambiente Docker. Porém, com a utilização de volumes, temos uma vantagem em relação aos bind mounts.

Escolha a alternativa que apresenta a vantagem do uso de volumes.

Volumes são gerenciados pelo Docker e independem da estrutura de pastas do sistema.

Alternativa correta! Desta maneira, a persistência de dados independe de como as pastas do sistema estão estruturadas.

Faça como eu fiz: Criando um volume

Agora criaremos nosso primeiro volume gerenciado pelo Docker, com o objetivo de garantir que nossos dados persistam mesmo sem uma preocupação com a estrutura de diretórios de nosso sistema. Dessa forma será possível armazenar e reutilizar arquivos entre execuções de containers, o que é muito útil para aplicações stateful.

Inicialmente, abra o terminal e execute o comando `docker volume ls` e veja que a saída está vazia, pois ainda não criamos nenhum volume até então.

Em seguida, execute o comando `docker volume create meu-volume` e execute novamente o comando `docker volume ls`. Veja que desta vez o nosso recém-criado volume está sendo exibido.

Com o volume criado, agora iremos executar um novo container vinculado ao volume. Para isso, execute o comando `docker run -it -v meu-volume:/app ubuntu bash`. Dentro do container, execute o comando `cd /app` para acessar o diretório e crie um arquivo com o comando `um-arquivo-qualquer`. Saia do container com o comando `Ctrl D` e execute mais uma vez o comando `docker run -it -v meu-volume:/app ubuntu bash`. Por fim, execute o comando `ls /app` e veja que o arquivo `um-arquivo-qualquer` criado anteriormente continua presente.

Utilizando tmpfs

Chegamos ao terceiro tipo que podemos utilizar de persistência com o Docker, nem tanto de dados, que são os tmpfs.

[00:11] Antes de seguirmos, algumas peculiaridades: a primeira é que ele só vai funcionar no *host* Linux. Então por isso é importante estarmos utilizando o Linux, porque diversas funcionalidades, como essa, por exemplo, são feitas para rodar em ambiente Linux.

[00:27] Então essa é a importância de você estar seguindo provavelmente esse curso no Linux também, porque é uma questão que já está disponível dentro desse ambiente.

[00:35] Mas vamos ver como o tmpfs funciona e o que vai mudar. Nós já executamos containers diversas vezes. E agora qual será a diferença?

[00:45] Temos toda a nossa parte de definir como um container vai funcionar. Mas como é que fazemos para utilizar agora um tmpfs?

[00:55] No fim das contas, se fôssemos olhar na documentação, mas eu não vou dar spoiler para vocês, nós vamos defini-lo de maneira bem simples, porque não vamos utilizar o `-v` agora. Ele possui uma *flag* própria já, que no caso é a `--tmpfs`.

[01:11] Se fizermos `docker run -it --tmpfs=/app ubuntu bash` e dermos um "Enter", seguido de um `ls`, veremos que ele criou a pasta "app".

[01:20] Mas repara uma coisa muito importante: diferente das outras vezes que criamos essa pasta através de volumes, agora ela está com um fundo verde, assim como a pasta tmp. O que será que isso quer dizer?

[01:34] Isso significa que essa pasta "/app" é temporária. Então ela está sendo escrita na memória do nosso *host*.

[01:44] Eu vou criar um arquivo dentro do meu diretório app, por exemplo: `touch app/um-arquivo-qualquer`. Vou dar um `ls app/`, para ver meu arquivo. Agora vou sair com `exit` e vou criar um novo. E se eu der um `ls app/` de novo, agora não tem nada dentro dele.

[02:08] Então qual é a utilidade prática desse tipo de armazenamento que não armazena? A ideia do tmpfs é basicamente persistir dados na memória do seu *host*, mas esses dados não estão sendo escritos na camada de *read-write*. Eles estão sendo escritos diretamente na memória do *host*.

[02:37] Isso é importante, por exemplo, quando temos algum dado sensível que não queremos persistir na camada de *read-write* por questões de segurança talvez, mas queremos tê-los dentro de alguma maneira. Nesses casos podemos utilizar o tmpfs.

[02:50] Então esses dados não serão escritos na camada de *read-write*, e sim ficar em memória temporariamente.

[02:55] Então é uma questão de segurança que seria interessante também em alguns cenários como, por exemplo, arquivos de senha ou algum arquivo que você não quer carregar durante a execução como um todo e manter na camada de escrita.

[03:11] Então o tmpfs é basicamente isso. Ele só funciona no *host* Linux. Mas assim como todas as outras abordagens que já vimos, tanto de *bind mount* quanto de volume, ao invés de fazermos a parte de tmpfs com a *flag* `--tmpfs`, podemos utilizar também a *flag* `--mount`, seguindo uma ideia bem parecida com a do *bind mount*, em que definimos o tipo que vamos utilizar.

[03:43] A ideia será a mesma: basta colocar o tipo como tmpfs. E o nosso *destination* nesse caso é o /app dentro do nosso container: `docker run -it --mount type=tmpfs,detination=/app ubuntu bash`.

[03:52] Se executarmos isso e fizermos um `ls`, mais uma vez está o nosso "/app" temporário, porque o fundo está verde, assim como nosso tmp. Se criarmos um arquivo com `touch app/um-arquivo-qualquer` e fizermos um `ls app/`, temos nosso arquivo. Só que se criarmos um novo container de novo, não vai ter nenhuma informação dentro do nosso app.

[04:17] Então essa é a ideia do tmpfs. Caso queiramos colocar algum dado temporário que não deve ser armazenado de maneira alguma na camada de *read-write*, podemos utilizar o tmpfs também.

[04:30] A ideia é basicamente essa. Na questão de persistência de dados no fim das contas nós vimos três possibilidades. Temos o tmpfs, que acabamos de ver.

[04:39] Temos os *bind mounts*, que foi o primeiro que vimos e que fazem um *bind*, uma ligação direta entre o sistema de pastas do nosso *host* e do nosso container.

[04:52] E também os volumes, que são a solução recomendada, inclusive, para utilização, porque eles são gerenciados pelo Docker e conseguimos ter um controle maior sobre isso, sem depender diretamente da estrutura de pastas do nosso *host* efetivamente.

[05:11] Essa parte de persistência de dados nós terminamos por aqui. E agora vamos pensar em algumas questões mais elaboradas, como podemos fazer um grande apanhado de tudo isso que já vimos para colocar alguma aplicação mais legal em produção e ver como isso vai funcionar interagindo com outras questões também.

Conhecendo a rede bridge

Voltando um pouco, lembra que nós vimos sobre a questão de como os containers são isolados em relação ao nosso *host* e como precisamos nos preocupar em como eles vão se comunicar?

[00:09] Porque no fim das contas estávamos debatendo aquela questão de que um sistema complexo é composto por diversas aplicações atualmente.

[00:16] Então pode ser que tenhamos uma aplicação Java se comunicando com uma C#, que se comunica com uma Nginx. Ou podemos pensar num caso clássico de uma aplicação *back-end* se comunicando com um servidor de banco de dados, por exemplo.

[00:28] Então se os containers estão isolados, como podemos lidar com essa questão da comunicação entre containers?

[00:37] Se voltarmos naquela questão dos *namespaces*, nós temos toda aquela parte que já provê isolamento para nós nas interfaces de rede. Mas como será que isso funciona dentro do Docker?

[00:50] Vamos voltar ao nosso terminal. E nesse momento podemos testar o nosso experimento clássico de execução de um container Ubuntu. Então vou fazer `docker run –it Ubuntu bash`. E vou colocar também um *bash* para executarmos.

[01:06] O que vai acontecer nesse momento? Já sabemos que o container vai ficar em execução. Mas eu vou abrir um novo terminal com um `docker ps`. Temos o nosso container que acabou de subir.

[01:21] Existe um comando interessante com que podemos inspecionar as configurações, os detalhes de um container quando ele já está em execução ou até mesmo em outras ocasiões também. É o `docker inspect`. Colocando o ID desse container e dando um "Enter", ele vai dar diversos detalhes.

[01:42] Já no final tem a parte que estamos procurando, que é a parte de networks, de redes. E dentro desse conjunto de redes ele tem uma chamada *bridge* que tem diversas configurações.

[01:56] Mas em que momento nós configuramos essa rede? A questão é que nós não configuramos. Quem fez isso foi o próprio Docker.

[02:03] Vamos fazer uma comparação. Eu vou abrir mais um terminal e executar mais um container do Ubuntu com `docker run –it ubuntu bash`. E vamos comparar a saída desses *outputs* de rede.

[02:25] Então vou abrir mais terminal e fazer um `docker ps` e um `docker inspect` com o ID desse outro Ubuntu que acabamos de criar.

[02:35] Repara que se colocarmos lado a lado, toda essa parte de rede que ele está mostrando é igual. A parte de IPAMconfig como *null*, o network ID é igual.

[Aula5_video1_imagem1]

[02:49] Então todos esses pontos dentro do nosso sistema, exceto o *endpoint* ID e o IP *address*, são iguais. Por quê? Isso significa então que esses containers no fim das contas estão na mesma rede.

[03:04] Mas será que conseguimos fazer algum tipo de comunicação entre eles, já que eles estão na mesma rede, que é um driver que o Docker está colocando para nós? Antes de pensar nisso, precisamos entender o que é a *bridge*.

[03:15] Então vou abrir mais um terminal para ficar tudo bem separado o que estamos fazendo. E existe, dentro de todo o arsenal de comandos que o Docker provê para nós, uma parte sobre redes. Temos o `docker run`, o `docker images`, o `docker build`, e temos o `docker network`.

[03:34] E como eu faço para listar as redes que o Docker já tem no sistema, criado de maneira automática? Basta eu fazer `docker network ls`. Ele mostra três redes para nós, uma se chama *bridge*, que tem seu ID, o *driver bridge*, e um escopo local. Tem uma que se chama *host*, que usa o *driver host* e o escopo também é local. E no fim das contas também temos uma última, que se chama *none*, que poderíamos não colocar nenhuma rede dentro do nosso container.

[04:10] Mas como isso vai funcionar no fim das contas? O que isso tudo significa e por que precisamos nos preocupar com isso?

[04:17] Se compararmos nossos IDs, pegando um dos meus *inspects*, nós temos que o ID da rede começa com 80a1db. Na sua máquina vai ser diferente. E repara que é exatamente a mesma rede que estamos vendo no nosso `network ls`.

[04:33] Isso significa que os nossos dois containers que criamos sem definir nenhuma rede foram colocados nessa rede padrão *bridge*, com esse nome e utilizando esse driver também de *bridge*.

[04:43] E o que isso significa? Significa que se, por exemplo, eu tentar acessar algum desses containers, como o container de ID 8ea67 e fizer um `docker ps` e um `docker inspet`, ele tem o IP *address* dele, que é 172.17.0.2.

[05:17] E se fizermos um `docker ps` no outro, que começa com b02, e fizermos um `docker inspect`, nós temos o IP *address* de 172.17.0.3.

[05:29] Então se eu tentar, por via das dúvidas, acessar o meu b02, cujo IP *address* termina com 03, e por algum motivo eu tentar comunicá-lo com o outro via IP, eu devo conseguir, já que eles estão na mesma interface de rede.

[05:55] Só que como é que isso vai funcionar? Como estamos usando uma imagem do Ubuntu, bem provavelmente, caso tentemos executar algum `ping`, ele não vai conseguir.

[06:04] Então é a velha etapa: nós precisaríamos usar uma imagem base que já contém o ping ou podemos simplesmente fazer `apt-get update`, e depois instalamos o ping para fazer esse experimento.

[06:16] Existem imagens que já vêm com ping, mas como estamos padronizando os nossos primeiros testes com a imagem do Ubuntu, para mantermos o padrão vale nós continuarmos com toda essa parte de utilizar o Ubuntu.

[06:28] Existem outras imagens voltadas para essa parte de teste de rede e afins que também você pode consultar no Docker Hub, mas como eu falei, só na questão de ping mesmo nós vamos fazer esses testes.

[06:40] Então ele vai atualizar os pacotes rapidamente e quando ele terminar vamos instalar o pacote do ping para que consigamos fazer essa comunicação. Assim que ele terminar todo o processo de atualização e instalarmos o ping eu volto.

[06:56] Nós fizemos o `apt-get update`, e logo depois para agilizar eu já executei o comando de instalação também, que foi `apt-get install iputils-ping`.

[07:11] Se eu der um “Enter” ele já vai ter instalado. Mas a ideia é só para fazermos a instalação do ping. Então se eu tentar agora dar um ping no 172.17.0.2, que no caso é o de início 8ea, ele vai fazer a comunicação sem nenhum problema.

[07:57] Então estamos conseguindo fazer essa comunicação entre containers via IP. Mas quais são os problemas que isso pode levantar? Porque estávamos fazendo uma comunicação diretamente via IP. Mas já vimos que os containers estão suscetíveis a reiniciar, a serem recriados e afins. E isso não vai garantir que o contêiner vai ter sempre o mesmo IP. Então teremos uma conexão muito instável nesse sentido.

[08:22] Precisamos ter uma maneira mais certa de fazer isso, como, por exemplo, via um DNS, talvez um *host name* seria interessante.

[08:29] Mas como vamos entender isso? Nós já entendemos primeiro o que são as redes, vimos que podemos containers que estão na mesma rede.

[08:37] Mas vamos aprofundar isso um pouco mais vendo a questão de como podemos criar a nossa própria rede e como ela vai se comportar nesse sentido. Só que isso nós faremos no próximo vídeo. Eu te vejo lá e até mais.

Redes do Docker

Começamos a entender e utilizar redes com o Docker a fim de comunicar diferentes containers e aplicações.

Marque as alternativas que contém os nomes das diferentes redes padrão já criadas pelo Docker.

Host. None. Bridge.

Alternativa correta! Esta é uma das redes que o Docker já cria por padrão.

Criando uma rede bridge

Como podemos fazer para ter uma comunicação mais estável entre containers? Porque vimos que o IP é uma coisa que não podemos garantir.

[00:10] Mas vamos ver um pouco a nossa coluna de informações sobre o container. Eu removi todos os containers com nosso comando clássico que já vimos anteriormente de `docker container rm $(docker ps -aq) --force`.

[00:25] Agora eu vou mais uma vez fazer `docker run –it ubuntu bash` e vou abrir um novo terminal. Se eu der um `docker ps`, que informação eu poderia ter que seria mais estável do que um IP?

[00:44] Eu poderia utilizar, por exemplo, o nome. Talvez você esteja se perguntando, e com total razão: esse nome não é gerado de maneira aleatória? Por exemplo, agora é o "angry_keller". Como teremos isso de maneira estável, como conseguiremos identificar isso?

[01:02] Nós podemos simplesmente definir os nossos próprios nomes para os containers, porque até então nós não fizemos isso. Quem fez essa criação de nome para nós foi o próprio Docker.

[01:12] Então se eu voltar em algum dos outros terminais e der mais uma vez um `docker rm` com o nosso comando de remover todos os containers, o que eu posso fazer é, no momento da execução de um container, definir um nome para ele com a *flag* `--name`.

[01:31] Por exemplo, vou dar o nome de "ubuntu1" para o container: `docker run -it --name ubuntu1`. E eu quero executá-lo com a imagem do Ubuntu e o comando `bash`, então `docker run -it --name ubuntu1 ubuntu bash`.

[01:40] Mas isso será o suficiente para conseguirmos comunicar dois containers via *host name*? Não. Nós ainda precisamos dar um passo além.

[01:47] Qual vai ser esse passo? Se olharmos o nosso `docker network ls`, nós temos as redes que já são padrão do Docker: a *bridge*, a *host* e a *none*. Mas para que consigamos fazer a comunicação entre containers via *host name* nós precisamos criar nossa própria rede.

[02:05] E como criamos nossa própria rede? Deve ser muito difícil. Na verdade é bem fácil. Basta executarmos o comando `docker network create`.

[02:15] Queremos criar uma rede que faça o papel de *bridge*, que é a rede padrão, mas será uma própria nossa para fazer a ponte entre os containers utilizando esse driver de *bridge*. Então `docker network create --driver bridge minha-bridge`. O nome dela é o último parâmetro que é passado.

[02:35] E agora nesse momento em que vamos criar o nosso container, além de definir o nome dele, vamos definir também a rede através do `--network minha-bridge`.

[02:49] E repara que se voltarmos ao outro terminal, fizemos `docker ps` e agora inspecionarmos esse container com `docker inspect`, ele vai mostrar que dentro da parte de network não está mais simplesmente aquela *bridge*. Está a "minha-bridge", que eu criei. Tem as outras informações dele, o IP está como 172.19.0.2 e com esse nome de "ubuntu1".

[03:16] Vamos agora tentar criar um outro container. Vou dar um "Ctrl + L" e vamos criar um container dentro dessa mesma rede.

[03:25] Eu vou colocar um `docker run -d` porque não vamos nos preocupar com o terminal dele. E vou colocar o nome de `--name pong`, só por uma piada que vai ser engraçada e que você já vai entender, `docker run -d --name pong --network minha-bridge ubuntu sleep 1d`.

[03:44] Coloquei o *sleep* de um dia só para manter o container em execução, não vamos nos preocupar com o terminal dele. Vou dar um "Enter". Ele criou. E agora se eu vier no terminal do meu ubuntu1, vamos fazer `apt-get update` de início. Ele vai fazer toda a atualização de repositórios e afins do sistema do meu container.

[04:13] Mas enquanto ele vai fazendo isso, se simplesmente fizermos um `docker inspect` no container que acabou de ser criado, que é o nosso pong, repara que assim como o nosso ubuntu1, ele está na rede "minha-bridge", com um IP diferente. Mas não estamos mais nos preocupado com o IP.

[04:30] Se fizermos um `docker ps` agora, temos o nosso ubuntu1 e o pong. Então conseguimos definir os nomes agora, conseguimos ter um controle sobre isso. E no momento em que eu tentar agora comunicar esse meu ubuntu1 com o pong, a ideia vai ser que se eu fizer simplesmente o comando de comunicação diretamente ele deve funcionar.

[04:56] Vamos ver como está o processo de atualização e vamos ver se isso vai funcionar 100%. Quando ele terminar de fazer a atualização dos pacotes nós faremos aquele mesmo processo de antes de instalar o ping com `apt-get install iputils-ping`.

[05:16] O processo de instalação do ping é bem rápido também. Quando ele terminar nós faremos nosso experimento para ver se tudo vai funcionar da maneira esperada.

[05:26] Ele terminou agora, vou dar um "Ctrl + L" para limpar. E vou fazer `apt-get install iputils-ping -y`. O `-y` é para ele não pedir a confirmação.

[05:35] E agora vamos fazer o nosso teste final. Vou dar um "Ctrl + L" mais uma vez e vou fazer um `ping pong`. Era essa a grande piada que eu queria fazer com vocês, por isso que eu coloquei o nome de pong no container.

[05:48] E repara que ele está fazendo a comunicação para o IP inclusive do container. Ele está mostrando que é a 172.19.0.3. Se fizermos um `docker ps` e um `docker inspect pong`, veremos que é o 172.19.0.3.

[06:07] Conseguimos agora comunicar dois containers via *host name* com a *user-defined bridge*, a rede que nós definimos através de criação.

[06:17] Mas como nós saberíamos disso? Isso é um tópico muito importante que está listado na documentação, em *use bridge networks*. Ele fala de todos os processos, eu não tirei isso do além.

[06:28] Ele mostra que nas redes *user-defined bridges*, ou seja, as redes que são criadas por usuários de *bridge*, a diferença é que elas provém essa resolução automática de DNS entre containers, que é basicamente o que nós estamos fazendo agora.

[06:42] Então conseguimos agora comunicar diferentes containers via *host name*. É um pouco mais fácil manter essa comunicação agora. E vamos entender outras questões sobre, por exemplo, o que é a rede *host* e a rede *null* para complementar nosso conhecimento, e depois vamos seguindo. Por esse vídeo é só. Eu te vejo na próxima e até mais.

As redes none e host

Agora veremos como funcionam as duas redes restantes. Vamos dar um `docker network ls`.

[00:06] Já vimos como funciona a *bridge*, e nós criamos nossa própria *bridge* também. Mas vamos olhar como funciona a rede *host*, que utiliza o driver *host*, e a rede *none*, que utiliza o driver *null*.

[00:19] Vamos começar pela rede *none*, que utiliza o driver *null*. Como é que ela funciona? Vamos exemplificar para ver como realmente ela impacta a vida do nosso container.

[00:31] Vamos executar um `docker run -d`. Coloquei o `-d` porque não vamos nos preocupar com questão de terminal em modo interativo. Em seguida vou falar que meu container será executado na minha rede chamada *none*, que já existe: `docker run -d --network none`.

[00:48] Vou executar a imagem do Ubuntu, e o comando que eu quero executar para o container se manter em execução é o *sleep*, como fizemos no início para manter o container em execução e não precisar travar o terminal para ele: `docker run -d --network none ubuntu sleep 1d`. Vou executar esse comando. Ele mostrou o ID completo do container.

[01:08] E se fizermos `docker inspect` nesse ID para vermos quais são as características desse container, repara que no final ele está falando que está utilizando agora o *none* e toda a descrição da rede desse *none*.

[01:25] Mas o que isso impactará diretamente nesse container? No fim das contas, quando utilizamos o driver *none*, estamos simplesmente falando que esse container não terá qualquer interface de rede vinculada a ele. Ele ficou completamente isolado a nível de rede.

[01:41] Nós não conseguimos fazer nenhum tipo de operação envolvendo a rede desse container, porque o driver dele é *none*, ele utiliza o driver *null* no fim das contas.

[01:53] O que precisamos fazer agora, caso queiramos fazer o contrário disso, por exemplo? Queremos que nosso container tenha interface de rede, e já vimos como fazer com a *bridge*, mas de uma maneira um pouco mais prática em alguns casos. Nós queremos fazer o contrário de não ter uma interface de rede, e sim ter uma interface de rede, mas vinculada ao nosso *host*, por exemplo.

[02:13] Vamos fazer uma leitura das nossa redes com `docker network ls`. Nós temos a nossa rede que utiliza o driver *host* e possui também o nome *host*.

[02:24] E como vai funcionar agora? Vamos fazer praticamente o mesmo teste. Se eu fizer um `docker ps`, eu vejo que não estou com mais nenhum container em execução além desse Ubuntu que eu acabei de criar.

[02:35] E agora eu vou fazer `docker run -d --network host aluradocker/app-node:1.0`.E antes do nome da imagem vou dizer que esse container será executado na rede *host*. Vamos ver o que vai acontecer. Vou copiar o ID do container e fazer `docker inspect`. Agora no final ele está informando que está utilizando a rede *host* na saída desse *inspect*.

[03:09] Só que o que isso muda na prática? Agora eu vou simplesmente abrir uma nova aba do meu navegador e tentar acessar essa aplicação.

[03:20] Mas não como eu vou acessar se eu não fiz o mapeamento de portas como já aprendemos? Se eu colocar "localhost" e definir qual porta eu quero acessar, vamos lembrar que a versão 1.0 da nossa aplicação app-node executava por padrão sempre na porta 3000.

[03:40] E depois nós parametrizamos as versões 1.1 e 1.2. Mas a 1.0 sempre era executada na porta 3000. Então se no meu navegador eu tentar acessar a aplicação na porta "localhost:3000", eu consigo.

[03:54] Mas por que eu consegui? Porque nós simplesmente agora retiramos quaisquer isolamentos que tinham entre a interface de rede do container e do *host*. Porque utilizando o driver *host* nós estamos utilizando a mesma rede, a mesma interface do *host* que está hospedando esse container, por assim dizer.

[04:14] Então caso tivesse alguma outra aplicação na minha porta 3000 com meu *host* em execução, eu não conseguiria fazer a utilização desse container dessa maneira, daria um problema de conflito de portas, porque a interface seria a mesma.

[04:29] Basicamente a ideia desse vídeo foi essa, para finalizar e ver como funcionam os outros dois drivers, o *host* e o *null* para que consigamos fechar toda essa parte de rede e não passar nada batido por nós.

[04:43] Agora veremos um pouco de prática para ver como fazer a comunicação de dois containers, mas em um ambiente um pouco mais elegante entre duas aplicações. Veremos isso no próximo vídeo. Eu te vejo lá e até mais.

Host vs None

Anteriormente, vimos sobre as duas redes Host e None, cada uma com sua respectiva finalidade.

Marque a alternativa abaixo que representa a diferença da utilização de cada uma dessas redes.

A rede host remove o isolamento entre o container e o sistema, enquanto a rede none remove a interface de rede.

Alternativa correta! Estes são os propósitos de utilizar estas redes.

Comunicando aplicação e banco

Agora vamos colocar um pouco de prática para vermos como funciona realmente a questão de comunicação de containers através da rede do Docker.

[00:10] Vamos fazer um `docker images`. Até então nós temos imagens que vamos utilizar a partir de agora nesse vídeo, que eu já baixei com o comando `docker pull`. Mas vou deixar o comando para você também baixar.

[00:20] Vou reexecutar, mas no caso ele não vai baixar porque eu já tenho, a imagem mongo na versão 4.4.6 e a "aluradocker/alura-books" na versão 1.0.

[00:34] Ênfase na versão 4.4.6 do mongo, e não na versão *latest*. Quando você for baixar a imagem do mongo você vai utilizar o comando `docker pull mongo:4.4.6` e dar o "Enter". Então ele fará o download dessa versão em específico.

[00:53] É a mesma coisa para a imagem alura-books. Então vai ser `docker pull aluradocker/alura-books:1.0`.

[01:04] Agora o primeiro passo que precisamos dar é comunicar o container que será gerado pela imagem alura-books com um banco de dados que vai ser gerado pela imagem do mongo, com um container.

[01:18] Mas como é que faremos essa comunicação? Através da rede do Docker. Então de início eu vou fazer um `docker run` no meu banco de dados: `docker run mongo:4.4.6`.

[01:31] Mas ainda faltam alguns detalhes. Por exemplo, eu não quero travar o meu terminal, então eu vou executar em modo *detached* com a flag `-d`.

[01:37] Eu vou definir que a minha rede vai ser a "minha-bridge", que foi a rede que criamos. Se fizermos um `docker network ls` veremos a rede que nós já criamos.

[01:48] Caso você tenha apagado, por favor, crie sua própria *bridge* com o comando `docker network create --driver bridge minha-bridge`. Se eu executar esse comando vai dar erro, porque a rede já existe. Mas a ideia é executar esse comando, caso você tenha apagado.

[02:06] E agora o que estamos fazendo é um `docker run -d` com o container nessa rede: `docker run -d --network minha-bridge mongo:4.4.6`.

[02:13] E o ponto agora é o seguinte: o nosso container de alura-books vai se comunicar com o banco de dados mongo. E como eles estarão numa rede *bridge* criada por nós, ou seja, criada manualmente, a comunicação poderá ser feita via *host name*.

[02:27] Só que como será o *host name* que a nossa aplicação alura-books está buscando? Qual é o nome de banco que ela está buscando para se conectar?

[02:36] Para isso eu também vou disponibilizar para vocês o código-fonte da aplicação que foi usada como base para gerar a imagem que será usada para gerar esse container.

[02:45] Mas o que importa é que no arquivo de configuração dessa aplicação, no *host* ele está procurando por um *host* chamado "meu-mongo". Então no momento em que essa imagem foi construída, esse arquivo estava definido dessa maneira.

[03:02] Isso significa que eu preciso que o *host name*, ou seja, o nome desse container, seja "meu-mongo". Então o comando vai ficar `docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6`. E a partir de agora, se eu der um "Enter" nesse comando, ele vai criar o container.

[03:17] E agora eu preciso fazer o *run* do nosso alura-books. Então `docker run aluradocker/alura-books:1.0`. E agora vamos adicionar os detalhes com os quais precisamos nos preocupar, como o `-d`; a rede, que precisa ser a mesma rede, para que os containers consigam se comunicar: `docker run -d --network minha-bridge aluradocker/alura-books:1.0`.

[03:43] O nome nesse caso é irrelevante, porque a aplicação está procurando pelo banco, e não o contrário. Não estou falando que sempre vai ser essa regra, mas nesse caso que estamos fazendo nós não precisamos nos preocupar com o nome desse container especificamente, então vou deixar como "alura-books": `docker run -d --network minha-bridge --name alurabooks aluradocker/alura-books:1.0´.

[04:04] E outro detalhe também é que precisamos fazer o mapeamento de portas, que vai ser a porta 3000 na porta 3000 também do nosso container: `docker run -d --network minha-bridge --name alurabooks –p 3000:3000 aluradocker/alura-books:1.0´.

[04:16] Eu preciso fazer o mapeamento de portas, porque eu não estou e nem posso nesse caso utilizar a rede *host*. Eu estou utilizando a rede "minha-bridge". Vou dar um "Enter" e ele vai executar.

[04:27] Vou abrir uma nova aba no meu navegador, e agora vamos fazer "localhost:3000", que vai acessar a nossa aplicação. E ela tem um *endpoint*, o "/seed", que vai popular o banco. E agora se atualizarmos a página, todos os dados do banco estão sendo carregados na nossa aplicação.

[04:55] Então a partir desse momento, se eu voltar ao terminal e simplesmente parar o container do meu mongo com o comando `docker stop meu-mongo`, os dados no navegador vão sumir, porque a comunicação com o banco parou.

[05:14] Se eu voltar ao terminal e executar `docker start meu-mongo`, fizer um *seed* novamente e recarregar a página, tudo volta ao normal.

[05:28] Então nós estabelecemos a comunicação entre dois containers que estão na mesma rede, e conseguimos ter um resultado real, bem próximo do que vemos no dia a dia de utilização de aplicações.

[05:47] Nós tivemos uma aplicação *back-end*, que também tem um *front*, se comunicando com o banco e trazendo os dados para o usuário no fim das contas.

[05:55] Só que precisamos levantar alguns questionamentos agora. Será que a melhor maneira é fazermos a inicialização de containers sempre manualmente? Porque eu tive que me preocupar em fazer o `docker run` de um, depois o `docker run` do outro.

[06:09] Será que quando vamos fazer as coisas realmente em produção nós sempre subimos tudo na mão? É um ponto sobre o qual precisamos pensar. Só que veremos isso nas próximas aulas. Eu te vejo lá e até mais.

Faça como eu fiz: Comunicação de containers

Agora iremos executar dois containers criados em uma mesma rede customizada, a fim de comunicar e trafegar informações entre uma aplicação e um banco de dados. Anteriormente, vimos que o papel da rede **bridge** é possibilitar a comunicação entre containers em um mesmo host. Chegou o momento de pôr isso em prática.

Inicialmente, abra o terminal e execute o comando `docker network ls`. Caso ainda não tenha criado a rede `minha-bridge`, execute o comando `docker network create --driver bridge minha-bridge`.

Em seguida, iremos executar o container responsável pelo banco de dados. Para isso, execute o comando `docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6`. Repare que estamos usando a versão **4.4.6**.

Precisamos agora executar o container responsável pela aplicação que irá se comunicar com o banco de dados. Para isso, execute o comando `docker run -d --network minha-bridge --name alurabooks -p 3000:3000 aluradocker/alura-books:1.0`. Repare que utilizamos a flag `-p` para em seguida validar o funcionamento da aplicação através de nosso host.

Em seu navegador, acesse a url `localhost:3000` e veja que foi possível carregar a página da aplicação. Para que os dados sejam carregados e armazenados no banco, acesse `localhost:3000/seed` e, em seguida, recarregue a página `localhost:3000`. Veja que as informações agora estão sendo exibidas por conta da comunicação entre aplicação e banco de dados.

Conhecendo o Docker Compose

Agora vamos voltar a um problema que já tivemos anteriormente, mas vamos ver o que aconteceu.

[00:06] O que nós fizemos ainda há pouco? Nós executamos aquele nosso container do aluradocker da imagem alura-books na versão 1.0 em conjunto com o mongo na versão 4.4.6.

[00:17] Para fazer isso nós precisamos de início executar o banco. O comando que executamos foi o `docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6`. E para executar nosso container do alura-books foi através do comando: `docker run -d --network minha-bridge --name alurabooks -p 3000:3000 aluradocker/alura-books:1.0`.

[00:43] E nós precisamos fazer tudo isso manualmente. Nós precisamos ter definido qual era o comando que iríamos executar e definido a ordem que queríamos.

[00:54] Então vamos parar para pensar: qual foi a motivação inicial que tivemos no curso? Uma das questões é que podemos ter diversas aplicações se comunicando entre si para construir um sistema ainda mais complexo.

[01:05] Mas o que está acontecendo agora é que se nós crescermos muito a nossa aplicação, o que vai acabar acontecendo, em algum momento teremos que subir diversos containers manualmente.

[01:16] Sempre que quisermos parar um container teremos que fazer `docker stop` ou um `docker rm` para remover um a um. Então para cada container vai ser um comando que precisaremos executar, além de nos preocuparmos com toda a questão da pilha de execução que teremos com todos esses containers.

[01:32] Existe uma solução já desenvolvida pela empresa do próprio Docker que vai nos ajudar a resolver esse tipo de situação, que no caso é o Docker Compose.

[01:46] O Docker Compose nada mais é do que uma ferramenta de coordenação de containers. Não confunda com orquestração, são coisas diferentes.

[01:53] Então o Docker Compose vai nos auxiliar a executar, a compor, como o nome diz, diversos containers em um mesmo ambiente, através de um único arquivo. Então vamos conseguir compor uma aplicação maior através dos nossos containers com o Docker Compose.

[02:08] E faremos isso através da definição de um arquivo yml, aquela extensão yml, ou yaml, caso você já tenha ouvido falar. E nada mais é do que um tipo de estrutura que vamos seguir baseado em indentação do nosso arquivo.

[02:32] Nós vamos seguir mais ou menos essa receita de definir uma versão, quais serão os nossos serviços, e também como fazer toda a parte de rede e comunicação dos nossos containers, só que agora através de um único arquivo. E conseguiremos coordenar isso diretamente dos comandos de como o Docker Compose faz isso.

[02:50] Um pequeno adendo é que caso você esteja no Windows nesse momento, quando você instala o Docker você já tem nesse momento o Docker Compose.

[03:00] Então caso você execute `docker-compose` no seu terminal, a princípio ele vai dar o "erro", mostrando todos os comandos que você pode executar. Ele já fala que você poderá definir e executar aplicações de múltiplos containers com o Docker Compose.

[03:22] Mas caso você esteja no Linux, como temos feito o curso desde o início, você precisará instalar o Docker Composer. Se viermos no terminal, não vamos conseguir executar nada com o Docker Compose. Ele não reconhece o comando `docker-compose`.

[03:45] E ele vai sugerir também a instalação através do snap ou do apt, mas nós não faremos isso. Nós vamos seguir a documentação nesse caso.

[03:51] Então eu vou abrir uma nova aba e vamos fazer o processo de instalação. Se eu digitar na barra de pesquisa "docker compose install Linux", ele vai ter a documentação oficial que eu vou deixar para vocês também.

[04:03] E caso você esteja no Windows e a instalação não tenha acontecido por algum motivo, é só marcar na página da documentação que você está utilizando o Windows. No nosso caso, estamos utilizando o Linux.

[04:15] Para instalar é bem simples, basta copiar o comando que ele fornece e voltar ao terminal. Vou fazer "Ctrl + L" para limpar e "Shift + Insert" para colar. Vou colocar minha senha, você vai colocar a sua, claramente, e vou dar um "Enter".

[04:30] Ele vai fazer todo o processo de baixar. E logo depois precisamos tornar o nosso Docker Compose executável, dar a permissão para ele com o comando do `chmod`.

[04:43] Nesse momento vou abrir um terminal novo. Se executarmos um `docker-compose` para executar, agora temos o mesmo *output* que tínhamos no Windows. Ele vai definir múltiplos containers para executar uma aplicação com o Docker.

[05:04] Agora que entendemos o que é o Docker Compose, que será essa ferramenta de composição e de coordenação de containers, precisamos transformar o que fizemos anteriormente num ambiente agora distribuído dessa maneira com o Docker Compose. Por esse vídeo nós terminamos. Eu te vejo no próximo e até mais.

Utilização do Docker Compose

Durante todo o curso, compreendemos a execução de containers através do comando `docker run`. Além disso, entendemos também como comunicar containers em conjunto com a rede `bridge` padrão e também criando nossa própria rede através dos comandos `docker network`.

Marque a alternativa que ilustra qual situação ainda precisamos resolver e como o Docker Compose auxilia neste problema.

O Docker Compose irá resolver o problema de executar múltiplos containers de uma só vez e de maneira coordenada, evitando executar cada comando de execução individualmente.

Alternativa correta! Este é o principal objetivo do Docker Compose.

Definindo os serviços

Agora precisamos traduzir esses dois comandos que já executamos anteriormente para subir o nosso alura-books e também o comando para subir o nosso mongo. Nós vamos traduzir, por assim dizer, para essa estrutura yml.

[00:17] Seguiremos uma estrutura bem parecida com a que está na própria documentação. Vamos definir uma versão para o yml que vamos utilizar; vamos definir quais são os serviços. A diferença é que não seguiremos exatamente igual à documentação, mas grande parte será bem parecida.

[00:32] O que eu vou fazer nesse momento é abrir um novo terminal e criar uma nova pasta na área de trabalho, chamada "ymls": `mkdir Desktop/ymls`. Eu vou acessá-la com `cd Desktop/ymls/` e vou abrir o VS Code dentro dela. Você pode usar o seu editor de texto favorito.

[00:58] No VS Code, dentro dessa pasta eu vou criar um “docker-compose.yml”. Repara que o VS Code é inteligente o suficiente para saber que esse arquivo se trata de algo relacionado ao Docker, ele colocou até o símbolo da baleia ao lado do nome.

[01:16] Vou criar o arquivo e a ideia é que dentro desse arquivo nós configuremos o nosso yml. Como vamos definir todas as configurações? Eu vou simplesmente, como eu falei de início, definir a versão do yml que vamos utilizar, que será a 3.9, então `version: "3.9"`.

[01:42] E agora nós vamos definir os nossos serviços, que no fim das contas serão nossos containers. Então quais serão os nossos serviços? Nós queremos repetir o que fizemos anteriormente, de subir o alura-books e o banco de dados.

[01:56] Então eu posso simplesmente colocar `services:`. E nesse momento vamos definir que temos um serviço chamado `mongodb`, por exemplo, mas eu poderia dar o nome que eu quisesse.

[02:09] E esse serviço vai ter as peculiaridades dele. Precisamos definir a partir de agora qual é a imagem que vamos utilizar para esse serviço, se queremos dar um nome para o container, se vamos colocá-lo em alguma rede.

[02:22] Nós queremos fazer isso. Eu quero que a imagem que ele vai utilizar para esse serviço seja a do mongo na versão 4.4.6, que é a que fizemos na nossa execução. Então `image: mongo:4.4.6`.

[02:38] Nós também queremos dar o nome do container de “meu-mongo”. Então vamos colocar `container_name: meu-mongo`. E queremos que ele esteja numa rede. Não precisamos manter a minha-bridge. Eu vou colocar uma rede chamada “compose-bridge”.

[03:03] Para a definição da rede eu faço `networks:`, dou um “Enter”, um “Tab” e um traço, para colocarmos o elemento dessa lista. E vou colocar `- compose-bridge`.

[03:16] Então agora eu defini minha rede, o nome do meu container e a imagem que eu quero utilizar, que foi a mesma coisa que definimos anteriormente quando fizemos manualmente no terminal.

[03:27] Agora eu preciso fazer a mesma coisa, só que para o meu alura-books. Então vou quebrar mais uma linha, para ficar um pouco mais visível. E eu vou definir o nosso serviço chamado alura-books. E dentro dele eu vou utilizar a imagem: `image: aluradocker/alura-books:1.0`. Depois o nome do container, que eu vou colocar como “alurabooks”: `container_name: alurabooks`. Depois a rede, que precisa ser a mesma, então vou colocar `compose-bridge`.

[04:04] E agora também eu vou precisar colocar o mapeamento de portas que fizemos anteriormente. Para definir as portas eu coloco `ports:`, e faço de maneira bem parecida com a rede: dou um “Enter”, um “Tab” e um traço e coloco a porta em que minha aplicação está executando, que é a 3000. E eu quero que ela rode na minha máquina também na porta 3000, então `- 3000:3000`.

[04:35] Basicamente o que já definimos agora foi que o alurabooks está rodando dentro do container na porta 3000 e vai rodar no nosso *host* na porta 3000; definimos a imagem; a rede; e o nome.

[04:52] Agora falta um pequeno detalhe, que é configurar essa rede, porque ela não existe até então. Então o que precisamos fazer no final é, alinhado com a parte de *services*, colocar `networks:`, dar um “Enter” e definir a minha rede, que é a `compose-bridge:`.

[05:11] Vou dar mais um “Enter” e vou informar o driver dela, que vai ser o *bridge*: `driver: bridge`.

[Aula6_video2_imagem1]

[05:18] E agora eu simplesmente vou abrir um terminal nesse diretório que eu já tenho, e vou executar o comando `docker-compose up`.

[05:28] Antes de executar esse comando vou fazer um `ls` para garantir que eu estou nesse diretório que o meu arquivo "docker-compose" foi criado com esse nome.

[05:37] Então eu vou colocar `docker compose up`. E ele vai criar. Não coloquei em modo *detached* para entendermos o que ele está fazendo. Ele está criando o banco.

[05:53] Ele já subiu todo o banco. E também, se formos subindo, conseguimos ver que ele mostra um *output* bem misturado, tanto do nosso "meu-mongo" quanto do nosso "alurabooks". Tem toda essa questão deles sendo executados.

[06:14] Agora temos garantia de que está tudo em execução. Se voltarmos ao nosso navegador e executarmos "localhost:3000", conseguimos acessar nossa aplicação. Populei o banco com um "/seed", e se eu atualizar a página, está tudo aparecendo.

[06:31] Mas apesar de tudo ter funcionado, ainda tem mais detalhes que podemos ver sobre como isso tudo funcionou.

[06:37] Só que antes de terminar essa aula, dentro do terminal que nós executamos e está travado, eu vou dar um "Ctrl + C". Repara que ele vai parar os dois containers, tanto o "alurabooks" quanto o "meu-mongo". E se voltarmos ao navegador e atualizarmos a página com "F5", veremos que ele já derrubou.

[06:54] Mas antes de terminarmos o assunto do sobre Docker Compose, ainda teremos mais um vídeo dando alguns detalhes sobre o que aconteceu e o que podemos fazer também utilizando essa ferramenta. Eu te vejo lá e até mais.

Parâmetros para serviços

Anteriormente, vimos como o Docker Compose pode ser útil para coordenar containers e como podemos usá-lo para definir diversos parâmetros acerca da execução de containers. Em conjunto com o comando `docker-compose up`, foi possível executar dois containers de uma só vez.

Quais alternativas abaixo contém parâmetros que podemos definir com o Docker Compose?

Nome do container.

Alternativa correta! O nome do container é um parâmetro que pode ser definido com o Docker Compose.

Rede que o container irá se conectar.

Alternativa correta! A rede é um parâmetro que pode ser definido com o Docker Compose.

Imagem a ser utilizada.

Alternativa correta! A imagem é um parâmetro que pode ser definido com o Docker Compose.

Faça como eu fiz: Coordenando containers

Agora chegou o momento de, em conjunto com o Docker Compose, iniciarmos dois containers de maneira coordenada na mesma rede, tudo a partir de um único comando e arquivo de definição. Com isso, o processo de deploy será bastante reduzido e ficará ainda mais prático.

Inicialmente, crie uma pasta chamada `ymls` em sua área de trabalho. Dentro dessa pasta, você deverá criar um arquivo chamado `docker-compose.yml`, que será responsável por conter todas as definições de como os nossos containers serão executados.

Com seu editor de texto favorito, edite o arquivo `docker-compose.yml` e defina a estrutura base para começarmos:

```
version: "3.9"
services:
```

Agora, dentro de `services`, iremos definir primeiramente o comportamento de nosso container responsável pelo banco de dados:

```
version: "3.9"
services:
  mongodb:
    image: mongo:4.4.6
    container_name: meu-mongo
    networks:
      - compose-bridge
```

Para definirmos o container responsável pela nossa aplicação que consumirá do banco, seguiremos a mesma ideia, porém, adicionaremos as instruções responsáveis pelo mapeamento de portas:

```
version: "3.9"
services:
  mongodb:
    image: mongo:4.4.6
    container_name: meu-mongo
    networks:
      - compose-bridge

  alurabooks:
    image: aluradocker/alura-books:1.0
    container_name: alurabooks
    networks:
      - compose-bridge
    ports:
      - 3000:3000
```

Por fim, também será necessário definir a rede dos containers. No caso, estamos utilizando a rede `compose-bridge`. Alinhado ao conteúdo da linha de `services`, adicione o seguinte conteúdo:

```
version: "3.9"
services:
  mongodb:
    image: mongo:4.4.6
    container_name: meu-mongo
    networks:
      - compose-bridge

  alurabooks:
    image: aluradocker/alura-books:1.0
    container_name: alurabooks
    networks:
      - compose-bridge
    ports:
      - 3000:3000

networks:
  compose-bridge:
    driver: bridge
```

Através do terminal, acesse o diretório que contém o seu arquivo `docker-compose.yml` e execute o comando `docker-compose up -d`. No navegador, acesse a url `localhost:3000/seed` e em seguida `localhost:3000` e veja que tudo continua funcionando como anteriormente.

Complementando o Compose

O que mais seria interessante saber sobre o Docker Compose? Uma coisa interessante é que no momento em que nós damos o `docker-compose up`, vimos que estamos subindo tudo de maneira meio indefinida.

[00:15] Mas existe uma instrução que podemos colocar, em conjunto com a documentação que eu vou deixar o link também para vocês. Nós podemos definir algumas instruções também.

[00:25] Além de todas que já colocamos, existe uma que se chama "depends_on". Ela expressa dependência entre serviços. No momento em que colocamos uma dependência de um serviço para outro, ele vai iniciar o serviço nessa ordem específica.

[00:46] No momento em que fizermos a execução, ele vai esperar o serviço subir. Mas tem um pequeno detalhe: o `depends_on` não vai esperar necessariamente a aplicação dentro do container estar pronta para receber as requisições. O que ele vai fazer é esperar o container ficar pronto, o que não significa que a aplicação dentro do container já está pronta.

[01:12] No momento em que viermos no nosso VS Code falarmos que agora nossa aplicação do alurabooks depende do nosso banco de dados, nós conseguimos fazer essa definição. Então faço `depends_on:`, dou um "Enter", um "Tab" e um `- mongodb`.

[01:31] Nós podemos agora colocar um `docker-compose up` no diretório mais uma vez. Antes vou salvar o arquivo no VS Code, vou dar um "Ctrl + L" no terminal e só então fazer `docker-compose up`.

[01:45] Ele vai fazer toda a definição. Meu mongo já está ok. E agora repara que apesar de no final ele ainda ter feito uma parte dividida entre as duas aplicações, teve uma redução em relação ao que tínhamos antes, de vários logs misturados. Nesse caso ele teve um isolamento um pouco maior.

[02:08] Como esperamos o container do mongo ficar pronto, só tivemos no final uma sobreposição de informações. Mas agora nós expressamos essa dependência entre as nossas aplicações.

[02:21] E tem um outro detalhe. Vamos dar um "Ctrl + C" mais uma vez. Nós podemos executar em modo *detached*, como tínhamos mencionado, então `docker-compose -d up`. No caso eu coloquei errado. Você tem que definir não só com o `-d` efetivamente, mas com `up -d`. Então vamos fazer `docker-compose up -d`. Ele vai inicializar e pronto, ele não travar o nosso terminal.

[02:49] Podemos fazer um `docker-compose ps` e ele vai mostrar os serviços que foram criados pelo Docker Compose de maneira mais organizada. Podemos fazer também um `docker-compose down` para ele remover. Ele vai fazer toda a etapa de remoção dos contêineres e da rede que foi criada.

[03:08] E só para finalizar essa definição toda de Docker Compose, o que é interessante sabermos e entendermos da documentação, para caso depois você for ler e não ficar perdido?

[03:22] Existem algumas coisas que seriam interessantes se conseguíssemos fazer, como a sessão de *deploy*, que permite fazer toda uma configuração de, por exemplo, número de réplicas: quantos containers de determinado serviço você quer; a parte de *placement*, como você vai ajustar isso a nível de paralelismo e tudo mais.

[03:43] Só que essa definição toda não funciona da maneira que estamos utilizando o Docker. Ele fala que essas configurações só tomam efeito no momento em que estamos fazendo isso com um *swarm*, que vai funcionar só com docker *stack deploy*.

[03:56] Isso significa que da maneira que estamos atualmente essas configurações, caso você tente testar na sua máquina e não funcionem, é porque você não está utilizando o Docker em modo *swarm*.

[04:06] Nós temos já um curso na Alura falando sobre isso, nós abordamos essas instruções com o Compose, e falamos sobre diversos detalhes também. Tem um curso da Alura só falando sobre Docker *swarm*.

[04:17] Sobre a parte da definição toda do Docker Compose e como ele funciona, é basicamente isso. Conseguimos subir e descer os serviços, fazer todas as nossas definições.

[04:27] Inclusive, conseguimos também fazer a utilização de volumes. Analogamente como fizemos com as redes, nós conseguiríamos definir, por exemplo, um volume para um container e fazer essa utilização também.

[04:40] Então caso você tenha alguma necessidade específica, a documentação é um lugar muito bom de se consultar. Eu falei para vocês de volumes, e podemos também fazer a utilização de volumes, com volumes - db para o que estamos definindo.

[04:55] Nós até poderíamos, por exemplo, subir uma imagem do Ubuntu e fazer sem nenhum problema como já fizemos anteriormente a definição de um serviço com um volume, com uma declaração bem parecida com a maneira que fizemos a nossa rede também.

[05:13] Essa parte toda de Docker Compose terminamos por aqui. Eu te vejo na próxima e até mais.

Conclusão

Parabéns por ter chegado até aqui. Eu espero que você tenha aproveitado o conteúdo desse curso onde vimos diversas questões sobre o Docker.

[00:06] Iniciamos com a questão de diferença entre máquinas virtuais e containers, abordamos essas duas questões.

[00:15] E a partir disso começamos a entender como os containers realmente funcionam. Vimos que containers funcionam como processos dentro do nosso sistema, e também que eles fazem toda a parte de isolamento através dos *namespaces*. Vimos todas essas questões bem importantes para o Docker e o funcionamento dele.

[00:32] A partir disso começamos a colocar a mão na massa. Voltamos ao terminal e entendemos comandos básicos, como comandos para listar os containers e comandos análogos como, por exemplo, o `docker ps`, e o `docker container ls`.

[00:45] Entendemos o comando `docker run` e vimos aos poucos o que ele fazia. Primeiro vimos que ele executava o container, mas depois vimos que ele fazia mais.

[00:54] Por exemplo, se colocarmos uma imagem que não temos alguma camada de que necessitamos, ele vai no Docker Hub e faz o download dessa camada para que a utilizemos dentro do nosso sistema.

[01:04] Vimos como o Docker faz a reutilização de camadas. Começamos a entender todos esses conceitos e também tivemos um exemplo mais visual a partir do momento que vimos o comando `docker history`, com o qual conseguimos ver como as camadas são distribuídas e o que são as camadas no momento da criação das imagens.

[01:24] Depois disso começamos a entrar em questões de aspecto de volume, e começamos a entender a questão da persistência de dados, como poderíamos criar nossas próprias imagens.

[01:33] E a partir disso também vimos a parte de redes, com o `docker network`. Compreendemos a utilização das redes, o que são as redes, os drivers e como comunicamos diferentes containers que estão numa mesma rede.

[01:44] E por fim vimos toda a parte do Docker Compose. Com `docker-compose` aprendemos a fazer a coordenação de múltiplos containers, subir múltiplos contêineres de uma vez só, fazendo a comunicação entre eles e depois também derrubar tudo de uma vez, através de um único comando, sem depender de diversos comandos manuais e subir um a um para fazer tudo o que queríamos fazer.

[02:09] Então durante todo esse curso fomos desenvolvendo passo a passo toda a etapa de criação, entendimento de um container, comunicação, persistência de dados e também coordenação.

[02:20] Eu espero que você tenha aproveitado esse curso. Eu te vejo numa próxima e até mais.