DevOps

UNIDADE 04

Docker 

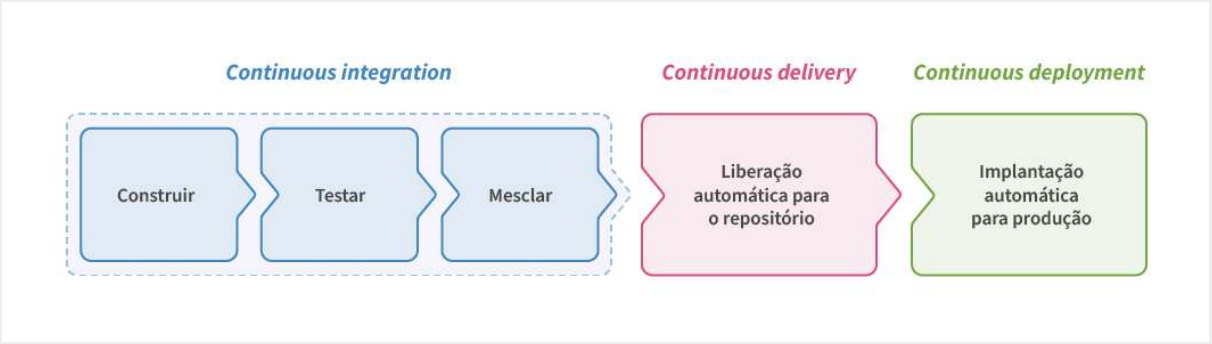
PIPELINES DE CI/CD

Lembrando o que comentamos anteriormente: CI/CD é um conjunto de práticas e ferramentas que permite aos desenvolvedores entregarem código rápido e de um jeito seguro e eficiente. Ele é composto de duas partes:

1. **CI** (***continuous integration***): aqui, a ideia é integrar pequenas partes do código em um repositório compartilhado várias vezes ao dia, garantindo que cada mudança seja testada e validada automaticamente.
2. **CD** (***continuous delivery***): aqui, montamos uma infraestrutura de um jeito que o código é sempre mantido em um estado que pode ser implantado em produção a qualquer momento, mas a implantação real ainda é feita **manualmente**.

a. Também existe outro termo CD: o ***continuous deployment***. O código é automaticamente implantado em produção assim que passa por todas as etapas de teste e verificação, **sem necessidade de intervenção manual**. É como se fosse um passo além do *continuous delivery*.

#ParaTodosVerem



Fonte: Red Hat, 2020 (Adaptado).

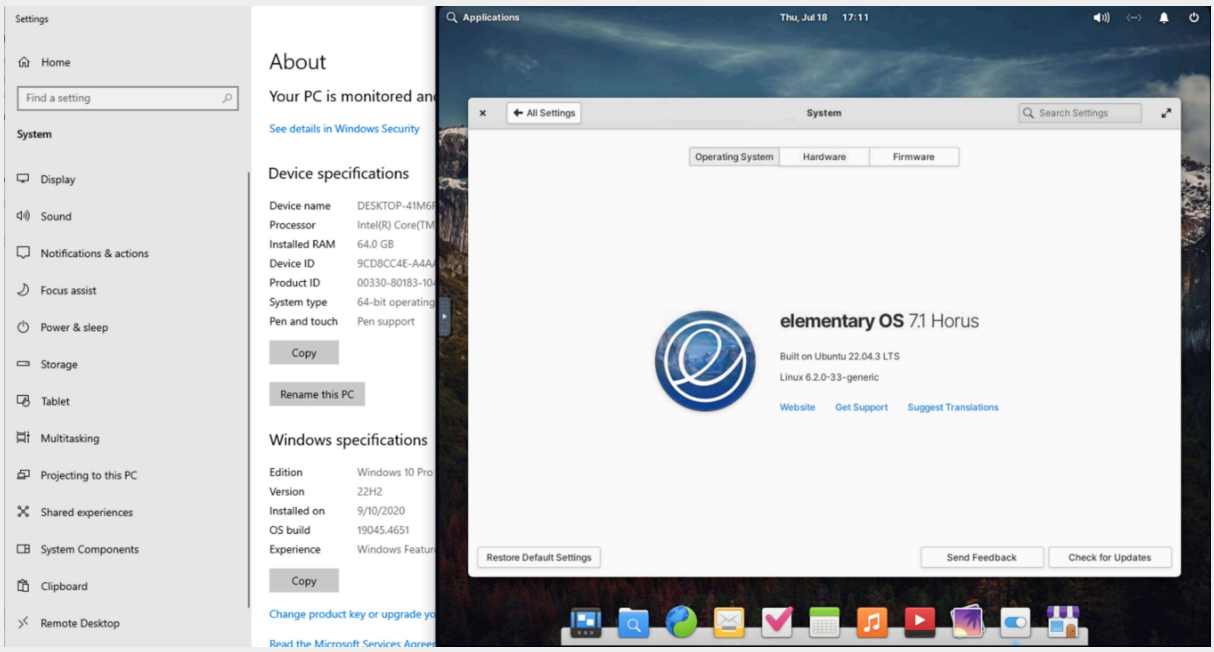
DICA

O CD no termo CI/CD significa *continuous delivery*, e não *continuous deployment*. Contudo, muita gente se confunde com os dois termos, incluindo a pessoa que pode te entrevistar para um novo emprego. Portanto, se alguém se perguntar sobre CI/CD em uma entrevista, é legal você falar da diferença entre os dois tipos de CD.

MÁQUINAS VIRTUAIS E CONTAINERS

Vamos começar do básico. Primeiro, vamos entender o que são as máquinas virtuais (VMs). Imagine que você tem um computador, mas gostaria de rodar vários sistemas operacionais diferentes nele ao mesmo tempo, como Windows, Linux e macOS. Uma VM é como um "computador dentro de um computador" que permite isso. Utilizando um *software* chamado hipervisor, você pode criar múltiplas VMs, cada uma com seu próprio sistema operacional, todas rodando simultaneamente no mesmo *hardware* físico. Isso é ótimo para testes, desenvolvimento e para rodar aplicações que necessitam de diferentes ambientes. Contudo, elas não são perfeitas: as VMs podem ser pesadas e consumir muitos recursos, pois cada uma precisa do seu próprio sistema operacional completo.

#ParaTodosVerem



Fonte: O autor (2024).

Essa imagem ilustra como é possível rodar múltiplos sistemas operacionais simultaneamente usando tecnologia de virtualização; nesse caso, rodando o elementary OS dentro de um ambiente Windows.

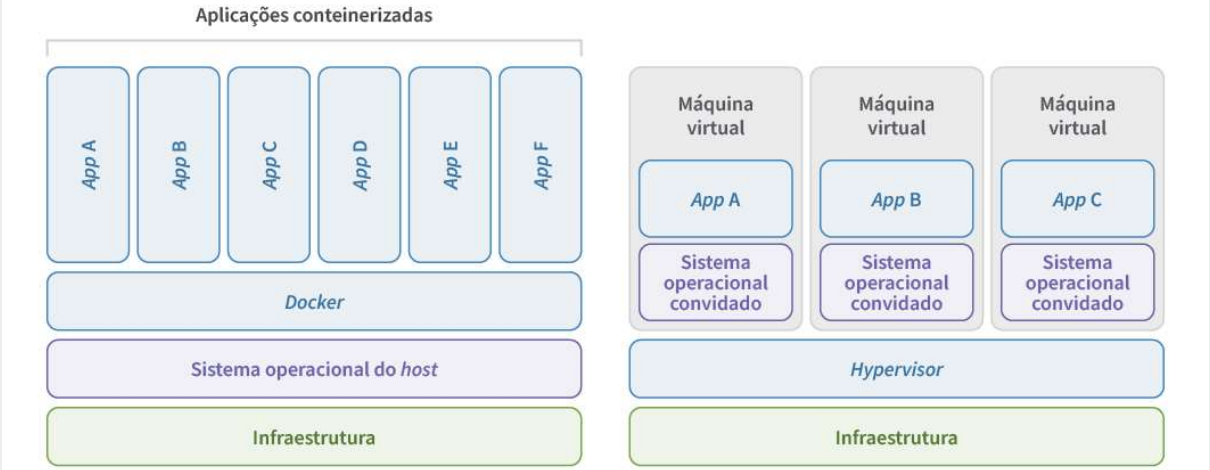
EXEMPLO

Um exemplo de execução de uma VM: aqui, tenho um computador que está rodando o Windows 10, enquanto em uma janela eu tenho o Linux (nesse caso, o elementary OS). No exemplo, o Windows é o *host* e o Linux é a VM. Podemos ter casos em que o contrário acontece ou, ainda, combinações diferentes como Windows + Mac, Windows + Windows, Linux + Mac, entre outros.

Agora, pense comigo: um sistema operacional completo tem muito mais coisas do que precisamos para rodar uma aplicação só, não é mesmo? Vamos supor que você queira colocar um *site* em funcionamento e escolheu o Linux como o servidor para suportá-lo. É provável que você não precise de **todas** as aplicações que vem por padrão quando você instala o Linux, não é? Você não precisaria de um LibreOffice, ou de um VLC Media Player. Não precisaria de um GIMP ou de um Avidemux. Nem ao menos precisaria de um papel de parede ou de um ambiente bonito: só a linha de comando resolveria.

É aí que entra o Docker, que é algo diferente de uma VM. O Docker também permite a você criar ambientes isolados, mas de uma forma muito mais eficiente. Em vez de virtualizar todo um sistema operacional, o Docker usa ***containers*** para virtualizar apenas o sistema operacional necessário para rodar a aplicação. Acho que a imagem abaixo ajuda a explicar melhor a diferença entre os dois.

#ParaTodosVerem



Fonte: Wikimedia Commons

Na imagem acima, a arquitetura do Docker está à esquerda, enquanto que a arquitetura das VMs está à direita. Perceba que todas as aplicações do Docker compartilham de um mesmo sistema operacional base, enquanto cada VM possui o seu próprio sistema operacional. Isso, na prática, faz com que as VMs sejam bem ineficientes.

Pense nos *containers* como se fossem umas "mini-VMs" que compartilham o mesmo *kernel* de um sistema operacional, mas com as suas próprias bibliotecas e dependências. Isso significa que os *containers* se iniciam rapidamente e consomem menos recursos.

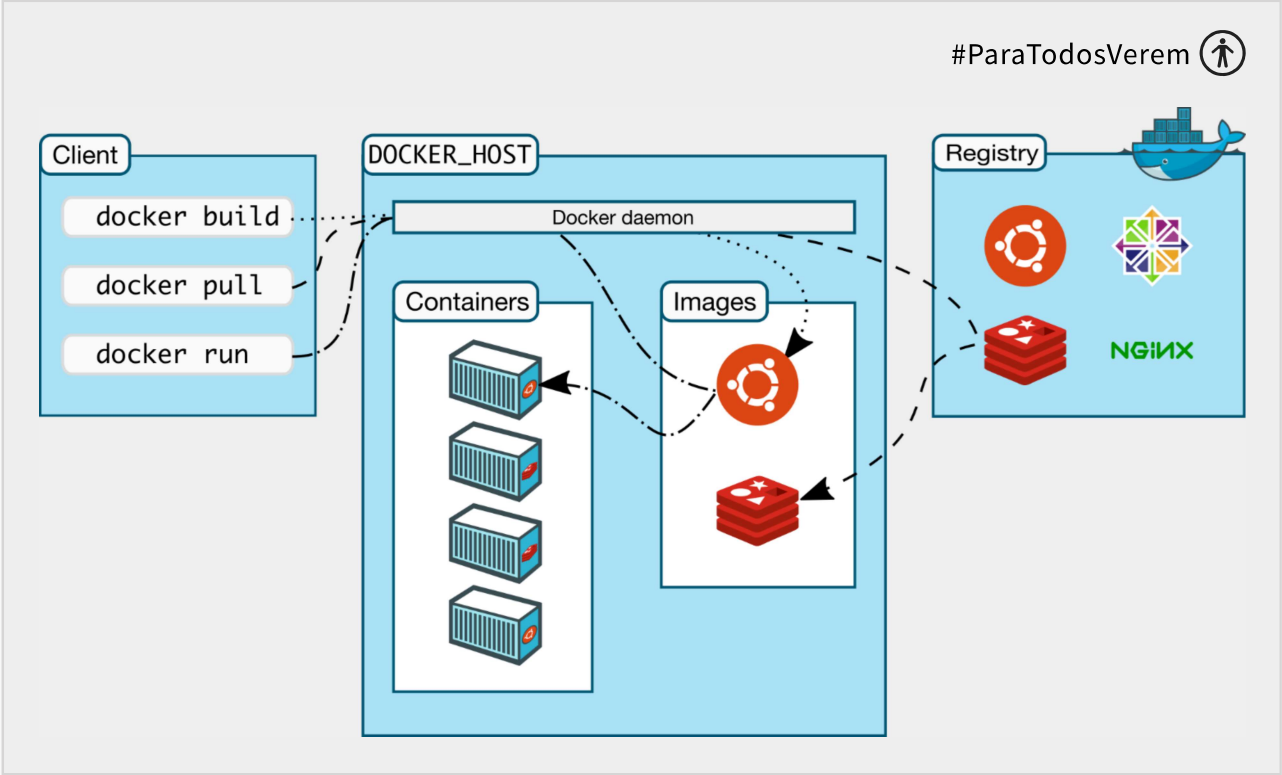
IMPORTANTE

Por isso o Docker é excelente para criar ambientes de desenvolvimento **consistentes**. Isto é: garante que a sua aplicação funciona da mesma forma em qualquer lugar, seja no seu computador, no servidor de testes ou na produção.

Resumindo: enquanto as VMs são ótimas para simular múltiplos sistemas operacionais completos, o Docker é perfeito para rodar aplicações de um jeito mais leve e mais rápido. As VMs oferecem um isolamento completo ao virtualizar um sistema operacional inteiro, enquanto o Docker oferece uma solução mais leve ao virtualizar apenas a aplicação e suas dependências.

COMPONENTES DO DOCKER

O que acha de entendermos melhor os principais componentes do Docker?



Fonte: nhumrich!/ Open Clipart

USANDO DOCKER

Agora, o que acha de usarmos o Docker juntos? Temos aqui duas videoaulas: na primeira, instalaremos o Docker em nosso computador. Depois, criaremos o nosso próprio *container*.

IMPORTANTE

O link gratuito de download do Docker Desktop é o https://www.docker.com/products/docker-desktop/.

Instalando e configurando o Docker

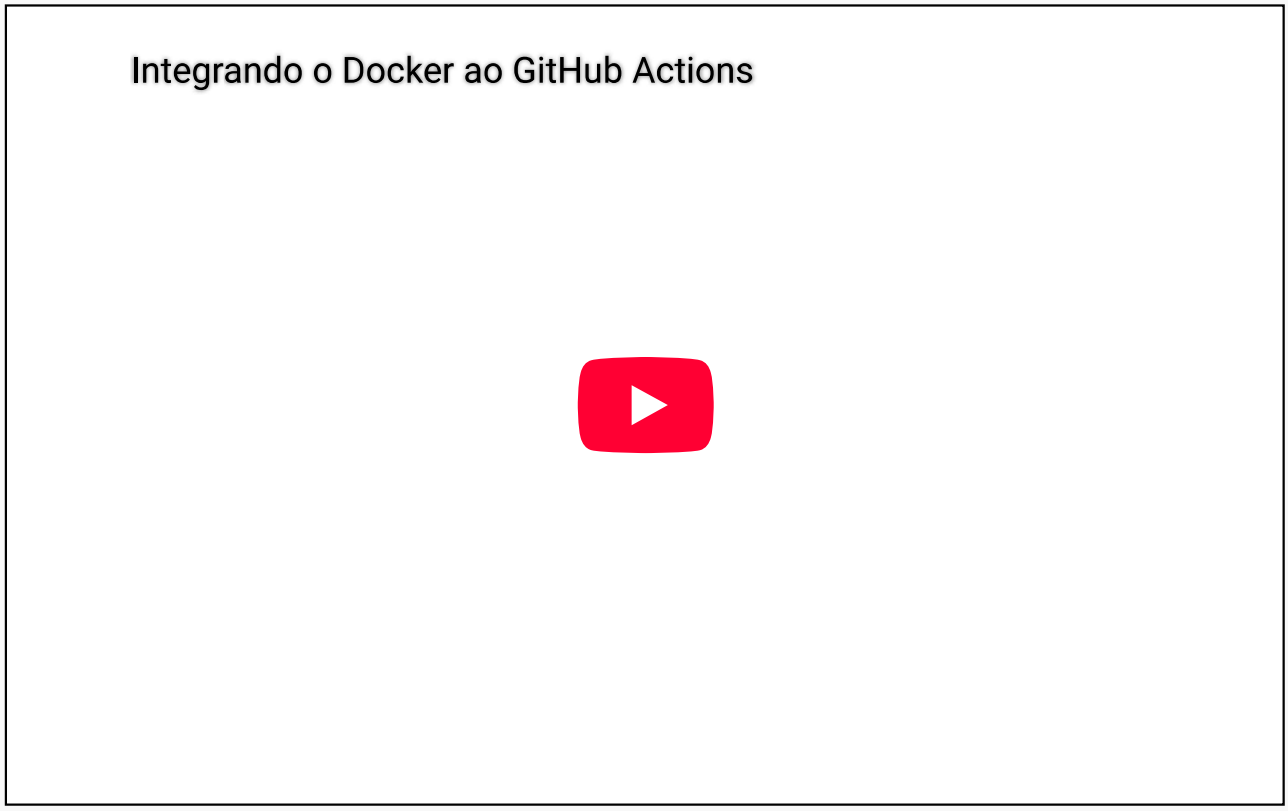
Criando o nosso próprio container

CASOS DE USO DO DOCKER

Acho que é importante que você saiba alguns casos de uso comuns do Docker no mercado. Por exemplo, a migração de sistemas entre uma infraestrutura para outra ou, ainda, para facilitar a escalabilidade de serviços ao criar mais *containers* de uma mesma aplicação quando necessário. Contudo, é importante você saber de outros três casos para o nosso contexto de DevOps: CI/CD, arquitetura de microsserviços, e ambientes de produção e *staging*. Vamos lá?

Exemplo 1: CI/CD

Lembra que eu comentei que o Docker é bem útil em CI/CD por fornecer um ambiente isolado e consistente para construir, testar e fazer o *deploy* das suas aplicações, certo? Pois bem, ao usar Docker com GitHub Actions, você pode configurar seus *workflows* para construir e testar seu código dentro de *containers* Docker. Como um exemplo, você pode usar um Dockerfile para definir o ambiente de *build* e testar sua aplicação dentro desse *container*. Em seguida, com um arquivo de *workflow* YAML no GitHub Actions, você pode definir *jobs* que constroem essa imagem Docker, executam testes e, se tudo estiver certo, fazem o *push* dessa imagem para um repositório de *container* como Docker Hub ou GitHub Container Registry. Isso assegura que cada mudança no código passe por um rigoroso processo de *build* e teste antes do *deploy*, o que melhora a confiabilidade do seu trabalho. O que acha de testarmos isso?



Exemplo 2: arquitetura de microsserviços

O Docker é bem útil para trabalharmos com uma **arquitetura de microsserviços**, algo que startups usam com uma certa frequência. Agora, o que é isso? Vamos supor que você está construindo uma cidade. Em vez de fazer um único prédio gigantesco que abrigue tudo, como lojas, escritórios e apartamentos, você decide fazer várias construções menores, cada uma com uma função específica. Uma construção pode ser um supermercado, outra pode ser uma escola e assim por diante. Cada construção é independente, mas juntas, elas formam a cidade. Sei que pode parecer óbvio, mas não era bem assim no passado: alguns anos atrás era comum termos aplicações **monolíticas**: um só projeto de *software* que fazia tudo, tal como aquele prédio gigante que comentei.

Já o exemplo dessas construções menores é a ideia básica por trás da arquitetura de microsserviços: você cria vários serviços pequenos e independentes, cada um responsável por uma funcionalidade específica. Por exemplo, em um *site* de *e-commerce*, você pode ter um microsserviço para o catálogo de produtos, outro para o carrinho de compras, outro para o pagamento, e assim por diante. Esses microsserviços se comunicam entre si através de APIs (interfaces de programação de aplicações). Isto permitea diferentes equipes trabalharem em diferentes serviços ao mesmo tempo, facilita a escalabilidade (você pode aumentar a capacidade de um serviço específico sem precisar mexer nos outros) e torna mais fácil a manutenção e atualização das partes do sistema, já que cada serviço pode ser desenvolvido, implantado e escalado de forma independente. Nesse exemplo, **dá para usar um container do Docker para cada microsserviço.**

Exemplo 3: ambiente de produção e aceitação/homologação/*staging*

Já falamos algumas vezes sobre ambiente de produção, não é? Vamos entender os diferentes ambientes de desenvolvimento de *software*: desenvolvimento, *staging* e produção, usando a analogia de um churrasco. Primeiro, imagine que você está planejando um churrasco para seus amigos. Contudo, temos um problema: você nunca fez um churrasco na sua vida. Antes de receber todos, você começa fazendo testes na sua própria churrasqueira ou no quintal alguns dias antes, experimentando diferentes cortes e tempos para assar a carne – esse é o **ambiente de desenvolvimento**. Aqui, você pode errar, ajustar os sabores e experimentar novas técnicas sem qualquer pressão. Isso, em um cenário real de TI, seria um desenvolvedor trabalhando na criação de alguma nova funcionalidade de um *software* já existente. O Docker ajuda aqui, fornecendo um ambiente consistente para cada desenvolvedor, garantindo que todos estejam testando e ajustando o seu código nas mesmas condições.

Agora, depois de aperfeiçoar suas técnicas no ambiente de desenvolvimento, você decide fazer um ensaio geral para o churrasco. Você monta a churrasqueira, prepara as carnes e cozinha a carne como se fosse o dia do churrasco, mas somente para as pessoas que moram com você – esse é o **ambiente de *staging*** (também chamado de **homologação** ou **aceitação**, dependendo da empresa). É uma réplica quase exata do evento real, em que você testa a apresentação, o tempo para assar a carne e a combinação dos acompanhamentos. Isso, em um cenário real de TI, seria um desenvolvedor testando o seu código em uma base de dados parecida com a verdadeira, mas sem estar disponível para nenhum usuário ainda. Com o Docker, você pode facilmente criar esse ambiente de *staging* que imita perfeitamente o evento real, garantindo que você possa identificar e corrigir quaisquer problemas antes de disponibilizar o que você fez para os seus usuários.

Deu tudo certo? Então finalmente chega o dia do churrasco. Os seus amigos chegam e você faz a carne – esse é o **ambiente de produção**. É o momento em que tudo precisa estar perfeito, pois qualquer falha aqui impactará diretamente seus convidados. Tudo o que foi testado e ajustado nos ambientes de desenvolvimento e *staging* agora é colocado em prática. No universo de TI, seria o momento em que disponibilizamos a aplicação para os usuários. Nessa analogia, usar Docker ajuda a garantir que o seu código funcione de forma consistente em todos os ambientes – afinal, se funcionou antes, deve funcionar aqui também. Por isso, usar Docker ajuda a minimizar riscos e garantir a qualidade do seu trabalho em todas as etapas do planejamento e execução.

DOCKER HUB

Se você trabalha com Docker também precisa saber da existência do Docker Hub, que funciona como um grande repositório de *containers* Docker, tal como o GitHub Container Registry que vimos anteriormente. Isso significa que, na prática, o Docker Hub é um serviço de registro com base em nuvem no qual você pode encontrar e compartilhar imagens Docker.

Gosto de pensar no Docker Hub como sendo a App Store, mas para *containers* Docker. No Docker Hub, você pode encontrar imagens para praticamente tudo – desde sistemas operacionais mínimos até complexas aplicações empresariais. Quando você precisa de uma imagem específica, como nginx ou mysql, você pode simplesmente puxar essa imagem do Docker Hub usando o comando docker pull.

DICA

Sempre que possível, use imagens oficiais do Docker Hub, pois elas são mantidas e atualizadas pela comunidade ou pela equipe oficial (ou seja: são mais seguras e com menos *bugs*). Além disso, escolha imagens-base leves, como a alpine, para reduzir o tamanho do *container* e melhorar a eficiência.

Além de baixar imagens, você também pode usar o Docker Hub para compartilhar suas próprias imagens. Depois de criar e testar uma imagem Docker, você pode empurrá-la para o Docker Hub com o comando **docker push**, tornando-a acessível a outros desenvolvedores.

DOCKER COMPOSE

Agora, e se você precisa trabalhar com vários containers Docker ao mesmo tempo e trabalhando **em conjunto**? Por exemplo: vamos supor que você está desenvolvendo uma aplicação que não é somente um simples serviço, mas uma combinação de diferentes serviços, como um servidor *web* + banco de dados + serviço de *cache*. Configurar e gerenciar todos esses serviços manualmente pode ser bem difícil, e é aí que entra o Docker Compose.

Com o Docker Compose, você pode definir e gerenciar todos esses serviços em um único arquivo YAML chamado **docker-compose.yml**. Nesse arquivo, você descreve os *containers* que compõem sua aplicação, especificando as imagens a serem usadas, como eles devem se conectar e quaisquer variáveis de ambiente necessárias.

IMPORTANTE

Sabemos que estamos passando vários tópicos de CI/CD para você. Entendemos que é preferível que você tenha mais propriedade no Docker do que conhecer um pouco do Docker e um pouco do Docker Compose. Portanto, caso você tenha tempo e interesse de se aprofundar no Docker Compose, recomendamos que leia a documentação oficial sobre ele aqui.

Conclusão

E chegamos ao fim de mais uma unidade de estudo. Durante esta semana, exploramos como o Docker revolucionou a maneira como desenvolvemos, testamos e implantamos aplicações. Acho que já está claro para você neste momento, mas aprender sobre Docker é essencial para qualquer pessoa que almeja uma carreira em TI, pois ele permite criar ambientes isolados e facilmente replicáveis para o desenvolvimento e execução de *software*. Isso elimina aqueles problemas tradicionais de "funcionou na minha máquina, e não sei o motivo de estar dando um erro no servidor", e garante que o *software* funcione da mesma maneira em qualquer ambiente, seja no computador do desenvolvedor, no servidor de testes ou na produção. Isso, afinal de contas, ajuda na **previsibilidade** do ciclo desenvolvimento de *software*.

Você também pode ter encontrado algumas vagas que requerem o conhecimento de Docker. De fato, isso bem valorizado no mercado de trabalho. Com a popularidade crescente de microsserviços e a necessidade de escalar aplicações de maneira eficiente, o Docker se tornou uma ferramenta bem útil para essa finalidade.

Referências

DOCKER DOCS. Overview of Docker workshop. Disponível em: https://docs.docker.com/guides/workshop/. Acesso em: 21 ago. 2024.

FREEMAN, E. **DevOps para leigos**. Rio de Janeiro: Editora Alta Books, 2021. E-book.

KIM, G.; BEHR, K.; SPAFFORD, G. **O projeto Fênix**. Rio de Janeiro: Editora Alta Books, 2020.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **Manual de DevOps**. Rio de Janeiro: Editora Alta Books, 2018.

O QUE é entrega contínua (CD)? Red hat, 26 nov. 2020. Disponível em: https://www.redhat.com/pt-br/topics/devops/what-is-continuous-delivery. Acesso em: 13 ago. 2024.