Aula 19 – Criando Imagens Docker Otimizadas (Dockerfile)

Bem-vindo(a) à Aula 19! Se você já trabalhou com Docker, sabe o quão transformador ele é para empacotar e distribuir aplicações. Contudo, assim como um carro potente precisa de ajustes finos para entregar sua melhor performance, suas imagens Docker também podem se beneficiar de uma otimização cuidadosa. Ignorar essa etapa pode levar a builds lentos, imagens gigantescas que consomem espaço e largura de banda, e, o que é pior, a vulnerabilidades de segurança desnecessárias.

Nesta aula, vamos mergulhar nas melhores práticas para construir Dockerfiles que não apenas funcionam, mas que são eficientes, leves e seguros. Você aprenderá a aplicar técnicas como multi-stage builds e a gerenciar camadas de cache de forma inteligente, transformando seus Dockerfiles em verdadeiras obras de engenharia. Ao final, você estará apto(a) a criar imagens que aceleram seus ciclos de desenvolvimento e deploy, além de fortalecer a segurança de suas aplicações em ambientes de produção.

Prepare-se para desvendar os segredos por trás de imagens Docker enxutas e robustas, um conhecimento essencial no cenário atual de arquiteturas distribuídas, microserviços e CI/CD. Vamos começar a otimizar!

O Desafio das Imagens Grandes e Vulneráveis

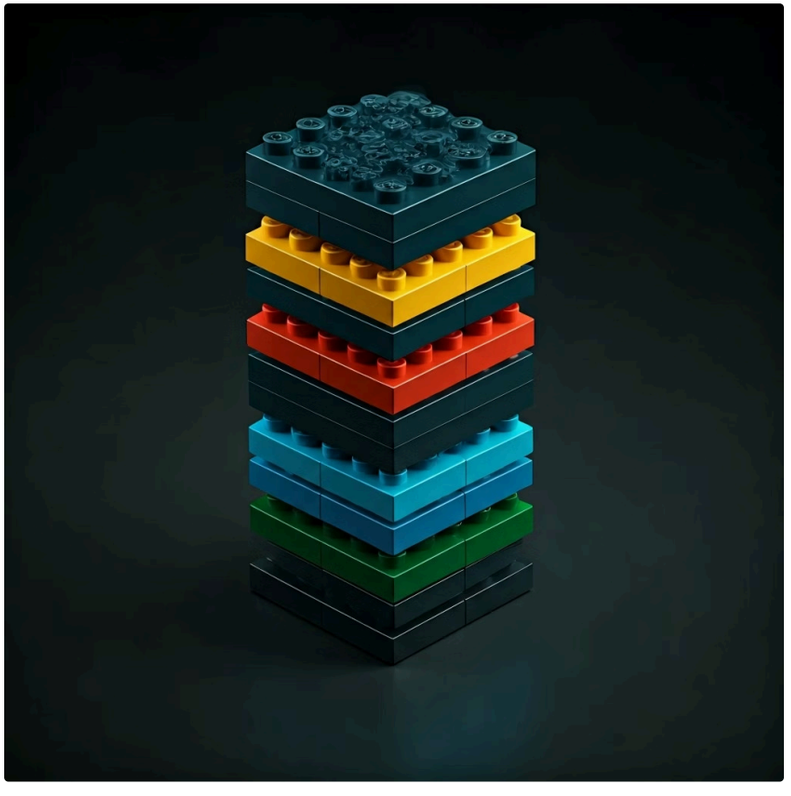
Imagine que você está se preparando para uma viagem importante. Você poderia levar consigo a casa inteira, com todos os móveis, eletrodomésticos e objetos pessoais, ou poderia empacotar apenas o essencial, de forma organizada e leve. No mundo do Docker, muitas vezes, sem perceber, acabamos "levando a casa inteira" para dentro de nossas imagens. Isso acontece quando incluímos dependências de desenvolvimento, ferramentas de build, arquivos temporários e outros itens que são cruciais durante a construção da aplicação, mas completamente desnecessários para sua execução em produção.

O problema de imagens Docker inchadas vai além do simples consumo de espaço em disco. Elas demoram mais para serem construídas, transferidas para os registries e baixadas pelos servidores, impactando diretamente a velocidade dos seus pipelines de CI/CD e o tempo de deploy. Além disso, cada pacote extra, cada biblioteca não utilizada, representa uma potencial superfície de ataque, aumentando o risco de vulnerabilidades de segurança que poderiam ser facilmente evitadas.

Nossa jornada nesta aula é justamente aprender a "fazer as malas" de forma inteligente, garantindo que suas imagens contenham apenas o que é estritamente necessário para a aplicação funcionar, de forma eficiente e segura.

Entendendo as Camadas do Docker: A Base da Otimização

Antes de otimizar, precisamos entender como o Docker constrói suas imagens. Pense em um Dockerfile como uma receita de bolo, onde cada instrução (FROM, RUN, COPY, EXPOSE, etc.) é um passo. Cada um desses passos, quando executado, cria uma nova "camada" no sistema de arquivos da imagem. Essas camadas são empilhadas uma sobre a outra, e o resultado final é a imagem completa.



Essa arquitetura de camadas é um dos pilares da eficiência do Docker. Ela permite o reuso. Se você tem várias imagens que compartilham a mesma base (por exemplo, FROM node:18-alpine), o Docker só precisa baixar e armazenar essa camada base uma vez. Mais importante ainda, durante o processo de build, o Docker utiliza um cache de camadas. Se uma instrução e seus arquivos de contexto não mudaram desde o último build, o Docker reutiliza a camada existente, economizando tempo e recursos.

Compreender esse mecanismo é crucial, pois a otimização de imagens Docker passa diretamente pela manipulação inteligente dessas camadas, seja para reduzir seu número, seu tamanho ou para maximizar o aproveitamento do cache. É como construir uma torre de LEGO: cada bloco é uma camada, e a ordem e o tipo de bloco importam para a estabilidade e eficiência da construção.

Boas Práticas Essenciais para Dockerfiles

A otimização de Dockerfiles começa com uma base sólida de boas práticas, que vão além das técnicas avançadas e se aplicam a qualquer projeto. Antes de pensar em multi-stage builds ou cache, é fundamental garantir que o seu Dockerfile esteja bem estruturado e limpo. Essas práticas são como os alicerces de uma construção: invisíveis, mas essenciais para a solidez de tudo que virá depois.

Escolha da Imagem Base

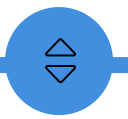
Optar por imagens menores e mais específicas, como as versões alpine de linguagens e frameworks (ex: node:18-alpine, python:3.9-alpine), pode reduzir drasticamente o tamanho inicial da sua imagem. Essas imagens são construídas sobre distribuições Linux mínimas, contendo apenas o essencial.

Uso do .dockerignore

Assim como o .gitignore, ele impede que arquivos e diretórios desnecessários (como node_modules local, .git, arquivos de teste) sejam copiados para o contexto de build do Docker, evitando que eles ocupem espaço nas camadas da imagem.

Agrupamento de Comandos

Agrupar comandos RUN sempre que possível, usando && para encadear instruções, ajuda a reduzir o número de camadas criadas. Cada RUN cria uma nova camada, então consolidar comandos relacionados em um único RUN pode tornar a imagem mais compacta.



Ordem das Instruções

Coloque as instruções que mudam com menos frequência (como a instalação de dependências do sistema) no início do Dockerfile, aproveitando melhor o cache de camadas, que veremos em detalhes mais adiante.

Multi-stage Builds: A Revolução da Otimização

Chegamos a uma das técnicas mais poderosas para otimizar imagens Docker: os **multi-stage builds**. Pense na construção de um carro. Você tem uma linha de montagem onde diversas peças são fabricadas, soldadas, pintadas. No final, o carro pronto sai da linha. Você não precisa levar toda a linha de montagem, as ferramentas de solda e os galões de tinta junto com o carro para dirigi-lo, certo?

O mesmo princípio se aplica às aplicações. Muitas vezes, para construir um aplicativo (especialmente em linguagens compiladas como Go, Java, ou até mesmo JavaScript com Webpack), precisamos de um ambiente de desenvolvimento robusto, com compiladores, SDKs, gerenciadores de pacotes e muitas dependências. No entanto, para *executar* a aplicação em produção, a maioria dessas ferramentas é completamente desnecessária. O problema é que, em um Dockerfile tradicional, todas essas ferramentas de build acabavam na imagem final, inchando-a e aumentando sua superfície de ataque.

> **Os multi-stage builds resolvem isso** permitindo que você defina múltiplos estágios dentro de um único Dockerfile. Cada estágio pode ter sua própria imagem base e suas próprias instruções. O truque é que você pode copiar artefatos (como o binário compilado ou os arquivos estáticos da sua aplicação) de um estágio anterior para um estágio final, descartando todo o ambiente de build intermediário.

O resultado é uma imagem final significativamente menor e mais segura, contendo apenas o que é essencial para a execução da aplicação.

Detalhando o Multi-stage Build na Prática

Para entender como os multi-stage builds funcionam, vamos analisar a sintaxe e o fluxo. Um Dockerfile com múltiplos estágios usa a palavra-chave AS para nomear cada estágio. O estágio final, que será a imagem de produção, copia apenas os artefatos necessários dos estágios anteriores.

Considere um exemplo de uma aplicação Go. No primeiro estágio, usamos uma imagem Go completa para compilar o código. No segundo estágio, usamos uma imagem base mínima, como scratch (a imagem mais leve possível, sem sistema operacional), ou alpine, e copiamos apenas o binário compilado do estágio anterior.

```
# Estágio 1: Build da aplicação
FROM golang:1.21-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp .

# Estágio 2: Imagem final de produção
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/myapp .
EXPOSE 8080
CMD ["./myapp"]
```

Estágio Builder

FROM golang:1.21-alpine AS builder define o estágio de build com todas as ferramentas necessárias.

Cópia Seletiva

COPY --from=builder /app/myapp . copia apenas o binário compilado do estágio anterior.

Imagem Final

Resultado: imagem minúscula, rápida para baixar e implantar, com superfície de ataque reduzida.

Neste exemplo, todo o ambiente Go, o cache de módulos e os arquivos-fonte ficam no estágio builder e não são incluídos na imagem final alpine:latest. Isso resulta em uma imagem de produção minúscula, que é mais rápida para baixar e implantar, e com uma superfície de ataque muito menor. Essa técnica é um divisor de águas para pipelines de CI/CD, onde a agilidade e a eficiência são cruciais.

Otimizando o Cache de Camadas (Layer Caching)

Além de reduzir o tamanho da imagem com multi-stage builds, outra estratégia fundamental para acelerar o processo de construção é otimizar o uso do cache de camadas do Docker. Pense no cache como uma memória de curto prazo do Docker. Quando você constrói uma imagem, o Docker armazena cada camada resultante de uma instrução. Se você fizer um novo build e uma instrução (e o contexto de arquivos associado a ela) não tiver mudado, o Docker não a executa novamente; ele simplesmente reutiliza a camada já existente no cache.

O grande segredo aqui é a ordem das instruções no seu Dockerfile. O cache é invalidado a partir da primeira instrução que muda. Isso significa que se você tem uma instrução que muda com frequência (como COPY . ., que copia todo o código-fonte da aplicação) no início do seu Dockerfile, todas as instruções subsequentes serão executadas novamente, mesmo que não tenham mudado. Isso é um desperdício de tempo e recursos, especialmente em pipelines de CI/CD onde builds são frequentes.

> **A estratégia é simples:** coloque as instruções que são menos propensas a mudar no início do Dockerfile e as que mudam com mais frequência (como o código da sua aplicação) mais para o final. Dessa forma, o Docker pode reutilizar o máximo de camadas possível do cache, acelerando significativamente os builds subsequentes.

É como preparar um prato: você faz a base que leva mais tempo e raramente muda primeiro, e só depois adiciona os ingredientes frescos que variam a cada preparo.

Exemplos Práticos de Otimização de Cache

Para ilustrar a importância da ordem das instruções no Dockerfile para o cache de camadas, vamos considerar um exemplo comum com uma aplicação Node.js.

Dockerfile NÃO Otimizado para Cache

```
FROM node:18-alpine
WORKDIR /app
COPY . . # Esta linha muda frequentemente
RUN npm install # Será executada toda vez que o código mudar
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Neste exemplo, qualquer alteração no código-fonte da aplicação (que é copiado pela instrução COPY . .) invalidará o cache a partir dessa linha. Isso significa que RUN npm install será executado novamente, mesmo que as dependências (definidas em package.json e package-lock.json) não tenham mudado. Instalar dependências pode ser um processo demorado, e refazê-lo desnecessariamente atrasa o build.

Dockerfile Otimizado para Cache

```
FROM node:18-alpine
WORKDIR /app
COPY package.json package-lock.json ./ # Copia apenas os arquivos de dependência
RUN npm install # Executa a instalação das dependências
COPY . . # Copia o restante do código
EXPOSE 3000
CMD ["node", "src/index.js"]
```

No Dockerfile otimizado, primeiro copiamos apenas os arquivos package.json e package-lock.json. Se esses arquivos não mudarem, a camada de RUN npm install será reutilizada do cache. Somente quando o código-fonte da aplicação (copiado pela segunda instrução COPY . .) mudar, o cache será invalidado a partir dessa linha. Isso garante que a instalação de dependências, que geralmente leva mais tempo, só ocorra quando for estritamente necessário, resultando em builds muito mais rápidos, especialmente em ambientes de CI/CD onde o código é frequentemente atualizado.

Reduzindo o Tamanho Final da Imagem

Mesmo com multi-stage builds e otimização de cache, ainda há outras táticas para garantir que sua imagem Docker seja o mais enxuta possível. Pense em uma mudança de casa: você já descartou os móveis velhos (multi-stage), e organizou as caixas de forma eficiente (cache), mas ainda pode reduzir o volume se jogar fora objetos que não usa, ou escolher caixas menores.

O tamanho da imagem final impacta diretamente o tempo de download, o consumo de armazenamento e, em alguns casos, até o custo em serviços de nuvem. Uma imagem menor é mais rápida para implantar e gerenciar. A primeira linha de defesa é sempre a escolha da **imagem base**. Já mencionamos as imagens alpine, que são excelentes para a maioria dos casos. Para aplicações Go, por exemplo, a imagem scratch pode ser usada como base final, pois ela é completamente vazia, exigindo que você adicione apenas o binário compilado e, se necessário, certificados SSL.

Outra técnica é a remoção de dependências e arquivos desnecessários após a instalação. Em sistemas baseados em apt (Debian/Ubuntu), comandos como apt-get clean e rm -rf /var/lib/apt/lists/* podem liberar bastante espaço. Para gerenciadores de pacotes como npm ou pip, certifique-se de que o cache de pacotes seja limpo após a instalação. Combinar comandos RUN com && e usar \ para quebrar linhas não só reduz o número de camadas, mas também permite que você execute comandos de limpeza na mesma camada em que os arquivos foram criados, garantindo que o espaço seja liberado antes que a camada seja finalizada.

Imagens Base Leves e Ferramentas de Limpeza

A escolha da imagem base é um dos fatores mais impactantes no tamanho final da sua imagem Docker. Existem diversas opções, cada uma com suas características e trade-offs.

scratch

É a imagem mais leve de todas, literalmente vazia. Ideal para aplicações Go compiladas estaticamente, onde o binário já contém todas as dependências. Não possui shell nem sistema de arquivos, o que a torna extremamente segura e pequena.

alpine

Baseada na distribuição Alpine Linux, é incrivelmente pequena (cerca de 5 MB) e segura. É uma excelente escolha para a maioria das aplicações que precisam de um sistema operacional mínimo e um gerenciador de pacotes (apk).

debian (slim)

Versões slim de imagens Debian oferecem um bom equilíbrio entre tamanho e compatibilidade. São maiores que Alpine, mas menores que as imagens Debian completas, e podem ser mais fáceis de usar para quem está acostumado com Debian/Ubuntu.

ubuntu

As imagens Ubuntu são maiores, mas oferecem ampla compatibilidade e acesso a uma vasta gama de pacotes. Geralmente, são usadas quando há uma dependência específica que não é facilmente encontrada em imagens mais leves.

Ferramentas de Limpeza

Além da escolha da base, é crucial incorporar ferramentas de limpeza no seu Dockerfile. Após instalar pacotes com apt-get, por exemplo, é uma boa prática limpar o cache do gerenciador de pacotes na mesma instrução RUN.

```
FROM debian:bullseye-slim
RUN apt-get update && \
    apt-get install -y --no-install-recommends <seus-pacotes> && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Essa abordagem garante que os arquivos temporários criados durante a instalação sejam removidos antes que a camada seja finalizada, evitando que eles contribuam para o tamanho da imagem.

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
scratch	Aplicações Go compiladas, binários estáticos	Imagem vazia, sem SO	FROM scratch
alpine	Aplicações Node.js, Python, Java, Go	Alpine Linux, gerenciador apk	FROM node:18-alpine
debian-slim	Aplicações com dependências Linux específicas	Debian Linux, versão minimalista	FROM python:3.9-slim-bullseye
apt-get clean	Limpeza de cache de pacotes	Gerenciador apt (Debian/Ubuntu)	RUN apt-get clean

Minimizando a Superfície de Ataque

A otimização de imagens Docker não se trata apenas de tamanho e velocidade; a segurança é um pilar igualmente importante. Uma imagem Docker "gorda" não é apenas lenta, mas também mais vulnerável. Cada pacote extra, cada ferramenta de desenvolvimento, cada porta aberta desnecessariamente, representa um ponto de entrada potencial para atacantes. Minimizar a superfície de ataque significa reduzir ao máximo as oportunidades para que algo dê errado.

Pense na sua aplicação como uma fortaleza. Você não deixaria todas as portas e janelas abertas, nem guardaria ferramentas de invasão dentro dela, certo? No contexto do Docker, isso se traduz em seguir o princípio do menor privilégio: conceder apenas as permissões e incluir apenas os recursos estritamente necessários para a aplicação funcionar.

> **Isso significa:** remover ferramentas de debug que não são usadas em produção, evitar a instalação de pacotes que não contribuem para a execução da aplicação e, crucialmente, não executar a aplicação como usuário root.

Ao adotar essas práticas, você não apenas torna suas imagens mais seguras contra explorações conhecidas, mas também dificulta a vida de atacantes que tentam encontrar novas vulnerabilidades. É um passo fundamental para construir aplicações robustas e confiáveis em um cenário de ameaças cibernéticas cada vez mais sofisticado.

Usuários Não-Root e Permissões

Um dos maiores riscos de segurança em containers é executar a aplicação como usuário root. Se um atacante conseguir explorar uma vulnerabilidade na sua aplicação, ele terá privilégios de root dentro do container, o que pode levar a um comprometimento muito mais sério do sistema host ou de outros containers. O princípio do menor privilégio dita que a aplicação deve rodar com o mínimo de permissões necessárias.

Para implementar isso, você deve criar um usuário não-root dentro do seu Dockerfile e configurar a aplicação para rodar com esse usuário.

```
FROM node:18-alpine
WORKDIR /app

# Cria um grupo e um usuário não-root
RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser

COPY package.json package-lock.json ./
RUN npm install
COPY . .

# Garante que os arquivos da aplicação pertencem ao usuário appuser
RUN chown -R appuser:appgroup /app

# Define o usuário que executará a aplicação
USER appuser

EXPOSE 3000
CMD ["node", "src/index.js"]
```

01

Criação do Usuário

adduser --system appuser cria um usuário de sistema appuser sem privilégios de login.

02

Definição do Usuário

A instrução USER appuser garante que todos os comandos subsequentes (incluindo o CMD que inicia a aplicação) serão executados como appuser.

03

Permissões Corretas

chown -R appuser:appgroup /app garante que o usuário appuser tenha as permissões corretas para acessar os arquivos da aplicação.

Essa prática é um pilar da segurança em containers, minimizando o impacto de possíveis brechas e alinhando-se com as melhores práticas de DevSecOps.

Varredura de Vulnerabilidades e Ferramentas

Mesmo com todas as boas práticas de otimização e segurança aplicadas no Dockerfile, a paisagem de ameaças está em constante mudança. Novas vulnerabilidades são descobertas diariamente em bibliotecas, sistemas operacionais base e pacotes. Por isso, a varredura contínua de vulnerabilidades em suas imagens Docker é uma etapa crucial e não negociável para manter a segurança das suas aplicações.

Essas ferramentas de varredura analisam as camadas da sua imagem, identificam os pacotes instalados e os comparam com bancos de dados de vulnerabilidades conhecidas (CVEs - Common Vulnerabilities and Exposures). Elas podem alertar sobre bibliotecas desatualizadas, configurações inseguras ou componentes com falhas de segurança. Integrar essas varreduras ao seu pipeline de CI/CD é uma prática de DevSecOps que permite identificar e corrigir problemas de segurança antes que as imagens cheguem à produção.

Ferramentas populares incluem:

Trivy

Uma ferramenta de código aberto leve e fácil de usar, que detecta vulnerabilidades em sistemas operacionais, linguagens de programação e configurações.

Clair

Desenvolvida pela CoreOS, é uma plataforma de análise de vulnerabilidades para imagens de containers.

Snyk

Oferece varredura de vulnerabilidades em código, dependências e imagens Docker, com integração em diversas plataformas.

A adoção dessas ferramentas e a automação da varredura são essenciais para manter um ambiente de containers seguro e atualizado, garantindo que suas imagens otimizadas não sejam apenas rápidas e leves, mas também resilientes contra ataques.

Revisão e Checklist de Otimização

Chegamos ao final da nossa jornada de otimização de imagens Docker. Vimos que criar imagens eficientes e seguras é um processo multifacetado, que envolve desde a escolha da imagem base até a implementação de práticas de segurança avançadas. A otimização não é um luxo, mas uma necessidade no desenvolvimento moderno, impactando diretamente a performance, os custos e a resiliência das suas aplicações.

Para consolidar o aprendizado, aqui está um checklist prático que você pode usar ao construir seus próximos Dockerfiles:

- **Escolha da Imagem Base**

  Opte por imagens alpine ou slim sempre que possível.

- **.dockerignore**

  Utilize-o para excluir arquivos e diretórios desnecessários do contexto de build.

- **Multi-stage Builds**

  Separe os estágios de build e runtime para descartar ferramentas de desenvolvimento.

- **Otimização de Cache**

  Ordene as instruções do Dockerfile para maximizar o reuso de camadas (menos frequentes primeiro).

- **Agrupamento de Comandos**

  Combine instruções RUN com && para reduzir o número de camadas.

- **Limpeza Pós-Instalação**

  Remova caches de pacotes e arquivos temporários na mesma instrução RUN.

- **Princípio do Menor Privilégio**

  Crie e use um usuário não-root para executar a aplicação no container.

- **Remoção de Ferramentas**

  Elimine ferramentas de debug, shells e pacotes desnecessários na imagem final.

- **Varredura de Vulnerabilidades**

  Integre ferramentas como Trivy ou Snyk ao seu pipeline de CI/CD.

Ao seguir este checklist, você estará construindo imagens Docker que são não apenas menores e mais rápidas, mas também significativamente mais seguras. Isso se traduz em deploys mais ágeis, menor consumo de recursos e maior confiança na sua infraestrutura de containers.

Consolidação

Nesta aula, desvendamos os segredos para criar imagens Docker que são verdadeiros exemplos de eficiência e segurança. Começamos entendendo a estrutura de camadas do Docker e a importância do cache, que são a base para qualquer otimização. Em seguida, mergulhamos nas boas práticas essenciais, como a escolha da imagem base e o uso do .dockerignore. O ponto alto foi a exploração dos multi-stage builds, uma técnica revolucionária para separar o ambiente de construção do ambiente de execução, resultando em imagens drasticamente menores. Complementamos com estratégias para otimizar o cache de camadas, garantindo builds mais rápidos, e discutimos como reduzir o tamanho final da imagem e, crucialmente, minimizar sua superfície de ataque através de usuários não-root e varreduras de vulnerabilidades.

> **Em prática:** Aplique o multi-stage build em seu próximo projeto. Revise seus Dockerfiles existentes e identifique oportunidades para usar imagens base mais leves. Crie um usuário não-root para sua aplicação. Integre uma ferramenta de varredura de vulnerabilidades em seu pipeline de CI/CD.

Autoavaliação

1. Qual das seguintes técnicas é mais eficaz para reduzir o tamanho final de uma imagem Docker, separando as dependências de build das dependências de runtime?
   a) Usar a instrução EXPOSE
   b) Implementar multi-stage builds
   c) Aumentar o número de camadas
   d) Copiar todos os arquivos para o contexto de build
2. Para maximizar o aproveitamento do cache de camadas do Docker, qual a melhor estratégia em relação à ordem das instruções no Dockerfile?
   a) Colocar as instruções que mudam frequentemente no início.
   b) Colocar as instruções que mudam com menos frequência no início.
   c) Usar apenas uma instrução RUN para todos os comandos.
   d) Ignorar o cache, pois ele não impacta significativamente o tempo de build.
3. Qual o principal benefício de executar uma aplicação Docker com um usuário não-root?
   a) Reduzir o tamanho da imagem.
   b) Acelerar o tempo de build.
   c) Minimizar a superfície de ataque e o impacto de uma possível exploração.
   d) Melhorar a compatibilidade com diferentes sistemas operacionais.
4. Qual ferramenta é comumente utilizada para varredura de vulnerabilidades em imagens Docker?
   a) Docker Compose
   b) Kubernetes
   c) Trivy
   d) Nginx
5. Explique como o uso do arquivo .dockerignore contribui para a otimização de imagens Docker, tanto em termos de tamanho quanto de segurança.

Gabarito: 1. b; 2. b; 3. c; 4. c

Próxima Aula

Na **Aula 20 – Orquestração de Múltiplos Containers com Docker Compose**, você aprenderá a gerenciar e orquestrar múltiplas aplicações em containers de forma eficiente, configurando redes, volumes e dependências entre serviços.

Recursos Adicionais

- **Documentação Oficial do Docker:** Para aprofundar nos detalhes de cada instrução e conceito.
- **Artigos sobre DevSecOps:** Para entender a integração de segurança no ciclo de desenvolvimento.
- **Tutoriais sobre Trivy/Snyk:** Para praticar a varredura de vulnerabilidades em suas imagens.

NOTA IMPORTANTE: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais e a documentação do Docker para verificar alterações e as práticas mais recentes.