DOCKERFILE

Um Dockerfile é um script de configuração usado para automatizar o processo de criação de imagens Docker. Essencialmente, um Dockerfile contém uma lista sequencial de instruções que são interpretadas de cima para baixo e precisam ser executadas para construir uma imagem. Cada instrução em um Dockerfile adiciona uma nova camada à imagem.

Um script de automação

Um Dockerfile age como um script de automação que especifica uma série de passos ou comandos que devem ser executados para construir uma imagem Docker. Neste sentido, ele é similar a scripts de shell ou scripts de build, onde cada comando é executado sequencialmente para configurar o ambiente, instalar dependências, copiar arquivos, e definir variáveis de ambiente, entre outras tarefas. O Dockerfile elimina a necessidade de execução manual desses passos, proporcionando uma forma automatizada e replicável de construir imagens Docker. Esse aspecto de automação é crucial em ambientes de integração contínua e entrega contínua (CI/CD), onde a consistência e a eficiência na construção de imagens são vitais.

Agnostidade relativo à Linguagens de Programação

Docker é considerado "independente de linguagem" ou "agnóstico em relação à linguagem". Esses termos capturam a ideia de que uma tecnologia, ferramenta ou abordagem pode ser utilizada com qualquer linguagem de programação, sem preferência ou dependência de uma linguagem específica. Isso acontece porque sua tecnologia de contêinerização encapsula uma aplicação e todas as suas dependências, através do Dockerfile, em uma imagem Docker, independente da linguagem de programação utilizada. Isso significa que um contêiner Docker pode executar aplicações escritas em qualquer linguagem, seja Python, Ruby, Java, ou qualquer outra, pois o que importa é que a imagem Docker contenha todas as bibliotecas, variáveis de ambiente e configurações necessárias para a execução da aplicação. Essa abordagem oferece uma grande flexibilidade, permitindo que desenvolvedores e equipes utilizem as linguagens com as quais se sentem mais confortáveis, sem se preocupar com conflitos de ambiente ou dependências na hora de implantar ou escalar suas aplicações. Portanto, a agnosticidade de linguagem do Docker simplifica o desenvolvimento, teste e implantação de software, promovendo uma maior eficiência e colaboração entre equipes multidisciplinares.

Flexibilidade e Adaptabilidade a Diversos Ambientes e Configurações

O Dockerfile suporta instruções para definir variáveis de ambiente (ENV), expor portas (EXPOSE), e definir argumentos de construção (ARG) que permitem a personalização das construções sem modificar o Dockerfile. Isso faz com que a criação de imagens Docker seja altamente flexível e adaptável a diferentes ambientes e configurações.

Consistência e “Na minha máquina funciona”

O uso de Dockerfiles facilita o desenvolvimento, o teste e a implantação de aplicações, garantindo a consistência entre os ambientes de desenvolvimento, teste e produção. Isso ajuda a reduzir os "funciona na minha máquina", pois as dependências e configurações da aplicação são encapsuladas na imagem Docker. Ao compartilhar o Dockerfile, desenvolvedores garantem que qualquer pessoa possa construir uma imagem Docker idêntica de forma rápida e sem esforço.

Uma Linguagem de Domínio Específico

O Dockerfile utiliza uma linguagem de domínio específico projetada exclusivamente para a tarefa de definir e criar imagens Docker. Esta DSL é composta por um conjunto limitado, mas poderoso, de instruções (como FROM, RUN, COPY, CMD, etc.) que são interpretadas pelo Docker para construir imagens. Cada instrução tem um propósito específico e uma sintaxe definida, tornando o Dockerfile altamente especializado para seu domínio de aplicação. Este aspecto da DSL faz do Dockerfile uma ferramenta poderosa para expressar a configuração de contêineres de maneira concisa e legível.

O Dockerfile é versionável

Ser versionável significa que você pode usar sistemas de controle de versão, como Git, para rastrear e gerenciar mudanças feitas nos Dockerfiles ao longo do tempo. Isso permite que equipes de desenvolvimento colaborem mais eficientemente, revisem mudanças históricas, revertam para versões anteriores quando necessário, e mantenham uma história clara da evolução da configuração de seus contêineres Docker. A versionabilidade dos Dockerfiles contribui significativamente para práticas de desenvolvimento ágil e integração contínua/entrega contínua (CI/CD), fornecendo uma maneira robusta de gerenciar a construção e implantação de aplicações contêinerizadas.

Referência Dockerfile

Instrução	**Descrição**
ADD	Adiciona arquivos e diretórios locais ou remotos.
ARG	Usa variáveis em tempo de construção.
CMD	Especifica comandos padrão.
COPY	Copia arquivos e diretórios.
ENTRYPOINT	Especifica o executável padrão.
ENV	Define variáveis de ambiente.
EXPOSE	Descreve em quais portas sua aplicação está ouvindo.
FROM	Cria um novo estágio de construção a partir de uma imagem base.

HEALTHCHECK	Verifica a saúde de um contêiner na inicialização.
LABEL	Adiciona metadados a uma imagem.
MAINTAINER	Especifica o autor de uma imagem.
ONBUILD	Especifica instruções para quando a imagem é usada em uma construção.
RUN	Executa comandos de construção.
SHELL	Define o shell padrão de uma imagem.
STOPSIGNAL	Especifica o sinal de chamada do sistema para sair de um contêiner.
USER	Define o ID de usuário e grupo.
VOLUME	Cria montagens de volume.
WORKDIR	Muda o diretório de trabalho.

Melhores práticas gerais para escrever Dockerfiles

Use builds multiestágio

Os builds multiestágio permitem reduzir o tamanho da sua imagem final, criando uma separação mais clara entre a construção da sua imagem e o produto final. Divida as instruções do seu Dockerfile em estágios distintos para garantir que o resultado final contenha apenas os arquivos necessários para executar a aplicação.

Usar vários estágios também permite construir de forma mais eficiente, executando etapas de build em paralelo. Veja mais em: Multi-stage builds | Docker Docs

Exclua com .dockerignore

Para excluir arquivos não relevantes para a construção, sem reestruturar seu repositório de origem, use um arquivo .dockerignore. Este arquivo suporta padrões de exclusão similares aos arquivos .gitignore. Veja mais em: Build context | Docker Docs

Crie contêineres efêmeros

A imagem definida pelo seu Dockerfile deve gerar contêineres tão efêmeros quanto possível. Efêmero significa que o contêiner pode ser parado e destruído, reconstruído e substituído com uma configuração mínima absoluta. Lembre do fator Processos nos 12 fatores para entender a importância de contêineres efêmeros: The Twelve-Factor App (traduzido) (12factor.net)

Não instale pacotes desnecessários

Evite instalar pacotes extras ou desnecessários apenas por serem agradáveis de ter. Por exemplo, você não precisa incluir um editor de texto em uma imagem de banco de dados.

Ao evitar instalar pacotes extras ou desnecessários, suas imagens têm complexidade reduzida, menos dependências, tamanhos de arquivos menores e tempos de build reduzidos.

Desacople aplicações

Cada contêiner deve ter apenas uma preocupação. Desacoplar aplicações em vários contêineres facilita a escalabilidade horizontal e a reutilização de contêineres.

Ordene argumentos de várias linhas

Sempre que possível, ordene argumentos de várias linhas alfabeticamente para facilitar a manutenção. Isso ajuda a evitar duplicação de pacotes e torna a lista muito mais fácil de atualizar. Veja como usar um proc

Aproveite o cache de build

Ao construir uma imagem, o Docker executa as instruções no seu Dockerfile, verificando se pode reutilizar a instrução do cache de build. Para mais informações sobre como o cache de build e como otimizar seus builds, veja: https://docs.docker.com/build/cache/

Fixe versões de imagens base

As tags de imagem são mutáveis, o que significa que um publicador pode atualizar uma tag para apontar para uma nova imagem. Para garantir a integridade da sua cadeia de fornecimento, você pode fixar a versão da imagem a um valor específico.

Construção da Imagem

A construção de uma imagem Docker inicia tipicamente com a instrução FROM, que define a imagem base sobre a qual a nova imagem será construída. Esta escolha é fundamental, pois determina o sistema operacional e o ambiente base disponível para a aplicação, como Python, Node.js, entre outros. Seguindo a definição da imagem base, a instrução RUN é frequentemente utilizada para executar comandos no shell da imagem, permitindo a instalação de pacotes e bibliotecas ou realizar configurações específicas necessárias para a aplicação. Para adicionar arquivos ou diretórios do contexto de construção ao sistema de arquivos da imagem, empregam-se as instruções COPY e ADD, sendo COPY a preferencial para a maioria das cópias por sua transparência e direção, enquanto ADD oferece funcionalidades adicionais como a descompressão automática de arquivos compactados e suporte a URLs. A instrução

CMD é utilizada para especificar o comando padrão que será executado ao iniciar um contêiner a partir da imagem criada, agindo como ponto de entrada padrão para a execução da aplicação. Esta instrução é vital, pois define o comportamento padrão do contêiner, mas pode ser sobrescrita por opções de linha de comando ao iniciar o contêiner.

Para enriquecer este processo, é essencial mencionar a importância da instrução WORKDIR, que estabelece o diretório de trabalho para as instruções RUN, CMD, ENTRYPOINT, COPY, e ADD subsequentes no Dockerfile. A prática de definir WORKDIR organiza melhor a estrutura de diretórios dentro da imagem e evita a necessidade de usar caminhos completos em comandos subsequentes, melhorando a legibilidade e consistência do Dockerfile.

Após estabelecer a base com FROM é recomendável definir um diretório de trabalho usando WORKDIR. Isso direciona onde os comandos RUN, COPY, ADD, CMD, e ENTRYPOINT serão executados, como em um diretório chamado /app. Assim, todos os comandos seguintes operam nesse contexto, simplificando ações como COPY . . (copiar do diretório atual no host para o diretório atual no contêiner, agora /app).

Incluir WORKDIR antes de copiar arquivos e executar comandos assegura que todos os passos de construção e execução da aplicação sejam claramente definidos e executados no contexto adequado, evitando erros comuns relacionados a caminhos de arquivos e diretórios. Além disso, ao definir um WORKDIR específico, a documentação implícita do Dockerfile é aprimorada, indicando a localização principal dos arquivos da aplicação dentro do contêiner. O WORKDIR não apenas define o contexto para as operações de cópia e execução, mas também cria o diretório se ele não existir, mitigando erros relacionados à ausência de diretórios esperados.

Instrução MAINTAINER (**Descontinuada**)

A instrução MAINTAINER **era** tradicionalmente usada para especificar o autor da imagem Docker, fornecendo um ponto de contato ou reconhecimento para quem criou a imagem. Por exemplo:

```
MAINTAINER John Doe <john.doe@example.com>
```

No entanto, é importante notar que a instrução MAINTAINER foi oficialmente descontinuada em favor de LABEL para manter informações de contato e autoria, devido à sua maior flexibilidade e capacidade de armazenar múltiplos dados.

LABEL permite adicionar metadados à imagem Docker como pares chave-valor. Essa instrução é mais versátil que MAINTAINER, pois pode conter múltiplas informações, como a versão da aplicação, a descrição, o mantenedor, e qualquer outra informação relevante. Por exemplo:

```
# Definir a imagem base
FROM node:14

# Substitui a instrução MAINTAINER
LABEL maintainer="John Doe <john.doe@example.com>"
LABEL version="1.0"
LABEL description="Uma aplicação Node.js de exemplo."
```

Instrução USER

A instrução USER especifica o ID de usuário (UID) ou o nome do usuário que será usado para executar o contêiner. Por padrão, os contêineres são executados com privilégios de root, o que pode representar um risco de segurança se a aplicação for explorada. Usar USER para especificar um usuário não-root para a execução da aplicação melhora a segurança, limitando o escopo das operações que podem ser realizadas dentro do contêiner.

```
# Definir usuário não-root
USER node
```

Neste exemplo, o contêiner será executado com as permissões do usuário node, que é comum em imagens baseadas no Node.js. Isso reduz o risco de ações maliciosas que podem afetar o sistema host.

Instrução ENV

ENV permite definir variáveis de ambiente que serão utilizadas dentro do contêiner. Essas variáveis podem ser usadas para configurar a aplicação de forma dinâmica, permitindo que valores como URLs de banco de dados, chaves de API, e outras configurações sejam modificadas sem necessidade de reconstruir a imagem.

```
# Definir variáveis de ambiente
ENV NODE_ENV=production \

DATABASE_URL=exemplo://usuario:senha@host:porta/banco_de_dados
```

Utilizar ENV facilita a gestão de configurações sensíveis e específicas do ambiente, seguindo as práticas recomendadas dos 12 fatores para configurações de aplicações.

```
# Definir a imagem base
FROM node:14

# Metadados da imagem
LABEL maintainer="John Doe <john.doe@example.com>"
LABEL version="1.0"
LABEL description="Uma aplicação Node.js de exemplo."

# Definir variáveis de ambiente
ENV NODE_ENV=production \

DATABASE_URL=exemplo://usuario:senha@host:porta/banco_de_dados

# Configurar o diretório de trabalho
WORKDIR /usr/src/app

# Copiar arquivos da aplicação
COPY . /usr/src/app

# Instalar as dependências da aplicação
RUN npm install

# Definir usuário não-root
USER node

# Definir o comando padrão para executar a aplicação
CMD ["node", "app.js"]
```