[Informações iniciais sobre instalação](#_Toc192765527)

[Tutorial de instalação do WSL 2](#_Toc192765528)

[Review sobre o Docker Desktop](#_Toc192765529)

[Docker Desktop - Otimizações para Linux, Mac e Windows](#_Toc192765530)

[Informações importantes de uma imagem-container de desenvolvimento](#_Toc192765531)

[Escolha do nome de arquivos Dockerfile](#_Toc192765532)

[Escolha da imagem FROM](#_Toc192765533)

[Determinando um usuário padrão](#_Toc192765534)

[Estudo de caso com 1 – imagem sem usuário definido](#_Toc192765535)

[Estudo de caso com 2 – imagem com usuário definido](#_Toc192765536)

[Diretório de trabalho](#_Toc192765537)

[Definindo o diretório de trabalho na imagem](#_Toc192765538)

[COPY e RUN](#_Toc192765539)

[Problema do COPY e RUN para algumas linguagens de programação](#_Toc192765540)

[EXPOSE](#_Toc192765541)

[CMD vs ENTRYPOINT](#_Toc192765542)

[Script de inicialização para resolução do problema COPY + RUN + volume](#_Toc192765543)

[Conteúdo do script](#_Toc192765544)

[Não dê permissão de execução para o script no host](#_Toc192765545)

[Lidando com o arquivo de script](#_Toc192765546)

[Arquivo Dockerfile final](#_Toc192765547)

[Estratégias para melhorar performance em sincronização de arquivos](#_Toc192765548)

[Escolha o nome do arquivo Docker compose](#_Toc192765549)

[Build e context](#_Toc192765550)

[Suba e teste o container](#_Toc192765551)

[Fazer build novamente em caso de alteração no Dockerfile](#_Toc192765552)

[CMD e Entrypoint no Docker compose](#_Toc192765553)

[Organizando os primeiros passos do projeto com Docker compose](#_Toc192765554)

[Instalando a lib mysql2](#_Toc192765555)

[Rode o index.js](#_Toc192765556)

[Alterando o package.json para facilitar](#_Toc192765557)

[Definição de argumentos e variáveis de ambientes](#_Toc192765558)

[Healthcheck e depends on](#_Toc192765559)

[restart: always](#_Toc192765560)

[restart: on-failure](#_Toc192765561)

[Networks e host.docker.internal](#_Toc192765562)

[Configurar o Network](#_Toc192765563)

[Configurar o Network com um Network criado manualmente](#_Toc192765564)

[Utilizar o host.docker.internal](#_Toc192765565)

[Include](#_Toc192765566)

[Informações importantes](#_Toc192765567)

[Extends](#_Toc192765568)

[tmpfs](#_Toc192765569)

[Profiles](#_Toc192765570)

[Modo watch](#_Toc192765571)

[Observação importante sobre o modo watch](#_Toc192765572)

[Modo Watch – exemplo mais complexo](#_Toc192765573)

[Outro modo de fazer ignore](#_Toc192765574)

[Detalhes importantes](#_Toc192765575)

[Docker compose override](#_Toc192765576)

[Docker compose Project name](#_Toc192765577)

[Comandos do dia a dia](#_Toc192765578)

[Como resolver problemas gerais que acontecem no Docker](#_Toc192765579)

[Docker compose com e sem o -d](#_Toc192765580)

[Docker compose run](#_Toc192765581)

[Docker compose build](#_Toc192765582)

[Logando no container com usuário root](#_Toc192765583)

[Docker compose config](#_Toc192765584)

[Variáveis de ambiente pré-definidas no docker compose](#_Toc192765585)

[Configurando os exports automaticamente no VSCode](#_Toc192765586)

[Configurando os exports no arquivo .env](#_Toc192765587)

[Copiando arquivos de e para o container](#_Toc192765588)

[Copiando arquivos com docker cp](#_Toc192765589)

# Informações iniciais sobre instalação

## Tutorial de instalação do WSL 2

Todas as informações importantes sobre a instalação do WSL 2 estão no tutorial <https://github.com/codeedu/wsl2-docker-quickstart>

Se você está utilizando o Docker Desktop, não se esqueça de ativar o modo autoMemoryReclaim para ajudar a economizar recursos no Windows.

Também há um vídeo no nosso canal do Youtube falando sobre as novidades do WSL 2, recomendado assistir: <https://youtu.be/-oxnRGhA9Mg?si=Dk6IT-qSR0Kf2T6P>.

## Review sobre o Docker Desktop

Recomendado assistir este vídeo para entender os motivos de usar o Docker Desktop, além do funcionamento em cada sistema operacional.

Link do vídeo: <https://youtu.be/99dCerRKO6s?si=TMvpv_iZy81UC2-Q>

## Docker Desktop - Otimizações para Linux, Mac e Windows

Na última aula eu falei sobre a instalação do Docker, mencionei sobre as das principais opções de usar Docker na sua máquina, Docker Engine vs Docker Desktop. Para quem vai começar a usar o Docker Desktop, é importante fazer algumas configurações para otimizar o uso de recursos dele na sua máquina. No vídeo que eu citei na aula anterior tem todas estas configurações, veja o vídeo: <https://www.youtube.com/watch?v=99dCerRKO6s>

Mais abaixo vou citar os pontos importante para você configurar

**- Linux:**

* Configure os recursos que a VM pode utilizar, você pode configurar em Resources o número de core do processor, memória RAM e o tamanho de disco que será usado para imagens, volumes e containers. Faça o devido ajuste conforme for necessário para sua máquina, a qualquer momento você pode reajustar estes valores.
* Configure o Resource Saver em Resources. Esta opção devolve CPU e memória RAM quando não há containers sendo executados, há um timer que será o intervalo de tempo que será verificando esta situação, aplique o intervalo conforme adequado, o valor padrão é a cada 5 min.

**- Mac:**

* Configure os recursos que a VM pode utilizar, você pode configurar em Resources o número de core do processor, memória RAM e o tamanho de disco que será usado para imagens, volumes e containers. Faça o devido ajuste conforme for necessário para sua máquina, a qualquer momento você pode reajustar estes valores.
* Configure o Resource Saver em Resources. Esta opção devolve CPU e memória RAM quando não há containers sendo executados, há um timer que será o intervalo de tempo que será verificando esta situação, aplique o intervalo conforme adequado, o valor padrão é a cada 5 min.
* Verifique na aba General, se o file sharing está marcado como VirtioFS. Isto é muito importante, porque vai trazer mais performance aos volumes Docker.

**- Windows:**

* Configure o Resource Saver em Resources. Esta opção devolve CPU e memória RAM quando não há containers sendo executados, há um timer que será o intervalo de tempo que será verificando esta situação, aplique o intervalo conforme adequado, o valor padrão é a cada 5 min.
* Configure no .wslconfig a opção autoMemoryReclaim. Esta opção fazerá o WSL devolver memória RAM ao Windows quando ver que o uso do WSL está baixo. Veja no tutorial de WSL para entender como configurar esta opção: [https://github.com/codeedu/wsl2-docker-quickstart?tab=readme-ov-file#liberar-memória-ram-do-wsl-2](https://github.com/codeedu/wsl2-docker-quickstart?tab=readme-ov-file#liberar-mem%C3%B3ria-ram-do-wsl-2)

Com estas configurações, o Docker Desktop terá um uso mais otimizado de CPU e memória.

# Informações importantes de uma imagem-container de desenvolvimento

Em desenvolvimento, existem necessidades específicas de certas configurações para satisfazer o que é necessário para desenvolvimento. Por exemplo, ferramentas como curl ou wget para fazer chamadas http.

O objetivo é prover containers que deem apoio ao desenvolvedor. Portanto em desenvolvimento não há tanta preocupação com o tamanho da imagem ou container. Ainda assim deve ser seguro e rápido. Além disso, apesar de ter ferramentas e configurações diferentes, deve ser o mais próximo possível da imagem de produção.

## Escolha do nome de arquivos Dockerfile

É possível ter múltiplos arquivos Dockerfile com nomes diferentes. Uma boa prática é que esses arquivos fiquem na raiz ou em uma pasta separada e nomeá-los adicionando um sufixo correspondente, por exemplo:

Dockerfile.base

Dockerfile.prod

Dockerfile.dev

Dockerfile.staging

Deve-se ter cuidado ao ter vários arquivos para que não se torne difícil de manter uma vez que as regras podem ficar segregadas.

Sendo assim, pode-se ter um arquivo Dockerfile.base servindo de base para os outros. Assim, cada um dos outros arquivos teriam um FROM apontando para a imagem em questão.

## Escolha da imagem FROM

A imagem deve ser mais próxima possível da imagem de produção ou a mesma, para evitar problemas ao publicar ou manter o software.

A imagem deve ser compatível com o ambiente de execução, ou seja, precisa suportar as ferramentas e libs que que a aplicação precisa para rodar.

As menores distribuições Linux:

* Alpine Linux (termina em :alpine ou -alpine)
* Debian Slim (Termina em :slim ou -slim)
* BusyBox (Termina em :busybox ou -busybox) – É um Linux padrão, só tem os comandos mais básicos.
* Scratch – É a imagem mais básica que tem. É usada quando se quer criar uma imagem do zero.
* Distroless – São imagens que não têm shell, não tem nada. É só a aplicação e a linguagem desejada, é uma imagem bem segura.

As imagens oficiais do Docker são, normalmente, as melhores opções pois são mantidas pela comunidade, atualizadas frequentemente e mais seguras. Por exemplo, caso necessite de uma imagem do Node, a imagem oficial que está no Dockerhub é a melhor.

Escolher uma imagem pequena é bom pois você começa com o mínimo de ferramentas e vai adicionando conforme necessário. Isso torna o ambiente mais limpo, organizado e seguro. Quanto menos coisas, menos problemas.

## Determinando um usuário padrão

Decidir qual será o usuário base da imagem ou container é uma decisão importante da imagem. Toda imagem vem com um usuário padrão e normalmente é o root. Não é bom utilizar o usuário root pois ele tem todas as permissões e pode fazer qualquer coisa no container.

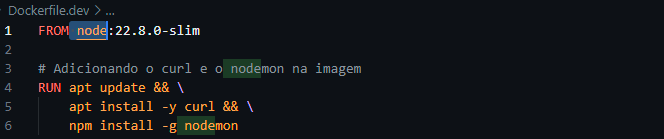
Algumas vezes é necessário utilizar o usuário root, mas é bom evitar pois é um risco de segurança. O container deve rodar em um ambiente limitado. Quanto mais limitado, mais seguro.

Muitas vezes têm-se um usuário não root para imagem de produção e um usuário root para imagem de desenvolvimento. Isso pode gerar uma acomodação em relação ao ambiente, não pensando nas limitações da imagem de produção. Quanto mais próximo os dois ambientes, melhor será.

Para desenvolvimento, ter um usuário não root numa situação em que serão sincronizados arquivos com o container todo o tempo é importante pois, se os arquivos forem criados dentro do container com o usuário root como padrão, pertencerão ao root. Se houver uma tentativa de editar esses arquivos na IDE haverá erro de permissão.

### Estudo de caso com 1 – imagem sem usuário definido

**1 – Criar uma imagem se usuário definido:**

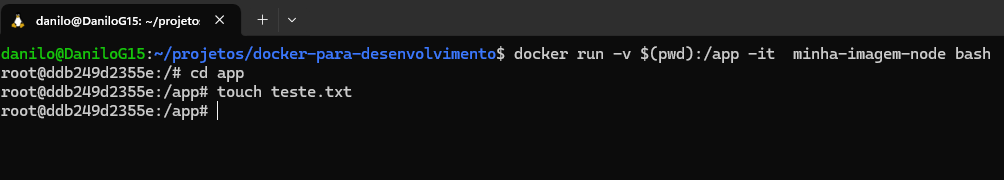




**2 – Criar um volume da pasta do projeto para dentro da imagem criada:**

* No diretório do projeto, rode no terminal
  + docker run -v $(pwd):/app -it minha-imagem-node bash
    - pwd – diretório current
    - :/app – pasta de destino dentro do container
    - -it e bash estão sendo usados para após criar o volume, executar diretamente o bash dentro da imagem

**3 – Criar um arquivo teste.txt dentro do container:**

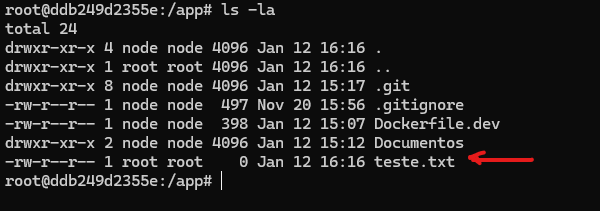
****

**4 – Tentar alterar o arquivo criado dentro do VSCode:**

Abrir o arquivo gerado, no VSCode e adicionar algum conteúdo.

Ao tentar salvar ocorrerá um permission denied, pois ele pertence ao usuário root.

**5 – Verificar dentro do container o usuário a qual pertence este arquivo:**

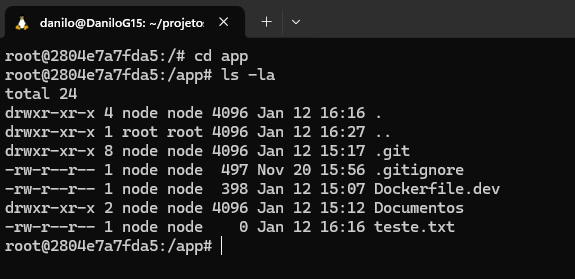


Conforme imagem, o arquivo pertence ao usuário root.

#### Solução do estudo de caso com permissão ao arquivo

* Na pasta onde se encontra o arquivo, em sua máquina, rodar(para o usuário danilo):
  + sudo chown danilo:danilo teste.txt

Dentro do container, verificar novamente o usuário que tem permissão de acesso ao arquivo:

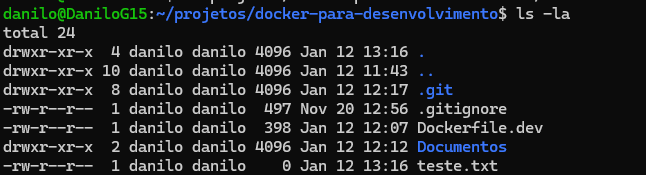


É possível notar que o usuário mudou para node (dentro do container).

***Observação sobre o usuário node:***

Esse usuário node, é o usuário criado pela imagem node (utilizada para criação da imagem minha-imagem-node) com o id 1000.

Já na máquina, o usuário mudou para danilo:

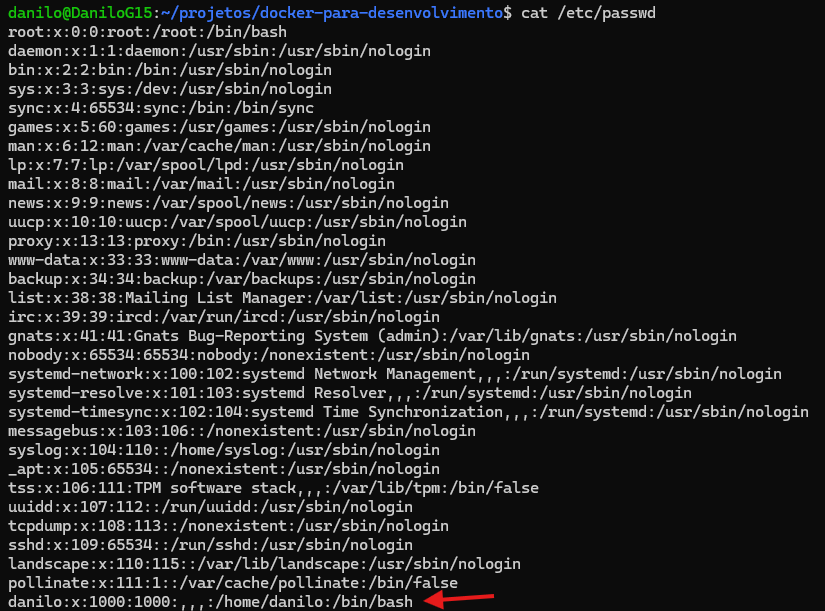


#### Relação entre o usuário do container e o usuário da máquina

O que ocorre acima é que, apesar dos nomes de usuários distintos, eles possuem o mesmo id, e é o id quem interessa.

Pode-se verificar executando o seguinte comando:

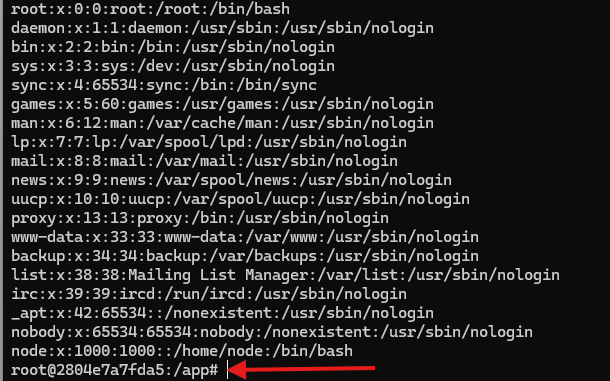
* + cat /etc/passwd

  
  
Os usuários do Linux possuem um id, que começa com o 0. (depois de x: )

Usuário 0 é o root. Do 1 até o 100 são usuários internos. Do 1000 em diante são nossos usuários criados para poder manipular o sistema operacional.

Como é possível observar, o usuário danilo possui o id 1000.

Rodando o mesmo comando dentro do container:



É possível observar que o usuário node possui o mesmo id.

Ao tentar alterar novamente o arquivo pelo VSCode, será possível concluir a alteração, uma vez que o usuário logado é o danilo.

**Caso seja desejado dar permissões para pastas e subpastas, deve-se utilizar o -R (recursivo):**

* + sudo chown danilo:danilo -R Documentos

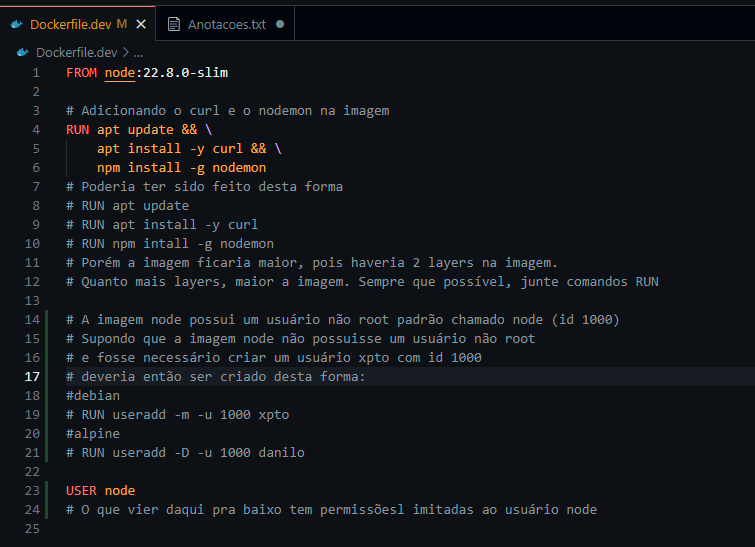
#### E se o usuário danilo não tivesse o id 1000 ?

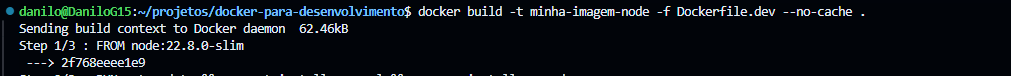
Neste caso, no Dockerfile deveria ser criado um usuário não root com um ID que corresponde ao id da máquina. Por exemplo (danilo possui id 1001):

* para debian:
  + RUN useradd -m -u 1001 meu-user-1001
* para alpine:
  + RUN useradd -D -u 1001 meu-user-1001

### Estudo de caso com 2 – imagem com usuário definido

**1 – Criar uma imagem com usuário definido:**

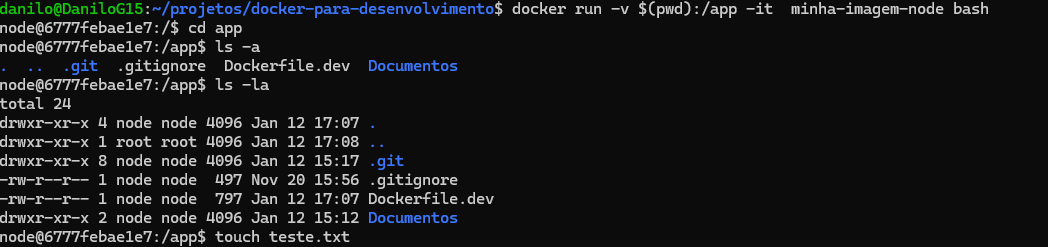




**2 – Criar um volume da pasta do projeto para dentro da imagem criada:**

* No diretório do projeto, rode no terminal
  + docker run -v $(pwd):/app -it minha-imagem-node bash
    - pwd – diretório current
    - :/app – pasta de destino dentro do container
    - -it e bash estão sendo usados para após criar o volume, executar diretamente o bash dentro da imagem

**3 – Criar um arquivo teste.txt dentro do container:**

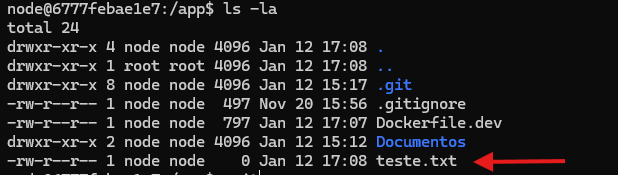


**4 – Tentar alterar o arquivo criado dentro do VSCode:**

Abrir o arquivo gerado, no VSCode e adicionar algum conteúdo.

Ao tentar salvar, ocorrerá com sucesso.

**5 – Verificar dentro do container o usuário a qual pertence este arquivo:**



Pode-se verificar que, diferente da situação do Estudo de caso 1, o arquivo teste.txt já pertence ao usuário node (id 1000).

Por isso foi possível a alteração do arquivo pelo VSCode, pois meu usuário logado no sistema é o 1000 também. (Relação entre usuário do container e usuário da máquina explicado no Estudo de caso 1).

# Diretório de trabalho

O diretório de trabalho é importante pois define onde o projeto será executado. Quando se entra no container via bash, sh ou qualquer outro terminal, ele abre no diretório de trabalho.

Além disso, outras ferramentas como Docker Desktop, VSCode e etc também reconhecem isso.

Não existe um local correto para o diretório de trabalho. Diferentes imagens podem variar o caminho.

O importante é que seja um diretório em que o usuário padrão tenha permissão de escrita.

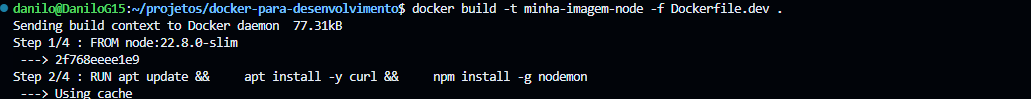
## Definindo o diretório de trabalho na imagem

**1 – Definir o diretório de trabalho**

Será definido o diretório /home/node/app. O diretório /app não existe, será criado pelo container. O usuário node tem permissão de escrita no diretório /home/node



**2 – Fazer o build novamente**

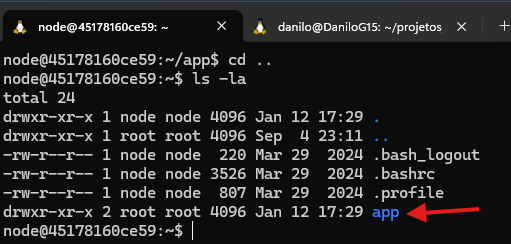
****

**3 – Rodar o container**

****

A pasta aberta no bash já aponta para o diretório de trabalho.

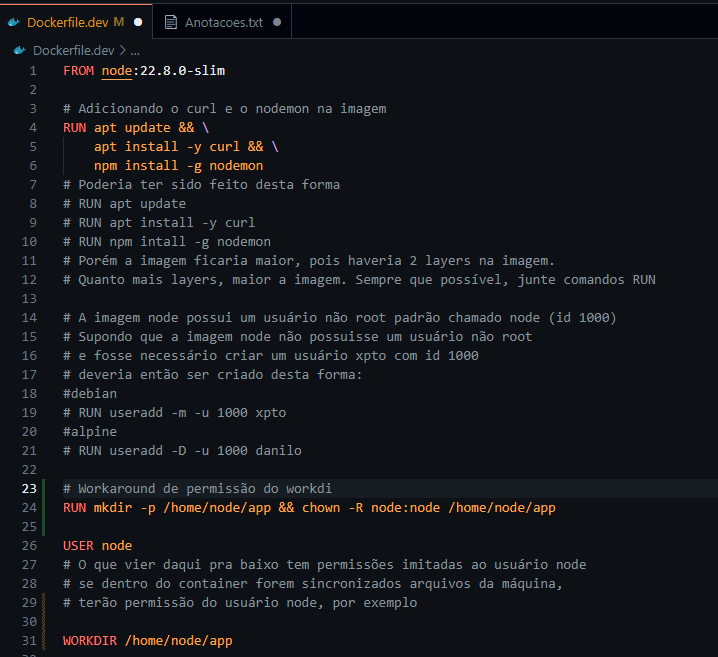
**4 – Verificar a quem a pasta pertence**

****

Neste caso, percebe-se que a usuário root é quem tem permissão de acesso à pasta app.

EM algumas versões do docker isso pode ocorrer. A versão do node utilizada neste exemplo é a Docker version 20.10.21, build baeda1f.

Caso ocorra esta situação, um workaround é criar o diretório e dar permissão no Dockerfile antes de mudar o usuário.



A imagem então deve ser criada novamente e a pasta verificada.



Na versão Docker version 27.4.1, build b9d17ea, esse problema não ocorreu.

# COPY e RUN

Para ilustrar e aproveitar o exemplo já utilizado anteriormente, tendo já uma imagem base, um usuário definido e o diretório de trabalho, será feita a instalação das bibliotecas da aplicação Node, ou seja, gerar o node\_modules e disponibilizar para o container.

Para isso será utilizado COPY, para copiar o arquivo package.json para a imagem e o comando RUN para instalar as dependências.

Exemplo de um package.json para teste:  
{

  "name": "app",

  "version": "1.0.0",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "description": "",

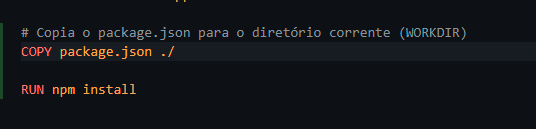
  "dependencies": {

    "express": "^4.21.2"

  }

}

No Dockerfile é acrescentado os comandos para copiar o package.json para dentro do container e rodar o npm install

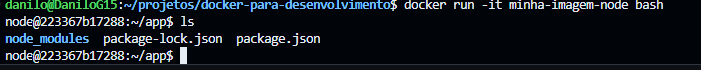


Roda o build da imagem novamente:

* + docker build -t minha-imagem-node -f Dockerfile.dev .

Assim o npm install será executado na imagem.

Uma vantagem desta prática é que o Docker faz caching da layer do comando RUN, ou seja, se o package.json não mudar, o Docker vai usar o cache e não vai executar o npm install, utilizando assim o node\_modules que está armazenado na layer gerada para a imagem, economizando assim, muito tempo .

Assim, ao rodar a imagem, é possível verificar que a node\_modules está dentro do container.  


## Problema do COPY e RUN para algumas linguagens de programação

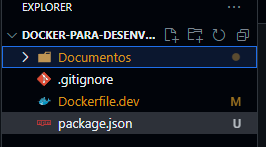
Em linguagens de programação que geram dependências que precisam ficar juntas ao código fonte, pode ocorrer um problema caso sejam utilizados volumes.

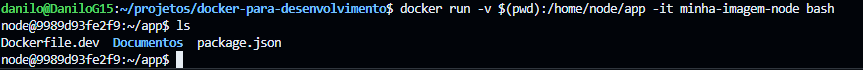
Ao iniciar o container, o docker vai pegar todo o conteúdo (em $(pwd)) e mandar para a pasta configurada (/home/node/app). Ou seja, node\_modules será apagado.

Para ilustrar, rode a imagem utilizando um volume:

* + docker run -v $(pwd):/home/node/app -it minha-imagem-node bash

Será possível ver que o /home/node/app foi substituído com o conteúdo do volume:





Para resolver essa situação será necessário um script que será executado ao subir o container. Esse script será explicado posteriormente, ele é quem irá gerar a node\_modules.

# EXPOSE

A instrução EXPOSE serve apenas para documentação, ela não faz nada. Isso é confuso pois ao colocar EXPOSE + porta (ex EXPOSE 3000) estaria sendo feito m bind com a porta e não é isso. Ela apenas está dizendo que nesta imagem, quando o container iniciar, essa porta será utilizada. Mas se você não utilizar, não acontecerá nada.

É uma boa prática utilizar o EXPOSE para deixar claro quais portas serão utilizadas.

Por exemplo:  
EXPOSE 3000 3001 3002

# CMD vs ENTRYPOINT

As instruções CMD e ENTRYPOINT são executadas quando o container é iniciado.

Saber a diferença entre as 2 é importante pois muda a forma como o container vai iniciar.

O CMD especifica o comando inicial do container, mas pode ser substituído a qualquer momento. Por exemplo:

* + docker run -it minha-imagem-node bash

A imagem está sendo rodada com o *bash* no final. O bash vai ser o comando inicial do container. O bash substitui o CMD.

Exemplo:

No Dokerfile acrescenta-se CMD [ "node", "-v"].  
Gere a imagem novamente.

* + docker build -t minha-imagem-node -f Dockerfile.dev .

Rode o container:

* + docker run -it minha-imagem-node

O resultado:



Rode o container com bash:

* + docker run -it minha-imagem-node bash

O resultado:



No ENTRYPOINT, o comando sempre vai rodar, não pode ser substituído.

Exemplo:

No Dokerfile acrescenta-se ENTRYPOINT [ "node", "-v"].  
Gere a imagem novamente.

* + docker build -t minha-imagem-node -f Dockerfile.dev .

Rode o container:

* + docker run -it minha-imagem-node

O resultado:



Rode o container com bash:

* + docker run -it minha-imagem-node bash

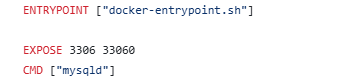
O resultado:



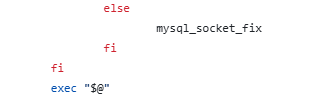
Isso ocorre porque o ENTRYPOINT não é substituído. O que acontece é que os comandos serão agregados ao ENTRYPOINT. Então é como se o comando fosse node -v bash.

Entretando os 2 comandos podem ser utilizados juntos.

Um exemplo com a imagem do Mysql:



O mysql utiliza um shell script como pré preparação para que algo seja rodado. Em determinado momento esse shell script possui uma instrução exec “$@”



Ele é o responsável por fazer com que as 2 instruções possam coexistir. O conteúdo de CMD será utilizado nesse $@.

Para desenvolvimento é interessante que se use o CMD para deixar o container sempre disponível e também para manter o container de pé. (Caso queira entrar com o bash, por exemplo).

Um exemplo de CMD que pode ser utilizado somente para manter o container de pé:

CMD [“tail”, “-f”, “/dev/null”] - Lê indefinidamente nada.

* tail – lê arquivos
* -f – indefinidamente
* /dev/null – dispositivo nulo do Linux

# Script de inicialização para resolução do problema COPY + RUN + volume

Para resolver o problema do COPY e RUN comentado anteriormente, onde a node\_modules é sobrescrita pelo volume, será necessário um script de inicialização.

A ideia é ter o container já pronto para desenvolver. Então é necessário que a node\_modules esteja disponível no momento em que o container ficar de pé.

As instruções abaixo não farão mais sentido, uma vez que o volume está sendo utilizado, pois, ao utilizar o volume, o conteúdo será copiado para dentro do container:

COPY package.json ./

RUN npm install

## Conteúdo do script

Deve-se criar um script de inicialização. Pode-se criar uma pasta script, na raiz do projeto, contendo o arquivo. No exemplo foi chamado de start.sh e possui o seguinte conteúdo:

#!/bin/bash

npm install

# Comando apenas para manter o container de pé

tail -f /dev/null

Para executar um script no Linux, é preciso que o arquivo possua o cabeçalho #!/bin/bash para dizer ao Linux qual programa será utilizado para executar o arquivo. Além disso, o arquivo precisa ter permissão de execução dentro do container, para tanto é necessário conceder essa permissão ao arquivo.

## Não dê permissão de execução para o script no host

Quando você monta um volume em um contêiner Docker, o sistema de arquivos do host é compartilhado com o contêiner. No entanto, o contêiner pode ter um ambiente diferente do host, incluindo o sistema operacional, as permissões de usuário e grupo, e as configurações de segurança. Aqui estão algumas razões pelas quais você pode enfrentar problemas ao tentar executar um script com permissões de execução definidas no host:

1. **Diferenças de Usuário e Grupo**: O usuário e grupo que você está usando no contêiner podem não ter as mesmas permissões que o usuário e grupo no host. Por exemplo, se o script no host tem permissões de execução apenas para o proprietário, e o contêiner está rodando como um usuário diferente, ele não poderá executar o script.
2. **Sistema de Arquivos Diferente**: O sistema de arquivos do contêiner pode ter diferentes políticas de segurança ou não suportar certas permissões que estão definidas no host.
3. **Contexto de Segurança**: Docker pode estar configurado com políticas de segurança adicionais, como AppArmor ou SELinux, que podem restringir a execução de arquivos montados de volumes.
4. **Permissões de Montagem**: Quando você monta um volume, as permissões de montagem podem ser diferentes. Por exemplo, se você montar um volume como somente leitura, não será possível alterar as permissões de execução dentro do contêiner.

## Lidando com o arquivo de script

Uma vez que não é uma boa ideia dar permissão de execução ao script no host, o script será copiado para dentro do container.

Será acrescido ao Dockerfile o seguinte (antes de definir o USER node):

* + COPY /script/start.sh /
  + RUN chmod +x /start.sh

Isso é feito antes do USER node, pois, ao utilizar COPY, mesmo que esteja dentro do contexto do USER node, o dono do arquivo fica sendo o root. Então é o usuário root quem precisa dar a permissão de execução.

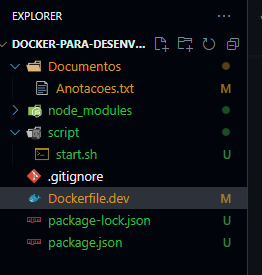
No final do arquivo deve ser adicionado:

* + CMD [ "/start.sh" ]

Deve-se criar uma nova imagem e subir o container (cuidado para não substituir o CMD com bash no final):

* + docker build -t minha-imagem-node -f Dockerfile.dev .
  + docker run -v $(pwd):/home/node/app -it minha-imagem-node

Repare que o node\_modules foi gerado:



## Arquivo Dockerfile final

FROM node:22.8.0-slim

RUN apt update && \

    apt install -y curl && \

    npm install -g nodemon

COPY script/start.sh /

RUN chmod +x /start.sh

USER node

WORKDIR /home/node/app

EXPOSE 3000

CMD [ "/start.sh" ]

## Estratégias para melhorar performance em sincronização de arquivos

Existe uma estratégia que pode ajudar a tornar o manuseio das pastas de libs como node\_modules mais simples em ambientes em que a sincronização destas pastas seja lento.

Esta estratégia é indicada só para situações em que você tem o comando de instalação das libs, como o npm install no início do container demorando muito para gerar a node\_modules, por exemplo. Veja o vídeo desta estratégia: <https://youtu.be/fC7ZDGHpTUQ?si=ymlIbbioNLrf2l08>

# Escolha o nome do arquivo Docker compose

Uma vez que temos múltiplos ambientes, podemos ter vários nomes de docker-compose também. Uma boa prática é deixar esses arquivos na raiz ou em uma pasta separada, acrescentado um sufixo referente ao ambiente. Exemplo:

* + docker-compose.dev.yaml
  + docker-compose.prod-yaml
  + docker-compose.staging.yaml

Deve-se tomar cuidado pois ter vários arquivos docker-compose pode fazer com que a aplicação fique difícil de manter, pois as regras acabam ficando segregadas.

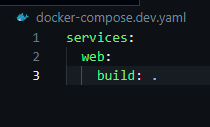
O docker-compose tem mais recursos para reuso de código do que o Dockerfile. (Recursos como include e extends, por exemplo).

# Build e context

* Crie um arquivo docker-compose.dev.yaml junto ao Dockerfile.dev



Com as seguintes configurações abaixo, o Docker iria ler o arquivo Dockerfile para fazer o build.



Como o Dockerfile possui o sufixo .dev, é necessário adicionar um context, conforme a seguir:  
services:

  web:

    build:

      context: .

      dockerfile: Dockerfile.dev

    volumes:

      - .:/home/node/app

O context: . significa que será considerada a pasta atual e o dockerfile: indica qual o Dockerfile a ser considerado.

**Importante:** Caso seja informado alguma pasta específica no context: , ao fazer um COPY no Dockerfile, será utilizado como referência o contexto definido em context.

## Suba e teste o container

Repare que o nome do arquivo deve ser informado devido ao sufixo dev.

* + docker compose -f docker-compose.dev.yaml up
  + docker compose -f docker-compose.dev.yaml exec web bash
    - web é o nome do serviço configurado

## Fazer build novamente em caso de alteração no Dockerfile

Caso algo seja alterado no dockerfile, pode-se fazer o seguinte para subir o container e executar um novo build, através do docker compose:

* + docker compose -f docker-compose.dev.yaml up –build

# CMD e Entrypoint no Docker compose

Exemplo de CMD e entrypoint no docker compose:

services:

web:

build:

context: .

dockerfile: Dockerfile.dev

command: node -v

entrypoint: ["executable", "arg"]

volumes:

- .:/home/node/app

Tanto o command quanto o entrypoint informado no arquivo docker-compose irá sobrescrever o CMD e o ENTRYPOINT do Dockerfile.

# Organizando os primeiros passos do projeto com Docker compose

Um pequeno projeto em Node com Express e banco de dados será feito para servir como exemplo de como utilizar o Docker compose para desenvolvimento.

Foi adicionado um banco de dados ao docker-compose.dev.yaml:

services:

  web:

    build:

      context: .

      dockerfile: Dockerfile.dev

    volumes:

      - .:/home/node/app

    ports:

      - 3000:3000

  db:

    image: mysql:8.0.30-debian

    environment:

      MYSQL\_ROOT\_PASSWORD: root

      MYSQL\_DATABASE: my\_database

Foi criada uma pasta src com um arquivo index.js dentro com o seguinte conteúdo:

const express = require('express')

const app = express()

const port = 3000

app.get('/', (req, res) => {

res.send('Hello World!')

})

app.get('/external-api', async function(req, res){

//const address = 'http://host.docker.internal:9000/products';

const address = 'http://external-api:9000/products';

const response = await fetch(address)

const data = await response.json()

res.send(data)

});

app.get('/test-db', async function(req, res){

const mysql = require('mysql2');

const connection = mysql.createConnection({

host : process.env.DB\_HOST,

user : process.env.DB\_USER,

password : process.env.DB\_PASSWORD,

database : process.env.DB\_DATABASE,

});

connection.connect(error => {

if (error) {

console.error('Error connecting to database: ', error);

return;

}

res.send('Connected to DB');

});

});

app.listen(port, () => {

console.log(`Example app listening on port ${port}`)

});

## Instalando a lib mysql2

Suba o container:

* + docker compose -f docker-compose.dev.yaml up

Entre no container:

* + docker compose -f docker-compose.dev.yaml exec web bash

Dentro do container, instale o mysql2:

* + npm install mysql2

## Rode o index.js

* + node src/index.js
  + teste entrando em [localhost:3000](http://localhost:3000/)

## Alterando o package.json para facilitar

Coloque o seguinte script no package.json

"scripts": {

"start": "nodemon src/index.js"

},

Sai do container, caso esteja nele, e entre novamente e execute o script:

* + docker compose -f docker-compose.dev.yaml exec web bash
  + npm start

# Definição de argumentos e variáveis de ambientes

Variáveis de ambiente podem ser definidas no arquivo docker-compose, Dockerfile ou na linha de comando ao fazer o docker build.

A variável de ambiente fica disponível em todo o ciclo de vida da imagem do container. (Tanto dentro dos arquivos docker-compose ou Dockerfile quanto dentro das imagens)

Os argumentos são valores que estão disponíveis apenas no processo de build da imagem. Pode ser alguma informação sensível que não deva estar disponível dentro do container ou um valor que não faz sentido estar dentro do container, por exemplo.

Exemplo de argumento no Dockerfile (versão do nodemon):

FROM node:22.8.0-slim

ARG NODEMON\_VERSION=3.1.7

# Adicionando o curl e o nodemon na imagem

RUN apt update && \

    apt install -y curl && \

    npm install -g nodemon@${NODEMON\_VERSION}

* + docker compose -f docker-compose.dev.yaml up –build

Em um ambiente para desenvolvimento é comum que haja mudança de valores de configuração por diversos motivos. Então definir variáveis de ambiente diretamente no arquivo docker-compose ou Dockerfile não é a melhor opção pois irá alterar o arquivo diversas vezes. Não é desejado que o arquivo fique sendo commitado, por exemplo.

Então o ideal é jogar as variáveis de ambiente em um arquivo env.

Conteúdo do .env:

DB\_HOST=db

DB\_USER=root

DB\_PASSWORD=root

B\_NAME=my\_database

Configuração do .env no docker-compose:

services:

…

    ports:

      - 3000:3000

    env\_file:

      - .env

Pode haver mais de um arquivo de variáveis de ambiente. Se fosse esse o caso poderia ser configurado desta forma:

services:

…

    ports:

      - 3000:3000

    env\_file:

      - .env

- .env.dev

Neste caso, as variáveis de ambiente de todos os arquivos serão processadas. Caso haja variáveis iguais entre os arquivos (mesmo nome), prevalece a que foi definido no último arquivo configurado na lista.

O que for definido no env\_file, é obrigatório que exista. Uma forma de definir um opção pra um arquivo env não obrigatório é a seguinte:

env\_file:

      - .env

- path: env.override

      required: false

# Healthcheck e depends on

Ao utilizar um depends\_on é indicado que um serviço depende de outro. Isso altera a ordem em que os serviços iniciam ao subir o container. Por exemplo:

services:

  web:

    build:

…

    env\_file:

      - .env

    depends\_on:

      - db

  db:

    image: mysql:8.0.30-debian

    # Poderia utilizar um env file para trazer mais dinamismo

    environment:

      MYSQL\_ROOT\_PASSWORD: root

      MYSQL\_DATABASE: my\_database

Desta forma acima, o serviço do banco irá subir primeiro.

Porém, o depends\_on não garante que o serviço esteja ok, ou seja, caso o serviço db esteja com erro, o serviço web irá subir da mesma forma. Para lidar com isso, pode-se utiliza o healthcheck da seguinte forma:

 …

depends\_on:

      db:

        condition: service\_healthy

  db:

    image: mysql:8.0.30-debian

    # Poderia utilizar um env file para trazer mais dinamismo

    environment:

      MYSQL\_ROOT\_PASSWORD: root

      MYSQL\_DATABASE: my\_database

    healthcheck:

      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]

      interval: 30s # interval - com que frequência o healthcheck é executado

      timeout: 5s # timeout - quanto tempo esperar antes de considerar o healthcheck falhado

      retries: 5 # retries - número de tentativas antes de considerar o container como falhado

      start\_period: 10s # start\_period - tempo de espera antes de iniciar o healthcheck

      #ou seja, o healthcheck será executado a cada 30 segundos, com um timeout de 5 #segundos, 5 tentativas e um tempo de espera inicial de 10 segundos

No caso acima, o node web só irá subir se o node db subir e estiver saudável.

Há uma opção em que se o serviço db falhar o serviço web irá reiniciar para tentar novamente:

depends\_on:

      db:

        condition: service\_healthy

restart: true

## restart: always

Ao utilizar restart: always, caso esse container falhe ao tentar subir, este irá tentar novamente.

db:

    image: mysql:8.0.30-debian

restart: always

Uma consequência do uso disto é que toda vez que a o docker inicia (desligar a máquina e ligar novamente, por exemplo) faz com que o db suba. Isso acontece pois o docker entende que o serviço deva ser iniciado toda vez que o docker inicie, devido ao Always.

## restart: on-failure

Ao utilizar restart: on-failure:5 é dito ao docker que em caso de falha ele irá tentar subir novamente, até 5 tentativas.

db:

    image: mysql:8.0.30-debian

restart: on-failure:5

# Networks e host.docker.internal

No Docker Compose, as networks são usadas para definir como os containers se comunicam entre si e com o mundo externo. Aqui estão alguns conceitos básicos sobre networks no Docker Compose:

No arquivo docker-compose.yml, você pode definir networks personalizadas que seus serviços usarão. Isso permite que você controle a topologia da rede e a comunicação entre os containers.

Tipos de Networks:

Bridge: É o tipo padrão de rede no Docker. Containers na mesma rede bridge podem se comunicar entre si usando o nome do serviço como hostname.

Host: Usa a rede do host diretamente. Isso significa que o container compartilha a stack de rede do host, o que pode ser útil para casos específicos de desempenho ou quando você precisa de acesso direto à rede do host.

None: Desativa a rede para o container. Isso é útil quando você deseja isolar completamente o container da rede.

Configuração no docker-compose.yml: Você pode definir networks na seção networks do arquivo docker-compose.yml e associar serviços a essas networks. Por exemplo:

version: '3'

services:

app:

image: myapp

networks:

- mynetwork

networks:

mynetwork:

driver: bridge

Comunicação entre Containers: Containers conectados à mesma network podem se comunicar entre si usando o nome do serviço como hostname. Isso facilita a configuração de aplicações distribuídas.

host.docker.internal: Este é um hostname especial que resolve para o endereço IP do host Docker. É útil para permitir que um container acesse serviços que estão rodando no host.

Para ilustração crie uma pasta chamada external-api e crie um novo arquivo docker-compose.external-api.yaml que apenas roda uma api simples em Node na porta 9000, para ilustração:

services:

  external-api:

    image: node:22.8.0-slim

    command: npx json-server --watch api.json --port 9000

    ports:

      - 9000:9000

    working\_dir: /home/node/app

    volumes:

      - ./:/home/node/app

Crie também um arquivo api.json com o seguinte conteúdo:  
{

    "products": [

        {

            "id": 1,

            "name": "Product 1",

            "price": 100

        },

        {

            "id": 2,

            "name": "Product 2",

            "price": 200

        },

        {

            "id": 3,

            "name": "Product 3",

            "price": 300

        }

    ]

}

Ao rodar essa nova aplicação, ela estará em uma rede diferente da aplicação Node já abordada em capítulos anteriores, pois estão em docker-compose diferentes.

Para que esses 2 containers possam sem comunicar existem algumas opções.

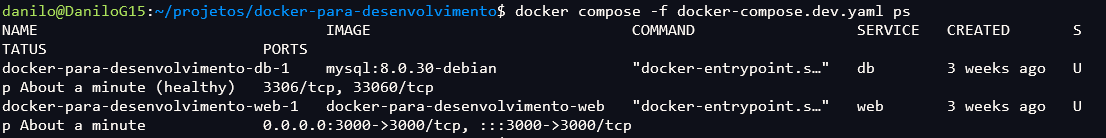
## Configurar o Network

Uma opção é criar uma rede em comum para esses 2 containers.

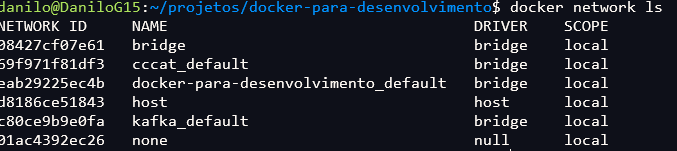
Para ilustração:

- Suba a aplicação do docker-compose.dev.yaml e execute o comando ps:

* + docker compose -f docker-compose.dev.yaml up
  + docker compose -f docker-compose.dev.yaml ps



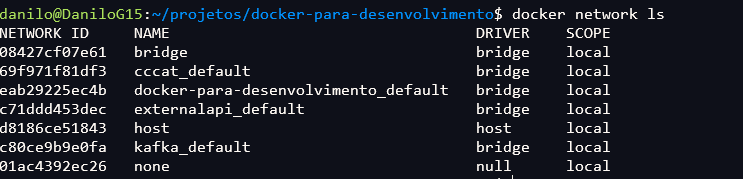
Repare que o Docker utiliza o nome da pasta, o nome do serviço e o número 1 como nome do container.

Através do comando docker network ls é possível ver as redes existente:  


Percebe-se que existe uma rede chamada docker-para-desenvolvimento-default, que é a rede que os containers do mesmo docker-compose vão compartilhar.

- Suba a aplicação docker-compose.external-api.yaml e verifique a rede:

* + docker compose -f docker-compose.external-api.yaml up
  + docker network ls



O nome da rede da aplicação external-api é externalapi\_default.

Configure a rede external-api no docker-compose.dev.yaml:

services:

web:

build:

…

depends\_on:

…

**networks:**

**- externalapi\_default**

db:

…

healthcheck:

...

**networks:**

**- externalapi\_default**

**networks:**

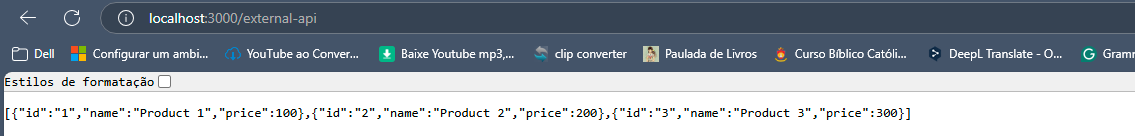
**externalapi\_default:**

**external: true**

Mate e suba o docker-compose.dev.yaml novamente e faça um npm start.

* + docker compose -f docker-compose.dev.yaml up
  + docker compose -f docker-compose.dev.yaml exec web bash
  + npm start

No navegador entre em localhost:3000/external-api:



Observe que retornará o json criado.

Esse endereço external-api foi definido no index.js já apresentado anteriormente.

app.get('/external-api', async function(req, res){

const address = 'http://external-api:9000/products';

const response = await fetch(address)

const data = await response.json()

res.send(data)

});

Pode-se reparar que a aplicação external-api é acessada através do hostname external-api, na porta 9000, com sucesso.

## Configurar o Network com um Network criado manualmente

Seria possível criar um network específico.

* + docker network create my-external-network

Alterar o docker-compose.external-api.yaml e o docker-compose.dev.yaml para utilizar essa rede criada.

## Utilizar o host.docker.internal

Uma outra alternativa é a utilização de um endereço que os containers irão sempre entender, independente de estarem em redes conjuntas ou separadas, ou mesmo para aplicações que não estão rodando em container. Algumas vezes é importante que uma aplicação que está em container comunique-se com uma aplicação que não está em container. Utilizar o IP da máquina seria uma opção para este caso, mas pode geral outros problemas, pois o IP pode mudar.

Então uma alternativa é o host.docker.internal que permite essa comunicação de forma simples e é um padrão na comunidade docker.

Para utilizar o host.docker.internal será necessário alterar o etc/hosts (Mac e Linux, no Windows C:\Windows\System32\drivers\etc\hosts).

O etc/hosts possui uma tabela de roteamento de nomes e ip.

Então no etc/hosts será adicionado: 127.0.0.1 host.docker.internal.

O etc/hosts do container que vai se comunicar também precisa ser alterado.

No docker compose isso é configurado adicionando tudo o que deve ir para o etc/hosts da seguinte forma:

 extra\_hosts:

      - "host.docker.internal:host-gateway"

O host-gateway é uma palavra chave que o Docker utiliza para colocar o IP do gateway .

Para teste, deve-se alterar docker-compose.dev.yaml, adicionando o extra\_hosts no serviço web e removendo quaisquer configuração previamente feita para network:

services:

web:

…

extra\_hosts:

- "host.docker.internal:host-gateway"

Alterar também o index.js conforme abaixo:

app.get('/external-api', async function(req, res){

**const address = 'http://host.docker.internal:9000/products';**

const response = await fetch(address)

const data = await response.json()

res.send(data)

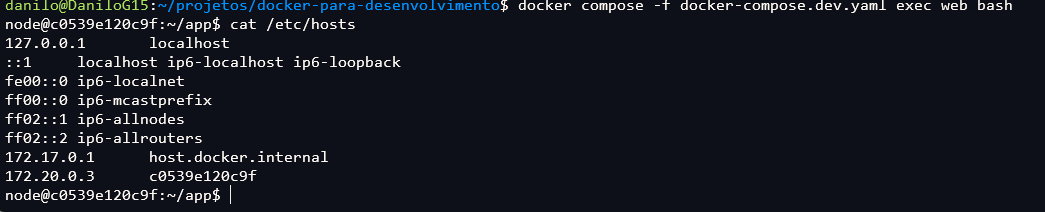
});

Pare o container e suba de novo:

* + docker compose -f docker-compose.dev.yaml up

Entre no container web e verifique o /etc/hosts:

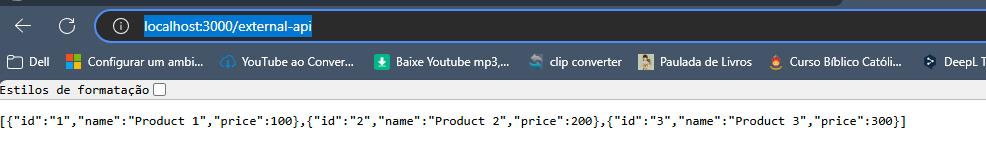
* + docker compose -f docker-compose.dev.yaml exec web bash



Execute o npm start:

* + npm start

Teste o <http://localhost:3000/external-api> novamente.



# Include

O Include foi introduzido no Docker Compose versão 2.20.3.

Serve para quando se deseja trabalhar com múltiplos arquivos docker compose.

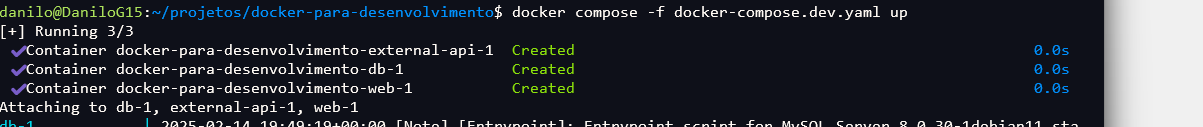
Para teste, deve-se alterar docker-compose.dev.yaml, adicionando, na primeira linha:

include:

  - ./external\_api/docker-compose.external-api.yaml

Ao subir o docker-compose.dev.yaml , o docker-compose.external-api.yaml também subirá:

* + docker compose -f docker-compose.dev.yaml up



Caso seja necessário entrar no container external-api, pode ser feito através do docker-compose.dev.yaml. Por exemplo:



## Informações importantes

- Ao utilizar o include, os containers passam a compartilharem a mesma rede.

Caso o index.ja fosse alterado novamente, conforme abaixo (retirando o host.docker.internal), iria funcionar:

app.get('/external-api', async function(req, res){

**const address = 'http://external-api:9000/products';**

const response = await fetch(address)

const data = await response.json()

res.send(data)

});

- Ao utilizar include, deve-se garantir que as configurações entre os containers não sejam conflitantes (por exemplo variáveis de ambientes) ,

# Extends

Com extends é possível estender docker composes e também coisas dentro do arquivo docker compose (caso em que tenha apenas um arquivo, por exemplo).

Importante saber que as configurações *volumes\_from* e *depends\_on* não são estendidadas, quando se estende as configurações de um serviço.

Para ilustrar o extends, crie um arquivo docker-compose-common.yaml com o seguinte conteúdo:

services:

  web\_common:

    build:

      context: .

      dockerfile: Dockerfile.dev

    volumes:

      - .:/home/node/app

    env\_file:

      - .env

    extra\_hosts:

     - "host.docker.internal:host-gateway"

Imagine um cenário onde haverá 2 aplicações Node rodando. Devemos alterar o arquivo docker-compose-dev.yaml para herdar as configurações do web\_common, além de criar outra aplicação Node:

services:

  web1:

    extends:

      file: docker-compose-common.yaml

      service: web\_common

    ports:

      - 3000:3000

    depends\_on:

      db:

        condition: service\_healthy

  web2:

    extends:

      file: docker-compose-common.yaml

      service: web\_common

    ports:

      - 3001:3000

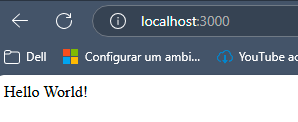
    depends\_on:

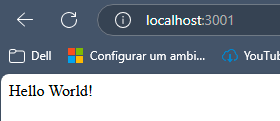
      db:

        condition: service\_healthy

Para testar:

* + docker compose -f docker-compose.dev.yaml up
  + docker compose -f docker-compose.dev.yaml exec web1 bash
    - npm start
  + docker compose -f docker-compose.dev.yaml exec web2 bash
    - npm start

Acessar uma api na porta 3000 e outra na porta 3001:  




Numa aplicação node é comum não expor o Node para internet e sim deixar um servidor na frente, como o Nginx.

Configure o Nginx após o web2:

 nginx:

    image: nginx:1.19.10-alpine

    profiles: [nginx]

    restart: always

    ports:

      - 4000:80

    volumes:

      - ./.docker/nginx/nginx.conf:/etc/nginx/nginx.conf

Crie as pastas .docker/nginx e o arquivo nginx.conf dentro de nginx com o seguinte conteúdo:

events {

}

http {

upstream myapp {

server web1:3000;

server web2:3000;

}

server {

listen 80;

location / {

proxy\_pass http://myapp;

}

}

}

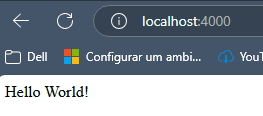
Essa configuração serve para balancear as requisições entre os 2 serviços, web1 e web2.

Mate o container e rode novamente.

Para testar:

* + docker compose -f docker-compose.dev.yaml up
  + docker compose -f docker-compose.dev.yaml exec web1 bash
    - npm start
  + docker compose -f docker-compose.dev.yaml exec web2 bash
    - npm start

Acesse a api na porta 4000 (porta configurada no nginx)



# tmpfs

tmpfs é muito importante para acelerar algumas vezes a execução dos containers. É um recurso que tem grande valor, por exemplo, para lidar com testes de integração por ser um processo de CI. Este recurso permite que um diretório do container fique totalmente na memória RAM.

Crie um arquivo docker-compose.test.yaml com o seguinte conteúdo:

include:

  - ./external-api/docker-compose.external-api.yaml

services:

  web1:

    extends:

      file: docker-compose.common.yaml

      service: web

    ports:

      - 3000:3000

    depends\_on:

      db:

        condition: service\_healthy

        #restart: true # se o container deve ser reiniciado se caso a dependência falhar

  db:

    image: mysql:8.0.30-debian

    #restart: on-failure:5 | always

    # poderia usar um env\_file para trazer mais dinamismo

    environment:

      MYSQL\_ROOT\_PASSWORD: root

      MYSQL\_DATABASE: my\_database

**tmpfs:**

**- /var/lib/mysql**

    healthcheck:

      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]

      interval: 30s # interval - com que frequência o healthcheck é executado

      timeout: 5s # timeout - quanto tempo esperar antes de considerar o healthcheck falhado

      retries: 5 # retries - número de tentativas antes de considerar o container como falhado

      start\_period: 10s # start\_period - tempo de espera antes de iniciar o healthcheck

      #ou seja, o healthcheck será executado a cada 30 segundos, com um timeout de 5 segundos, 5 tentativas e um tempo de espera inicial de 10 segundos

No exemplo acima, o diretório de armazenamento do Mysql está em memória acelerando assim o IO.

# Profiles

Imagine um cenário onde se deseja rodar apenas alguns containers e não todos.

Para ilustrar, será criado um novo serviço no docker-compose.dev.yaml. O phpadmin é uma interface que permite consultar dados do banco de dados.

phpmyadmin:

image: phpmyadmin:5.2.1-apache

environment:

PMA\_HOST: db

PMA\_PORT: 3306

PMA\_USER: root

PMA\_PASSWORD: root

ports:

- 8080:80

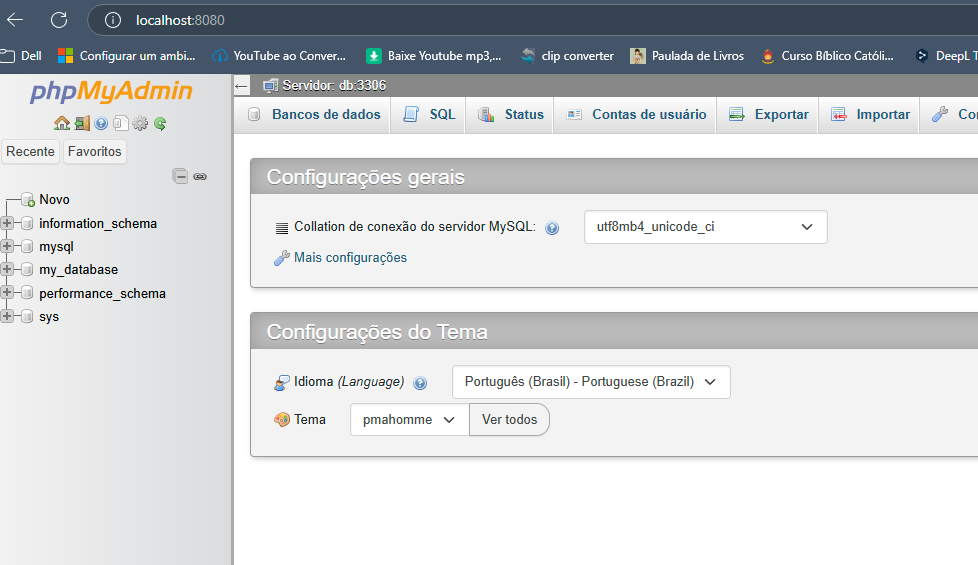
depends\_on:

db:

condition: service\_healthy

Suba o container:

* + docker compose -f docker-compose.dev.yaml up
  + Teste o phpMyAdmin: localhost:8080



Imagine que este serviço phpadmin somente seja desejado em certas situações, como por exemplo, debug da aplicação. Para isto existe o recurso de *profile*.

O profile vai criar um perfil dentro do docker compose fazendo com que, caso o container tenha um perfil, este só será executado caso o perfil seja apontado no comando junto com o up.

Como exemplo será adicionado um perfil *debug* dentro do phpmyadmin.

phpmyadmin:

image: phpmyadmin:5.2.1-apache

profiles: [debug]

environment:

PMA\_HOST: db

PMA\_PORT: 3306

PMA\_USER: root

PMA\_PASSWORD: root

ports:

- 8080:80

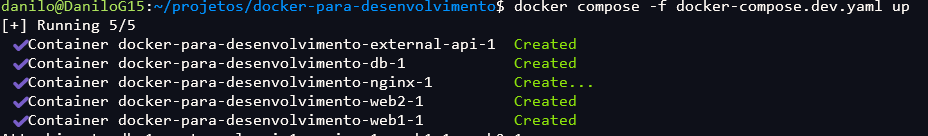
depends\_on:

db:

condition: service\_healthy

Mate o container e suba novamente:

* + docker compose -f docker-compose.dev.yaml up

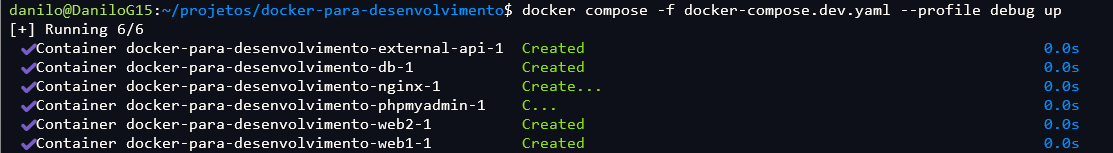


Verifique que o phpadmin não subiu pois não foi especificado o perfil debug, ou seja,o que está configurado com profiles só roda quando especificado.

Mate o container e suba novamente com a opção do profile debug:

* + docker compose -f docker-compose.dev.yaml --profile debug up

Repare que neste caso o container subiu:



Acrescente agora o profile *nginx* nos serviços nginx e web2:

 web2:

    profiles: [nginx]

...  
nginx:

    image: nginx:1.19.10-alpine

    profiles: [nginx]

...

Mate o container e suba novamente com a opção do profile debug e nginx:

* + docker compose -f docker-compose.dev.yaml --profile debug --profile nginx up

Deve-se observar os casos em que um serviço depende de outro. No exemplo, o phpadmin depende do db. Mas imagine que fosse ao contrário. Que o db dependesse de phpadmin. O profile debug está configurado no phpadmin. Então se subir o container sem informar o profile, ocorrerá erro, pois o db dependeria do phpadmin, o qual não irá subir.

Também é válido observar que a utilização de muitos perfis dificulta a manutenção e utilização.

Não é aconselhado, por exemplo, ter profiles para diferentes ambientes no mesmo arquivo. É melhor que se tenha um arquivo docker compose diferente para cada ambiente. O profile seria pra definir uma perspectiva que aquele ambiente seria rodado, por exemplo, rodar um ambiente de teste na perspectiva de debug.

# Modo watch

O watch é um modo em que o docker compose irá sincronizar coisas com o container.

Uma observação a se fazer é que o watch pode-se confundir com volume a primeiro momento. O volume não foi feito para desenvolvimento, é utilizado, porém não ideal. O volume é I/O bloqueante e são 2 processos trabalhando, 1 para olhar o que mudou na máquina e outro para o container para a realização da sincronia.

O watch parece com o volume, mas não é. Foi feito para desenvolvimento, não é I/O bloqueante

Observe o seguinte serviço:

 nginx:

    image: nginx:1.19.10-alpine

    profiles: [nginx]

    restart: always

    ports:

      - 4000:80

    volumes:

      - ./.docker/nginx/nginx.conf:/etc/nginx/nginx.conf

Há um volume que está sincronizando o nginx.conf. Caso alguma mudança seja feita no arquivo nginx.conf, será necessário matar o container e subir de novo para que a sincronização ocorra.

Com o watch este tipo de situação é evitado.

Um detalhe é que sempre deve-se trabalhar com o build, não da pra usar o watch usando a imagem diretamente (exemplo: image: nginx:1.19.10-alpine).

Para exemplificar, crie um Dockerfile dentro de .docker/nginx, com o seguinte conteúdo:

FROM nginx:1.19.10-alpine

Duplique o arquivo docker-compose.dev.yaml e renomeie para docker-compose.watch.yaml.

Altere o serviço nginx para:

 nginx:

    build: ./.docker/nginx

    profiles: [nginx]

    restart: always

    ports:

      - 4000:80

    develop:

      watch:

        - action: sync+restart

        path: ./.docker/nginx/nginx.conf

        target: /etc/nginx/nginx.conf

Existem os seguintes modos de sincronia:

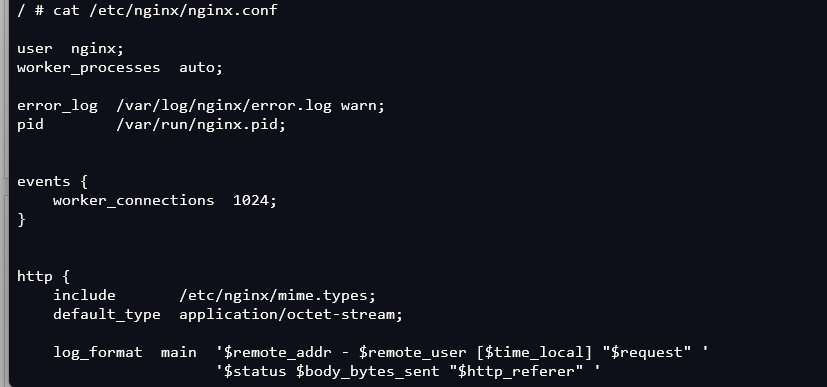
* sync+restart realiza a sincronização, reinicia o container e não faz rebuild;
* sync apenas sincroniza o arquivo da máquina com o container;
* rebuild: realiza a sincronização, reinicia o container e faz rebuild;

Suba o container novamente (observe o --watch no fim do comando):

* docker compose -f docker-compose.watch.yaml --profile nginx up --watch

Uma observação é que quando o container sobe, a sincronia não é feita. Seria necessário salvar o arquivo nginx.conf (na máquina) para que ocorra a sinconia. Para visualizar rode:

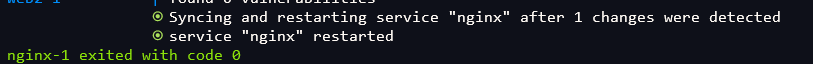
* docker compose -f docker-compose.watch.yaml --profile nginx exec nginx sh
* cat /etc/nginx/nginx.conf



Repare que o arquivo não é o mesmo que consta em : ./.docker/nginx/nginx.conf.

Abra o arquivo ./.docker/nginx/nginx.conf e o salve.

Repare que foi feita a sincronia:





Para que isso se resolve poder ter um COPY no Dockerfile, ou então utilizar uma instrução experimental x-initialSync: true:

nginx:

    build: ./.docker/nginx

    profiles: [nginx]

    restart: always

    ports:

      - 4000:80

    develop:

      watch:

        - action: sync+restart

**x-initialSync: true**

          path: ./.docker/nginx/nginx.conf

          target: /etc/nginx/nginx.conf

Suba o container novamente e verifique o arquivo:

* docker compose -f docker-compose.watch.yaml --profile nginx up --watch
* docker compose -f docker-compose.watch.yaml --profile nginx exec nginx sh
* cat /etc/nginx/nginx.conf

Caso tivesse sido utilizado o COPY, cada vez que o arquivo fosse alterado, o rebuild deveria ser feito. (modo rebuild)

## Observação importante sobre o modo watch

Não é recomendado sincronizar pastas que possuem muitos arquivos, como por exemplo a node\_modules. Apesar do modo watch ser mais eficiente que o volume, ele trabalha com multi processos. Sendo assim, se houver muitos arquivos, haverá um uso de CPU intenso.

Então, acaba-se tornando muito comum utilizar o .dockerignore quando se utiliza o modo watch, para que sejam ignoradas pastas indesejadas.

## Modo Watch – exemplo mais complexo

Crie a seguinte estrutura na raiz do projeto: mais-exemplos/modo-watch/nestjs-exemplo.

Crie o seguinte .dockerignore, dentro de nestjs-exemplo:

node\_modules/

.git/

docker-compose.watch.yaml

Dockerfile

Crie o seguinte Dockerfile, dentro de nestjs-exemplo:

FROM node:22.8.0-slim

RUN apt update && apt install -y procps

USER node

RUN mkdir -p /home/node/app

WORKDIR /home/node/app

COPY --chown=node:node package\*.json .

RUN npm install

COPY --chown=node:node . .

CMD [ "tail", "-f", "/dev/null" ]

Crie o docker-compose.watch.yaml:

services:

  app:

    build: .

    command: npm run start:dev

    restart: always

    ports:

      - 3000:3000

    develop:

      watch:

        - action: sync

          path: .

          target: /home/node/app

        - action: rebuild

          path: package.json

Repare que o Dockerfile irá copiar todos os arquivos para imagem. Por isso não está sendo utilizado o x-initialSync.

O modo watch irá sincronizar tudo o que está do diretório corrente, exceto o que está ignorado no .gitignore.

Essas 2 linhas finais do docker-compose:

        - action: rebuild

          path: package.json

indicam que sempre que o package.json for alterado, será feito um rebuild.

Baixe os demais arquivos que deverá compor este exemplo em [fc4-docker-para-desenvolvimento/mais-exemplos/modo-watch/nestjs-example at main · devfullcycle/fc4-docker-para-desenvolvimento · GitHub](https://github.com/devfullcycle/fc4-docker-para-desenvolvimento/tree/main/mais-exemplos/modo-watch/nestjs-example)

Suba o container para teste:

* docker compose -f docker-compose.watch.yaml up --watch

### Outro modo de fazer ignore

Ao invés de utilizar o .gitignore (boa prática recomendada), uma outra forma de ignorar arquivos / diretórios é da seguinte forma:

 develop:

      watch:

        # sync, sync+restart, rebuild

        - action: sync

          path: .

          target: /home/node/app

          ignore:

            - node\_modules/

            - .gitignore

### Detalhes importantes

No processo da imagem, uma node\_modules será gerada dentro do container. É possível também gerar uma node\_modules na pasta raiz do projeto, executando os comandos necessários. Isso é necessário, pois senão a IDE não vai dar auto complete, não vai entender certos imports, o que torna o desenvolvimento mais custoso.

É importante lembrar que as 2 node\_modules não são as mesmas. A node\_modules não está sendo sincronizada.

Outro ponto importante é que o modo watch não tem um binding bidirecional. Por exemplo, se for criado um arquivo qualquer dentro do container - touch teste.txt – este arquivo ficará apenas dentro do container. O processo é unidirecional – host-> container.

# Docker compose override

O Docker Compose Override é uma funcionalidade do Docker Compose que permite modificar ou complementar a configuração definida no arquivo docker-compose.yml sem alterar o arquivo original. Isso é feito através de um arquivo adicional chamado docker-compose.override.yml.

Por padrão, quando você executa o comando docker-compose up, o Docker Compose automaticamente carrega o docker-compose.override.yml, combinando suas configurações, se o nome do arquivo for docker-compose.yaml ou docker-compose.yml ou compose.yaml ou compose.yaml. Isso é útil em ambientes de desenvolvimento, onde você pode querer alterar algumas configurações, como volumes, variáveis de ambiente ou portas, sem impactar a configuração padrão usada em produção.

Por exemplo, no docker-compose.override.yml, você pode mapear volumes locais para facilitar o desenvolvimento, ou expor portas adicionais para depuração. Essa abordagem mantém o arquivo principal limpo e focado na configuração padrão, enquanto o arquivo de override lida com as necessidades específicas do ambiente de desenvolvimento.

Suponha que, para algum teste, deseja-se expor o serviço de banco de dados configurado em docker-compose.dev.yaml na porta 33060 e também disponibilizar um volume. Isso não deve ser feito no arquivo original, para que o mesmo não seja alterado tendo sua porta exposta e nem que haja volumes.

Para tanto, cria-se um arquivo chamada docker-compose.override.yaml com o seguinte conteúdo:

services:

  db:

    ports:

      - 33060:3306

    volumes:

      - ./.docker/mysql/dbdata:/var/lib/mysql

Repare que possui apenas o que deve conter de diferente ( o que será adicionado ou sobrescrito).

Para testar suba o container docker-compose.dev.yaml:

* + docker compose -f docker-compose.dev.yaml -f docker-compose.override.yaml up

Repare que o parâmetro -f foi utilizado 2 vezes. O arquivo informado em -f mais a direita sobrescreverá as propriedades do da esquerda.

Uma outra forma de rodar o container, sem informar os arquivos, seria assim:

* export COMPOSE\_FILE=docker-compose.dev.yaml:docker-compose.override.yaml
* docker compose up

Uma dica, caso queria ver a configuração final que rodará, pode-se utilizar o seguinte comando:

* docker compose config (caso tenha feito o export COMPOSE\_FILE)
* docker compose -f docker-compose.dev.yaml -f docker-compose.override.yaml config(caso não tenha feito o export COMPOSE\_FILE)



# Docker compose Project name

O "Docker Compose Project Name" é um conceito no Docker Compose que se refere ao nome do projeto que agrupa todos os serviços definidos em um arquivo docker-compose.yml. Por padrão, o nome do projeto é derivado do nome do diretório em que o arquivo docker-compose.yml está localizado. No entanto, você pode especificar um nome de projeto personalizado usando a opção -p ou --project-name ao executar comandos do Docker Compose.

Por exemplo, se você tiver um arquivo docker-compose.yml em um diretório chamado meu\_projeto, o nome do projeto padrão será meu\_projeto. Se você quiser usar um nome diferente, pode executar um comando como docker-compose -p nome\_personalizado up, e o Docker Compose usará nome\_personalizado como o nome do projeto.

O nome do projeto é importante porque é usado como um prefixo para os nomes dos contêineres, redes e outros recursos criados pelo Docker Compose, ajudando a evitar conflitos entre diferentes conjuntos de serviços que podem estar sendo executados simultaneamente no mesmo ambiente.

Imagine que haja a necessidade de rodar os containers em branches diferentes. Isso muitas vezes pode ser problemático. O Project name ajuda bastante neste cenário.

# Comandos do dia a dia

## Como resolver problemas gerais que acontecem no Docker

Certos problemas são comuns de acontecer no Docker, independente de S.O.

No momento de um docker build, docker compose up ou outros, pode ser que não seja executado corretamente.

Caso seja verificado que não tem erro no comando ou no próprio dockerfile, podem ser problemas do próprio docker que está executando. Por exemplo, uma determinada aplicação não consegue achar de forma alguma o banco de dados que o Docker subiu. Mesmo matando o container e rodando de novo, não funciona.

Para estes problemas, uma possível solução é fazer o restart do docker.

* + sudo service docker restart

Muitos problemas já são solucionados desta forma.

Para situações que não resolvem, pode ser necessário reiniciar o S.O. (Caso de Windows utilizando wsl, por exemplo, pode-se fazer um - wsl shutdown – pelo PowerShell.

## Docker compose com e sem o -d

Fazer um docker compose up -d faz com que o terminal fique livre.

Utilizando sem o -d, o terminal continua exibindo os logs do container.

Caso utilize o -d e depois seja necessário ver os logs pode-se fazer um docker compose logs.

## Docker compose run

Com docker compose up, será levantado o container inteiro.

Com docker compose run é possível selecionar apenas algum serviço específico para subir. É possível também passar comnados.

Exemplos:

* docker compose -f docker-compose.dev.yaml run db
* docker compose -f docker-compose.dev.yaml run web1 bash

Um detalhe é que ao utilizar o run, caso o serviço tenha uma dependência, esta será executada primeiro. Outro detalhe é que com run, os logs não são exibidos

## Docker compose build

Com docker compose build é possível fazer build de um determinado serviço.

Por exemplo:

* docker compose -f docker-compose.dev.yaml build web1
* docker compose -f docker-compose.dev.yaml build web1 db

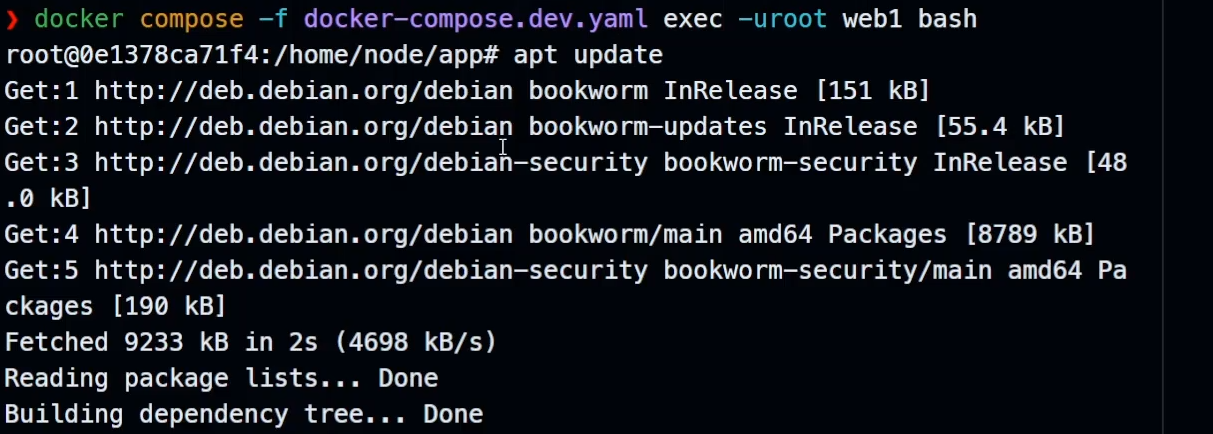
## Logando no container com usuário root

Como boa prática, utiliza-se um usuário específico no Dockerfile, ao invés de utilizar o root.

Imagine um cenário onde seja necessário instalar algo pontual no container em que é necessário o usuário root. Não é produtivo ficar mudando o Dockerfile a todo momento.

Para acessar o container utilizando o usuário root pode-se utilizar ***exec -uroot*** (ou -u root).

Por exemplo: docker compose -f docker-compose.dev.yaml exec -uroot web1 bash



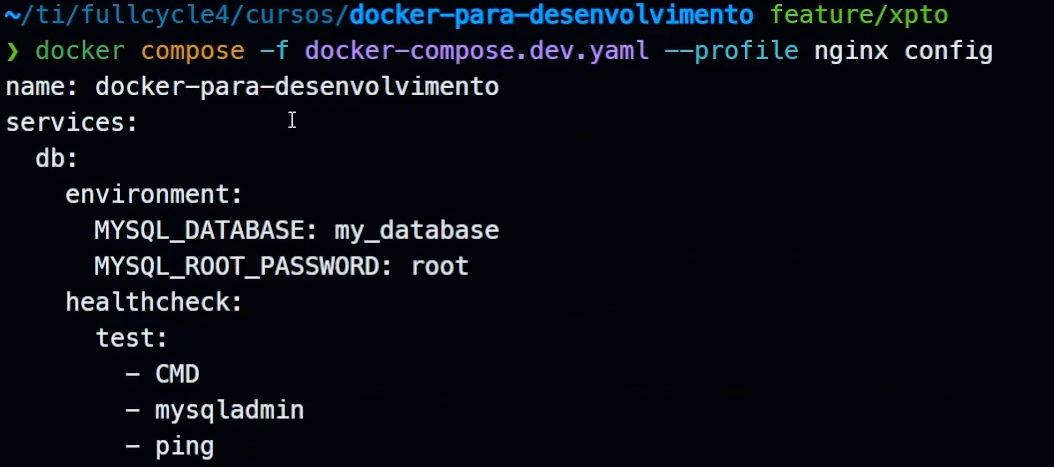
## Docker compose config

Docker compose config é o comando utilizado para exibir como será levantado os containers.

É útil em situações específicas como por exemplo pra visualizar como ficaria para um determinado profile ou então casos em que se utiliza o docker-compose.override.yaml.

Exemplos:

* docker compose -f docker-compose.dev.yaml --profile nginx config



## Variáveis de ambiente pré-definidas no docker compose

As variáveis de ambiente pré-definidas do Docker Compose são utilizadas para facilitar a configuração e a parametrização dos serviços definidos em um arquivo docker-compose.yml. Elas permitem que você personalize o comportamento dos contêineres sem precisar modificar diretamente o arquivo de configuração. Isso é especialmente útil para adaptar a configuração a diferentes ambientes (como desenvolvimento, teste e produção) ou para compartilhar configurações comuns entre diferentes projetos.

Algumas variáveis de ambiente comuns no Docker Compose incluem:

* COMPOSE\_PROJECT\_NAME: Define o nome do projeto, que é usado como prefixo para os nomes dos contêineres, redes e volumes criados pelo Compose.
* COMPOSE\_FILE: Especifica o caminho para um ou mais arquivos docker-compose.yml a serem usados.
* COMPOSE\_PATH\_SEPARATOR: Define o separador de caminho para múltiplos arquivos de configuração no COMPOSE\_FILE.

Essas variáveis ajudam a tornar o uso do Docker Compose mais flexível e adaptável a diferentes cenários.

Link com todas as opções:

[Pre-defined environment variables | Docker Docs](https://docs.docker.com/compose/how-tos/environment-variables/envvars/)

Exemplos de uso:

Ao fazer - export COMPOSE\_FILE=docker-compose.dev.yaml, pode-se então fazer apenas um docker compose up ao invés de docker compose -f docker-compose.dev.yaml up.

A questão é que sempre que abrir um novo terminal, preciará fazer os exports novamente.

### Configurando os exports automaticamente no VSCode

Crie na raiz do projeto uma pasta chamada .vscode. Crie dentro desta pasta o arquivo settings.json com o seguinte conteúdo:

"terminal.integrated.profiles.linux": {

        "dev": {

             "path": "zsh",

             "args": ["-l"],

             "env": {

                 "COMPOSE\_FILE": "docker-compose.dev.yaml"

             }

        }

    }

Ao abrir o terminal do VSCode, o COMPOSE\_FILE já terá sido configurado como variável de ambiente.

### Configurando os exports no arquivo .env

Uma outra opção seria configurar as variáveis de ambientes pré-definidas no próprio arquivo .env.

Por exemplo:

DB\_HOST=db

DB\_USER=root

DB\_PASSWORD=root

B\_NAME=my\_database

COMPOSE\_FILE=docker-compose.dev.yaml:docker-compose.override.yaml

COMPOSE\_PROFILES=debug,nginx

COMPOSE\_PROJECT\_NAME=xpto

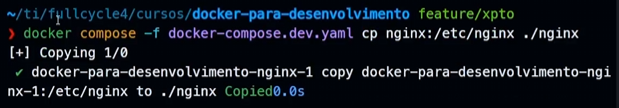
## Copiando arquivos de e para o container

Muitas vezes o container está rodando e acaba sendo necessário copiar alguma coisa específica para dentro do container ou do container para o host.

Para copiar um arquivo de dentro do container para o host:

* docker compose -f docker-compose.dev.yaml cp nginx:/etc/nginx ./nginx

O comando acima copia do serviço nginx a pasta /etc/nginx para a pasta local ./nginx.



Para copiar um arquivo do host para dentro do container:

* docker compose -f docker-compose.dev.yaml cp package.json nginx:/temp

### Copiando arquivos com docker cp

Um forma mais antiga é utilizar o docker cp.

O docker cp é um comando útil no dia a dia para transferir arquivos ou diretórios entre o host e um container. Ele funciona de forma semelhante ao comando cp do Linux, mas com a capacidade de especificar um container como origem ou destino.

**Sintaxe**

docker cp [opções] <container\_id>:<caminho\_origem> <caminho\_destino>

ou

docker cp [opções] <caminho\_origem> <container\_id>:<caminho\_destino>

**Exemplos:**

**Copiar do container para o host:**

docker cp my\_container:/path/in/container /path/on/host

Isso copia o arquivo ou diretório de /path/in/container no container my\_container para /path/on/host no host.

**Copiar do host para o container:**

docker cp /path/on/host my\_container:/path/in/container

Isso copia o arquivo ou diretório de /path/on/host no host para /path/in/container no container my\_container.