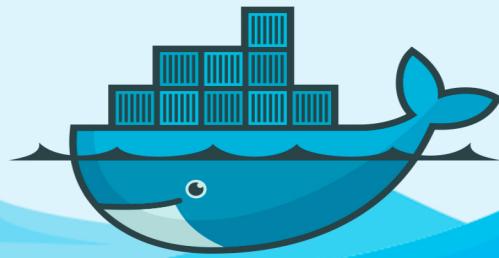


# FullCycle



# docker

E DOCKER COMPOSE NA PRÁTICA

**GUIA RÁPIDO**



# **Guia Rápido: Docker e Docker Compose na Prática**

Full Cycle

Esse livro está à venda em <http://leanpub.com/docker-e-docker-compose-na-pratica>

Essa versão foi publicada em 2023-04-11



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2019 - 2023 Full Cycle

# Conteúdo

Introdução . . . . .	1
O que é Docker? . . . . .	2
Mãos à obra: Projeto Laravel . . . . .	4
Por que utilizar o Docker? . . . . .	5
Primeiros passos com Docker . . . . .	6
Dockerfile e Docker Compose . . . . .	10
Dockerfile . . . . .	10
Docker-compose . . . . .	11
Volumes . . . . .	15
Conectando ao Banco de Dados . . . . .	18
Declarções Adicionais no Dockerfile . . . . .	25
WORKDIR . . . . .	25
RUN . . . . .	25
COPY . . . . .	25
ADD . . . . .	26
CMD . . . . .	26
Build da imagem no Docker Hub . . . . .	27
Conclusão . . . . .	29

# Introdução

Durante muito tempo, tarefas como configurar um ótimo ambiente de desenvolvimento foram consideradas um grande desafio para desenvolvedores de todos os níveis. Softwares, compiladores, IDEs, banco de dados e principalmente a compatibilidade com os ambientes de produção faziam com que os profissionais tivessem a produtividade reduzida gerando preocupações adicionais além as de produzir um bom código.

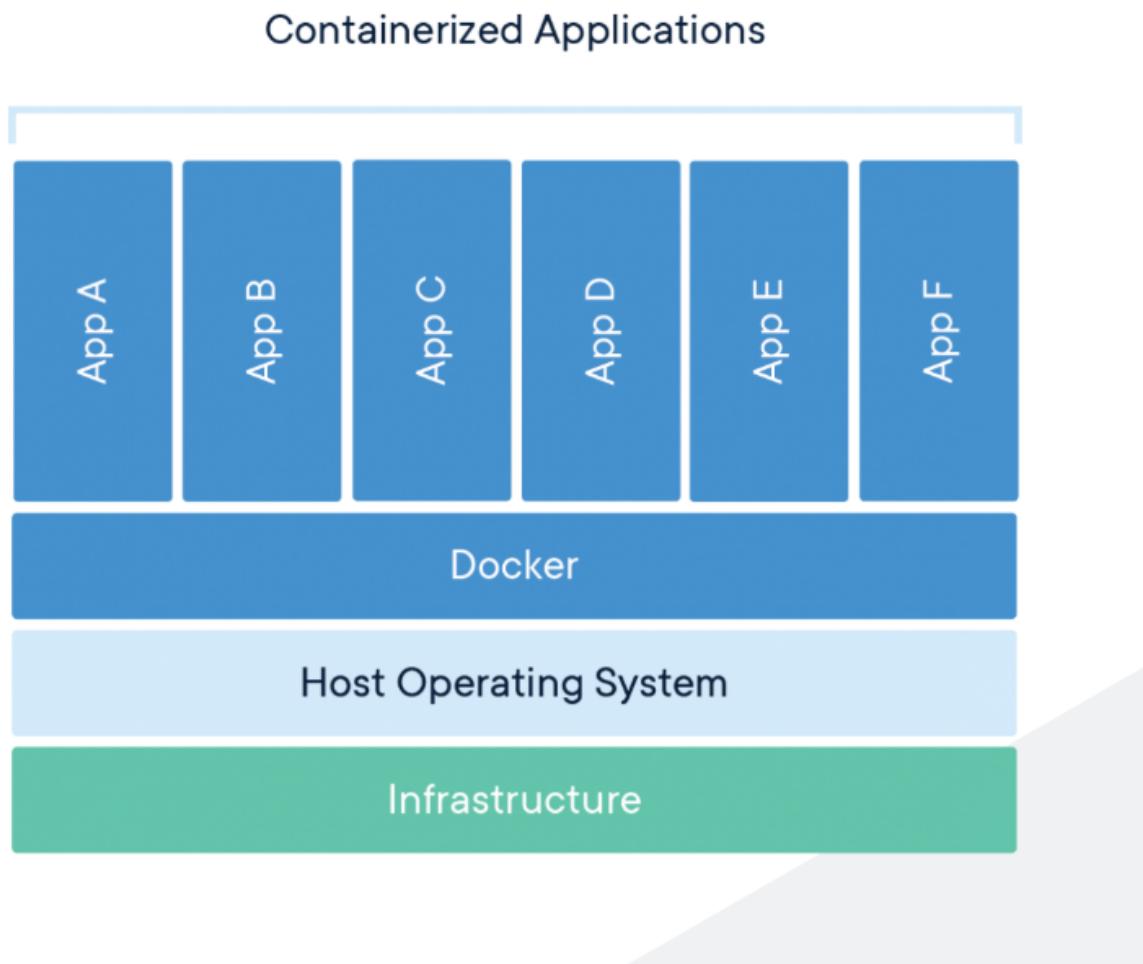
Nesse guia rápido, trataremos essencialmente do Docker, uma ferramenta que definitivamente mudou completamente a forma que todos nós desenvolvedores trabalhamos. Através de isolamento de namespaces, utilização de cgroups e um sistema de arquivos baseado em camadas, o Docker possibilita que subamos em questão de segundos um ambiente completo e idêntico ao de produção.

Vamos então colocar a mão na massa.

# O que é Docker?

Docker é uma plataforma Open Source que possibilita o empacotamento de uma aplicação dentro de um container. Uma aplicação consegue se adequar e rodar em qualquer máquina que tenha essa tecnologia instalada.

De acordo com o [www.docker.com](http://www.docker.com), um container é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de maneira rápida e confiável de um ambiente de computação para outro. Uma imagem de container do Docker é um pacote de software leve, autônomo e executável que inclui tudo o que é necessário para executar um aplicativo: código, tempo de execução, ferramentas do sistema, bibliotecas do sistema e configurações.



Disponível para aplicativos baseados em Linux e Windows, o software conteinerizado sempre será executado da mesma maneira, independentemente da infraestrutura. Os containeres isolam o software de seu ambiente e garantem que ele funcione de maneira uniforme, apesar das diferenças, por exemplo, entre desenvolvimento e preparação.

O Docker, especificamente, foi feito para ser trabalhado com o Linux, porém, é totalmente possível trabalharmos com o Docker no MAC e no Windows, já que o MAC possui um sistema de virtualização próprio e o Windows, pelo menos na versão Professional, possui o Hyper V.

No Windows, temos duas opções para rodar o Docker: se você possuir o Windows na versão Professional, basta ativar o Hyper V, em outras versões, porém, essa opção de ativação do Hyper V não está disponível, dessa forma, portanto, deveremos utilizar Docker Toolbox. Resumidamente, o Docker Toolbox sobe uma máquina virtual em seu sistema, instala uma versão do Linux e o seu Docker irá “conversar” com essa máquina virtual.

Não iremos abordar a instalação do Docker, pois podemos partir do pressuposto de que somos todos desenvolvedores e que nós todos temos a capacidade de instalar o Docker.

Vamos resolver um problema pontual de muitos desenvolvedores: Configuraremos um ambiente de desenvolvimento utilizando o Docker e o Docker Compose. Para isso, utilizaremos como exemplo o setup de uma aplicação baseada no framework PHP Laravel.

# Mãos à obra: Projeto Laravel

Para iniciar o processo, criaremos um simples projeto Laravel. Esse projeto será chamado de laravel-docker. Para isso, faça o download do “composer” (gerenciador de pacotes do PHP. Para isso acesse: getcomposer.org).

Em seu terminal digite o comando:

```
composer create-project --prefer-dist laravel/laravel laravel-docker
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos
$ composer create-project --prefer-dist laravel/laravel laravel-docker
```

O Laravel será instalado em sua máquina e em seguida será criado o projeto laravel-docker. (esse processo poderá levar alguns minutos).

Após a instalação, acessaremos o projeto:

```
cd laravel-docker
ls
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos
$ cd laravel-docker

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ ls
app/      composer.json  database/    public/     routes/     tests/
artisan*   composer.lock  package.json  readme.md   server.php  vendor/
bootstrap/ config/       phpunit.xml  resources/  storage/   webpack.mix.js

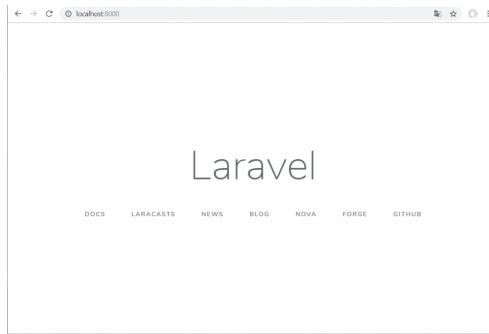
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ |
```

Note que o projeto já está rodando em nossa máquina. Se dermos o comando a seguir, poderemos comprovar isso:

```
php artisan serve
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ php artisan serve
Laravel development server started: <http://127.0.0.1:8000>
[Tue Jul 23 16:49:57 2019] 127.0.0.1:56384 [200]: /favicon.ico
[Tue Jul 23 16:50:03 2019] 127.0.0.1:56387 [200]: /favicon.ico
```

Acessando o localmente de nosso navegador, poderemos ver a aplicação rodando:



## Por que utilizar o Docker?

Nesse momento você deve estar se perguntando: “Mas é tão fácil criar um projeto Laravel, localmente, pra que eu preciso do Docker”

A resposta é bem simples: Para garantir que outros desenvolvedores possam trabalhar exatamente no mesmo ambiente que você, independente da máquina utilizada. Todos terão os mesmos recursos, por exemplo, mesma versão do PHP, MySQL, recursos computacionais e etc.

Com isso, aquela história de “Na minha máquina funciona”, com certeza vai acabar.

# Primeiros passos com Docker

Agora que já temos nosso projeto instalado, vamos checar rapidamente se o Docker já está disponível em nosso ambiente, caso não, faça a instalação em: [www.docker.com](http://www.docker.com).

Digite o comando `docker` em seu terminal. Você poderá notar que uma lista de outros comandos aparecerão. Isso significa que ele está corretamente instalado:

```
$ docker
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker

Usage: docker [OPTIONS] COMMAND
A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "C:\\Users\\rafae\\.docker")
  -D, --debug          Enable debug mode
  -H, --host list      Daemon socket(s) to connect to
  -l, --log-level string
                       Set the logging level
                       ("debug"|"info"|"warn"|"error"|"fatal")
                       (default "info")
  --tls                Use TLS; implied by --tlsverify
  --tlscacert string   Trust certs signed only by this CA (default "C:\\Users\\rafae\\.docker\\machine\\machines\\default\\ca.pem")
  --tlscert string     Path to TLS certificate file (default "C:\\Users\\rafae\\.docker\\machine\\machines\\default\\cert.pem")
  --tlskey string       Path to TLS key file (default "C:\\Users\\rafae\\.docker\\machine\\machines\\default\\key.pem")
  --tlsverify          Use TLS and verify the remote (default true)
  -v, --version         Print version information and quit

Management Commands:
  builder    Manage builds
  config     Manage Docker configs
  container  Manage containers
  image      Manage images
  network   Manage networks
  node       Manage Swarm nodes
  plugin    Manage plugins
  secret     Manage Docker secrets
  service   Manage services
  stack     Manage Docker stacks
  swarm     Manage Swarm
  system    Manage Docker
  trust     Manage trust on Docker images
  volume   Manage volumes

Commands:
  attach     Attach local standard input, output, and error streams to a running container
  build      Build an image from a Dockerfile
  commit    Create a new image from a container's changes
  cp        Copy files/folders between a container and the local filesystem
  create    Create a new container
  deploy    Deploy a new stack or update an existing stack
  diff      Inspect changes to files or directories on a container's filesystem
  events    Get real time events from the server
  exec      Run a command in a running container
  export   Export a container's filesystem as a tar archive
  history  Show the history of an image
  images   List images
  import   Import the contents from a tarball to create a filesystem image
  info     Display system-wide information
  inspect  Return low-level information on Docker objects
  kill     Kill one or more running containers
  load     Load an image from a tar archive or STDIN
  login    Log in to a Docker registry
  logout   Log out from a Docker registry
  logs     Fetch the logs of a container
  pause    Pause all processes within one or more containers
  port     List port mappings or a specific mapping for the container
  ps       List containers
  pull     Pull an image or a repository from a registry
  push     Push an image or a repository to a registry
  rename   Rename a container
  restart  Restart one or more containers
  rm      Remove one or more containers
  rmi     Remove one or more images
  run     Run a command in a new container
  save    Save one or more images to a tar archive (streamed to STDOUT by default)
  search   Search the Docker Hub for images
  start    Start one or more stopped containers
  stats   Display a live stream of container(s) resource usage statistics
  stop     Stop one or more running containers
  tag      Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top     Display the running processes of a container
  unpause  Unpause all processes within one or more containers
  update   Update configuration of one or more containers
  version  Show the Docker version information
  wait    Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ |
```

## Fixando alguns conceitos básicos do Docker: Containeres e Imagens.

**Container:** é o local onde a sua aplicação ficará rodando.

**Imagen:** É como um snapshot. Outros desenvolvedores com acesso a esta imagem, terão os mesmos recursos que você utiliza e configurou em seu container.

Para que você possa trabalhar com o Docker, é extremamente necessário que você conheça seus principais comandos. Vamos lá!

O primeiro comando é o `docker ps`, esse comando irá lhe mostrar quais os containers que foram criados e estão rodando:

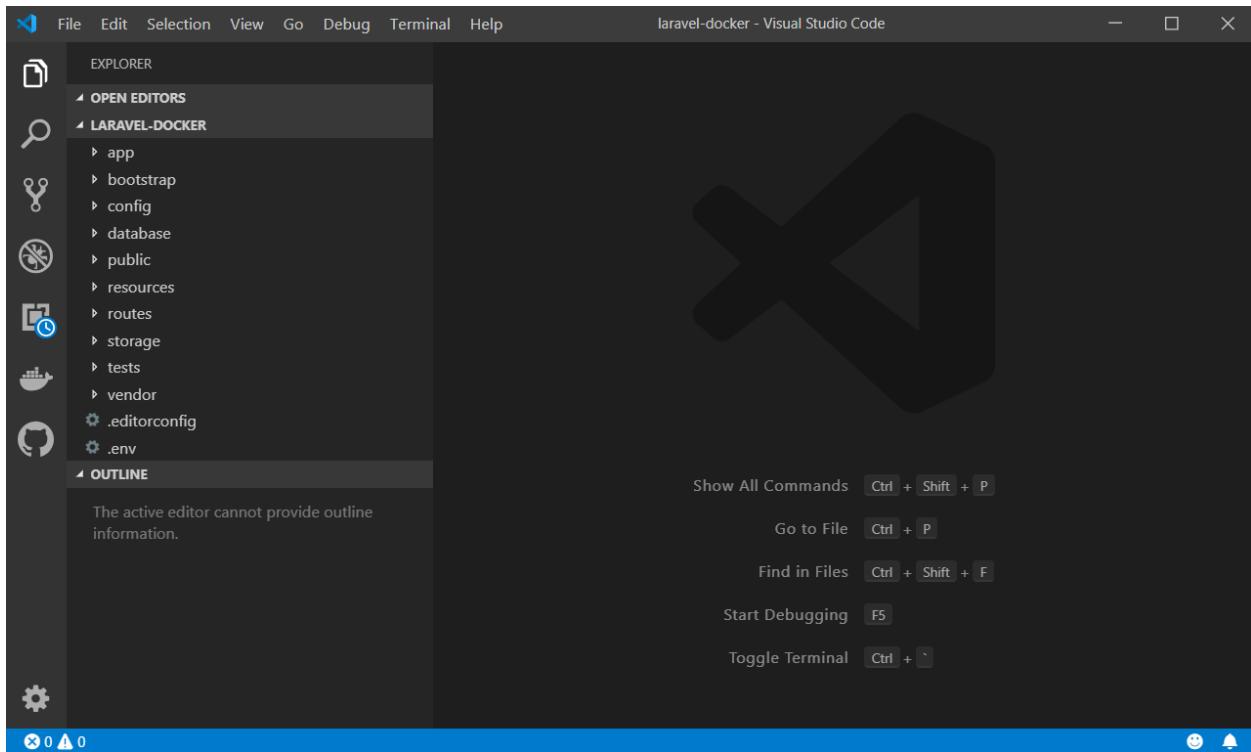
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker	CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAME
S							

Um segundo comando, muito importante, é o `docker images`, esse comando mostra quais imagens foram criadas:

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker	REPOSITORY	TAG	IMAGE ID	CREATED	SIZE

Note que ainda não criamos nenhum container e, consequentemente nenhuma imagem, por isso nenhuma informação foi apresentada.

Vamos abrir esse projeto utilizando nosso editor de código. Nesse caso, trabalharemos com o Visual Studio Code.



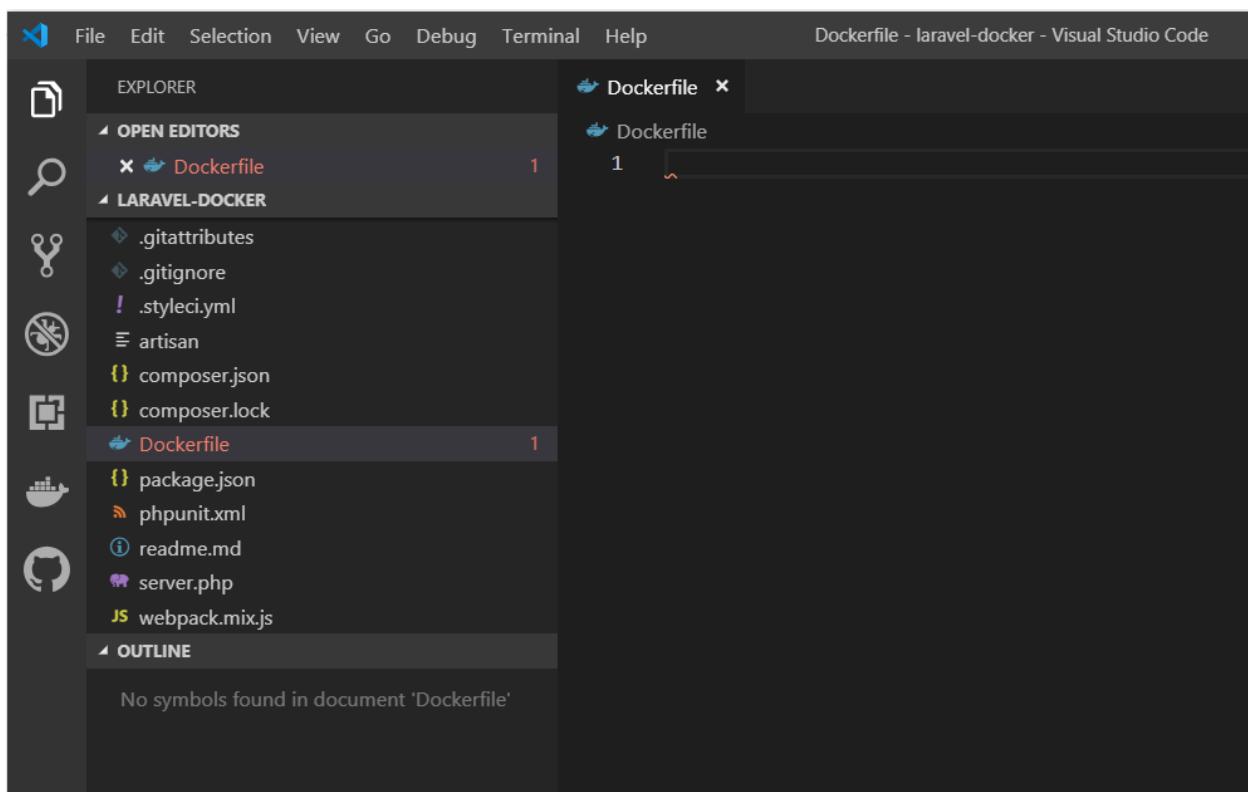
# Dockerfile e Docker Compose

## Dockerfile

Durante essa nossa jornada, trabalharemos com um arquivo chamado de Dockerfile.

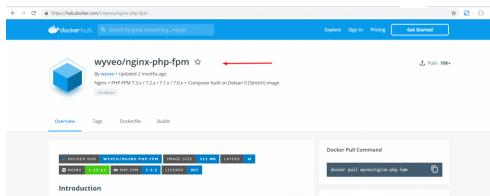
“O Dockerfile é um arquivo de texto que contém as instruções necessárias para criar uma nova imagem de contêiner. Essas instruções incluem a identificação de uma imagem existente a ser usada como uma base, comandos a serem executados durante o processo de criação da imagem e um comando que será executado quando novas instâncias da imagem de contêiner forem implantadas.”  
(Fonte: [www.docker.com](http://www.docker.com))

Vamos então criar um arquivo chamado Dockerfile.



Por padrão, utilizaremos uma imagem Docker que nos trará o PHP-FPM e o Nginx já configurados. Você poderá encontrar essa imagem acessando <https://hub.docker.com><sup>1</sup> e pesquisando por **nginx php fpm**:

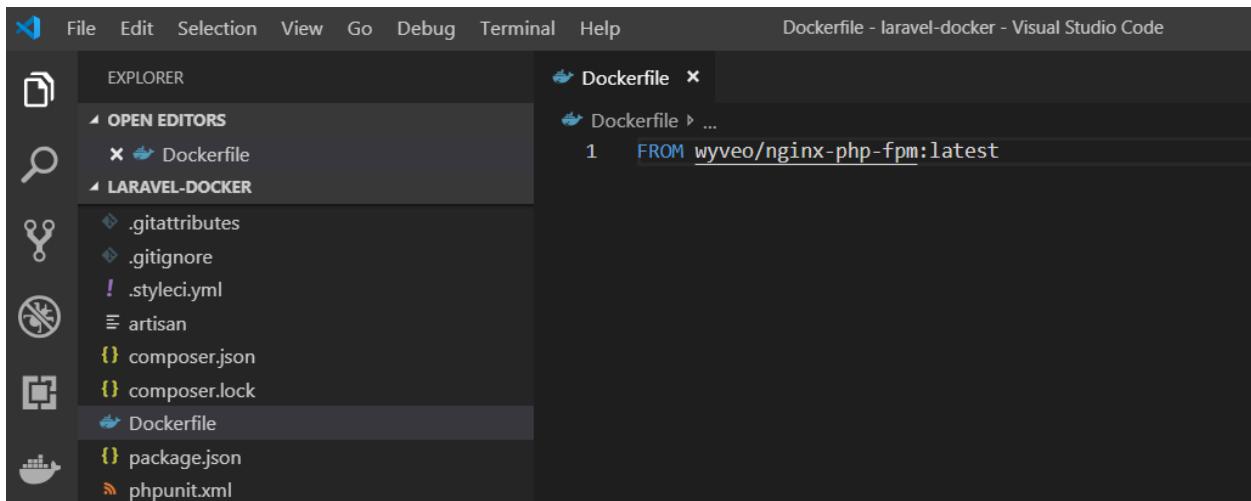
<sup>1</sup><https://hub.docker.com>



Em nossa IDE, no arquivo *Dockerfile* que criamos, inserimos o seguinte código:

```
1 FROM wyveo/nginx-php-fpm:latest
```

A instrução `FROM` define a imagem de container que será usada durante o processo de criação de nova imagem, `wyveo/nginx-php-fpm` é o endereço da imagem e `latest` indica que queremos a versão mais atual dessa imagem.



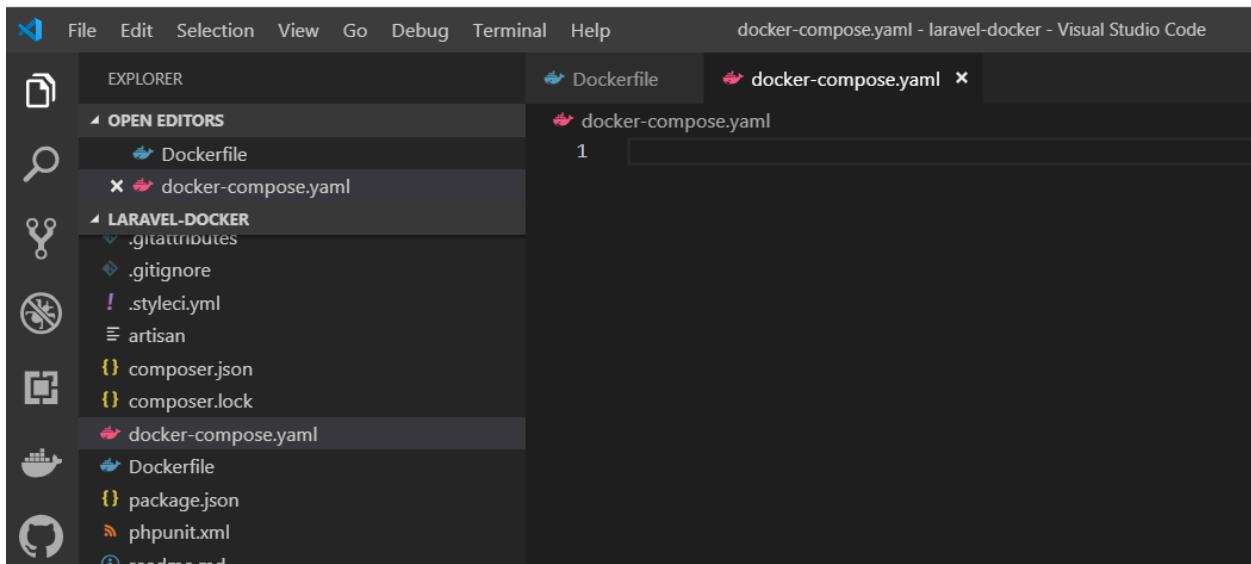
É necessário explicar que esse código traz uma instalação crua do nginx com PHP-FPM. Por padrão, o nginx, quando instalado dessa forma, mantém seu Document Root no seguinte caminho:

```
1 /usr/share/nginx/html
```

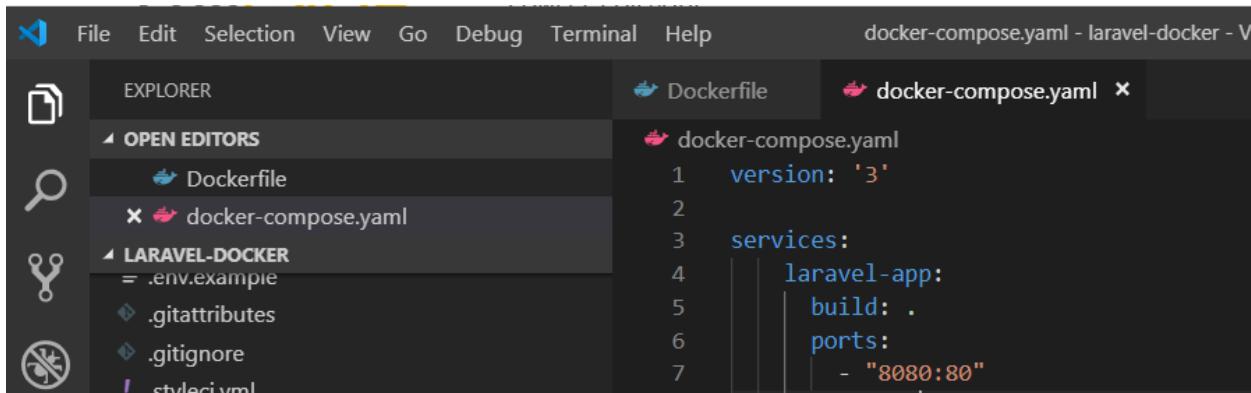
## Docker-compose

O docker-compose é uma ferramenta do Docker que, a partir de diversas especificações, permite subir diversos containeres e relacioná-los através de redes internas.

Para isso, vamos iniciar criando um arquivo chamado de `docker-compose.yaml`.



No arquivo *docker-compose*, declaramos a seguinte estrutura:



**version**: declara a versão do docker compose

**services**: declara quais serviços serão rodados, nesse caso, chamaremos de laravel-app.

**build**: declara o nome da imagem, ou, no caso, se declararmos o .., ele irá “chamar” a imagem declarada no Dockerfile.

**ports**: realiza a liberação das portas. Nesse exemplo, queremos que seja liberada a porta 8080, porém, quando acessada, seja feito um redirecionamento para a porta 80 de nosso container. Logo, toda vez que acessarmos o localhost com a porta 8080 o Docker redirecionará a requisição para a porta 80 do nginx criado no container.

Acessando o nosso terminal, subiremos o container com o comando:

```
$ docker-compose up -d
```

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker-compose up -d
```

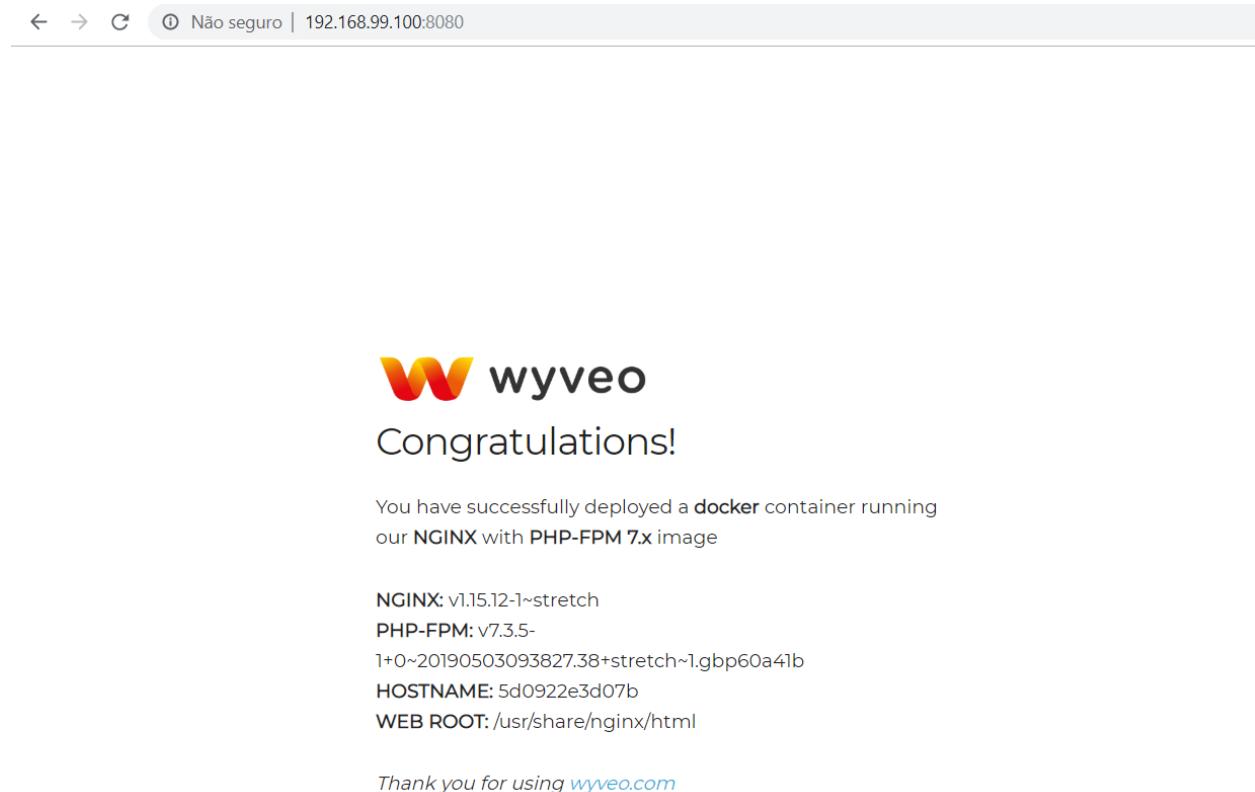
Note que após o comando, será apresentada as seguintes informações:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker-compose up -d
Building laravel-app
Step 1/1 : FROM wyveo/nginx-php-fpm:latest
--> 9536a4b2f181
Successfully built 9536a4b2f181
Successfully tagged laravel-docker_laravel-app:latest
Image for service laravel-app was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
laravel-docker_laravel-app_1 is up-to-date.

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ |
```

O docker-compose up irá rodar o docker-compose, baseado em nosso *docker-compose.yaml* e com o -d o container é inicializado em segundo plano e podemos utilizar o nosso terminal para outros comandos.

Acessamos o nosso browser e já podemos ver o nosso nginx rodando:



Lembando que, em nosso *docker-compose.yaml*, indicamos que acessaremos a porta 8080 de nossa máquina e essa acessará a porta 80 do nosso container.

Nosso container rodando:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS
PORTS              NAMES
5d0922e3d07b        laravel-docker_laravel-app   "/start.sh"   3 days ago      Up 3 days
          0.0.0.0:8080->80/tcp    laravel-docker_laravel-app_1
```

# Volumes

O Docker possui um mecanismo de gerenciamento de volumes que com ele é possível compartilharmos um volume da nossa máquina com o container.

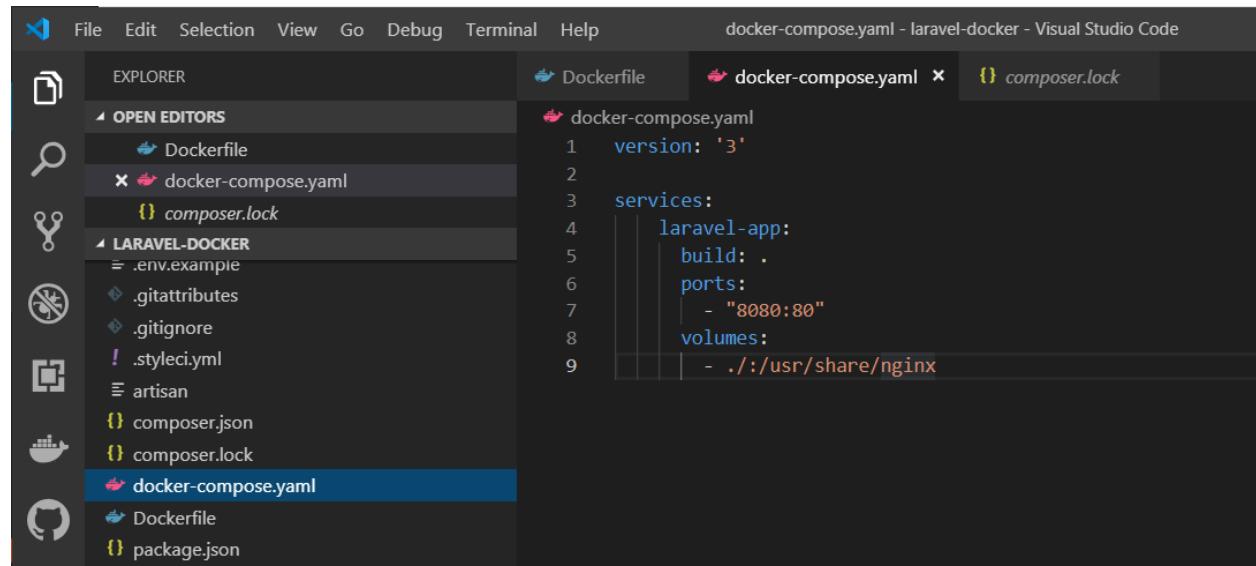
Em nosso *docker-compose.yaml* iremos declarar

```
1 volumes:  
2   - ./:/usr/share/nginx
```

Quando criamos esse volume, tudo o que estiver em nossa pasta será montado dentro do nosso container, ou seja, tudo o que modificarmos em nossa pasta será compartilhado com o container, porém, caso “matarmos” o container, ainda teremos os arquivos em nossa máquina.

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker  
$ ls  
app/           composer.lock      Dockerfile     readme.md    storage/  
artisan*        config/          package.json  resources/   tests/  
bootstrap/      database/       phpunit.xml  routes/     vendor/  
composer.json   docker-compose.yaml public/      server.php  webpack.mix.js
```

E assim fica o nosso *docker-compose.yaml*:



## Vamos testar?

Com o comando docker-compose up -d --build, subiremos as modificações realizadas em nosso *docker-compose.yaml*:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker-compose up -d --build
Building laravel-app
Step 1/1 : FROM wyveo/nginx-php-fpm:latest
--> 9536a4b2f181
Successfully built 9536a4b2f181
Successfully tagged laravel-docker_laravel-app:latest
laravel-docker_laravel-app_1 is up-to-date

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$
```

Para evitar que tenhamos um erro 404 no nginx (pois ele está buscando por padrão a pasta html), criaremos um link simbólico apontando a pasta public de nosso projeto Laravel para html.

---

### 404 Not Found

---

nginx

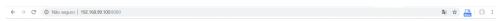
Esse link simbólico é criado rodando: ln -s public html em seu terminal. O link simbólico, na verdade, funciona como um atalho, pois toda vez que acessarmos a pasta html na verdade estaremos acessando a pasta public:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ ln -s public html
```

Note a pasta html:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ ls
app/      composer.json  database/    html/       public/     routes/    tests/
artisan*   composer.lock  docker-compose.yaml  package.json  readme.md  server.php  vendor/
bootstrap/ config/       Dockerfile    phpunit.xml  resources/ storage/  webpack.mix.js
```

Agora, finalmente podemos visualizar a imagem do Laravel sendo executada.

A screenshot of a web browser window. The address bar at the top shows the URL "http://127.0.0.1:8000". The main content area of the browser is completely blank, displaying only the "Laravel" logo centered on the page.

Laravel

HOME LARAVELISTS NEWS DIVE NEWS FEATURES

# Conectando ao Banco de Dados

Agora que já temos o nosso Laravel rodando, iremos realizar algumas declarações em nosso *docker-compose.yaml*.

Criaremos um serviço chamado:

```
1 mysql-app:
```

Nesse serviço vamos declarar que utilizaremos uma imagem do MySQL. Essa imagem pode ser facilmente encontrada no <https://hub.docker.com><sup>2</sup>.

```
1 image: mysql:5.7.22
```

O nosso MySQL também utilizará portas:

```
1 ports:
2   - "3306:3306"
```

Com essa declaração estamos dizendo que, tanto a porta de nossa máquina quanto a porta de nosso container serão as mesmas.

A imagem do MySQL foi preparada para que possamos trabalhar com variáveis de ambiente. Utilizaremos uma variável de ambiente que cria o banco de dados com uma senha, facilitando todo o trabalho.

```
1 environment:
2   MYSQL_DATABASE: laravel
3   MYSQL_ROOT_PASSWORD: laravel
```

Dando prosseguimento, para que essas máquinas possam conversar entre si, é necessário que compartilhemos uma rede interna:

```
1 networks:
2   -app-network
```

Esse serviço deve ser criado em nosso laravel-app e em nosso mysql-app.

Declaramos, também, fora dos serviços a criação da rede propriamente dita:

---

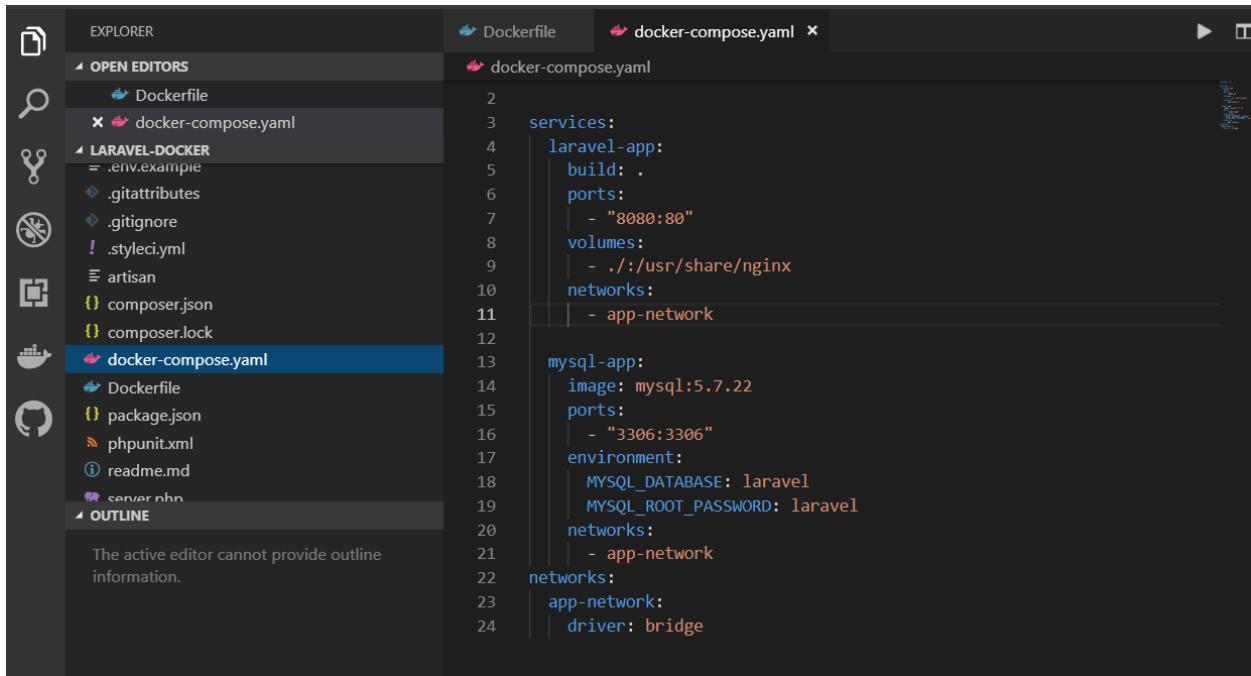
<sup>2</sup><https://hub.docker.com>

```

1 networks:
2   app-network:
3     driver: bridge

```

Esse é o resultado final.



```

EXPLORER Dockerfile docker-compose.yaml
OPEN EDITORS Dockerfile docker-compose.yaml
LARAVEL-DOCKER .env.example .gitattributes .gitignore .styleci.yml artisan composer.json composer.lock
docker-compose.yaml Dockerfile package.json phpunit.xml readme.md server.php
OUTLINE

The active editor cannot provide outline information.

Dockerfile
services:
  laravel-app:
    build: .
    ports:
      - "8080:80"
    volumes:
      - ./:/usr/share/nginx
    networks:
      - app-network

mysql-app:
  image: mysql:5.7.22
  ports:
    - "3306:3306"
  environment:
    MYSQL_DATABASE: laravel
    MYSQL_ROOT_PASSWORD: laravel
  networks:
    - app-network
  networks:
    app-network:
      driver: bridge

```

ide\_networks

Pronto para testar?

Primeiro, vamos derrubar nosso container:

```

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker-compose down
Removing laravel-docker_laravel-app_1 ... done
Removing network laravel-docker_app-network
Network laravel-docker_app-network not found.

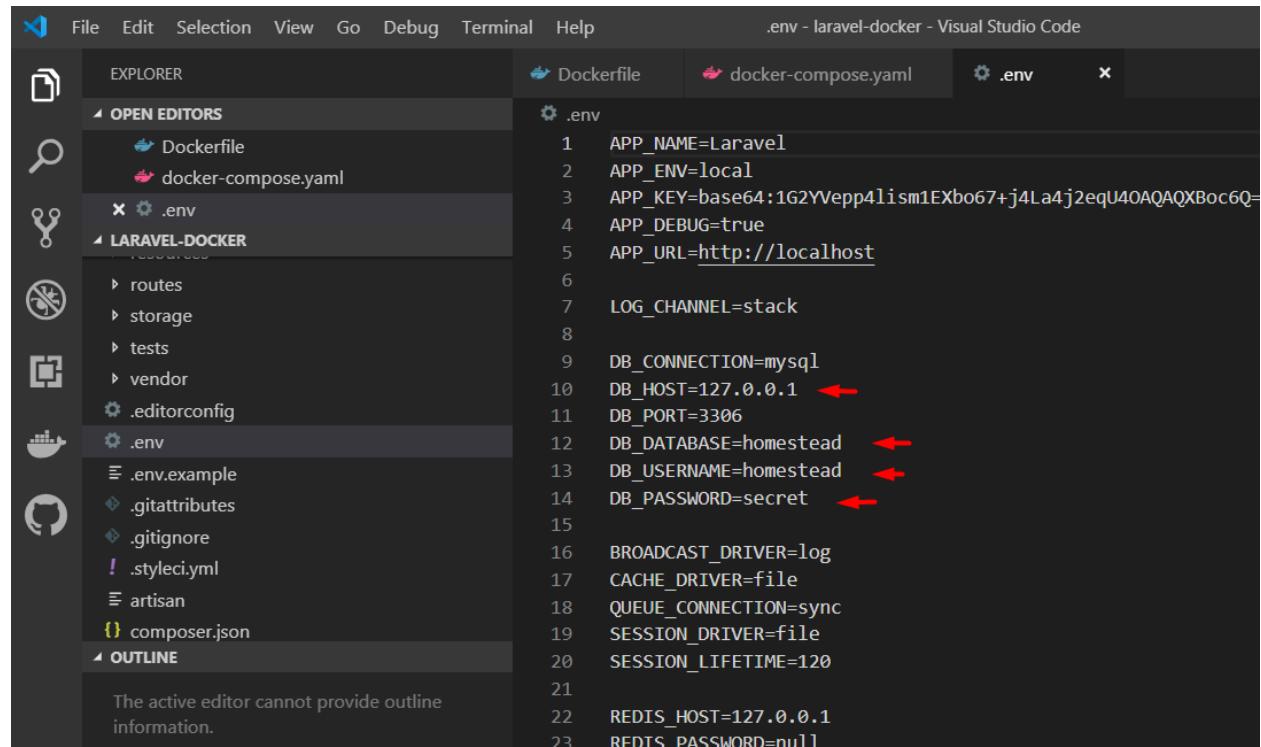
```

Em seguida, subimos novamente:

```
Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ docker-compose up -d --build
Creating network "laravel-docker_app-network" with driver "bridge"
Building laravel-app
Step 1/1 : FROM wyveo/nginx-php-fpm:latest
--> 9536a4b2f181
Successfully built 9536a4b2f181
Successfully tagged laravel-docker_laravel-app:latest
Pulling mysql-app (mysql:5.7.22)...
5.7.22: Pulling from library/mysql
Creating laravel-docker_mysql-app_1 ... done
Creating laravel-docker_laravel-app_1 ... done

Rafael@DESKTOP-51JQ04R MINGW64 ~/Documents/Projetos/laravel-docker
$ |
```

Agora, para que o Laravel possa se conectar com o container do MySQL, vamos rapidamente editar o arquivo .env de nosso projeto. Ele é responsável por todas as configurações desse framework. Nesse caso, vamos alterar as credenciais de acesso ao banco de dados de acordo com o que foi informado no docker-compose.yaml:



Iremos alterar o nome de nossa conexão:

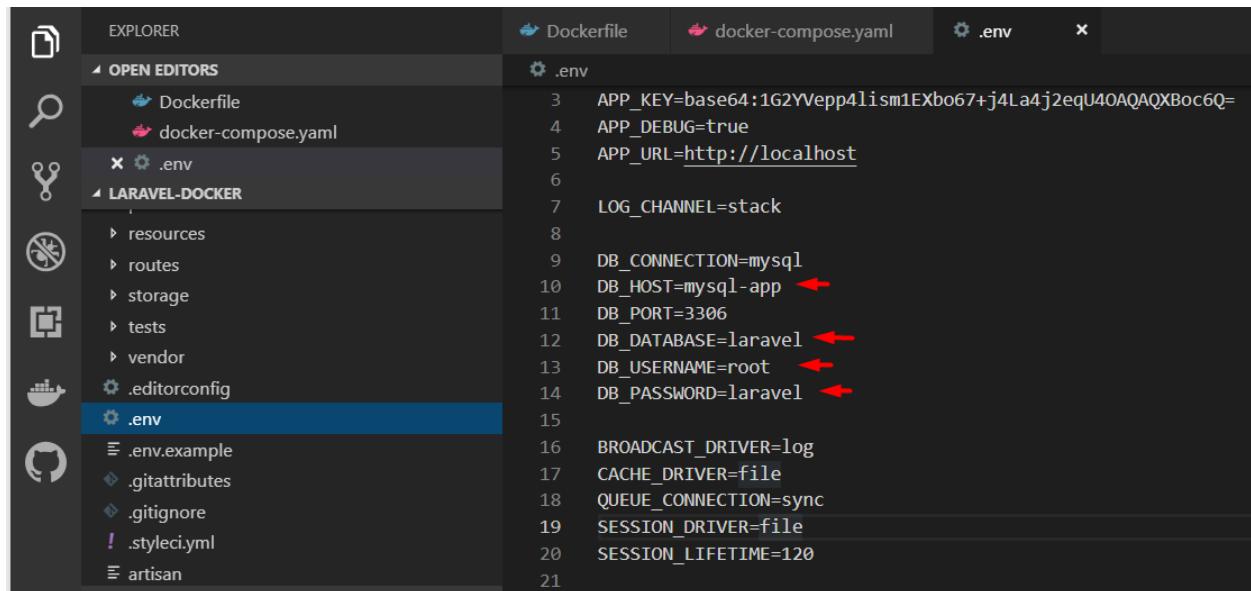
```
1 DB_HOST=127.0.0.1
```

Deveremos alterar para termos o mesmo nome de nossa conexão, ou seja:

```
1 DB_HOST=mysql-app
```

Alteramos, também:

```
1 DB_DATABASE=homestead para DB_DATABASE=laravel
2 DB_USERNAME=homestead para DB_USERNAME=root
3 DB_PASSWORD=secret para DB_PASSWORD=laravel
```



Vamos testar a conexão:

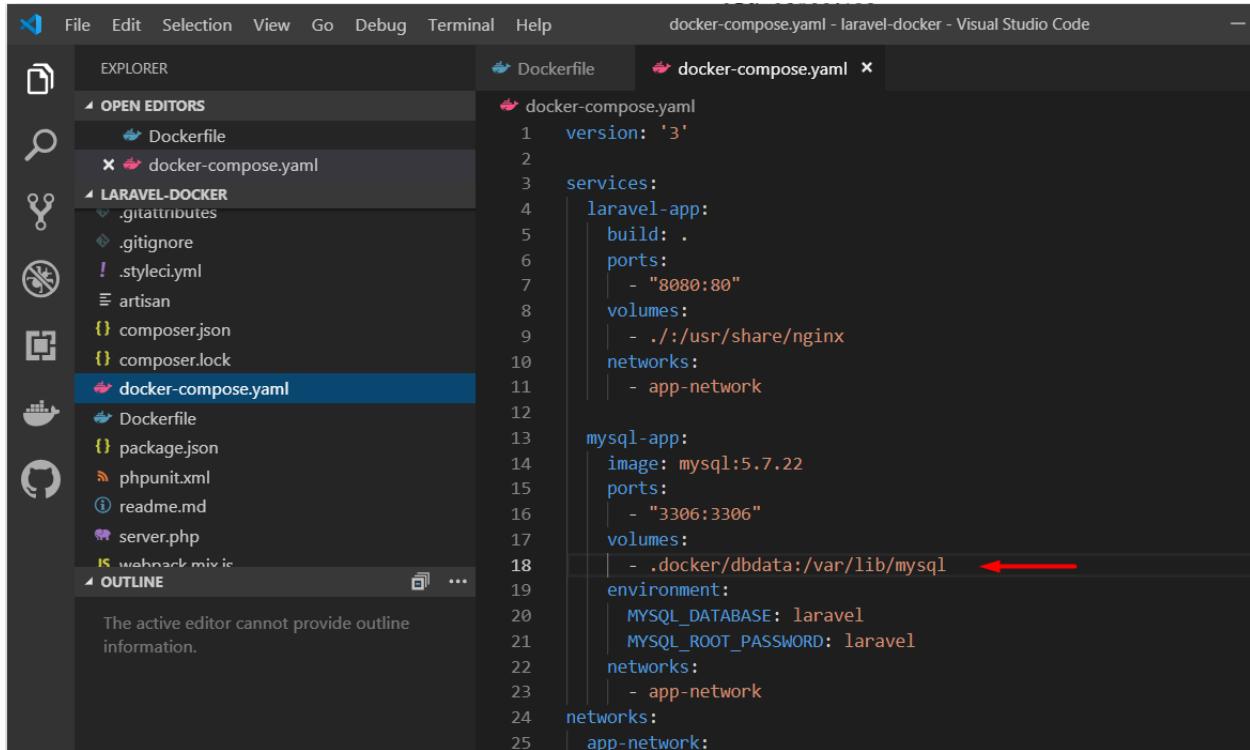
```
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker exec -it laravel-docker_laravel-app_1 bash
root@31907fee7c7f:/# cd /usr/share/nginx
root@31907fee7c7f:/usr/share/nginx# php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.06 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.01 seconds)
root@31907fee7c7f:/usr/share/nginx# |
```

Agora, deveremos criar o volume no mysql-app, para que o nosso banco não seja perdido caso “matarmos” o nosso container:

Criaremos uma pasta oculta chamada `.docker` quando declararmos o nosso volume:

```
1 volumes:  
2   - .docker/dbdata:/var/lib/mysql
```

Na prática, ficará assim em nossa IDE:



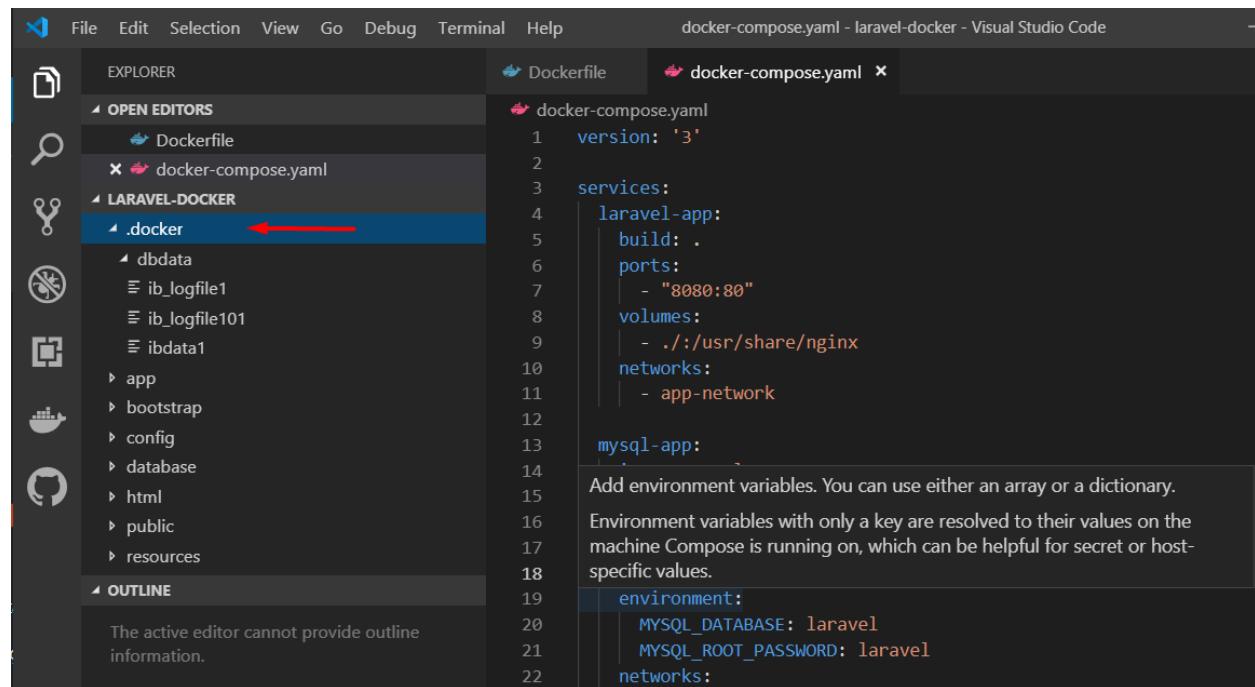
```
File Edit Selection View Go Debug Terminal Help docker-compose.yaml - laravel-docker - Visual Studio Code  
EXPLORER Dockerfile docker-compose.yaml  
OPEN EDITORS Dockerfile docker-compose.yaml  
LARAVEL-DOCKER .gitattributes .gitignore .styleci.yml artisan composer.json composer.lock docker-compose.yaml Dockerfile package.json phpunit.xml readme.md server.php webpack.mix.js  
OUTLINE The active editor cannot provide outline information.  
version: '3'  
services:  
  laravel-app:  
    build: .  
    ports:  
      - "8080:80"  
    volumes:  
      - ./:/usr/share/nginx  
    networks:  
      - app-network  
  
  mysql-app:  
    image: mysql:5.7.22  
    ports:  
      - "3306:3306"  
    volumes:  
      - .docker/dbdata:/var/lib/mysql ←  
    environment:  
      MYSQL_DATABASE: laravel  
      MYSQL_ROOT_PASSWORD: laravel  
    networks:  
      - app-network  
  networks:  
    app-network:  
      app-network:
```

volume\_mysql

Se paramos o nosso container e depois subirmos novamente, poderemos verificar que um novo arquivo foi adicionado em nossa IDE:

```
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker-compose down
Stopping laravel-docker_laravel-app_1 ... done
Stopping laravel-docker_mysql-app_1 ... done
Removing laravel-docker_laravel-app_1 ... done
Removing laravel-docker_mysql-app_1 ... done
Removing network laravel-docker_app-network
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker-compose up -d --build
Creating network "laravel-docker_app-network" with driver "bridge"
Building laravel-app
Step 1/1 : FROM wyveo/nginx-php-fpm:latest
--> 9536a4b2f181
Successfully built 9536a4b2f181
Successfully tagged laravel-docker_laravel-app:latest
Creating laravel-docker_laravel-app_1 ... done
Creating laravel-docker_mysql-app_1 ... done
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ |
```

Novo arquivo adicionado:

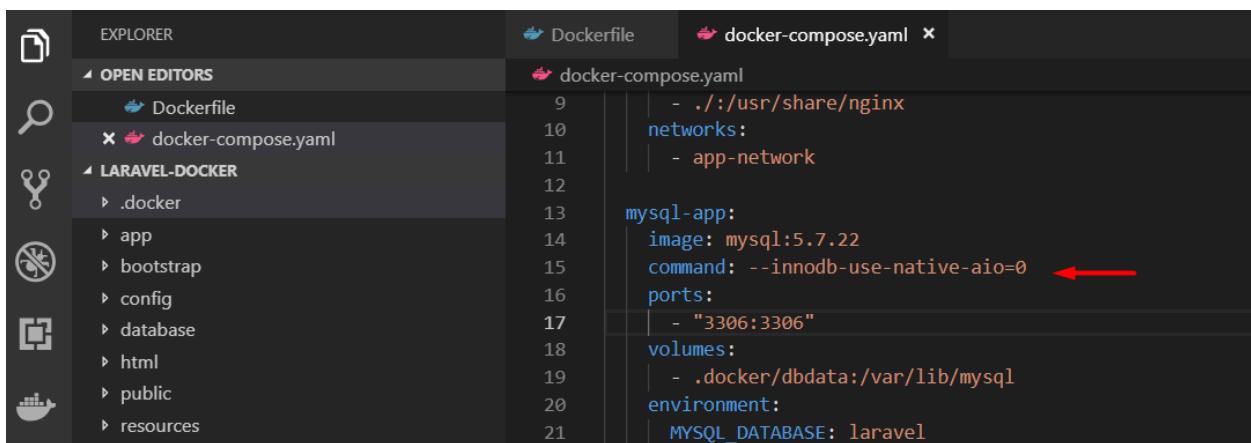


Se dermos novamente um `php artisan migrate`, aparecerá a informação de que não temos novos

arquivos, pois já estarão salvos em nossa máquina.

```
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker exec -it laravel-docker_laravel-app_1 bash
root@22d4b57f0902:/# cd /usr/share/nginx
root@22d4b57f0902:/usr/share/nginx# php artisan migrate
Nothing to migrate.
root@22d4b57f0902:/usr/share/nginx# |
```

É importante explicar que, caso esteja utilizando o Windows, recomenda-se uma declaração específica em nosso *docker-compose.yaml*:



```
EXPLORER Dockerfile docker-compose.yaml
OPEN EDITORS Dockerfile docker-compose.yaml
LARAVEL-DOCKER .docker app bootstrap config database html public resources
Dockerfile
9  | - ./:/usr/share/nginx
10 | networks:
11 |   - app-network
12 |
13 mysql-app:
14   image: mysql:5.7.22
15   command: --innodb-use-native-aio=0 ←
16   ports:
17     - "3306:3306"
18   volumes:
19     - .docker/dbdata:/var/lib/mysql
20   environment:
21     MYSQL_DATABASE: laravel
```

Sem essa declaração, para usuários do Windows, provavelmente o volume do nosso *mysql-app* não será montado da forma correta.

Agora, nossa aplicação Laravel já está pronta e totalmente disponível para desenvolvermos o nosso projeto.

# Declarações Adicionais no Dockerfile

## WORKDIR

“A instrução WORKDIR define um diretório de trabalho para outras instruções Dockerfile, como RUN, CMD e também o diretório de trabalho para executar instâncias da imagem do container.” (Fonte: [www.docker.com](http://www.docker.com)).

Nosso *Dockerfile*, ficará assim:

```
1 FROM wyveo/nginx-php-fpm:latest
2 WORKDIR /usr/share/nginx/
```

## RUN

“A instrução RUN especifica os comandos a serem executados e capturados na nova imagem de contêiner. Esses comandos podem incluir itens como a instalação de software, a criação de arquivos e diretórios e a criação da configuração do ambiente.” (Fonte: [www.docker.com](http://www.docker.com))

Como exemplo, em nossa aplicação:

```
1 FROM wyveo/nginx-php-fpm:latest
2 WORKDIR /usr/share/nginx/
3 RUN rm -rf /usr/share/nginx/html
4 RUN ln -s public html
```

A declaração RUN rm -rf /usr/share/nginx/html elimina a pasta html

A declaração RUN ln -s public html cria automaticamente o link simbólico.

## COPY

“A COPY é uma instrução que copia os arquivos e diretórios para o sistema de arquivos do container. Os arquivos e diretórios devem estar em um caminho relativo ao Dockerfile.” (Fonte: [www.docker.com](http://www.docker.com)).

Iremos abordar essa declaração de forma detalhada em nosso projeto, quando realizarmos o build da imagem para o Docker Hub.

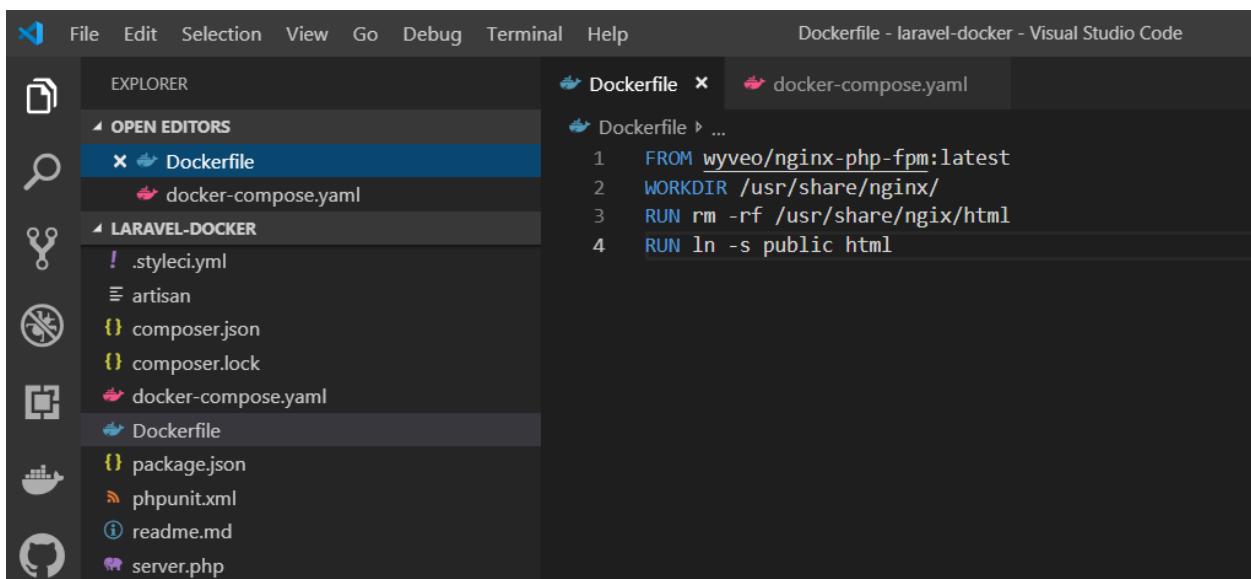
## ADD

“A instrução ADD é como a instrução de cópia, mas com ainda mais recursos. Além de copiar arquivos do host para a imagem de contêiner, a instrução ADD também pode copiar arquivos de um local remoto com uma especificação de URL.” (Fonte: [www.docker.com](http://www.docker.com)).

## CMD

“A instrução CMD, define o comando padrão a ser executado durante a implantação de uma instância da imagem do contêiner. Por exemplo, se o contêiner estiver hospedando um servidor Web NGINX, o CMD pode incluir instruções para iniciar o servidor Web com um comando como nginx.exe. Se várias instruções CMD forem especificadas em um Dockerfile, somente a última será avaliada.” (Fonte: [www.docker.com](http://www.docker.com)).

Em nosso projeto, o resultado será:



The screenshot shows the Visual Studio Code interface with the title bar "Dockerfile - laravel-docker - Visual Studio Code". The left sidebar is the Explorer view, showing files like ".styleci.yml", "artisan", "composer.json", "composer.lock", "Dockerfile", "package.json", "phpunit.xml", "readme.md", and "server.php". The main editor area shows the Dockerfile content:

```
FROM wyveo/nginx-php-fpm:latest
WORKDIR /usr/share/nginx/
RUN rm -rf /usr/share/nginx/html
RUN ln -s public html
```

# Build da imagem no Docker Hub

Para finalizar esse nosso guia rápido, realizaremos o build de nossa imagem no Docker Hub, dessa forma, outros desenvolvedores poderão utilizar a mesma imagem que criamos.

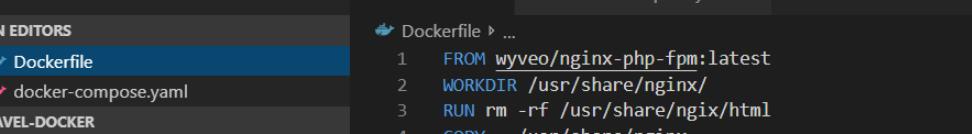
Nessa etapa, iremos declarar o COPY:

```
1   COPY . /usr/share/nginx
```

Em seguida utilizaremos, novamente o RUN:

```
1 RUN chmod -R 775 /usr/share/nginx/storage/*
```

Em nossa IDE ficara dessa maneira:



The screenshot shows the Visual Studio Code interface with the title "Dockerfile - laravel-docker - Visual Studio Code". The left sidebar contains icons for File, Edit, Selection, View, Go, Debug, Terminal, Help, Explorer, Open Editors, Laravel-Docker, and Docker. The "OPEN EDITORS" section lists "Dockerfile" (selected) and "docker-compose.yaml". The "LARAVEL-DOCKER" section lists ".styleci.yml", "artisan", "composer.json", "composer.lock", "docker-compose.yaml", "Dockerfile" (selected), "package.json", "phpunit.xml", and "readme.md". The main editor area displays the Dockerfile content:

```
FROM wyveo/nginx-php-fpm:latest
WORKDIR /usr/share/nginx/
RUN rm -rf /usr/share/nginx/html
COPY . /usr/share/nginx
RUN chmod -R 775 /usr/share/nginx/storage/*
RUN ln -s public html
```

Em nosso terminal, rodaremos o seguinte comando:

```
docker build -t seulogin/laravel-image:latest .
```

Sendo `seulogin/laravel-image`: latest . o nome de nossa imagem.

O comando digitado fará com que seja realizado o build com todas as alterações que fizemos em nosso *Dockerfile* e, feito o build, poderemos publicar a nossa imagem em nosso Docker Hub.

Após o build, faremos o login em nosso Docker Hub:

```
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over
to https://hub.docker.com to create one.
Username: rafatrevisani
Password:
WARNING! Your password will be stored unencrypted in C:\Users\rafae\.docker\config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

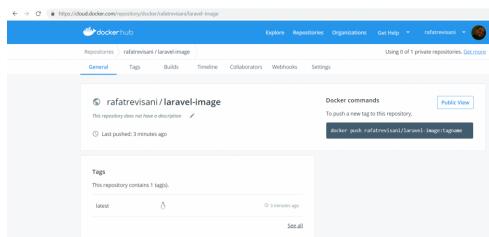
Login Succeeded
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ
```

Feito o login, faremos o push da imagem com o comando “docker push + nome da nossa tag”:

```
docker push seulogin/laravel-image:latest
```

```
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ docker push rafatrevisani/laravel-image:latest
The push refers to repository [docker.io/rafatrevisani/laravel-image]
d17451802f43: Pushed
f39a805fe410: Pushed
bae1dd18eb52: Pushed
8b17c9348060: Mounted from wyveo/nginx-php-fpm
b1d61a6d1bc4: Mounted from wyveo/nginx-php-fpm
244a4b78f4de: Mounted from wyveo/nginx-php-fpm
d47fa9804970: Mounted from wyveo/nginx-php-fpm
663ee81370b6: Mounted from wyveo/nginx-php-fpm
8c380c3bebcd: Mounted from wyveo/nginx-php-fpm
f94641f1fe1f: Mounted from wyveo/nginx-php-fpm
latest: digest: sha256:31187dcf6dbd2ab383f893f8af864bf14d48dbf0fbe1ef01c10a9c8724764dca size: 2410
Rafael@DESKTOP-51JQ04R ~/Documents/Projetos/laravel-docker
λ
```

Acessando o nosso Docker Hub, poderemos ver a nossa imagem:



Agora, a nossa imagem já está disponível a outros desenvolvedores e dessa forma, os mesmos poderão trabalhar ela.

# Conclusão

Como todos pudemos verificar, trabalhar com o Docker e o Docker Compose não foi nenhum bicho de sete cabeças.

Lembre-se que esse processo não serve somente para o Laravel, mas sim para qualquer linguagem e framework de sua escolha.

Que tal trabalhar em sua própria imagem agora?