# Mục đích của docker làm gì?

Docker là 1 kho chưa image, là 1 docker registry

Nhiệm vụ chính là khi ta tạo ra 1 ứng dụng, 1 trang web như trang web trước ta làm w3\_band, rồi đóng gói tất cả lại vào trong 1 image, và push lên registry,

Lưu ý là image đó đã có sẵn suorce code, cấu hình store,…

Mà khi ra run lên thì ứng dụng sẽ chạy cho ta như này.



Nhưng để run lên từ image đó, ta lại ko dùng các component của docker để run, mà ta dùng **Kubernetes**

Kubernetes: hỗ trợ môi trường chạy cho container

* Pod : tránh bị down container
* Quản lý container dễ dàng - replicas

Docker-image: là build lại image từ image offical và có setup 1 số config như port, hay volume để khi run lên n tương tác với volume trên host để ta cập nhật dữ liệu ánh xạ giữa host và container

Docker thường dùng docker-compose để run và có cấu hình mount phân vùng dữ liệu, thì cơ chế này lại ko hay bằng kubernetes.

Docker chỉ hay khi build image, đóng gói image thì ko nên dùng share folder vì khi run lên ở server khác n sẽ ko khớp với đường dẫn ở host đó

Còn kubernetes sẽ có share volume như pvc,nfs,…

|  |  |
| --- | --- |
| Dockerfile  FROM python:3.6-alpine  WORKDIR /app  # Tạo ra biến môi trường tên là PORT với giá trị 5555  ENV PORT 5555  COPY . .  RUN pip install -r requirements.txt  CMD ["python", "app.py"] | **docker-compose.yml**  version: "3.7"  services:  app:  image: learning-docker/python:v1  ports:  - "5000:5555"  restart: unless-stopped |

Tức là chỉ dùng dockerfile build image, và docker file ko nên xét nhiều share volume vì khi run image, đường dẫn volume nhớ đâu ở host khác lại ko đúng

Image ở dockerfile chỉ là mở rộng thêm tính năng, môi trường ở 1 image offical (ví dụ 1 image build ra chứa cả nginx và nodejs)

* Volume ta nên chơi(set) ở bước chạy, run image, vậy lựa chọn của kubernetes, run image riêng thiết lập volume ở kubernetes riêng.
* Và dùng docker-compose để run image và thiết lập port, volume share là quá tù

Và việc chỉnh sửa lại các env như thay port, sửa mount volume trên container của kuber rất nhanh, và lại ít down time

Còn vs docker-compose thì ko được edit trực tiếp vào container khi n đang chạy mà phải down rồi up lại => down time ứng dụng

Nếu ta đòi 1 image mà có thiết lập ứng dụng từ trước như w3\_band thì ko thể, bởi vì dữ liệu image mang chỉ là dữ liệu lúc build thôi, nếu ta update ứng dụng sau này thì image lại là của ứng dụng cũ.

* Ta bắt buộc phải dùng volume để ánh xạ và cập nhật dữ liệu liên tục

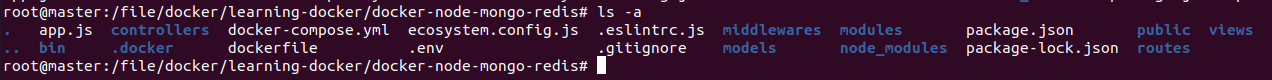
Khi ta chạy project bên trong docker container, khi container khởi động lại thì **mọi thứ sẽ mất và reset lại như ban đầu**, tức là data trong database sẽ mất, session của những user đang đăng nhập cũng sẽ mất.

Do đó ở bài này ta sẽ tạo ra những folder để lưu lại dữ liệu để khi container có khởi động lại thì dữ liệu của ta vẫn sẽ được lưu lại nhé (việc này tiếng anh gọi là **persist data**), để làm được việc đó thì ta **mount** những folder chứa dữ liệu này vào trong container dùng **volumes** nhé

Còn đối vs kuber thì container sẽ được quản lý bởi pod và giúp các container chia sẻ file vs nhau trong cùng 1 pod và dữ liệu sẽ ko mất khi container down

Dù docker hay kuber, thì để run app thì ta chỉ cần mount path chạy của container mount với folder source code, folder source code chứa app và thư viện

Vd source code trên host serrver



Và quan trọng ta phải config thêm env và volume

# Docker

**Những điều Docker mang lại**

Dưới đây là những thứ mình tổng hợp lại về những điều tốt đẹp của Docker:

1. Docker cho các bạn 1 môi trường mục tiêu cụ thể, ví dụ các bạn cần môi trường Ubuntu, có PHP 7.2 có NodeJS 8.0, có mysql 5.2. Docker sẽ giúp bạn có được điều đó.
2. **Môi trường trong Docker độc lập so với môi trường gốc**: Docker sẽ tạo ra cho các bạn môi trường "ảo" trong đó các bạn có thể chạy project của mình, bất kể hệ điều hành gốc của các bạn có là gì. Do đó, kể cả bạn ở Win hay Mac thì vẫn có thể chạy project dưới môi trường Ubuntu hay bất cứ môi trường nào (mà hiện tại Docker support) bạn cần.
3. Một môi trường Docker sau khi được định nghĩa nó sẽ là "bất biến". Bạn có thể setup ở bất kì đâu bất kì máy nào với môi trường giống hệt bạn đã định nghĩa.
4. Mỗi một project sẽ có một file cấu hình cụ thể, 10 năm sau bạn đọc lại project thì vẫn biết để chạy được project đó thì cần những gì và cần làm gì. Bạn chỉ cần đưa file cấu hình cho người khác là họ sẽ tự biết phải làm gì để chạy
5. Vì mỗi project chúng ta có thể setup ở một môi trường riêng, nên các project sẽ không bị xung đột, chia sẻ tài nguyên lằng nhằng (nếu như ta không muốn). Từ đó giảm thiểu tối đa sự phụ thuộc lẫn nhau, cài đặt, thêm bớt sửa xóa các thư viện, cấu hình cũng sẽ không bị ảnh hưởng tới các project khác.
6. Docker có thể giúp tự động Heal (tự hồi phục, khởi động lại) nếu trong trường hợp có lỗi.

## Dừng! Trước đây đã có máy ảo rồi mà??

Nghe xong phần bên trên có thể các bạn nhận ra sự na ná mà trước đây những máy ảo như VMWare hay Virtual Box cũng có thể làm được như: tạo ra các môi trường ảo, độc lập với hệ điều hành gốc.

Nhưng 1 trong những trở ngại lớn nhất của máy ảo theo mình thấy đó là sự nặng nề, cài đặt rất phức tạp, và rất chậm. Dùng máy ảo ta phải cắt cứng 1 lượng tài nguyên cho nó, dẫn tới hệ điều hành gốc và máy ảo phải chia sẻ phần cứng và làm giảm hiệu năng của cả 2. Việc cài đặt môi trường ở máy ảo tổn nhiều thời gian, khi gặp lỗi phải cài lại thì các bạn sẽ thấm nỗi đau .

Với Docker, Docker sẽ tạo ra "môi trường ảo" chứ không phải hẳn 1 máy ảo nữa, do đó sẽ chia sẻ phần cứng với môi trường gốc bên ngoài, RAM, CPU,.. tốc độ sẽ như nhau, chia sẻ file (nếu ta muốn) với môi trường gốc. Đồng thời việc tạo một môi trường "ảo" trong Docker cũng rất nhanh chóng và dễ dàng để làm với chỉ 1 hoặc 1 vài command.

Do đó các bạn cần rõ ràng về cái mà Docker mang lại và cái mà máy ảo mang lại: 1 thứ tạo ra cho ta môi trường ảo chia sẻ phẩn cứng với môi trường gốc, một thứ tạo ra cho chúng ta cả 1 chiếc máy ảo với mọi thứ từ phần cứng, đến file system,... tách biệt so với máy thật. (xồi, nghe xong đầu óc cứ thấy ảo ảo thật thật  ). Các bạn xem hình dưới để thấy so sánh vui, nhưng thực tế về việc dùng Docker so với dùng máy ảo nhé.

## Tại sai tôi cần Docker?

Dưới đây là một số suy nghĩ của mình vì sao các bạn nên yêu Docker :

* Khi project của chúng ta có nhiều người cùng làm, người dùng Win, người dùng Mac, người dùng Linux, nhưng cách để cài các phần mềm, thư viện để có thể chạy được project lại khác nhau ở mỗi nền tảng, đồng thời khi cài có thể làm thay đổi project. Dẫn tới việc project chạy không đúng, sai ở các nền tảng khác nhau.
* Sự sợ hãi nhất của mình ngày xưa đó là code Local ngon lắm rồi, nhưng lên server thì ngủm củ tỏi .
* Mỗi project cần rất nhiều thứ đi kèm: MySQL, Redis, các extensions,... Và việc nhớ cài cho đúng từng cái đã rất mệt, nhớ cách cấu hình cho đúng lại càng mệt hơn. Và sự SỢ HÃI lớn nhất là khi cài mà bị lỗi, mà lỗi xong xóa đi cài lại lại không được như cũ.
* Khi chúng ta có nhiều project, dùng chung nhiều thứ, dẫn tới khi muốn thay đổi, sửa đổi 1 tài nguyên dùng chung nào đó sẽ làm các project liên quan ảnh hưởng.
* Và 1 trường hợp mình thấy đau khổ nhất. Đó là khi ta chuyển môi trường (chuyển server chẳng hạn), ta cần bê nguyên cục project cũ và chuyển sang môi trường mới. Lúc này NỖI SỢ  mới thực sự hiện rõ. Làm sao để có thể chuyển toàn bộ data, cấu hình lại từ đầu với hàng tỉ bước, cài hàng tỉ thứ
* .... bla và blo

Tất cả những điều trên Docker sẽ giúp các bạn giải quyết một các rất chi là đơn giản, dễ dàng và cute (như người viết bài này vậy )

Thời điểm hiện tại việc dùng Docker sẽ giúp ta sử dụng được rất nhiều tiện ích đi kèm:

* Kubernetes: Tool để quản lý các project chạy bằng Docker, tự động Heal khi có lỗi, tự động scale, tự động deploy, tự động và tự động
* CI/CD: Phần này mình sẽ trình bày trong series này, áp dụng CI CD giúp ta chỉ cần code, và commit, còn lại các việc như test, check lỗi, và triển khai (deploy) ra môi trường thực tế đều được làm tự động, không cần phải cài các tool lằng nhằng để auto-deploy nữa.
* Sự hỗ trợ đến tận chân răng của các nền tảng quản lý source code như Github, Gitlab, Bitbucket,... cho các ứng dụng Docker lại càng làm tăng thêm giá trị cho Docker

## Image

Image là 1 trong những đơn vị cơ bản nhất trong Docker. 1 Image sẽ định nghĩa cho 1 môi trường và những thứ có trong môi trường đó. Ứng dụng của ta muốn chạy được thì cần phải có Image

Ví dụ trong Image ta có thể định nghĩa các thành phần như sau:

* Môi trường: Ubuntu 16.04
* PHP 7.2
* MySQL 5.2

Thì toàn bộ những thứ bên trên (Ubuntu, PHP, MySQL) ta có thể đóng gói lại thành 1 Image. Có thể giữ làm của hồi môn, hoặc đem public cho người khác có thể dùng được.

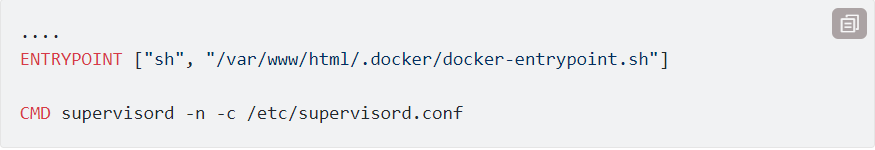
## Alpine



## Mỗi file Dockerfile chỉ cho phép có 1 CMD, vậy nếu tôi muốn có nhiều CMD thì sao?

Thì hiện tại Docker chỉ cho phép chạy 1 CMD khi khởi động container, nhưng nếu CMD của các bạn phức tạp thì ta có thể dùng tới ENTRYPOINT

ENTRYPOINT, có thể hiểu theo nghĩa đen là "điểm bắt đầu", nó có thể chạy khá giống như CMD nhưng cũng có thể dùng để cấu hình cho CMD trước khi ta chạy CMD. Một trường hợp thường dùng với ENTRYPOINT là ta dùng 1 file shell script .sh để cấu hình tất tần tật những thứ cần thiết trước khi khởi chạy container bằng CMD



# Docker resistry and repository

### Docker Registry

**Docker Registry** là một dịch vụ máy chủ cho phép lưu trữ các docker image của cá nhân, công ty, team,… Dịch vụ **Docker Registry** có thể được cung cấp bởi tổ chức thứ 3 hoặc là dịch vụ nội bộ được xây dựng riêng nếu bạn muốn. Một số dịch vụ Docker Registry phổ biến như :

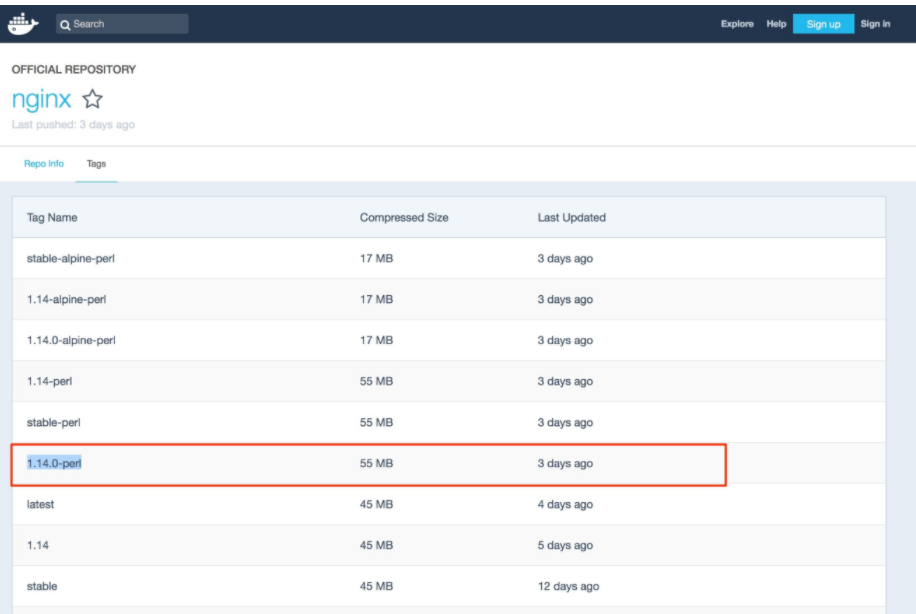
* Azure Container Registry
* Docker Hub
* Quay Enterprise
* Google Container Registry.
* AWS Container Registry

Docker sẽ tự tìm ở trên một public Registry (Registry là nơi lưu trữ Docker image, các bạn tưởng tượng nó như Github lưu code, nhưng đây là lưu Image)

### Docker Repository

**Docker Repository** là một tập hợp các docker image liên quan nhau với tên giống nhau, chỉ khác nhau giữa các giá trị thẻ (tag). **Thẻ (tag)** là một giá trị ID (số hoặc chữ) được gắn kèm thông tin theo [**Docker Image**](https://cuongquach.com/tag/docker-image) trong một repository để đánh dấu thể hiện thông tin khác nhau giữa các Docker Image. **Ví dụ** : version 1.1 , 1.2 hoặc latest.

Dễ hình dung nhất là khi bạn sử dụng command ‘**docker pull**‘ để download **Docker Image** nào bạn mong muốn. **Ví dụ** : ‘docker pull nginx:latest‘, thì Docker client sẽ download image có gắn thẻ tag là ‘latest’ nằm trong ‘nginx’ Docker Hub Repository.



# Run app bằng docker

* Dòng đầu tiên **FROM**: ta bắt đầu từ 1 Image có môi trường Alpint và đã cài sẵn Python phiên bản 3.6. Xem danh sách Image Python ở đâu, các bạn check ở [link chính thức](https://hub.docker.com/_/python) này nhé
* Lí do sao lại chọn Alpine mà không phải Ubuntu hay Debian, CentOS,.... Thì ở [bài trước ở mục này](https://viblo.asia/p/dockerize-ung-dung-nodejs-RnB5pxEG5PG#_cac-ban-phan-phoi-linux-va-cach-chon-image-sao-cho-dung-8) mình đã phân tích rồi nhé. Đồng thời xuyên suốt series này mình sẽ luôn dùng môi trường hệ điều hành Alpine Linux nhé
* Tiếp theo trong file Dockerfile ta có **WORKDIR**: ý là ta sẽ chuyển đến đường dẫn là **/app** bên trong Image, nếu đường dẫn này không tồn tại thì sẽ tự động được tạo luôn nhé
* Tiếp theo ta **COPY** toàn bộ file từ folder ở môi trường gốc (bên ngoài - folder **docker-python**) và đưa vào trong đường dẫn **/app** bên trong Image
* Tiếp tới là ta cài đặt dependencies, cần cài những thứ gì thì ta đề cập sẵn ở file **requirements.txt** rồi (câu lệnh này các bạn có thể xem nó xêm xêm như **npm install** trong NodeJS nhé)
* Cuối cùng là ta dùng **CMD** để chỉ command mặc định khi một container được khởi tạo từ Image: ở đây ta sẽ khởi động file [**app.py**](http://app.py/)

2 bước:

* Tạo image cho riêng mình bằng dockerfile
* Run bằng docker-compose

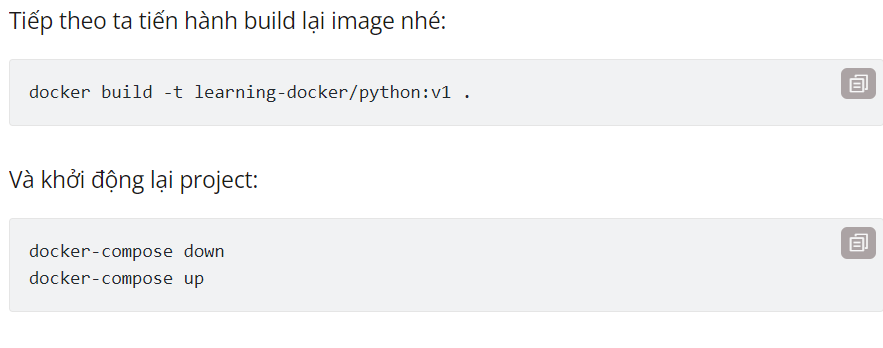
|  |  |
| --- | --- |
| Dockerfile  FROM python:3.6-alpine  WORKDIR /app  COPY . .  RUN pip install -r requirements.txt  CMD ["python", "app.py"] | **docker-compose.yml**  version: "3.7"  services:  app:  image: learning-docker/python:v1  ports:  - "5000:5000"  restart: unless-stopped |

Nhưng bây giờ bài toán đặt ra là muốn đổi port phải làm thế nào

## Biến môi trường (ENV)

Thay vì để biến port là số, sao t ko để là biến **x** nào đó.

|  |  |
| --- | --- |
| Dockerfile  FROM python:3.6-alpine  WORKDIR /app  # Tạo ra biến môi trường tên là PORT với giá trị 5555  ENV PORT 5555  COPY . .  RUN pip install -r requirements.txt  CMD ["python", "app.py"] | **docker-compose.yml**  version: "3.7"  services:  app:  image: learning-docker/python:v1  ports:  - "5000:5555"  restart: unless-stopped |



## Biến môi trường ở docker-compose

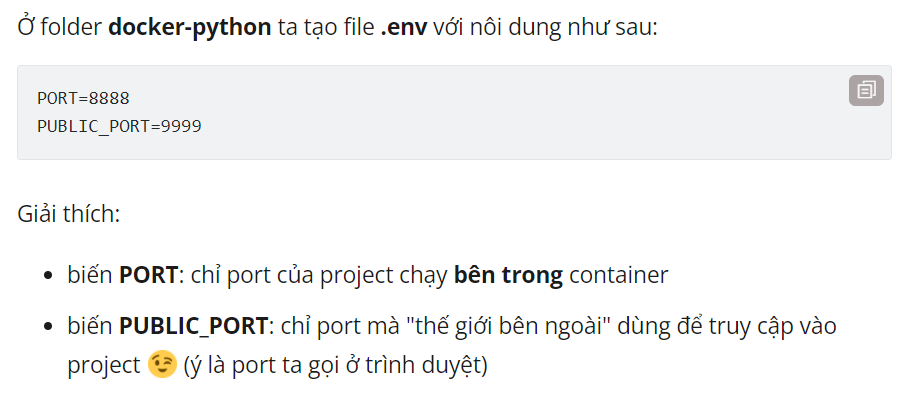
Để không phải build lại image mỗi lần ta đổi port, ta sẽ khai báo biến môi trường ở **docker-compose.yml** nhé. Tại sao:

* Biến môi trường ở file Dockerfile sẽ được khai báo khi ta build image
* Biến môi trường ở file docker-compose.yml sẽ được khởi tạo **khi container được khởi tạo**, tức là khi ta chạy **docker-compose up**. Do đó để thay đổi biến môi trường ta chỉ cần **down và up** là xong



## Cách tốt hơn để tạo biến môi trường (chuẩn cuối)

**docker-compose** support ta một cách đơn giản hơn, tiện hơn để khởi tạo biến môi trường, đó là đặt ở file **.env** (giống y như Laravel , cũng đồng nghĩa với việc nếu ta dockerize project Laravel thì ta chỉ cần duy nhất 1 file chung là **.env**). Khi chạy project thì **docker-compose** sẽ tự tìm xem có file **.env** hay không và load các biến trong đó.



Ta sửa lại file **docker-compose.yml** như sau:

version: "3.4"

services:

app:

image: learning-docker/python:v1

ports:

- "${PUBLIC\_PORT}:${PORT}"

restart: unless-stopped

environment:

PORT: ${PORT}

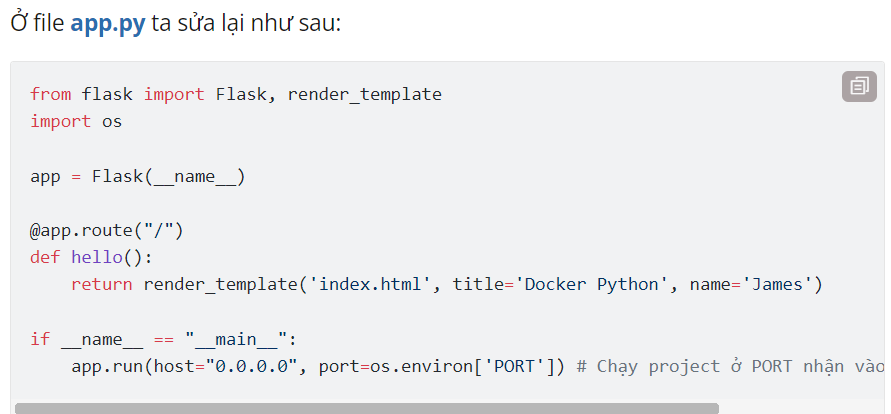
Sau đó ta khởi động lại project nhé:

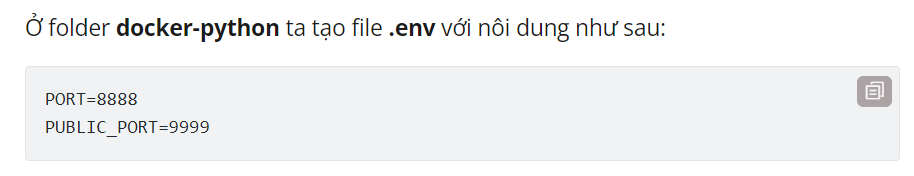
docker-compose down

docker-compose up

## chuẩn cuối: 2 bước

* Tạo .env
* Sửa tý app.py
* Khai báo docker-compose.yaml có biến .env (biến x của port )





# [Dockerize ứng dụng VueJS, ReactJS](https://viblo.asia/p/dockerize-ung-dung-vuejs-reactjs-ORNZqxwNK0n)

## Dockerize Project VueJS

Thế tại sao bài này lại có. Bắt nguồn từ điều này: đối với project Vue hoặc React là dạng full frontend, không có tí backend nào, chúng chỉ được chạy trên trình duyệt. Mà khi ra tới trình duyệt, thì thứ trình duyệt hiểu chỉ là **HTML, CSS, JS**.

Do đó để Dockerize ứng dụng Vue/React việc của ta là chỉ cần lấy được những file build cuối cùng cần thiết để chạy ở trình duyệt còn những thứ khác có hay không có, không quan trọng , làm như thế thì Image của chúng ta sẽ giảm được size xuống, giảm thiểu tối đa những thứ không cần thiết bên trong Image

Build image dockerfile

# build stage

FROM node:13-alpine as build-stage

WORKDIR /app

COPY . .

RUN npm install ## các bạn có thể dùng yarn install .... tuỳ nhu cầu nhé

RUN npm run build

# production stage

FROM nginx:1.17-alpine as production-stage

COPY --from=build-stage /app/dist /usr/share/nginx/html

## CMD ["nginx", "-g", "daemon off;"]

docker build -t learning-docker/vue:v1 .

* Ở file Dockerfile chúng ta chia làm 2 stage (giai đoạn) khi build image: **build stage** và **production stage**
* Ở **build stage** ta bắt đầu từ image tên là **node:13-alpine**, nếu các bạn không biết nó là gì thì đọc bài [dockerize ứng dụng NodeJS](https://viblo.asia/p/dockerize-ung-dung-nodejs-RnB5pxEG5PG) của mình nhé. Để đặt tên cho từng giai đoạn ta dùng từ khoá **as** nhé
* Trong build stage ta bắt đầu từ đường dẫn **/app**, sau đó copy toàn bộ file ở folder hiện tại ở môi trường ngoài, tức folder **docker-vue** vào bên trong đường dẫn ta set ở WORKDIR tức **/app** bên trong image.
* Tiếp theo ta chạy **npm install** như thường lệ để cài dependencies và cuối cùng là build project
* Ok build xong giờ tiến tới **production stage**: nơi ta định nghĩa cách chạy project
* Ở **production stage** ta bắt đầu với image tên là **nginx....** đặt tên stage này là **production-stage** với từ khoá **as**
* Thế **nginx** ở đây là cái gì thế ???? Project VueJS hay ReactJS khi chạy sẽ cần một webserver để có thể chạy được nó, và Nginx ở đây chính là webserver. Ở local có cần Nginx gì đâu nhỉ? Vì ở local khi chạy **npm run dev** thì các nhà phát triển VueJS đã thiết lập sẵn cho chúng ta 1 local webserver rồi nhé, nhưng khi chạy thật thì **không** nên dùng nhé, phải có 1 webserver xịn, và Nginx thì rất là xịn nhé
* Sau đó, phần này quan trọng nè, ta COPY từ **build-stage** lấy folder ở đường dẫn **app/dist** chính là "những file build cuối cùng cần thiết để chạy ở trình duyệt", ta lấy folder đó và copy vào đường dẫn **/usr/share/nginx/html**, đây chính là nơi Nginx sẽ tìm tới và trả về cho user khi user truy cập ở trình duyệt
* Và cuối cùng ta có CMD khởi động Nginx

**docker-compose.yml**

version: "3.7"

services:

app:

image: learning-docker/vue:v1

ports:

- "5000:80"

restart: unless-stopped

docker-compose up

## (Noted) Có cách nào để không phải chia Dockerfile thành 2 stage?

* Như ở trên mục đích của chúng ta là cần phải build được project, sau đó lấy "những file build cuối cùng" tức là lấy được folder **dist** ở Vue hoặc **build** ở bên React và gửi nó tới Nginx và bảo "Nginx cậu show hàng hộ tớ nếu có user hỏi thăm đến tớ"
* Vậy vấn đề ở đây là làm cách nào có thể build được project mà không cần dùng thêm một stage như ở Dockfile, sau đó chuyển nó tới cho anh bạn thân Nginx là ok rồi

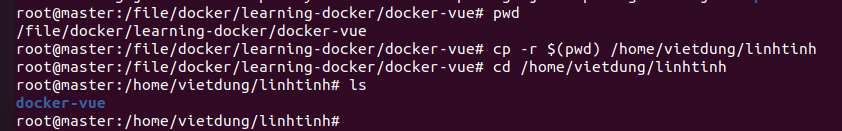
Docker hỗ trợ chúng ta có thể tạo ra một "container tạm thời" (intermediate container)

Container này sau khi làm xong nhiệm vụ thì sẽ tự được xoá đi.

docker run --rm -v $(pwd):/app -w /app node:13-alpine npm install && npm run build

ví dụ đang ở đường dẫn kia và dùng lệnh cp $(pwd)

cp $(pwd) = cp /file/docker/learning-docker/docker-vue



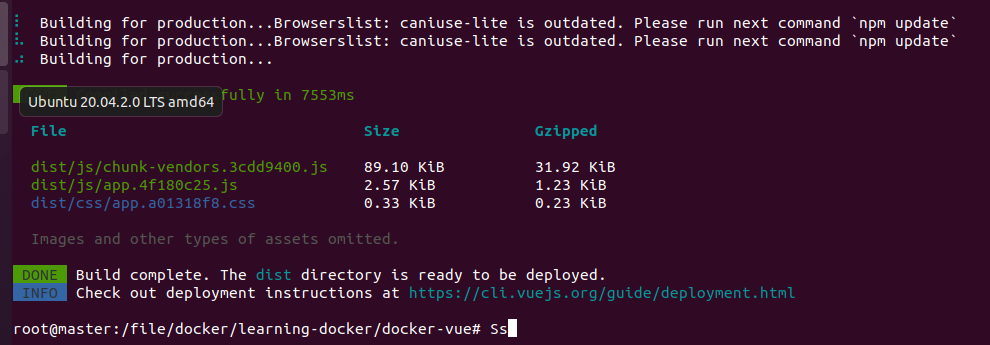
Giải thích nè:

* Ở câu lệnh trên ta chạy câu lệnh docker tạo ra 1 container với option **--rm** ý bảo "chạy xong chú tự xoá đi nhé"
* Tiếp theo ta có option **-v** tức là **volume**, ồ volume là gì mới à nha. Ở [bài đầu tiên](https://viblo.asia/p/li-do-toi-yeu-docker-ORNZqxRMK0n#_volume-11) mình đã nói tới rồi nhé, các bạn cứ bình tĩnh dần dần ta sẽ học nó nhiều hơn
* Sau **-v** là $(pwd):/app, ở đây ý ta bảo là đưa toàn bộ file ở folder hiện tại ở môi trường gốc ánh xạ vào trong đường dẫn /app trong Image ( $(pwd) trả về đường dẫn hiện tại) , việc này gọi đúng thuật ngữ thì là **mount**
* Tiếp theo ta có option **-w** chính là WORKDIR nhé
* Tiếp theo ta có **node:13-alpine**: tương đương với **FROM:node:13-alpine**
* Sau đó là các command ta cần chạy để build project.

Sau khi command trên chạy thành công thì các bạn sẽ thấy ở folder **docker-vue** xuất hiện cả folder **node\_modules** và folder **dist**. Đây chính là điều mà **volume** trong Docker mang lại.



Volume giúp ánh xạ file ở môi trường ngoài vào trong Docker container, và ánh xạ này là ánh xạ 2 chiều, ngoài thay đổi thì trong thay đổi, trong thay đổi thì ngoài thay đổi theo. Đó là lí do vì sao khi command trên chạy xong ở bên ngoài ta lại có kết quả như vậy. Dần dần các bạn sẽ hiểu **volume** nó là gì nhé



# Dockerize ứng dụng Laravel

* Để chạy được code PHP ta cần 1 bạn tên gọi là "handler", ý là code PHP không phải cứ thế ăn ngay chạy luôn được, cần phải có một anh bạn đảm nhận nhiệm vụ thực thi code PHP. Và ở thời điểm hiện tại thì PHP-FPM là phổ biến nhất, nên ta sẽ dùng PHP-FPM
* Thường thường hay thực tế là mình thấy là hầu hết thì ta sẽ dùng cùng với một webserver như Apache hay Nginx trong việc vận hành ứng dụng PHP (Laravel). Vì Nginx hiện tại cực kì phổ biến và mạnh mẽ vượt trội nhiều so với Apache. Nên ta sẽ dùng Nginx nhé

Do đó bài này ta sẽ tách ứng dụng thành 2 phần như sau:

* Một phần chứa PHP-FPM, trong đó có cài tất cả mọi thứ liên quan như composer, thư viện, setup, vì php-fpm đảm nhiệm vai trò chính trong việc chạy code
* Một phần là webserver Nginx đóng vai trò như là 1 anh gác cửa, đứng ở bên ngoài, khi có request gửi đến anh gác cửa anh ấy sẽ làm một số nhiệm vụ và chuyển request vào cho php-fpm ở bên trong xử lý

# Set master image

FROM php:7.2-fpm-alpine

# Set working directory

WORKDIR /var/www/html

# Install PHP Composer

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer

# Copy existing application directory

COPY . .

docker build -t learning-docker/laravel:v1 .

Mình sẽ giải thích những điểm quan trọng ở file Dockfile nhé:

* Đầu tiên ta bắt đầu từ image có tên **php:7.2-fpm-alpine** (mình lấy [ở đây](https://hub.docker.com/_/php))
* Tiếp theo ta chuyển đến đường dẫn **/var/www/html**
* Tiếp đó ta cài composer, để ta có thể cài các thư viện php
* Tiếp nữa là ta COPY toàn bộ folder hiện tại ở môi trường ngoài vào đường dẫn hiện tại trong Image, chính là WORKDIR



## Cấu hình docker-compose

Ta tạo file **docker-compose.yml** với nội dung như sau nhé:

version: '3.4'

services:

#PHP Service

app:

image: learning-docker/laravel:v1

restart: unless-stopped

volumes:

- ./:/var/www/html

#Nginx Service

webserver:

image: nginx:1.17-alpine

restart: unless-stopped

ports:

- "8000:80"

volumes:

- ./:/var/www/html

- ./nginx.conf:/etc/nginx/conf.d/default.conf

* ở service **app** mình có **mount** (hay hiểu đơn giản hơn đó là ánh xạ) 1 volume từ folder hiện tại vào bên trong container ở đường dẫn **/var/www/html**, để làm gì thì các bạn xem ở cuối bài nhé
* ở service **webserver** ta vừa mount folder hiện tại vào bên trong **/var/www/html** đồng thời cũng mount thêm file **nginx.conf** ở folder hiện tại vào bên trong container ở đường dẫn **/etc/nginx/conf.d/default.conf**. Vì để nginx có thể hiểu được php và vận hành cho đúng thì ta cần thêm 1 chút config nhỏ để khi được khởi chạy, Nginx sẽ đọc file config và vận hành cho đúng, file config đó sẽ được ta định nghĩa ở phần tiếp theo nhé

## Cấu hình Nginx

Mọi bài trước build xong Image là chạy được rồi nhưng bài này chúng ta cần phải cầu hình Nginx thêm chút nữa, vì lát nữa ta dùng Nginx như webserver nhé

Ở folder gốc **docker-laravel** chúng ta tạo file **nginx.conf** với nội dung như sau:

server {

listen 80;

index index.php index.html;

error\_log /var/log/nginx/error.log;

access\_log /var/log/nginx/access.log;

root /var/www/html/public;

location ~ \.php$ {

try\_files $uri =404;

fastcgi\_split\_path\_info ^(.+\.php)(/.+)$;

fastcgi\_pass app:9000;

fastcgi\_index index.php;

include fastcgi\_params;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_param PATH\_INFO $fastcgi\_path\_info;

fastcgi\_hide\_header X-Powered-By;

}

location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

}

Ở trên các bạn chú ý dòng **fastcgi\_pass app:9000;**: khi có request gửi đến nginx, nginx sẽ chuyển request đó tới PHP-FPM đang lắng nghe ở host tên là **app** và port **9000**.

Giải thích chút nhé :

Host tên là **app** ở đâu ra vậy? đó chính là địa chỉ của service **app** tên này sẽ phải trùng khớp với tên của service ta định nghĩa ở file **docker-compose nhé**

Cổng 9000 ở đâu ra đây, có thấy ở file **docker-compose** có map port gì đâu nhỉ? Thì các bạn xem lại phần cấu hình Dockerfile bên trên đoạn cuối chúng ta có vọc thử image **php:7.2-fpm-alpine** và thấy có đoạn **EXPOSE 9000**, đây là điểm rất quan trọng mà mình muốn các bạn để ý.

**EXPOSE 9000 nhằm mục đích muốn nói là: tôi chấp nhận cho container khác giao tiếp với tôi ở port 9000**

Các bạn chú ý sự khác nhau giữa mapping port ở các bài trước và EXPORT port ở bài này nhé:

* Mapping port nhằm giúp thế giới bên ngoài giao tiếp được với container
* Export port nhằm giúp các container có thể giao tiếp được với nhau, ở trên **webserver** sẽ cần phải gửi request tới **app**, nên app phải expose port 9000, nhưng **webserver** không cần expose vì chẳng có ai gọi đến nó cả.
* Ở thế giới bên ngoài sẽ không gọi được vào service **app** ở port 9000 nhưng **webserver** thì có thể nhé

## Tóm lại

Ta có các file

1. Dockerfile – để build image

# Set master image

FROM php:7.2-fpm-alpine

# Set working directory

WORKDIR /var/www/html

# Install PHP Composer

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer

# Copy existing application directory

COPY . .

1. Docker-compose

version: '3.4'

services:

#PHP Service

app:

image: learning-docker/laravel:v1

restart: unless-stopped

volumes:

- ./:/var/www/html

#Nginx Service

webserver:

image: nginx:1.17-alpine

restart: unless-stopped

ports:

- "8000:80"

volumes:

- ./:/var/www/html

- ./nginx.conf:/etc/nginx/conf.d/default.conf

1. env

cp .env.example .env

1. nginx.config

server {

listen 80;

index index.php index.html;

error\_log /var/log/nginx/error.log;

access\_log /var/log/nginx/access.log;

root /var/www/html/public;

location ~ \.php$ {

try\_files $uri =404;

fastcgi\_split\_path\_info ^(.+\.php)(/.+)$;

fastcgi\_pass app:9000;

fastcgi\_index index.php;

include fastcgi\_params;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_param PATH\_INFO $fastcgi\_path\_info;

fastcgi\_hide\_header X-Powered-By;

}

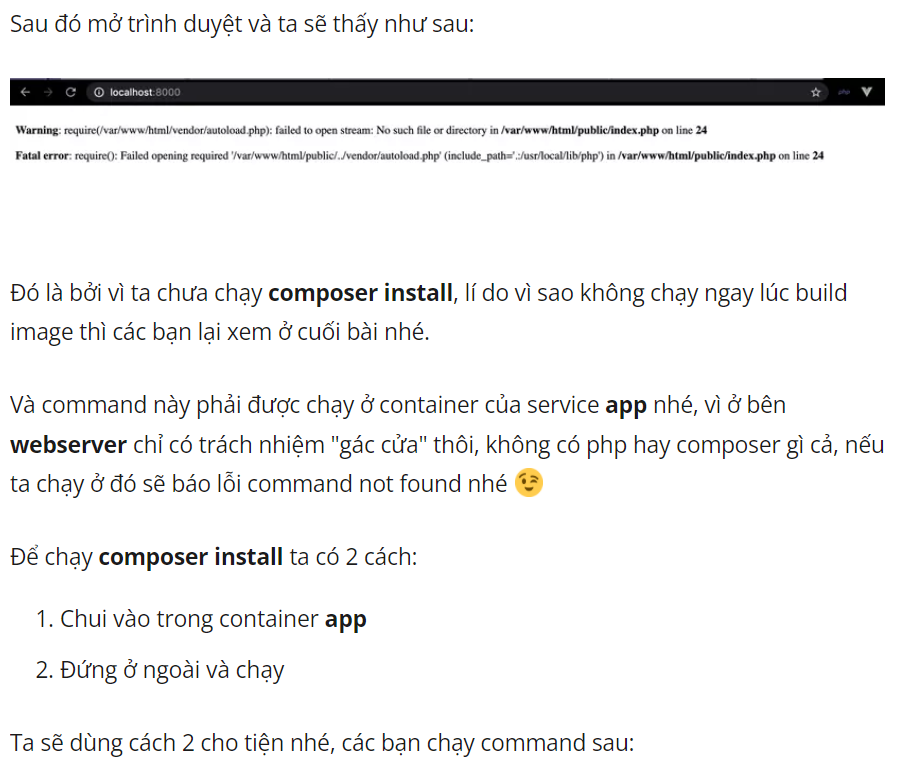
location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

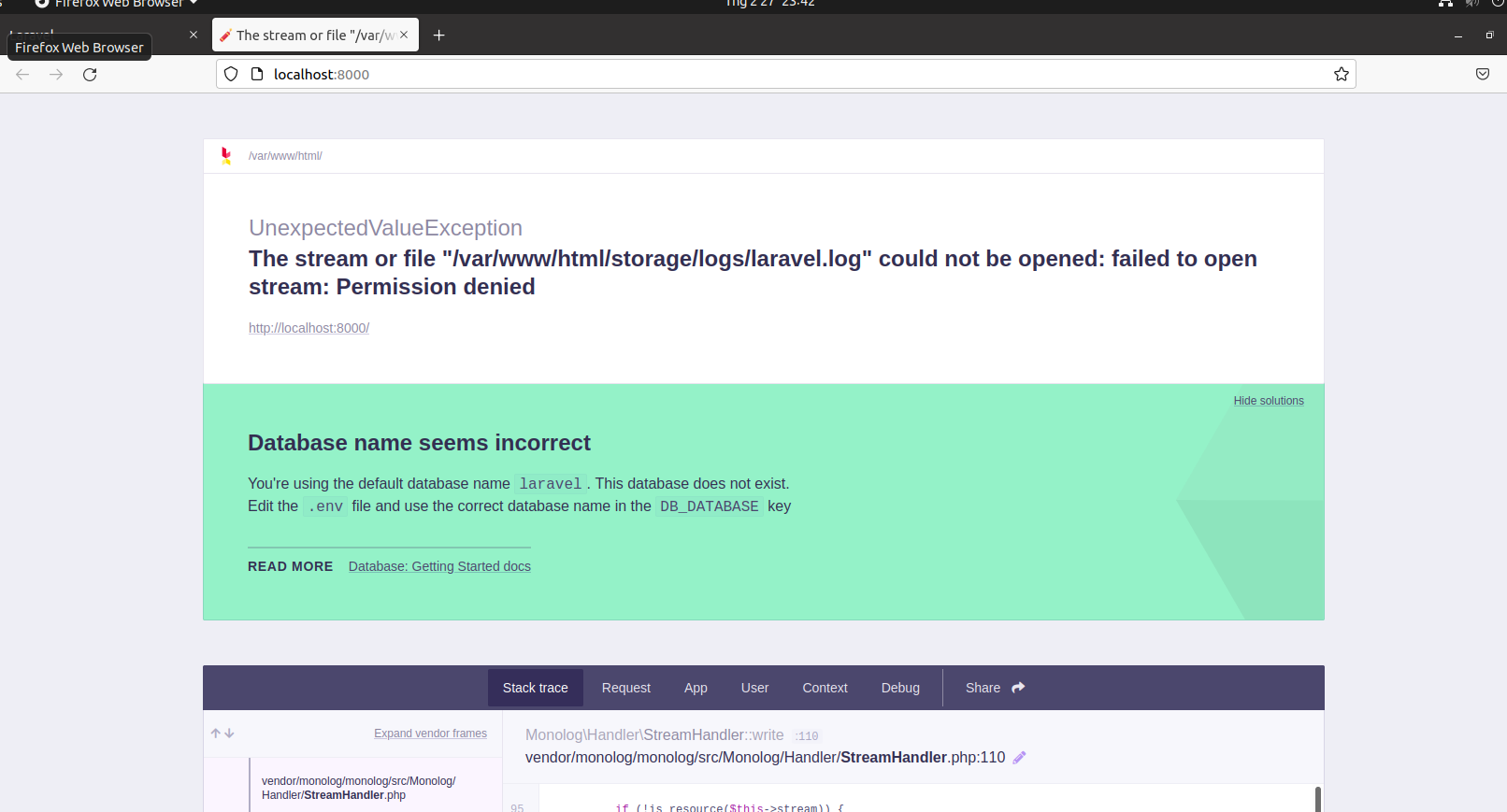
}

Rồi giờ docker-compose up



docker-compose exec app composer install

Chờ command chạy xong ta quay lại F5 trình duyệt và thấy:

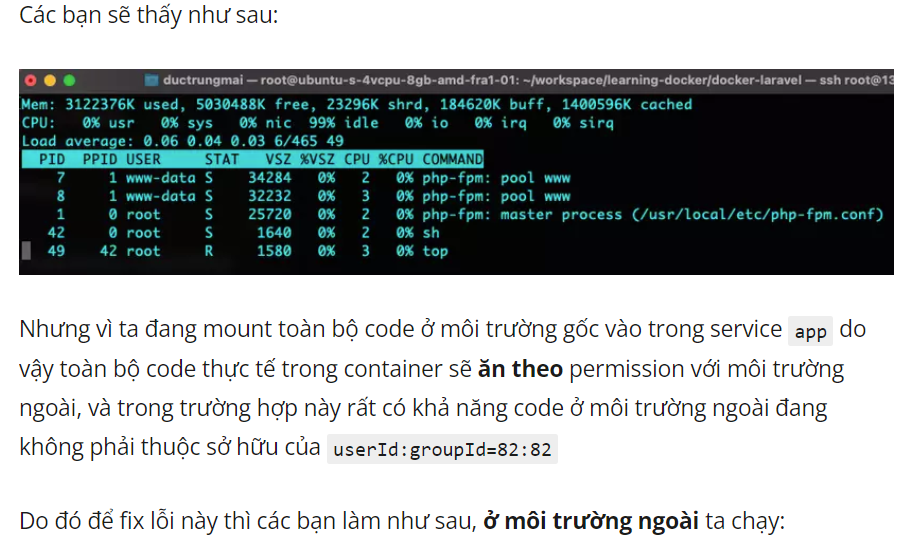


Đó là vì ta chưa generate key cho project

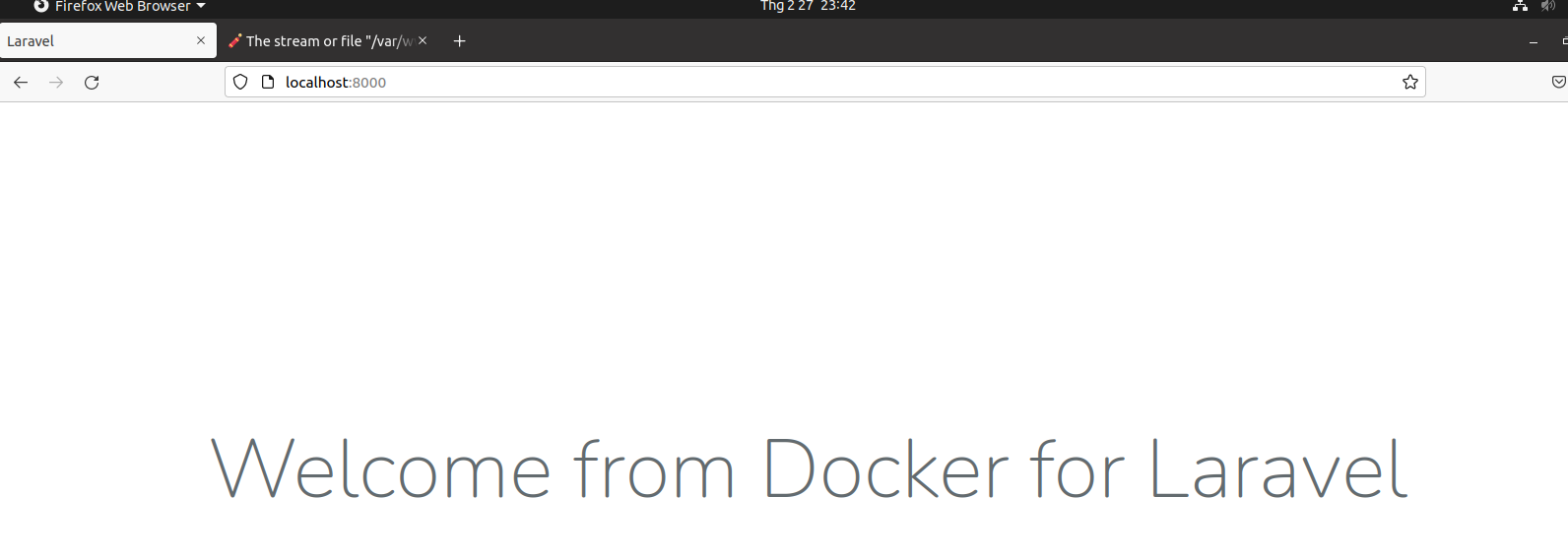
docker-compose exec app php artisan key:generate

## Nếu bạn gặp lỗi Permission Denied

Bởi vì container service app thì php-fpm nó được chạy bằng user www-data với userId:groupId là 82:82, kiểm chứng bằng cách chạy:



sudo chown -R 82:82 .



# Dockerize Project NodeJS, MongoDB, Redis, Passport

**docker-node-mongo-redis**

Project này có chức năng gì:

* Trang đăng nhập, đăng kí
* Bên trong có thêm mới sản phẩm, liệt kê danh sách sản phẩm ứng với user đăng nhập

Project này kiến trúc có những gì:

* Ta chỉ có 2 model là User và Product
* Dùng MongoDB để lưu trữ dữ liệu
* Dùng Redis để lưu trữ session của user đăng nhập
* Để xử lý Login/Logout ta dùng PassportJS
* Để xử lý upload file ta dùng Multer

ta định nghĩa ở **docker-compose.yml**:

* Có MongoDB là database -> ta có service **db**, dùng image mongo được build sẵn
* Có redis để lưu session của user -> ta có serivce **redis**, dùng image redis được build sẵn
* Phần còn lại là serivce để chạy project nodejs của chúng ta và kết nối tới 2 service bên trên -> ta chỉ cần cấu hình dockerfile cho service này, ta gọi là **app**

## Cấu hình Dockerfile

FROM node:13-alpine

WORKDIR /app

COPY . .

RUN npm install

# Development

CMD ["npm", "run", "dev"]

# Production

# RUN npm install -g pm2

# CMD ["pm2-runtime", "ecosystem.config.js", "--env", "production"]

Build image

docker build -t learning-docker/docker-node-mongo-redis:v1 .

## Data persistent

Ở bài này ta có databse là MongoDB để lưu trữ dữ liệu, ta có Redis để lưu trữ session của user.

Khi ta chạy project bên trong docker container, khi container khởi động lại thì **mọi thứ sẽ mất và reset lại như ban đầu**, tức là data trong database sẽ mất, session của những user đang đăng nhập cũng sẽ mất.

Do đó ở bài này ta sẽ tạo ra những folder để lưu lại dữ liệu để khi container có khởi động lại thì dữ liệu của ta vẫn sẽ được lưu lại nhé (việc này tiếng anh gọi là **persist data**), để làm được việc đó thì ta **mount** những folder chứa dữ liệu này vào trong container dùng **volumes** nhé

Ở folder gốc, các bạn tạo cho mình folder **.docker** (để ý dấu chấm ở đầu nhé). Trong **.docker** ta tạo folder **data**, trong **data** ta tạo 2 folder tên là **db** (cho mongodb) và **redis** cho redis

## Cấu hình docker-compose

**ocker-compose.yml** với nội dung như sau nhé:

version: "3.4"

services:

app:

image: learning-docker/docker-node-mongo-redis:v1

volumes:

- ./:/app # mount từ môi trường gốc vào trong để nếu các bạn thay đổi code thì bên trong sẽ tự động cập nhật

environment: # phần này ta định nghĩa ở file .env nhé

- DB\_HOST=${DB\_HOST}

- DB\_NAME=${DB\_NAME}

- REDIS\_HOST=${REDIS\_HOST}

- REDIS\_PORT=${REDIS\_PORT}

- PORT=${PORT}

ports:

- "${PORT}:${PORT}" # phần này ta định nghĩa ở file .env nhé

restart: unless-stopped

depends\_on:

- redis

- db

db:

image: mongo

volumes:

- .docker/data/db:/data/db

restart: unless-stopped

redis:

image: redis:5-alpine

volumes:

- .docker/data/redis:/data

restart: unless-stopped

Giải thích một số chỗ mới như sau

* Ở service **db** cho mongodb ta dùng image tên **mongo** (bản chính thức, các bạn search google là ra nhé)
* Tiếp đó ta mount volume từ folder **.docker/data/db** ta tạo ở phần trước vào bên trong container ở đường dẫn **data/db** và lát nữa khi khởi chạy Mongo sẽ tự tìm đến **/data/db** và load những dữ liệu vào trong database nhé
* Sao mình lại biết được đường dẫn **/data/db** ở trong container mà viết ở đây, thì ở phần mô tả của image <https://hub.docker.com/_/mongo>  họ đã nói rõ rồi nhé
* Điều tương tự cho service redis các bạn xem ở bên dưới nhé

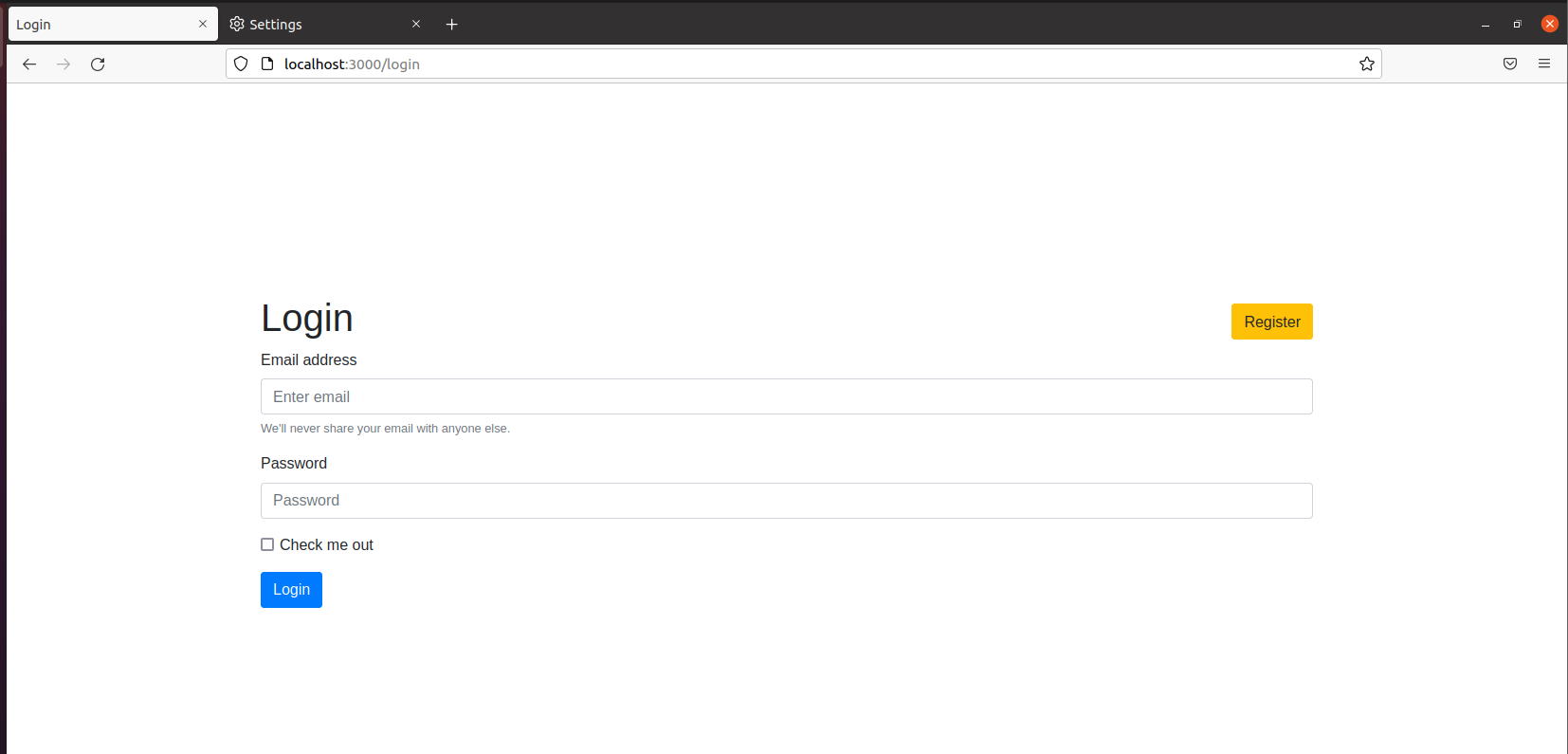
Có 1 điều các bạn để ý là ờ service **app** chúng ta có để một trường tên là **depends\_on** (phụ thuộc vào), ý bảo là service **app** sẽ phụ thuộc vào 2 service **db** và **redis**, điều này thực tế sẽ xảy ra như sau:

* Khi chạy **docker-compose up** thì service **db** và **redis** sẽ khởi động trước service **app**
* Khi chạy **docker-compose up app** thì đồng thời sẽ tạo ra 2 service **db** và **redis** (dù khi khởi động ta chỉ nói là "khởi động mỗi service app")
* Khi chạy **docker-compose stop** thì service **app** sẽ bị stop trước 2 service kia

Note: mặc dù service **app** khởi động sau service **db** nhưng sẽ không đảm bảo service **db** sẵn sàng ngay thời điểm service **app** khởi động và kết nối tới nhé, vì service **db** tốn 1 khoảng thời gian nhỏ từ lúc khởi động đến lúc sẵn sàng cho kết nối. Đó là lí do vì sao ở file **app.js** mình lại phải có đoạn code **connectionRetry** để thử kết nối từ NodeJS tới MongoDB cho tới khi nào thành công

Note tiếp: Vì bản chính thức image của Mongo không có cho Alpine (các bạn có thể tự cấu hình rồi build) nhưng ở đây mình dùng luôn image được build sẵn, image này được build dựa trên môi trường hệ điều hành Ubuntu (lát nữa khi chạy các bạn exec vào trong chạy **cat /etc/os-release** sẽ thấy).

Chắc bạn sẽ thắc mắc: có service thì chạy trên môi trường HĐH Alpine có service chạy trên HĐH Ubuntu, có được hay không? Thì câu trả lời là chạy bình thường như cân đường hộp sữa nhé, các bạn tưởng tượng chúng giống như kiểu API, API viết bằng NODEJS vẫn sẽ gọi được API viết bằng Python như thường, miễn là trả ra cái gì đó chung kiểu như dữ liệu JSON là được (đó là mình ví dụ thế



Lưu ý cài thêm node\_module nếu chưa có node\_modules

docker run --rm -v $(pwd):/app -w /app node:13-alpine npm install

Command trên sẽ tạo ra 1 container tạm thời từ image node:13-alpine, chạy npm install và mount trực tiếp file ra môi trường ngoài, kết qủa là khi command kia chạy xong thì môi trường ngoài của ta sẽ có node\_modules. Các bạn có thể hiểu đơn giản là nó cũng như việc ta chạy npm install thông thường vậy, nhưng giờ ta dùng sự hỗ trợ của Docker để chạy, như thế ở môi trường ngoài ta không cần cài thêm NodeJS để chạy npm

Ok rồi đó, các bạn restart lại project bằng cách chạy command sau:

Docker-compose up

Sau đó chúng ta mở trình duyệt ở địa chỉ localhost:3000 (cổng 3000 ta khai báo ở file .env nhé), nếu thấy như sau là oke rồi đó

## Còn khi ta deploy trên production

 Ta chỉ cần tạo file **docker-compose.yml** với nội dung như sau:

version: "3.4"

services:

app:

image: registry.gitlab.com/maitrungduc1410/learning-docker:node\_mongo\_redis\_production

volumes:

- ./public/images:/app/public/images # ta chỉ cần mount mỗi folder image thôi nhé

environment: # phần này ta định nghĩa ở file .env nhé

- DB\_HOST=${DB\_HOST}

- DB\_NAME=${DB\_NAME}

- REDIS\_HOST=${REDIS\_HOST}

- REDIS\_PORT=${REDIS\_PORT}

- PORT=${PORT}

ports:

- "${PORT}:${PORT}" # phần này ta định nghĩa ở file .env nhé

restart: unless-stopped

depends\_on:

- redis

- db

db:

image: mongo

volumes:

- .docker/data/db:/data/db

restart: unless-stopped

redis:

image: redis:5-alpine

volumes:

- .docker/data/redis:/data

restart: unless-stopped

image là ta vừa chạy oke trên môi trường test rồi, giờ build lại image rồi đẩy lên registry

Sau đó ta tạo file **.env** với nội dung như sau:

PORT=3000

DB\_HOST=db

DB\_PORT=27017

DB\_NAME=my\_db

REDIS\_HOST=redis

REDIS\_PORT=6379

Và cuối cùng là chạy project thôi   , giờ phút của sự thật đến rồi, ta chạy command sau:

docker-compose up

**.dockerignore** file này

# Bỏ qua node\_modules vì lát nữa ở Dockerfile sẽ chạy npm install

node\_modules

# Bỏ qua folder .docker vì folder này chỉ có service db và redis mới cần (ta đang build image cho service app)

.docker

# Bỏ qua ảnh mà chúng ta đã upload lên trong quá trình test

public/images/\*

# Bỏ qua file .env, file này dùng cho docker-compose khi chạy nên không cần đưa vào image

.env

# Bỏ qua Dockerfile không đưa vào image vì file này chỉ dùng tại thời điểm build image

Dockerfile

# Bỏ qua file docker-compose vì file này chỉ dùng tại thời điểm khởi chạy project

docker-compose.yml

tức là bỏ đi những cái trong đây

# Dockerize ứng dụng chat realtime với Laravel, Nginx, VueJS, Laravel Echo, Redis, SocketIO

## Tổng quan ứng dụng

Ứng dụng này đã được mình làm và deploy chạy production <https://realtime-chat.jamesisme.com/login>  các bạn có thể vào và dùng thử

Tổng quan:

* Ứng dụng có các phòng chat để user có thể join vào
* Mỗi phòng chat sẽ có 1 hộp (box) chat chung, tất cả user trong phòng có thể nhắn tin vào đây
* Cùng với đó là danh sách user có ở trong phòng chat, click chọn vào 1 user bất kì để nhắn tin riêng vơi người đó
* Cứ mỗi 1 phút sẽ tự động xuất hiện tin nhắn chào mừng của Bot (dùng **Laravel Schedule Task**)
* Detect user đang gõ, hay đã xem tin nhắn,...
* Các bạn có thể chọn biểu cảm tin nhắn như Facebook Messager: Love, Haha, Wow,.....

# Docker HEALTHCHECK

version: "3.4"

services:

app:

image: learning-docker/docker-node-mongo-redis:production

volumes:

- ./public/images:/app/public/images

environment: # phần này ta định nghĩa ở file .env nhé

- DB\_HOST=${DB\_HOST}

- DB\_NAME=${DB\_NAME}

- REDIS\_HOST=${REDIS\_HOST}

- REDIS\_PORT=${REDIS\_PORT}

- PORT=${PORT}

ports:

- "${PORT}:${PORT}" # phần này ta định nghĩa ở file .env nhé

restart: unless-stopped

depends\_on:

- redis

- db

networks:

- db-network

- redis-network

healthcheck:

test: wget --quiet --tries=1 --spider http://localhost:${PORT} || exit 1z

interval: 30s

timeout: 10s

retries: 5

db:

image: mongo

volumes:

- .docker/data/db:/data/db

restart: unless-stopped

networks:

- db-network

healthcheck:

test: echo 'db.runCommand("ping").ok' | mongo db:27017/speech-api --quiet

interval: 30s

timeout: 10s

retries: 5

redis:

image: redis:5-alpine

volumes:

- .docker/data/redis:/data

restart: unless-stopped

networks:

- redis-network

healthcheck:

test: ["CMD", "redis-cli","ping"]

interval: 30s

timeout: 10s

retries: 5

#Docker Networks

networks:

db-network:

driver: bridge

redis-network:

driver: bridge

Giải thích

Ở service **app** ta định nghĩa **HEALTHCHECK** với nội dung như sau:

* Ta dùng command **wget --quiet --tries=1 --spider**[**http://localhost**](http://localhost/)**: ${PORT} || exit 1z** (ở đây ta dùng localhost vì command này được chạy **bên trong** container nhé)
* Mình dùng luôn **wget** được cung cấp sẵn, command trên sẽ tạo 1 request đến app của chúng ta nếu status trả về là 200 là ok coi như "anh khoẻ" , nếu status trả về khác 200 thì coi như không khoẻ . Các bạn có thể dùng **curl** cũng được nhé
* Ta có **interval** là 30 giây, tức là command trên sẽ được chạy lần đầu là 30 giây sau khi container được khởi tạo (nhằm mục đích cho container của ta sẵn sàng cho việc "kiểm tra sức khoẻ", chứ nhiều lúc test ngay thì kết qủa có thể không đúng ). Sau đó cứ mỗi 30 giây tiếp theo ta lại "kiểm tra sức khoẻ" tiếp
* Ta có **timeout** là 10 giây, ý là nếu việc kiểm tra sức khoẻ kéo dài quá 10 giây, thì coi như là "anh không còn khoẻ nữa" . Điều này có thể xảy ra khi chúng ta tạo request tới 1 URL nào đó cần nhiều thời gian phản hồi. Nên các bạn cần chọn 1 con số thích hợp nhé
* Cuối cùng là ta có **retries** (thử lại) 5 lần: nếu như 1 container bị test failed (không khoẻ), thì Docker sẽ tiếp tục test thêm liên tục 5 lần để xem có chắc chắn là không khoẻ không,

