

* 이 튜토리얼은 WSL이 아니라 순수 리눅스에서 해보길 추천한다. WSL 없는 순수 윈도우에서 할 경우, 경로를 username 하위로 잡아줘야 작동함.

windows terminal 설치

WSL 설치

이후 Docker 설치

윈도우의 경우, Terminal을 열어, WSL Ubuntu를 키자. 그리고 /home/사용자이름 폴더에 들어간다.

이동한 후, 알코 강의를 듣는 경우

```
git clone https://gitlab.com/yalco/practice-docker.git
```

이 프로젝트는 Svelte 프레임워크로 진행되어있다. React, Vue 같은건데, 신생이다. frontend 폴더 안에 Svelte 라이브러리로 코딩한 src/App.svelte 파일이 메인이고, 빌드한 결과물은 public/build 안에 있다.

하고자 하는 건, node.js 로 돌아가는 http-server란 프로그램으로 웹서버를 실행해 이 파일들로 웹사이트를 띄울 것이다.

```
docker run -it node
```

이러면 좀 시간이 지난 후에 nodejs가 실행된다.

docker images 를 쳐보면(꼭 그 폴더에 없어도 된다. docker 는 전역이다.) node image 가 다운받아진것을 볼 수 있다.

대체 무슨 일이 일어난걸까? 바로 nodejs 가 docker hub 에 이미지로 있기 때문이다.



우리가 개발할 때 필요한 것들이 이미지로 좀 있다. ubuntu도 이미지가 있고, postgres, nginx, python, httpd(apache http server), docker(도커허브에 도커 이미지도 있음) 등등

아니 그럼 이미지란 대체 뭘까?

이미지는 리눅스 컴퓨터의 특정 상태를 캡처해서 박제해놓은것을 말한다.

즉, node 이미지는 리눅스에 node.js 가 설치된 상태가 박제되어있는것.

그럼 이미지가 어떻게 내 컴퓨터에 깔린거임?

도커는 일단 내 컴퓨터에서 그 이미지를 찾아보고, 없으면 docker hub로부터 해당 이름으로 등록된 이미지를 다운받게된다.

```
docker run -it node
```

여기서 run은 이미지가 없을 경우에 다운받은 이미지를 해동해서 내 컴퓨터의 컨테이너로 만든다. 이미지를 복사하는것이기때문에, 하나의 이미지로 컨테이너를 몇개씩이든 만들 수 있음

-it : 이 컨테이너를 연 다음 그 환경 안에서 cli를 사용한다. 컨테이너를 만든 다음 그 안에 있는 근무자와 컨테이너에 난 창문으로 대화를 하겠다는 뜻.

그리고 이 이미지는 컨테이너가 만들어져 설치되면(해동에서 깨어나자마자) 바로 node 명령어가 실행되도록 설계되어있다. 그래서 우리가 터미널에 node 명령어를 쳤을 때와 동일한 화면이 나오는 것.

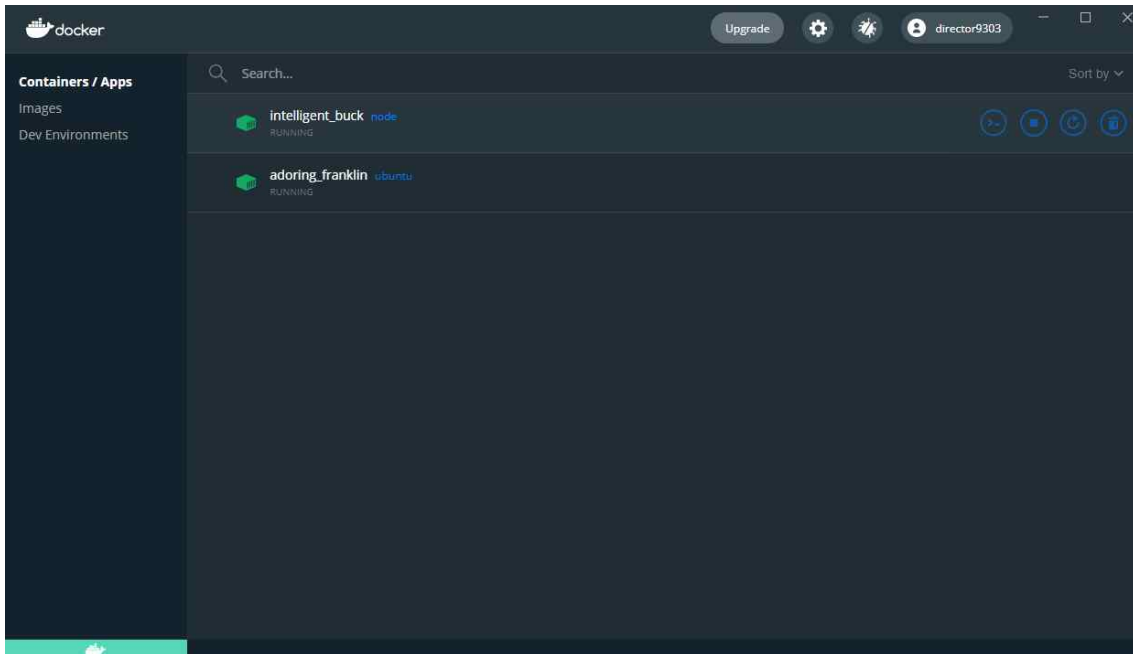
.exit 하고 나가서 다시 docker run -it node 명령어를 실행시키면, 이젠 내가 이미 node 이미지를 가지고있기때문에 다운받을필요없이 바로 실행시킨다.

이미지가 아닌 컨테이너를 보려면, docker ps

현재 실행중인 도커 컨테이너를 보여준다. 만약 전체 컨테이너를 보고싶다면

```
docker ps -a
```

작동중인 터미널을 꺼버리면 컨테이너가 사라진다. 이걸 좀 더 쉽게 관리를 하려면 gui 환경에서 관리하면 된다.



시작 정지로 관리하면 됨.

```

~/practice-docker/frontend on P master docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS      NAMES
fedfd03f2e7a   node     "docker-entrypoint.s..." 3 minutes ago  Up 3 minutes          intelligent_buck

```

id fedfd03f2e7a, 그리고 랜덤으로 생성된 이름인 intelligent_buck 로 제멋대로지어진 컨테이너 하나를 볼 수 있다.

이제 컨테이너 안을 좀 살펴보자.

`docker exec -it 컨테이너명 bash`

이 컨테이너 내에서 bash shell을 실행시킨다는 뜻

```

~/practice-docker/frontend on P master ?1 docker exec -it intelligent_buck bash
root@fedfd03f2e7a:/#

```

이제 우린 컨테이너안에 접속한 것. 컨테이너 자체가 리눅스였던것이다. 실제 이 컨테이너에 zsh을 깎다던지 하는 각종 조작을 할 수 있다. 정확하게는, 컨테이너 내부를 통해 가상의 리눅스 환경으로 들어간 것. 그래서 실제 리눅스는 아니라서 모든게 다 깔려있는건 아니다. 이 리눅스환경은 도커 데스크탑 프로그램으로 구현되고 있는 것일 뿐이다. 윈도우, 리눅스, 맥 어떤 os를 써도, 리눅스 가상환경으로 똑같이 동작한다. docker hub 에 ubuntu 이미지를 받아 컨테이너로 만들면 내 ubuntu 하나가 더 생기는 것이다.

그럼 이렇게 생각할수도 있다. ubuntu 이미지를 그대로 받아서 컨테이너로 만든 후, 적당한

서버 환경을 세팅하고 올리면 되지 않을까?

정답.

이렇게 실컷 가지고 논 우분투를 이제 이미지로 바꿔보자

```
$ docker commit 컨테이너이름 도커아이디/만들고자하는이미지이름
```

```
$ docker commit adoring_franklin director9303/jony_web_dev_setting
```

그럼 시간이 좀 지난후에 결과가 뜨고, 진짜 이미지가 된다.

푸시하려면 로그인해서 올리면 됨

```
$ docker login
```

```
$ docker push director9303/jony_web_dev_setting
```

근데 이런식으로 사용하는게 도커의 전부일까? 당연히 아니다. 보다 섬세한 컨테이너 활용을 위해 프로젝트폴더 안에 있는 Dockerfile 들을 사용할 것이다.

우선, 프로젝트폴더 안에는 frontend, backend, database 가 있다. 각각엔 모두 Dockerfile 이 있다.

Dockerfile은 나만의 이미지를 만들기위한 설계도라고 이해하면 된다. 옹? 노드 이미지를 이미 받아서 컨테이너로 만들었는데 왜 내걸 또 만드는거임?

컴퓨터에서 http-server를 돌리려면 http-server 가 전역으로, `npm install -g http-server` 로 깔려있어야.



```
frontend > Dockerfile > ...
1 FROM node:12.18.4
2
3 # 이미지 생성 과정에서 실행할 명령어
4 RUN npm install -g http-server
5
6 # 이미지 내에서 명령어를 실행할(현 위치로 잡을) 디렉토리 설정
7 WORKDIR /home/node/app
8
9 # 컨테이너 실행시 실행할 명령어
10 CMD ["http-server", "-p", "8080", "./public"]
11
12 # 이미지 생성 명령어 (현 파일과 같은 디렉토리에서)
13 # docker build -t {이미지명} .
14
15 # 컨테이너 생성 & 실행 명령어
16 # docker run --name {컨테이너명} -v $(pwd):/home/node/app -p 8080:8080 {이미지명}
```

line1: FROM 우리가 만들 이미지는 이 버전의 node 이미지에 덧붙여서 만드는 튜닝버전임을 의미

line4: RUN 이미지를 생성하는 과정에서 실행할 명령어. 여기서 npm을 통해 전역으로 http-server를 설치하는데, node가 깔려있으니깐 당연히 작동한다.

line7: WORKDIR 명령어를 실행할 위치. line10이 여기서 실행됨.

line10: CMD line7에서 실행될 명령.

RUN은 이미지를 생성하는 과정에서 실행되는 것. 즉, 이미지에서 컨테이너를 실행하는 시점엔 이미 http-server가 설치되어 있도록 하는 것.

CMD는 이미지로부터 컨테이너가 만들어져 가동될 때 기본적으로 바로 실행되는 명령어 line13: 이미지 생성 명령어. docker build -t {이미지명} .

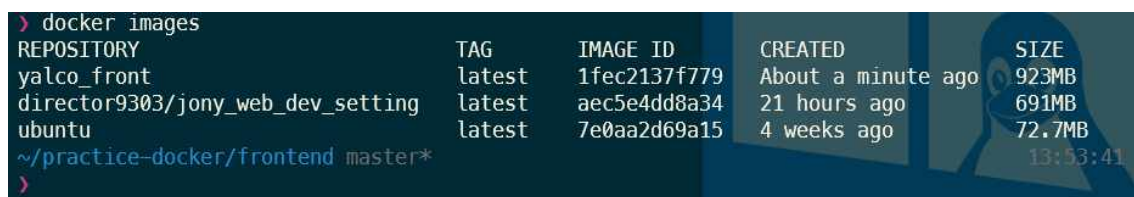
여기서 . 은 실행시키고자 하는 Dockerfile 의 상대경로를 의미한다. 파일명이 Dockerfile이면 따로 명시해 줄 필요가 없다.

line16: 컨테이너 생성 및 실행 명령어.

```
docker run --name {컨테이너명} -v $(pwd):/home/node/app -p 8080:8080 {이미지명}
```

이제 이 파일로 이미지를 만들어보자.

```
docker built -t yalco_front .
```



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
yalco_front	latest	1fec2137f779	About a minute ago	923MB
director9303/jony_web_dev_setting	latest	aec5e4dd8a34	21 hours ago	691MB
ubuntu	latest	7e0aa2d69a15	4 weeks ago	72.7MB

이미지가 성공적으로 만들어졌다. 이제 컨테이너를 만들면 될 것이라 생각된다.

지금 상황까진, WORKDIR 까지 성공적으로 만들어진것이라 볼 수 있다.

```
docker run --name {컨테이너명} -v $(pwd):/home/node/app -p 8080:8080 {이미지명}
```

이제 컨테이너 생성, 실행 명령을 하게 되면 CMD 부분이 실행될 것이다. 이때는

--name 옵션을 썼기 때문에 직접 컨테이너 명령을 지정해줘야한다. 왜냐면, 여러 도커 컨테이너를 만들것인데 각각의 역할이 다르기 때문이다.

-v 옵션은 volume 의 약자이다. 도커에서 볼륨은 컨테이너와 특정 폴더를 공유하는것을 말한다. 이게 뭘 소리냐면, 코드는 내 컴퓨터에서 짜는거니깐 컴퓨터(집)에 있는것인데, 코드를 짜서 그 파일을 거실탁자에 놓으면 그 탁자는 node.js가 일하는 컨테이너의 home칸 node 책상의 app 서랍과 연결되어 있는 것. 그러면 컨테이너가 언제, 몇 개가 만들어지던간에 각 컨테이너의 app 서랍에서는 거실에 둔 파일들로 얼마든지 서비스를 실행할 수 있다. 즉, \$(pwd), frontend 폴더 안의 내용들이, /home/node/app 으로 들어간다는 뜻. 이 이후에야 CMD의 내용을 실행시킬 수 있는데, 컨테이너안엔 원래 ./public이 없었는데 -v명령 이후에 컨테이너 안에 생겼기 때문이다.

-p는 포트다. 집의 내선번호를 컨테이너의 것과 연결해줘야된다. 여기서 둘 다 8080인데, 만약 같은 이미지의 다른 컨테이너를 실행시키면 충돌이 나니깐, 집의 내선번호만 바꿔줘야 된다.

8080:8080

8081:8080

...

마지막으로 컨테이너로 실행할 이미지를 지정해 준다.

127.0.0.1 은 localhost를 뜻한다.

방문자 기록부

백엔드와 연결되지 않았습니다.

짬!

이제 다른것도 시도해보자.

```
hyeonmin ~:/GoogleBackup/_coding/practice/practice-docker/frontend master docker stop $(docker ps -aq)
docker rm $(docker ps -aq)
docker image prune -a
```

이미지와 컨테이너를 삭제한다는 뜻

이제 Database로 가보자.

```

1 FROM mysql:5.7
2
3 # 이미지 환경변수들 세팅
4 # 실전에서는 비밀번호 등을 이곳에 입력하지 말 것!
5 # 서버의 환경변수 등을 활용하세요.
6 ENV MYSQL_USER mysql_user
7 ENV MYSQL_PASSWORD mysql_password
8 ENV MYSQL_ROOT_PASSWORD mysql_root_password
9 ENV MYSQL_DATABASE visitlog
10
11 # 도커환경에서 컨테이너 생성시 스크립트를 실행하는 폴더로
12 # 미리 작성된 스크립트들을 이동
13 COPY ./scripts/ /docker-entrypoint-initdb.d/
14
15 # 이미지 빌드 명령어 (현 파일과 같은 디렉토리에서)
16 # docker build -t {이미지명} .
17
18 # 실행 명령어 (터미널에 로그 찍히는 것 보기)
19 # docker run --name {컨테이너명} -it -p 3306:3306 {이미지명}
20
21 # 실행 명령어 (데몬으로 실행)
22 # docker run --name {컨테이너명} -p 3306:3306 -d {이미지명}

```

mysql 이미지를 개조할 것.

line6 - 9 : ENV 명령어로 생성될 컨테이너 안에 환경변수를 미리 지정. 어떤 사용자, 비밀번호, DB명을 사용할지 미리 정해두는것. 당연히 실전에선 보안요소를 이런 곳에 작성하지 않는다.

line13: COPY ./scripts/ 안에 있는 파일들을 이미지 내부의 /docker-entrypoint-initdb.d/ 폴더에 복사한다. 컨테이너로 실행될 때 작동함.

./scripts/ 를 자세히 보면, create_table.sql 과 insert_data.sql 이 있는데, 까보면 다음과 같다.

```

1 CREATE TABLE visits (
2   visitor_name varchar(25) not null,
3   visit_datetime datetime default NOW() not null
4 );
5
6
7 INSERT INTO visits (visitor_name) VALUES ('John');
8 INSERT INTO visits (visitor_name) VALUES ('Mark');
9 INSERT INTO visits (visitor_name) VALUES ('Robin');

```

테이블을 생성하고 기본적으로 들어갈 데이터들을 생성

그럼 COPY 와 volume 은 무슨 차이인가?

COPY는 RUN처럼 이미지를 생성하는 과정에서 해당 이미지 안에 특정 파일을 미리 넣어두는

것.

volume 은 CMD처럼, 컨테이너가 생성되어 실행될 때 그 내부의 폴더를 외부의 것과 연결하는 것.

여기선 왜 volume 대신 COPY를 썼냐면, script 파일들은 컨테이너 초기화 과정에 필요하기 때문에 image 안에 미리 넣어두는것이 맞다.

이제 터미널에서 명령어를 쳐보자.

아까와 조금 다른 점은, volume 이 없다는 것. 컨테이너명과 포트만 지정하면 되는데, 실전에선 데이터를 유지해야되니깐 데이터 폴더를 -v 옵션으로 집의 데이터 창고랑 볼륨공유를 할 것이다.

mysql은 3306포트가 기본이니 3306:3306 으로 연결하겠다.

`docker run --name {컨테이너명} -p 3306:3306 {이미지명}`

근데 이렇게 하면 지금 쓰고있는 터미널은 못쓴다. container 실행시키느라

돌아가고있기때문. 그래서 -d 옵션을 쓸건데, -d 는 demon 의 줄임말이고, 백그라운드에서 실행시키라는 뜻.

이렇게 하면 생성된 컨테이너의 id만 표시되고, 터미널은 그대로 사용 가능하다.

```
mysql> SHOW TABLES
-> ;
+-----+
| Tables_in_jony |
+-----+
| visits          |
+-----+
1 row in set (0.00 sec)

mysql> SELECT * FROM visits
-> ;
+-----+-----+
| visitor_name | visit_datetime |
+-----+-----+
| John         | 2021-05-28 10:47:34 |
| Mark         | 2021-05-28 10:47:34 |
| Robin        | 2021-05-28 10:47:34 |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```


DB 까보면 매우 잘 작동함

컨테이너가 일 잘하는지 살펴보려면

`docker logs -f` 컨테이너이름 써주면 됨.

```
ge cleaner thread priority can be changed. See the man page of setpriority().
2021-05-28T10:47:36.995684Z 0 [Note] InnoDB: Highest supported file format is Barracuda.
2021-05-28T10:47:37.003914Z 0 [Note] InnoDB: Creating shared tablespace for temporary table
s
2021-05-28T10:47:37.004131Z 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physica
lly writing the file full; Please wait ...
2021-05-28T10:47:37.017074Z 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
2021-05-28T10:47:37.017442Z 0 [Note] InnoDB: 96 redo rollback segment(s) found. 96 redo rol
back segment(s) are active.
2021-05-28T10:47:37.017468Z 0 [Note] InnoDB: 32 non-redo rollback segment(s) are active.
2021-05-28T10:47:37.018635Z 0 [Note] InnoDB: 5.7.34 started; log sequence number 12672337
2021-05-28T10:47:37.019361Z 0 [Note] InnoDB: Loading buffer pool(s) from /var/lib/mysql/ib_
buffer_pool
2021-05-28T10:47:37.020135Z 0 [Note] Plugin 'FEDERATED' is disabled.
2021-05-28T10:47:37.024123Z 0 [Note] InnoDB: Buffer pool(s) load completed at 210528 10:47:
37
2021-05-28T10:47:37.026901Z 0 [Note] Found ca.pem, server-cert.pem and server-key.pem in da
ta directory. Trying to enable SSL support using them.
2021-05-28T10:47:37.026939Z 0 [Note] Skipping generation of SSL certificates as certificate
files are present in data directory.
2021-05-28T10:47:37.027314Z 0 [Warning] CA certificate ca.pem is self signed.
2021-05-28T10:47:37.027356Z 0 [Note] Skipping generation of RSA key pair as key files are p
resent in data directory.
2021-05-28T10:47:37.027692Z 0 [Note] Server hostname (bind-address): '*'; port: 3306
2021-05-28T10:47:37.027779Z 0 [Note] IPv6 is available.
2021-05-28T10:47:37.027799Z 0 [Note] - '::' resolves to '::';
2021-05-28T10:47:37.027808Z 0 [Note] Server socket created on IP: '::'.
2021-05-28T10:47:37.033400Z 0 [Warning] Insecure configuration for --pid-file: Location '/v
ar/run/mysqld' in the path is accessible to all OS users. Consider choosing a different dir
ectory.
2021-05-28T10:47:37.038594Z 0 [Note] Event Scheduler: Loaded 0 events
2021-05-28T10:47:37.038949Z 0 [Note] mysqld: ready for connections.
Version: '5.7.34' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Serve
r (GPL)
2021-05-28T10:49:59.174068Z 2 [Note] Access denied for user 'root'@'localhost' (using passw
ord: N0)
```

실시간으로 로그 찍힘 `ctrl + c` 로 빠져나올수있음

마지막으로 백엔드쪽을 살펴보자.

```

1 FROM python:3.8.5
2
3 # 이미지 생성 과정에서 실행할 명령어
4 RUN pip3 install flask flask-cors flask-mysql
5
6 # 이미지 내에서 명령어를 실행할(현 위치로 같을) 디렉토리 설정
7 WORKDIR /usr/src/app
8
9 # 컨테이너 실행시 실행할 명령어
10 CMD ["python3", "backend.py"]
11
12 # 이미지 생성 명령어 (현 파일과 같은 디렉토리에서)
13 # docker build -t {이미지명} .
14
15 # 컨테이너 생성 & 실행 명령어
16 # docker run --name {컨테이너명} -v $(pwd):/usr/src/app -p 5000:5000 {이미지명}

```

line1: 파이썬 도커이미지 가져옴

line4: flask 관련 library 설치

line7: 디렉토리 설정

line10: 컨테이너 실행시 실행 명령어

그럼, 지금까지 frontend, database, backend 에 있는 Dockerfile로 미시적 환경을 구성한것을 살펴봤다. 이제부터 거시적으로 살펴볼 것이다.

지금까지 한 방식으로 부족한데, 매번 이런식으로 요소를 하나하나 이미지 다운받고, 적절한 명령어로 컨테이너를 실행해야되기에 번거롭다. 또한, 백과 프론트는 다른 컨테이너이기때문에 네트워크도 분리되어있어 통신 불가능.

요소들을 연결할 것이다. 거시적 설계도인 docker-compose.yml

```

1  version: '3'
2  services:
3    database:
4      # Dockerfile이 있는 위치
5      build: ./database
6      # 내부에서 개방할 포트 : 외부에서 접근할 포트
7      ports:
8        - "3306:3306"
9    backend:
10     build: ./backend
11     # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
12     volumes:
13       - ./backend:/usr/src/app
14     ports:
15       - "5000:5000"
16     # 환경변수 설정
17     environment:
18       - DBHOST=database
19    frontend:
20     build: ./frontend
21     # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
22     volumes:
23       - ./frontend:/home/node/app
24     ports:
25       - "8080:8080"

```

여기선 docker-compose를 사용할 것인데, docker-compose와 docker engine 의 버전이 잘 맞아야한다.

<https://docs.docker.com/compose/compose-file/compose-versioning/>

구글에 docker-compose compatibility matrix 치면 나오고, 알코 기준에선 compose 3버전이 가장 적합했지만 우리의 도커엔진은 20.10.6 이기 때문에 그에 맞는 3.8버전으로 바꿔주겠다.

Compose file format	Docker Engine release
Compose specification	19.03.0+
3.8	19.03.0+
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+

frontend, database, backend 의 세 요소는 docker-compose에선 services 항목의 내부 항목들로 들어간다.

```

version: '3.8'
services:
  database:
    # Dockerfile이 있는 위치
    build: ./database
    # 내부에서 개방할 포트 : 외부에서 접근할 포트
    ports:
      - "3306:3306"
  backend:
    build: ./backend
    # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
    volumes:
      - ./backend:/usr/src/app
    ports:
      - "5000:5000"
    # 환경변수 설정
    environment:
      - DBHOST=database
  frontend:
    build: ./frontend
    # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
    volumes:
      - ./frontend:/home/node/app
    ports:
      - "8080:8080"

```

services 안에 database, backend, frontend 항목명이 각 서비스들의 이름, 그리고 이들간의 네트워크에서 각 호스트명이 된다.

build는 Dockerfile이 있는 위치를 말한다. docker-compose.yml 파일의 위치로부터의 상대경로이다.

ports 는 내부개방포트:외부접근포트 이다. run 명령어를 실행시킬때마다 줘야할 옵션들을 여기에 작성하는 것.

backend와 frontend 엔 volume 항목이 추가되었고, 추가로 환경변수(environment)까지 설정했다. 이는 backend.py 코드에 해당 환경변수(DBHOST)가 들어가기 때문이다.

상수이기때문에, database 라는 이름이 들어가는데, database가 아니라 jony로 바뀌어도 잘 작동할 것이다.

```

app.config['MYSQL_DATABASE_HOST'] = os.getenv('DBHOST', 'localhost')

```

이제 터미널에서 docker-compose.yml 파일로 이동해서, docker-compose up 실행. 긴 시간이 지나고, 다음과 같이 나온다. 해당 터미널은 로그를 출력하기때문에 쓸 수 없다. 로그가 필요없다면 -d를 쓸 것.



슬프게도... 세 모듈 연동은 실패했다. localhost:8080 접속시 프론트에서 접속된것과 똑같은 증상이 뜨고, 연동은 안되었다. WSL의 문제일수도 있겠다는 생각은 들었다. (WSL로 도커를 쓸 때 대응해야되는 오류가 꽤 많다고 한다.) 그래서 윈도우 사용자는 AWS를 이용하는걸 추천했다.

참고로 리눅스에선 docker 설치 후 docker-compose도 따로 설치해야.