

Docker 실습

Window 환경, NodeJS와 Spring에서 실습

목차

- Docker 기본 문법 및 이미지 생성, 컨테이너 실행
- Docker 데이터 종류, 볼륨 그리고 바인드 마운트
- SSH
- AWS EC2에 컨테이너 실행하기
- AWS ECS에 컨테이너 실행하기
- Spring 프로젝트 도커 이미지 만들기

Docker 기본 문법 및 빌드

Docker file 문법

Docker file은 여러 줄의 Configuration으로 구성되어 있다.

- FROM: 사용할 base image를 선택함
- WORKDIR: 도커 이미지 안에서 어떤 경로에서 이것을 실행 할 것인지 명시함. 도커 내부적으로 디렉토리가 존재한다.
- COPY: 현재 WORKDIR 경로로 복사하기. 빈번히 변경되는 파일일수록 마지막에 작성해주는 게 좋다.
- RUN: 동작 커맨드를 입력
- ENTRYPOINT: CMD 대신 입력됨

```
1 FROM node:16-alpine
2
3 WORKDIR /app
4
5 COPY package.json package-lock.json ./
6
7 RUN npm install
8
9 ENTRYPOINT [ "node", "index.js" ]
```

Docker의 Directory

- Docker는 local 컴퓨터와 별개로 자체적인 directory를 가지고 있다.

How to access /var/lib/docker in windows 10 docker desktop?

Asked 3 years ago Modified 1 month ago Viewed 29k times

3. If you are seasoned enough, you may find the actual location of the virtual disk of all the data in your Windows directory.

 ext4.vhdx

```
C:\Users\your_name\AppData\Local\Docker\wsl\data\
```

- /var/lib/docker라는 저장 위치에 이미지가 저장되지만, 이 폴더는 vhdx라는 가상머신 안에 있다. Docker를 통한 간접 접근이 가능하다.

Docker file 문법 – COPY

```
COPY [--chown=<user>:<group>] <src>... <dest>  
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

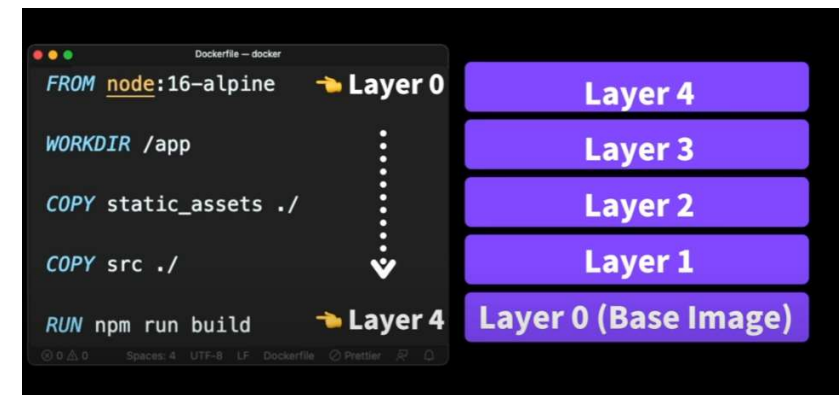
- COPY에는 두 가지 파라미터가 필요하다.
- 첫 번째 파라미터에는 외부 경로가 들어간다. 현재 local에서 넣어줄 파일이나 폴더를 지정한다. 단순 '.'점으로 표현하면 현재 도커 파일이 속해있는 모든 폴더와 파일을 지칭한다.
- 두 번째 파라미터에는 내부 경로가 들어간다. 도커 자체 내부 경로 앞서 언급한 vhdxf파일 안에 숨겨져 있음 './'로 표현하면 현재 WORKDIR을 경로로 지정한다. '/app'이라고 적으면 app이라는 폴더를 생성해서 거기에 복사해준다.

Docker file 문법 – RUN

- 터미널에 입력하는 동작 커맨드를 넣을 수 있다.
- nodeJS의 경우 npm install을 실행
- 커맨드에 node index.js를 넣는 것은 바람직하지 않다. 왜냐하면 도커 파일은 이미지 설정을 위한 문법만 들어가야 함. 이미지를 실행하는 게 아니라 이미지를 기반으로 컨테이너를 실행해야 한다.
- 실행 커맨드를 도커 파일에 넣고 싶으면 RUN을 사용하는 게 아니라 CMD를 사용해야 한다.
- CMD ["node", "index.js"]를 넣으면 컨테이너가 실행될 때 이 커맨드가 실행된다. => 즉 서버를 띄운다. 'CMD' 대신 'ENTRYPOINT'를 쓸 수도 있다.

Docker file layer

- Docker file은 layer로 구성되어 있다.
- 이는 Docker의 명령어가 cache를 사용하기 때문. 변경되지 않은 layer는 cache에 있는 결과를 그대로 가져오기 때문에 빠르게 실행된다.
- 더 낮은 layer가 바뀌면 그 보다 높은 후속 layer들은 값이 변경되지 않더라도 다시 연산이 실행된다. 캐시를 사용하지 않는다.
- 그러므로 자주 바뀌는 설정은 높은 후속 layer에 위치 시키는 게 좋다.



Docker image 만들기

- "docker build"라는 명령어를 사용하면 docker file을 참조하여 도커 이미지를 만든다.

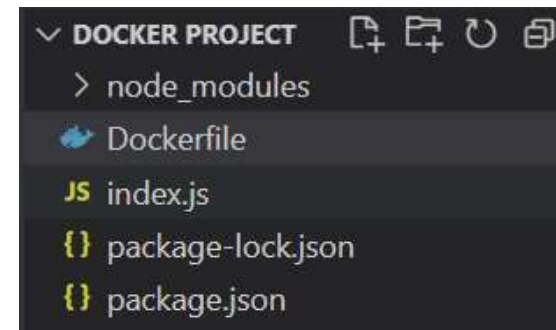
```
PS D:\docker project> docker build -f Dockerfile -t docker-test-image .
```

- 도커 커맨드

- f: 도커 파일 이름

- t: 만들어질 도커 이미지 이름

- "."은 현재 도커 파일이 있는 디렉토리 위치를 의미한다. 현재는 최상위 디렉토리에 도커 파일이 있음



Docker Command error

- “docker build” 했는데, 아무 것도 실행되지 않음.
- 에러 메시지: the docker daemon is not running.
- Docker Desktop 응용프로그램을 실행하지 않아서 그랬던 것.
- Docker 프로그램이 돌아가야 docker build 할 수 있다.

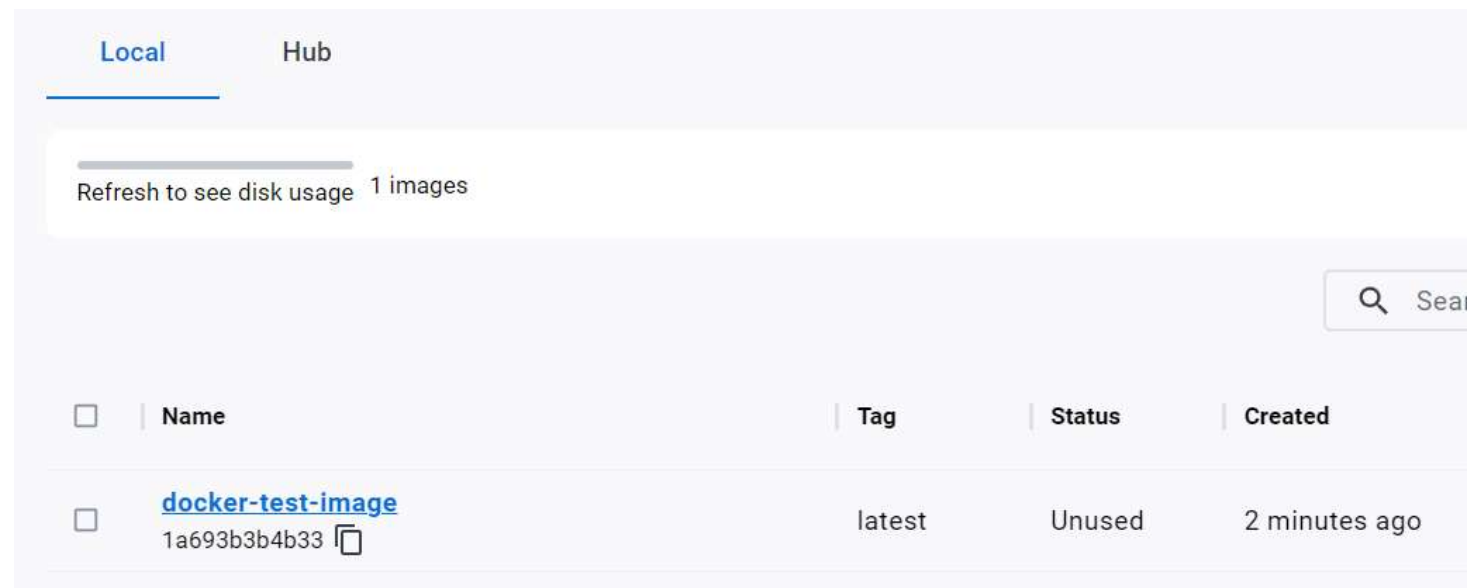
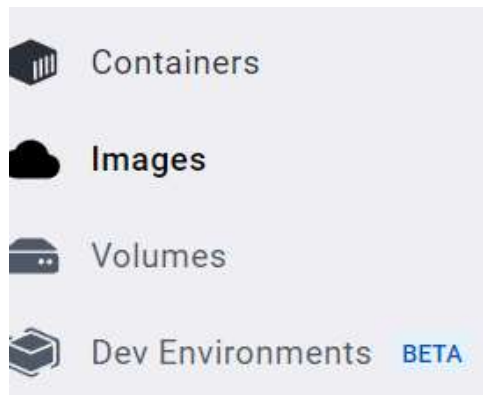
```
PS D:\docker project> docker build -f Dockerfile -t docker-test-image .  
error during connect: This error may indicate that the docker daemon is not running.:  
mswap=0&networkmode=default&rm=1&shmsize=0&t=docker-test-image&target=&ulimits=null&v  
the file specified.
```



Docker image 만들기

```
PS D:\docker project> docker build -f Dockerfile -t docker-test-image .
```

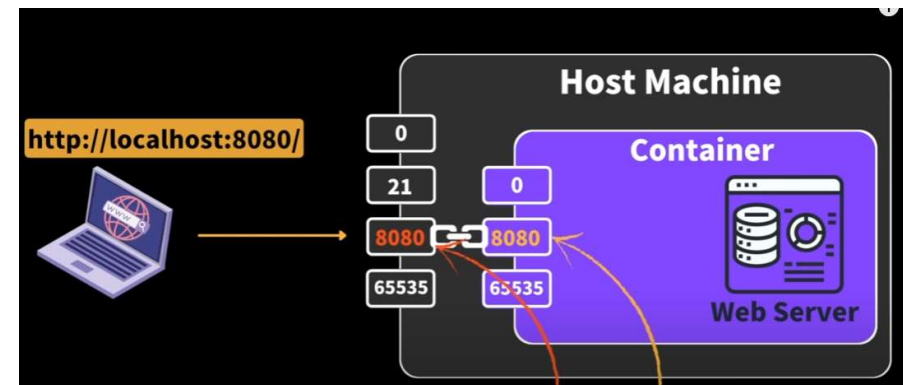
- “docker build”라는 명령어가 온전히 작동되면 아래와 같이 Docker Desktop 프로그램의 이미지 탭에서 볼 수 있음



Docker Container 실행

- Docker image를 이용해서 Container를 실행하기
- -d: detached를 의미. 터미널이 기다리지 않고, 다른 일을 할 수 있도록 함. 안 써도 됨.
- -p: publish를 의미. Host의 port와 container의 port를 연결해주는 작업

```
PS D:\docker project> docker run -d -p 8080:8080 docker-test-image  
439a740e7beb84e47314b15068dedf22f40998741d2fca18bd9746aa9d733b2c
```



Docker Container error

- "docker ps"를 실행하면 현재 작동중인 container를 볼 수 있다.
- 하지만 아무 것도 돌아가지 않았다.

```
PS D:\docker project> docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
PS D:\docker project> docker ps
```

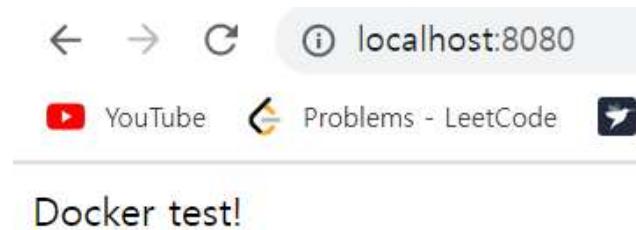
```
6
7  RUN npm ci
8
9  COPY index.js .
10
11 ENTRYPOINT [ "node", "index.js" ]
12
```

- 원인은 실행 파일을 도커에 추가하지 않아서 에러가 나서 Exit()
- COPY index.js . 를 추가하고 다시 이미지를 빌드하니 정상 작동됨

```
PS D:\docker project> docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED        STATUS        PORTS        NAMES
439a740e7beb   docker-test-image  "node index.js"         4 seconds ago  Up 3 seconds  0.0.0.0:8080->8080/tcp
```

Docker Container 통해 서버 실행

- Local 터미널에서 직접 'npm run'을 하지 않았는데, Docker가 컨테이너를 띄워서 실행했기 때문에 node 서버가 동작한다.



- 이를 조금 응용해서 생각하면, local 컴퓨터에 node를 설치하지 않아도 Docker를 사용하여 node를 실행할 수 있다는 말이다.
- 혹은 local 컴퓨터에 설치된 node와는 다른 버전의 node를 도커 상에서 실행 시킬 수 있다.

(참고) Docker 이미지의 특성

- 소스 코드를 수정해도 이미지에는 자동으로 반영되지 않는다. 변경된 소스 코드를 반영하려면 이미지를 다시 새로 Build 해야 한다.
- 이미지는 읽기 전용. 처음에 build할 때만 코드가 반영되고, 이후로는 수정할 수 없음.

(참고) 컨테이너 관련 docker 명령어

- Docker stop "name"을 하면 해당 name을 가지고 있는 컨테이너를 중지시킨다.
- Docker ps -a 를 실행하면, 중지된 컨테이너도 포함하여 보여준다.
- Docker start "name"을 하면 해당 name을 가지고 있는 중지된 도커 컨테이너를 재실행 해준다.
- Docker run을 할 때 커맨드에 '--rm'을 추가하면 이 컨테이너가 실행되고 나서 중지될 때 자동으로 삭제가 된다.
- Docker run에서 -name "이름"을 추가하면, 해당 컨테이너의 이름이 "이름"으로 설정된다.

Docker 데이터 종류, 볼륨 그리고
바인드 마운트

Docker 데이터 종류

- 세 종류의 데이터
 - application(code + 환경): 읽기 전용. 이미지에 저장된다. 한 번 만들어지면 변경 불가.
 - 임시 앱 데이터: 읽기, 쓰기 가능. 컨테이너에 저장된다. 메모리에 저장되고 컨테이너 동작 중에서만 유효함.
 - 영구 앱 데이터: 읽기, 쓰기 가능. 컨테이너가 종료되어도 사라지면 안 되는 데이터. 컨테이너에 저장하지만, 볼륨의 도움을 받는다.

컨테이너에서의 저장

- 컨테이너 어플리케이션을 실행해서 데이터를 저장하면 컨테이너 상에서 저장이 일어나지만 이것은 컨테이너 저장소에 저장된 것이다. 컨테이너가 종료가 되면 데이터는 사라진다.

Your Feedback

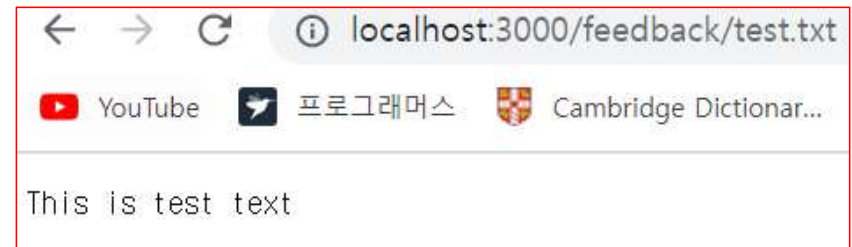
Title

Document Text

This is test text

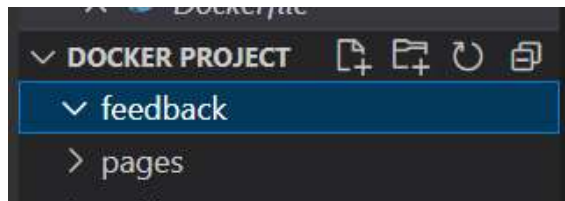
Save

저장하면 컨테이너 상에서는 저장되었지만



컨테이너에서 저장의 한계

- 컨테이너 어플리케이션에서 데이터를 저장해도 로컬 환경에서 저장되진 않는다.
- 이미지 빌드 이후에는 로컬 파일과 컨테이너 사이의 연결이 없다. 이미지가 한 번 빌드가 되면 로컬 파일의 변경점을 알지도 못하고 영향을 주지도 못함.
- 로컬 파일 시스템과 컨테이너는 단절되어 있다.
- 컨테이너가 삭제되면 저장된 모든 데이터가 날라간다.



로컬 컴퓨터의 feedback 폴더에는 아무것도 저장되지 않음

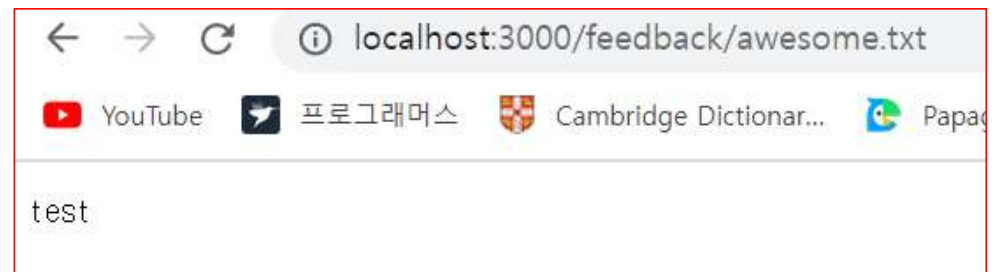
볼륨

- 볼륨은 호스트 컴퓨터에 있는 폴더.
- 볼륨을 통해 컨테이너 내부 폴더와 컨테이너 외부 폴더를 연결 할 수 있다.
- 컨테이너에 볼륨을 추가할 수 있는데, 컨테이너가 제거되어도 볼륨은 사라지지 않는다.

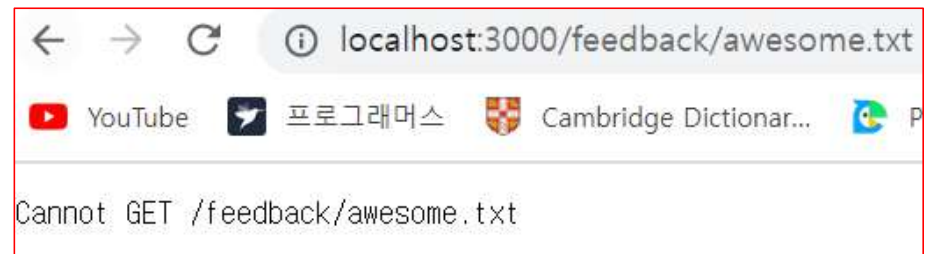
볼륨 사용 방법

- VOLUME ["/app/feedback"] <- 도커 이미지 내 저장 위치를 지정한다.

```
8
9 COPY . .
10
11 EXPOSE 80
12
13 VOLUME [ "/app/feedback" ]
14
15 CMD [ "node", "server.js" ]
```



- 하지만 이렇게만 사용하면 도커 컨테이너가 종료되면 볼륨이 사라진다.



볼륨 종류

- 볼륨에는 크게 두 가지 타입이 있음
 - 익명 볼륨: 컨테이너 종료 시 사라짐, 특정 컨테이너에 종속됨.
 - 네임드 볼륨: 컨테이너 종료 후에 볼륨이 유지됨, 하나의 컨테이너에만 연결되지 않음.
- 네임드 볼륨을 사용하면 컨테이너 재시작하면 볼륨이 복구되고 폴더가 복구되어 해당 폴더 데이터를 계속 사용할 수 있다.
- Dockerfile에서는 명명된 볼륨을 생성할 수는 없다.
- 대신 컨테이너가 실행할 때 명명된 볼륨을 생성해야 한다.

명명 볼륨 생성 방법

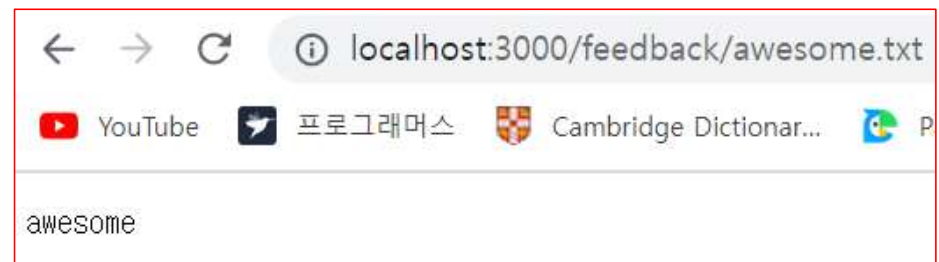
- 일단 도커 파일에 있는 볼륨 문법은 모두 제거한다.
- 이미지를 빌드하고 컨테이너를 run할 때 '-v'를 추가한다.

```
PS D:\docker\docker project> docker run -d -p 3000:80 --rm --name feedback-app -v feedback:/app/feedback feedback-node:volumes  
c52b3527e38e6e41efba5dc9b4909124ebaa3ccb1784215c4af6a17c5ccb8437
```

```
PS D:\docker\docker project> docker stop feedback-app  
feedback-app
```

```
PS D:\docker\docker project> docker run -d -p 3000:80 --rm --name feedback-app -v feedback:/app/feedback feedback-node:volumes  
a721d996fd73164e919b52746542e55fd99062faabd1e96bc6bed46b96621dc3
```

- 다시 실행하면 데이터가 남아있음
- 명명 볼륨은 사라지지 않기 때문



바인드 마운트

- 도커 컨테이너에서 데이터를 보존하기 위해서는 두 가지 방법이 있다. 하나는 볼륨, 다른 하나는 바인드 마운트
- 볼륨의 위치는 도커에 의해 관리되고 개발자는 그 위치를 알 수 없으나, 바인드 마운트의 위치는 개발자에 의해 관리된다.
- 호스트 컴퓨터의 특정 위치에 바인드 마운트를 설정하면 컨테이너는 항상 최신 코드에 액세스 할 수 있음
- 그러므로 바인드 마운트는 영구적이고 편집 가능한 데이터에 적합.
- 바인드 마운트를 사용하면 내부 HTML 소스를 변경한 것을 이미지로 빌드하지 않아도 실시간으로 컨테이너에 반영이 된다.
- 바인드 마운트를 사용하면 특정 데이터를 호스트 머신 파일 시스템에서 아웃소싱 할 수 있다.

바인드 마운트 사용 방법

- 기본 사용 문법은 '-v'를 사용하는 것은 볼륨과 똑같다. 하지만 전체 코드를 복사해야 하기 때문에 도커 이미지 내 경로는 "/app"으로 지정해주고, 호스트 주소 부분에는 호스트 디렉토리 위치를 넣어준다.
- 공백, 특수문자 에러를 피하기 위해 "(따옴표)"를 써서 주소 구문 전체를 감싸준다.

```
> docker run -d -p 3000:80 --rm --name feedback-app -v feedback:/app/feedback -v  
"D:/docker/docker project:/app" feedback-node:volumes  
4764ed0da64f4e58c094d365b6f1c0dc773e25a30cb91e537a98bdf8c2a98d3b
```

[ERROR] 컨테이너가 바로 사라짐

- 컨테이너를 실행시켰는데 곧바로 컨테이너는 사라짐.

```
PS D:\docker\docker project> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

- 로컬에 노드 모듈이 없는데, 이것이 바인드 마운트를 통해 도커 이미지 내 폴더에 덮어씌워졌기 때문



사이트에 연결할 수 없음

localhost에서 연결을 거부했습니다.

다음 방법을 시도해 보세요.

- 연결 확인
- 프록시 및 방화벽 확인

ERR_CONNECTION_REFUSED

새로고침

해결방법: 익명의 볼륨 사용

- 익명의 볼륨을 사용해서 이미지 상의 /app/node-modules 폴더를 지정해준다. 그러면 컨테이너 정상작동.

```
PS D:\docker\docker project> docker run -d -p 3000:80 --rm --name feedback-app -v feedback:/app/feedback -v "D:/docker/docker project:/app" -v /app/node_modules feedback-node:volumes 9bbab69f1be3c39277efbcf64d1e3656eed6e175ec6542e19447037ab5ce10d8
```

```
PS D:\docker\docker project> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
	NAMES		
9bbab69f1be3	feedback-node:volumes	"docker-entrypoint.s..."	3 seconds ago

- 컨테이너 실행할 때 충돌이 있는 경우 더 긴 내부 경로를 우선시한다.
- 즉, npm install 할 때 생성되는 node_modules 폴더는 살아남는다. 덮어씌우기 당하지 않음.

익명 볼륨, 명명 볼륨, 바인드 마운트 문법

- 익명 볼륨: 아무 콜론 없이 -v만 사용

`docker run -v /app/data...`

- 네임드 볼륨: -v 사용하고 콜론 사용하여 콜론 앞에 경로가 아닌 것이 붙음

`docker run -v data:/app/data...`

- 바인드 마운트: -v 사용하고 콜론 사용하여 콜론 앞에 로컬 머신 경로가 붙음

`docker run -v /path/to/code:/app/code...`

SSH

SSH

- SSH: secure shell의 줄임말. Local 에서 원격 호스트로 연결하기 위한 프로토콜.
- HTTP나 FTP처럼 응용 계층 프로토콜이고, client와 server가 통신하기 위한 프로토콜이지만, SSH는 한 쌍의 키를 사용하여 보안 접속을 한다는 차이점이 존재한다.
- Public key를 통해 전송 전에 내용을 암호화하고, 전송이 완료되면 private key를 통해 내용을 복호화한다.
- 암호화하여 데이터를 전송하기 때문에 통신 중간에 패킷을 가로채어도 이해할 수 없는 문자가 보여진다.
- SSH의 기본 포트는 22번이다 (HTTP의 기본 포트는 80번)

(+추가) HTTPS와 SSL

- HTTP는 데이터를 암호화하지 않고 전송하므로 악의적으로 감청이나 데이터 탈취를 당할 수도 있다.
- 데이터 암호화를 위해 HTTP + SSL(또는 TLS)를 결합하여 사용하는데, 이것을 HTTPS라고 한다.
- SSL은 사실 예전 버전이고 요즘은 TLS를 사용하여 암호화한다.
- SSL(secure socket layer)
- MAC이나 Linux에는 내장되어 있다.
- Windows에는 WSL 2나 PuTTY를 설치해야 한다



HTTPS는 TCP 위에 놓인 SSL 위의 HTTP

(+ 추가) SSH vs SSL

- 공통점: 보안을 위해 암호화를 사용한다. (암호화 기법은 각자 다름)
Public Key Infrastructure를 사용한다.

SSH	SSL
Command 동작을 위해 사용	정보 전송을 위해 사용
22번 포트 사용	443번 포트 사용
Client는 authentication이 필요함 Username, password 필요	서버 측 authentication만 있으면 됨 Username, password 없음

Container를 EC2에서 실행하기

EC2 인스턴스 생성

- Docker를 올릴 EC2 인스턴스를 생성하고 키 페어를 저장

인스턴스 시작

시작하려면 클라우드의 가상 서버인 Amazon EC2 인스턴스를 시작하십시오.

인스턴스 시작 ▼

서버 마이그레이션 ↗

참고: 인스턴스는 아시아 태평양 (서울) 리전에서 시작됩니다.

인스턴스 시작 정보

Amazon EC2를 사용하면 AWS 클라우드에서 실행되는 가상 머신 또는 인스턴스를 생성할 수 있습니다. 아래의 간단한 단계에 따라 빠르게 시작할 수 있습니다.

이름 및 태그 정보

이름

docker-test-server

[추가 태그 추가](#)

▼ 키 페어(로그인) 정보

키 페어를 사용하여 인스턴스에 안전하게 연결할 수 있습니다. 인스턴스를 시작하기 전에 선택한 키 페어에 대한 액세스 권한이 있는지 확인하세요.

키 페어 이름 - 필수


docker_test

 새 키 페어 생성

로컬 디스크 (D:) > docker > docker aws ec2 key

이름

수정한

 docker_test.pem

2023-0:

WSL 2 설치

- SSH 연결하여 로컬에서 가상 머신을 연결하기 위해 WSL 2가 필요하다.
- 윈도우에서 제공하는 WSL 2 설치법을 따라 Ubuntu 설치
- 터미널에서 WSL을 실행

인스턴스에 연결 정보

다음 옵션 중 하나를 사용하여 인스턴스 i-03f0da491e7003c93 (docker-test-server)에 연결

EC2 인스턴스 연결



Session Manager

SSH 클라이언트


EC2 직렬 콘솔

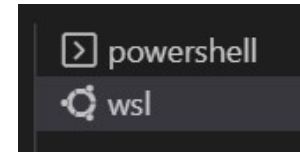
인스턴스 ID

 i-03f0da491e7003c93 (docker-test-server)

1. SSH 클라이언트를 엽니다.
2. 프라이빗 키 파일을 찾습니다. 이 인스턴스를 시작하는 데 사용되는 키는 docker_test.pem입니다.
3. 필요한 경우 이 명령을 실행하여 키를 공개적으로 볼 수 없도록 합니다.
 `chmod 400 docker_test.pem`
4. 퍼블릭 DNS을(를) 사용하여 인스턴스에 연결:
 `ec2-43-201-72-202.ap-northeast-2.compute.amazonaws.com`

예:

 `ssh -i "docker_test.pem" ec2-user@ec2-43-201-72-202.ap-northeast-2.compute.amazonaws.com`



[ERROR] unprotected private key file






- AWS에 나와 있는 SSH 명령을 그대로 가져다 썼으나 에러가 발생.
- Private key이 너무 공개되어 있어서 생긴 문제

```
master@LAPTOP-4EF20ME7:/mnt/d/docker/docker project$ chmod 400 docker_test.pem
master@LAPTOP-4EF20ME7:/mnt/d/docker/docker project$ ssh -i "docker_test.pem" ec2-
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                WARNING: UNPROTECTED PRIVATE KEY FILE!                @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0555 for 'docker_test.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "docker_test.pem": bad permissions
```

[ERROR] unprotected private key file

- 하지만 stack overflow를 아무리 찾아봐도 해결법은 나오지 않았다.
- 이미 chmod 400을 해줬는데도 해결이 안 됨.
- SSH 커맨드 관련한 지식이 있어야 해결 가능할 듯 함.


1863





Answer recommended by AWS

The problem is a wrong set of permissions on the file.

It is easily solved by executing: `chmod 400 mykey.pem`

This solution is taken from AWS instructions:

Your key file must not be publicly viewable for SSH to work. Use this command if needed:

```
chmod 400 mykey.pem
```

400 protects it by making it read only and only for the owner.

EC2 인스턴스 연결

- 굳이 SSH로 연결할 필요 없는 듯.
- 그냥 연결 버튼 누르면 알아서 연결해주고 브라우저 상에서 EC2 머신에 리눅스 커맨드 입력할 수 있게 해준다.

[EC2 인스턴스 연결](#) | [Session Manager](#) | [SSH 클라이언트](#) | [EC2 직렬 콘솔](#)

인스턴스 ID
i-0f7e6a401dd003939 (docker-test)

퍼블릭 IP 주소
3.35.176.56

사용자 이름
인스턴스를 시작하는 데 사용되는 AMI에 정의된 사용자 이름을 입력합니다. 사용자 지정 사용자 이름을 정의하지 않은 경우 기본 사용자 이름인 ec2-user를(를) 사용합니다.

참고: 대부분의 경우 기본 사용자 이름 ec2-user는(는) 정확합니다. 하지만 AMI 사용 지침을 읽고 AMI 소유자가 기본 AMI 사용자 이름을 변경했는지 확인하십시오.

[취소](#) [연결](#)

```
 _ | _ | _ )
 _ | ( _ | /
 _ | \ _ | _ |
                                Amazon Linux 2 AMI

https://aws.amazon.com/amazon-linux-2/
[ec2-user@ip-172-31-11-71 ~]$
```

EC2 커맨드 입력

- EC2 머신 필수 패키지 업데이트

```
[ec2-user@ip-172-31-11-71 ~]$ sudo yum update -y
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
amzn2-core
No packages marked for update
```

- EC2 머신에도 도커 설치

```
[ec2-user@ip-172-31-11-71 ~]$ sudo amazon-linux-extras install docker
Installing docker
```

- EC2 에서 도커 동작시키기

```
[ec2-user@ip-172-31-11-71 ~]$ sudo service docker start
Redirecting to /bin/systemctl start docker.service
```


도커 이미지를 리모트 머신에 올리기

- 크게 두 가지 방법

1. Docker file을 포함한 모든 소스코드를 원격 장치에 복사 -> 이미지 구축을 원격 장치에서 함.
 2. Local에서 이미지를 빌드하고 그 이미지를 원격 장치에 배포
- 2번의 방법이 일반적이다. 1번은 번거로운 작업이 많음. 그래서 이미지를 그냥 hub에 올려서 사용하는 게 낫다.

Docker Hub에 이미지 올리기

- 반드시 저장소 이름을 태깅해줘야 push 할 수 있다.

```
PS D:\docker\docker project> docker tag docker-test-image millwheel/node-docker-test
```

- 저장소 이름 사용해서 docker hub에 push

```
PS D:\docker\docker project> docker push millwheel/node-docker-test
Using default tag: latest
The push refers to repository [docker.io/millwheel/node-docker-test]
9692008c2a7a: Pushed
187888a0b65b: Pushed
383dbcff459b: Pushed
```

Docker commands

[Public View](#)

To push a new tag to this repository,

```
docker push millwheel/node-docker-test:tagname
```

EC2에서 도커 이미지 불러오기

- EC2도 똑같은 컴퓨터이므로 도커 기본 문법을 사용해서 이미지를 불러들이면 된다.
- Local 환경에 이미지가 없으면 Docker hub을 방문해서 이미지를 자동으로 다운로드함. Run을 실행했으므로 컨테이너가 자동으로 돌아감.
- Sudo를 붙여줘야 실행된다.

```
[ec2-user@ip-172-31-11-71 ~]$ sudo docker run -d --rm -p 8080:8080 millwheel/node-docker-test
Unable to find image 'millwheel/node-docker-test:latest' locally
latest: Pulling from millwheel/node-docker-test
63b65145d645: Pull complete
a67f65df360b: Pull complete
6112f742730b: Pull complete
```

```
[ec2-user@ip-172-31-11-71 ~]$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
e62ac9c08df0	millwheel/node-docker-test	"node index.js"	2 minutes ago

퍼블릭 ip에 접근해서 실행을 확인

- 그냥 퍼블릭 ip 긁어와서 주소창에 넣고 접속하면 아무 것도 실행되지 않는다. 왜냐하면 EC2 인스턴스는 기본적으로 웹과 연결이 끊어져 있기 때문이다. (보안 때문에 끊어져 있음. SSH로만 접근 가능)
- 외부에서 접속 할 수 있도록 보안 그룹을 설정해줘야 한다.

보안 그룹

- 인바운드: 서버가 내부 데이터를 보여줄 때 제한. 기본적으로 SSH만 열려 있다. Source 0.0.0.0/0은 전세계에 열려 있다는 것을 의미.

인바운드 규칙 (1/1)					
<input type="text" value="Q 보안 그룹 규칙 필터"/>					
IP 버전	유형	프로토콜	포트 범위	소스	
IPv4	SSH	TCP	22	0.0.0.0/0	

- 아웃 바운드: 서버가 외부 데이터를 가져오는 제한. 기본적으로 모든 트래픽을 허용하므로 모든 프로토콜을 이용해서 아무 데이터를 가져올 수 있다.

아웃바운드 규칙 (1/1)						
<input type="text" value="Q 보안 그룹 규칙 필터"/>						
안 그룹 규칙 ID	IP 버전	유형	프로토콜	포트 범위	대상	
r-0ab9af2b740f1a2ac	IPv4	모든 트래픽	전체	전체	0.0.0.0/0	

인바운드 규칙 추가

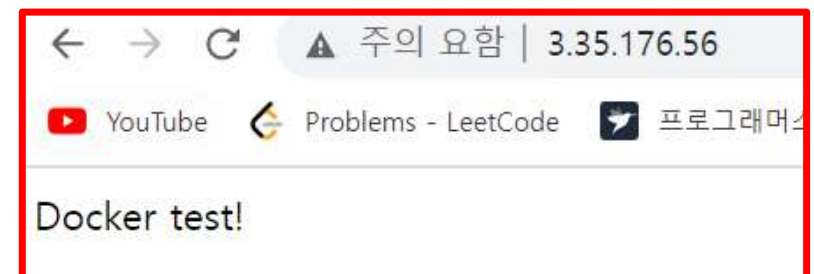
- 인바운드 규칙에 HTTP를 추가해준다. 주소는 anywhere로 설정.

HTTP TCP 80 Anywh... 0.0.0.0/0 X

어플리케이션에 지정된 포트 번호
(코드 상의 포트 번호)

```
[ec2-user@ip-172-31-11-71 ~]$ sudo docker run -d --rm -p 80:8080 millwheel/node-docker-test  
9457ebb8e5cd37b57d182ff2e03401a3d2df3893fab471626bd6ff99a56602a3
```

현재 호스트의 포트 번호



EC2 사용 방식의 단점

- 보안도 신경 써야 하고, 방화벽을 관리해야 한다
- 트래픽이 많이 발생하면 더 강력한 하드웨어로 교체해야 한다.
- 운영체제를 최신 상태로 유지해야 한다.
- EC2 원격 컴퓨터의 구성에 대한 책임을 전적으로 다 지게 된다.
- SSH를 통해 EC2에 접속하여 도커를 설치해야 한다. <- 상당히 번거로운 작업임

Container를 ECS에 올리기

ECS

- AWS의 Fully managed container service
- ECS 사용하면 EC2에서 해야 하는 번거로운 작업을 피할 수 있다.
- 특정 프로그램을 설치할 필요도 없고, 운영체제를 최신 상태로 유지할 필요도 없다.
- 생성 관리, 업데이트, 모니터링, 스케일링이 모두 자동으로 된다.

AWS ECS 생성하기

- 사용자 정의 컨테이너 시작
- 컨테이너 이름은 임의로 작성하고, 이미지 란에는 이미지가 위치한 곳 정보를 적어야 한다. 이미지가 docker hub에 있다면 저장소 이름만 써도 되고 docker hub이 아닌 다른 저장소를 쓴다면, 호스팅 도메인을 함께 적어줘야 한다.

sample-app

이미지 : httpd:2.4
메모리 : 0.5GB (512)
CPU : 0.25 vCPU (256)

nginx

이미지 : nginx:latest
메모리 : 0.5GB (512)
CPU : 0.25 vCPU (256)

tomcat-webserver

이미지 : tomcat
메모리 : 2GB (2048)
CPU : 1 vCPU (1024)

custom 구성

이미지 : --
메모리 : --
CPU : --

컨테이너 이름*

node-demo

이미지*

millwheel/node-docker-test

AWS ECS 생성하기 - 포트 매핑

- 포트 매핑에는 80을 넣어줘야 한다. 아래와 같이 다른 포트 번호(8080)을 넣어주면 작동이 안 된다.
- 코드에서도 80으로 설정해줘야 하고, AWS 포트 매핑 설정에서도 80으로 설정해줘야 한다.

```
app.listen(8080, () => console.log('is running'))
```

포트 매핑 컨테이너 포트

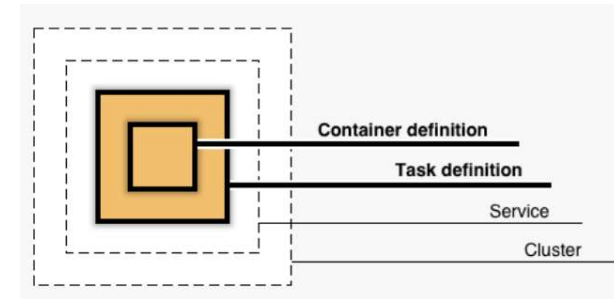
8080

tcp

+ 포트 매핑 추가

AWS ECS Task Definition

- Task Definition은 AWS에서 컨테이너를 시작하는 방법을 정의하는 것. 서버 스펙을 정의한다.
- Task는 EC2 인스턴스와 비슷하다. EC2와의 차이점은 ECS Task는 서버를 자동으로 관리해준다는 부분



작업 정의

[편집](#)

작업 정의는 애플리케이션에 대한 블루프린트이며, 속성을 통해 하나 이상의 컨테이너를 설명합니다. 일부 속성은 작업 수준에서 구성되지만 대부분의 속성은 컨테이너별로 구성됩니다.

태스크 정의 이름	first-run-task-definition	i
네트워크 모드	awsvpc	i
작업 실행 역할	새로 생성	i
호환성	FARGATE	i
작업 메모리	0.5GB (512)	
작업 CPU	0.25 vCPU (256)	

AWS ECS Service and cluster

- 서비스: 여러 Task를 하나로 묶어서 실행하는 단위
- 클러스터: 서비스가 실행되는 전체 네트워크. 하나의 클러스터에 여러 컨테이너를 넣을 수 있다.

클러스터 구성

Fargate 클러스터의 인프라는 AWS에서 완전하게 관리됩니다. 개별 Amazon EC2 인스턴스 관리 및 구성 없이 컨테이너가 실행됩니다

Fargate와 표준 ECS 클러스터의 차이점을 알아보려면 [Amazon ECS 설명서](#)를 참조하십시오.

클러스터 이름

default

클러스터 이름은 리전마다 계정별로 고유합니다. 최대 255개의 문자(대문자 및 소문자), 숫자, 하이픈 및 밑줄이 허용됩니다.

VPC ID 자동으로 새로 생성



서브넷 자동으로 새로 생성



AWS ECS 어플리케이션 실행

- Task에 있는 public ip를 가져와서 주소 창에서 실행하면 정상 작동된다. 아무런 설치가 필요 없음

네트워크

네트워크 모드 awsvpc

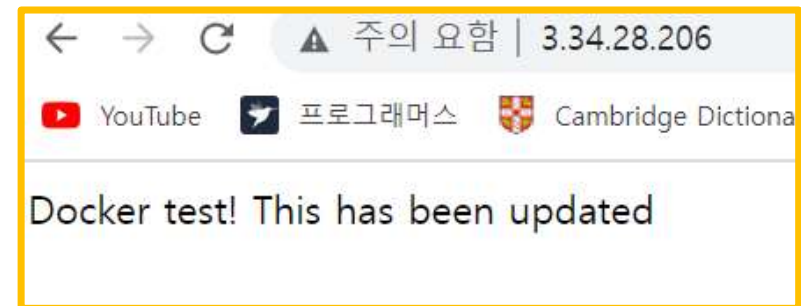
ENI ID [eni-021ff9ea2456ee9b0](#)

서브넷 ID subnet-02375f50a9a3f40c6

프라이빗 IP 10.0.1.165

퍼블릭 IP 3.34.28.206

Mac 주소 06:98:79:bf:25:62



Spring project docker image

Spring project home controller 생성

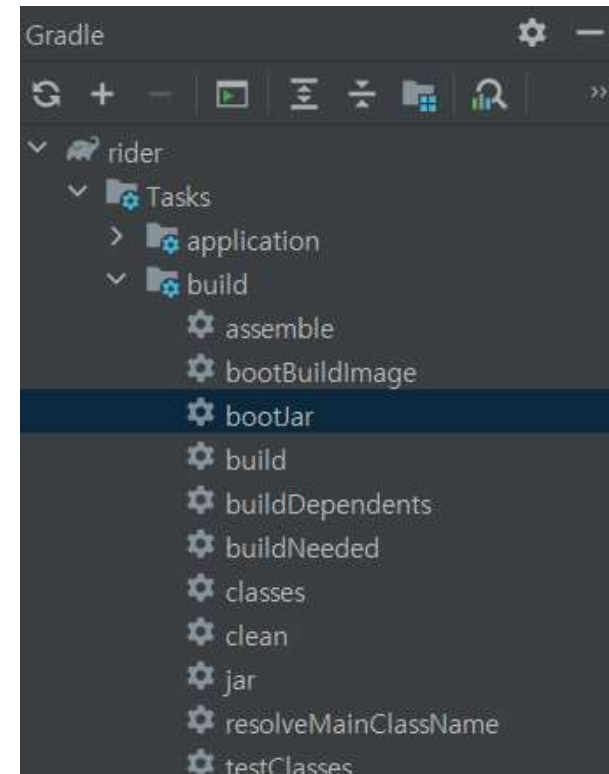
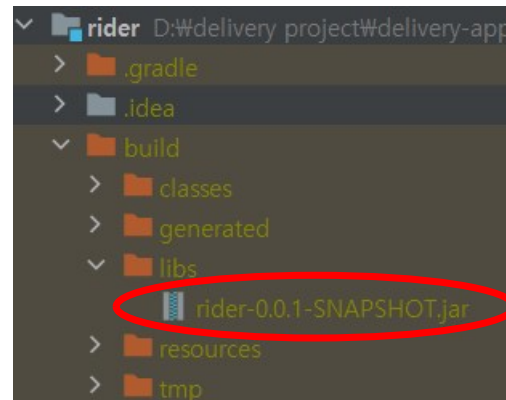
- 제작 중인 delivery application 의 rider service에서 도커 이미지 업로드 테스트
- 테스트를 위한 홈페이지 컨트롤러 생성

```
no usages  Sijun
@RestController
public class HomeController {

    no usages  Sijun
    @GetMapping("/")
    public String home(){
        return "Server is activated successfully";
    }
}
```

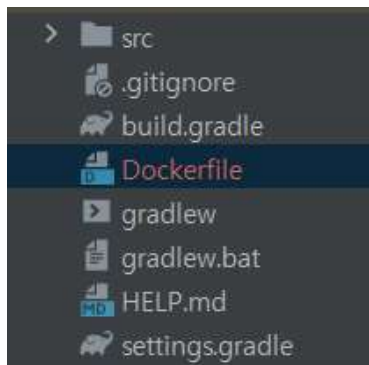

Gradle로 bootJar를 통해 .jar 파일 생성

- jar 파일은 여러 개의 자바 클래스 파일과, 클래스들이 이용하는 관련 리소스(텍스트, 그림 등) 및 메타데이터를 하나의 파일로 모아서 자바 플랫폼에 응용 소프트웨어나 라이브러리를 배포하기 위한 소프트웨어 패키지 파일 포맷
- 쉽게 말해 하나의 압축 포맷
- .jar 파일을 생성해야 docker file에서 읽어서 실행 할 수 있다.



Docker file 생성

- 프로젝트 최상단에 docker file을 생성
- Docker file은 어느 위치에 생성해도 상관없지만 Gradle로 빌드한 jar를 읽을 수 있도록 path를 조정하는 작업이 필요하다.
- ARG로 jar 파일이 있는 위치를 정하고 COPY로 DOCKER 저장소에 복사한 후 ENTRYPOINT로 명령어 실행



```
RiderApplication.java x HomeController.java x Dockerfile x
1 FROM openjdk:19-jdk-alpine
2 ARG JAR_FILE=/build/libs/*.jar
3 COPY ${JAR_FILE} app.jar
4 ENTRYPOINT ["java", "-jar", "/app.jar"]
5 EXPOSE 8080
```

Docker image build and container run

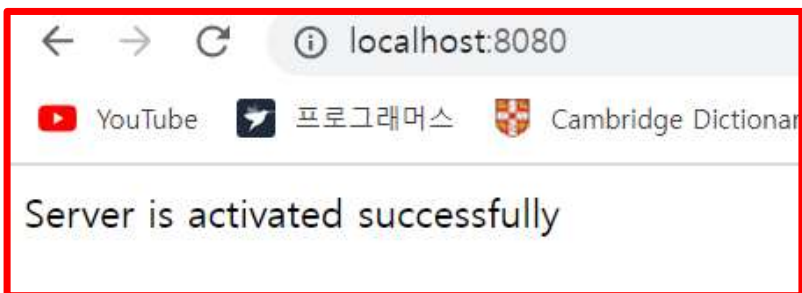
- 이미지 빌드

```
PS D:\delivery project\delivery-application\rider> docker build -t docker-test-image-spring .  
[+] Building 25.0s (8/8) FINISHED
```

```
PS D:\delivery project\delivery-application\rider> docker images  
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE  
docker-test-image-spring latest      a01bdb035331  6 seconds ago 357MB
```

- 컨테이너 실행 - 로컬 환경에서 동작 성공


```
PS D:\delivery project\delivery-application\rider> docker run -d --rm -p 8080:8080 docker-test-image-spring  
f280dd26b1f198e0d0e3092d13fc6458744a8c5245bfd0a649da481a8a0b011d
```




Docker hub image upload


- 도커 허브에 이미지 업로드 - 성공

```
PS D:\delivery project\delivery-application\rider> docker tag docker-test-image-spring millwheel/spring-docker-test
PS D:\delivery project\delivery-application\rider> docker push millwheel/spring-docker-test
Using default tag: latest
The push refers to repository [docker.io/millwheel/spring-docker-test]
```

 **millwheel / spring-docker-test**



Description

This repository does not have a description 

 Last pushed: a few seconds ago

Tags IMAGE INSIGHTS INACTIVE [Activate](#)

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
 latest		Image	---	a few seconds ago

[See all](#) [Go to Advanced Image Management](#)

AWS ECS에 컨테이너 실행

- ECS에서 클러스터 생성 및 이미지 불러오기

▼ 표준

컨테이너 이름*

이미지*

포트 매핑

컨테이너 포트	프로토콜
<input type="text" value="80"/>	<input type="text" value="tcp"/>

+ 포트 매핑 추가

컨테이너 정의

아래에서 컨테이너의 이미지를 선택하여 빠르게 시작하거나, 사용할 컨테이너 이미지를 정의합니다.

sample-app

이미지 : httpd:2.4
메모리 : 0.5GB (512)
CPU : 0.25 vCPU (256)

nginx

이미지 : nginx:latest
메모리 : 0.5GB (512)
CPU : 0.25 vCPU (256)

tomcat-webserver

이미지 : tomcat
메모리 : 2GB (2048)
CPU : 1 vCPU (1024)

docker-test-spring 구성

이미지 : millwheel/spring-docker-test
메모리 :
CPU :

작업 상태: Running Stopped

이 페이지의 필터

작업	작업 정의	마지막 상태	원하는 상태	그룹	시작 유형
880c62f0306042cd...	first-run-task-definiti...	RUNNING	RUNNING	service:docker-test-...	FARGATE

[ERROR] 웹사이트 실행 안 됨

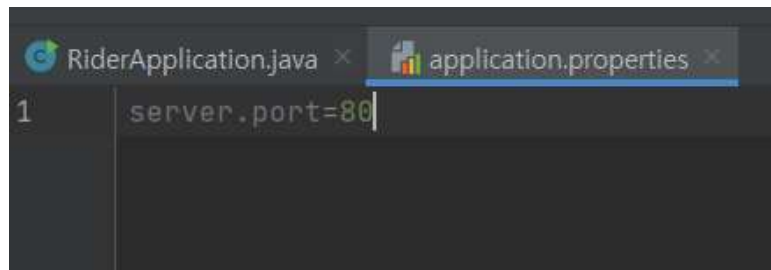
- ECS에서 클러스터 생성 및 이미지 불러오기

원하는 작업 상태: **Running** Stopped

이 페이지의 필터 시작 유형 ALL

	작업	작업 정의 ...	컨테이너 ...	마지막 상...	원하는 상...	시작 위치 ...
<input type="checkbox"/>	880c62f03...	first-run-tas...	--	RUNNING	RUNNING	2023-03-21...

```
"portMappings": [  
  {  
    "hostPort": 80,  
    "protocol": "tcp",  
    "containerPort": 80  
  }  
],
```



사이트에 연결할 수 없음

15.165.79.192에서 연결을 거부했습니다.

다음 방법을 시도해 보세요.

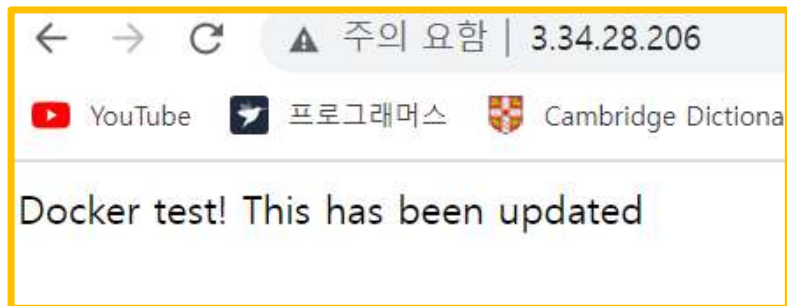
- 연결 확인
- 프록시 및 방화벽 확인

ERR_CONNECTION_REFUSED

[ERROR] NodeJS OK, But Spring NO

- NodeJS에서는 멀쩡하게 돌아가는 서버
- Spring 이미지를 가져오면 동작하지 않음
- ECS 문제는 아니고 Spring과 docker에서 문제가 발생한 것일 수 있음

Node JS로 돌렸을 때



사이트에 연결할 수 없음

15.165.79.192에서 연결을 거부했습니다.

다음 방법을 시도해 보세요.

- 연결 확인
- 프록시 및 방화벽 확인

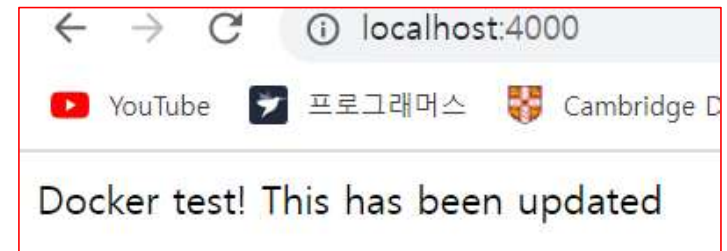
ERR_CONNECTION_REFUSED

[ERROR] 포트 설정 반영이 안 됨

- NodeJS에서 포트 설정은 코드 내에 있기 때문에 곧바로 Docker에서도 80번 포트를 쳐다보게 된다.

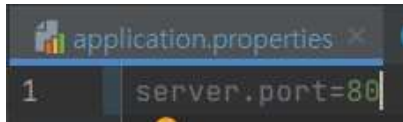
```
}  
});  
});  
  
app.listen(80);
```

```
PS D:\docker\docker project> docker logs dreamy_solomon  
Server is running
```



[ERROR] 포트 설정 반영이 안 됨

- 하지만 spring에서는 application.properties에 server.port=80을 해줬음에도 불구하고 도커 이미지를 빌드해서 컨테이너로 돌리면 포트 번호가 80번이 아닌 8080으로 돌아간다.

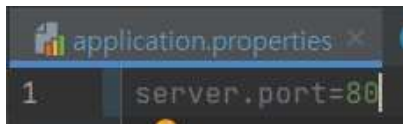


```
PS D:\delivery project\delivery-application\rider> docker logs docker-spring-test-1
```

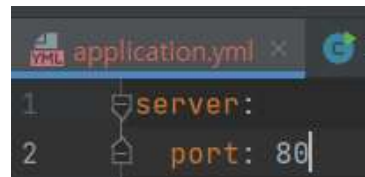
```
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
main] msa.rider.RiderApplication : Started RiderApplication in 2.173 seconds (process running for 2.613)
```

[ERROR] 포트 설정 반영이 안 됨

- application.properties가 아니라 yml로 설정을 해도, 혹은 코드 내부적으로 포트를 설정해도 여전히 port는 80으로 설정되지 않고 8080으로 돌아간다.
- Local 빌드에서는 80으로 돌아가지만, 딱 도커 이미지로 빌드만 하면 8080으로 돌아감 (설정이 전혀 적용이 안 됨)



```
1 server.port=80
```



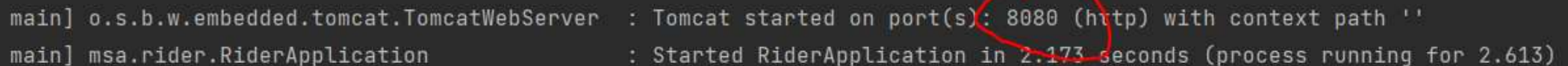
```
1 server:
2   port: 80
```



```
@SpringBootApplication
public class RiderApplication {

    no usages  Sijun +1

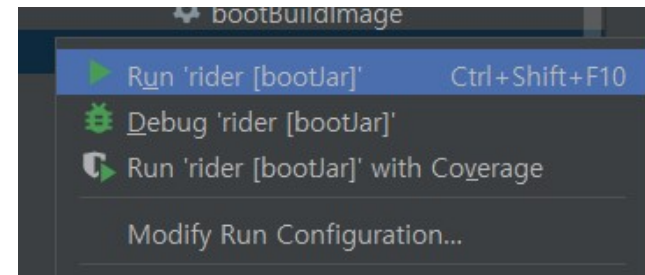
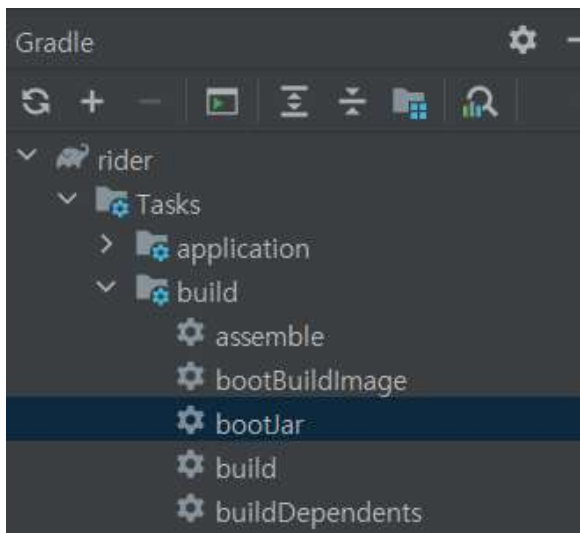
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(RiderApplication.class);
        app.setDefaultProperties(Collections.singletonMap("server.port", "80"));
        app.run(args);
    }
}
```



```
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
main] msa.rider.RiderApplication : Started RiderApplication in 2.173 seconds (process running for 2.613)
```

[해결] 알고 보니 gradle build 문제

- gradle 빌드를 수동으로 해줘야 build/lib에 jar파일이 생성된다.
- Server.port = 80으로 지정해놓고 다시 수동으로 build를 하지 않아서 소스코드의 변경사항이 반영되지 않았음.
- 다시 BootJar로 빌드 실행한 후 jar 파일 업데이트



```
: Starting Servlet engine: [Apache Tomcat/10.1.5]
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 100ms
: Tomcat started on port(s): 80 (http) with context path ''
: Started RiderApplication in 1.812 seconds (process running ...)
```

포트 변경
반영 성공

왜 수동으로 BootJar 실행해야 하나

- Gradle build와 BootJar의 차이를 Stack overflow에서 설명

`build` is a lifecycle task contributed by the [Base Plugin](#). It is

Intended to build everything, including running all tests, producing the production artifacts and generating documentation. You will probably rarely attach concrete tasks directly to `build` as `assemble` and `check` are typically more appropriate.

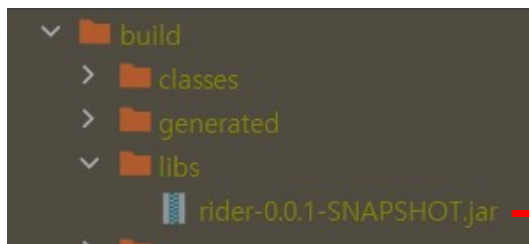
`bootJar` on the other hand is a specific task added by Spring Boot Gradle plugin that, when the `java` plugin is present, attaches itself to the `assemble` lifecycle task.

The `assemble` task is automatically configured to depend upon the `bootJar` task so running `assemble` (or `build`) will also run the `bootJar` task.

[\(Packaging Executable Jars\)](#)

왜 수동으로 BootJar 실행해야 하나

- Java 소스 코드로 작성된 파일을 실행하여 서버를 띄우려면 jar파일로 빌드를 해서 띄워야 한다.
- BootJar에 의해 생성된 executable jar 파일은 어플리케이션 실행에 필요한 모든 의존성을 함께 빌드한 파일이다.
- 근데 Docker로 이미지를 만들려면 이 executable jar 파일이 필요하다.
- 그러므로 BootJar를 실행해서 executable jar 파일 생성



```
FROM eclipse-temurin:17-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

스프링 공문서에 있는 Dockerfile 문법 예시