

제 목	04장 Docker 이미지 생성	
상세내용	Dockerfile 활용해 이미지 직접 만들기	

## 1. Dockerfile이란?

### ✓ Dockerfile이란?

Docker 이미지는 Dockerhub을 통해 다운받아서 사용할 수 있다. 이 Docker 이미지들도 누군가 만들어서 Dockerhub에 올려놓은 것이다. 그럼 도대체 이 Docker 이미지는 어떻게 만드는 걸까?

| **Dockerfile**이라는 걸 활용해서 Docker 이미지를 만들 수 있다.

Dockerhub에 올려놓은 Docker 이미지가 아닌, 나만의 Docker 이미지를 만들고 싶을 수 있다. 예를 들어, 내가 만든 Spring Boot 프로젝트가 있다. 내가 만든 Spring Boot 프로젝트 자체를 Docker 이미지로 만들고 싶을 수 있다. 이럴 때에도 Dockerfile을 활용하면 나만의 Docker 이미지를 만들 수 있게 된다.

정리하자면, **\*\*Dockerfile\*\***이란 **\*\*Docker 이미지를 만들게 해주는 파일\*\***이다

## 2. FROM : 베이스 이미지 생성

### ✓ 의미

**FROM** 은 베이스 이미지를 생성하는 역할을 한다. Docker 컨테이너를 **특정 초기 이미지**를 기반으로 추가적인 셋팅을 할 수 있다. 여기서 얘기한 '**특정 초기 이미지**'가 곧 **베이스 이미지**이다.

쉽게 설명하면 ~

우리가 윈도우 컴퓨터를 새로 사서 실행시켜보면 기본 프로그램들(인터넷, 그림판, 메모장 등)이 많이 깔려있다. 베이스 이미지도 이와 똑같다. 컨테이너를 새로 띄워서 미니 컴퓨터 환경을 구축할 때 기본 프로그램이 어떻게 깔려있으면 좋겠는 지 선택하는 옵션이라고 생각하면 된다.

누군가는 JDK가 깔려있는 컴퓨터 환경이 셋팅되기를 바랄 수도 있고, 누군가는 Node가 깔려있는 컴퓨터 환경이 셋팅되기를 바랄 수도 있다. 필요에 따라 베이스 이미지를 고르면 된다.

## ✓ 사용법

```
# 문법
FROM [이미지명]
FROM [이미지명]:[태그명]
```

- 태그명을 적지 않으면 해당 이미지의 최신(latest) 버전을 사용한다.

## [실습] FROM : 베이스 이미지 생성

### 🔗 JDK 17 베이스 이미지로 컨테이너 띄워보기

#### 1. Dockerfile 만들기

```
# Dockerfile => JDK 17
FROM openjdk:17-jdk
```

#### 2. Dockerfile을 기반으로 이미지 만들기

##### ▶ Dockerfile로 이미지(Image) 생성하는 문법

```
# docker build -t [이미지명]:[태그명] [Dockerfile이 존재하는 디렉터리 경로]

$ docker build -t sample .
$ docker build -t sample:1.0 .
```

- 태그명을 적지 않으면 latest로 설정된다

```
$ docker build -t my-jdk17-server .
```

#### 3. 이미지를 기반으로 컨테이너 띄우기

```
$ docker run -d my-jdk17-server
```

#### 4. 컨테이너 조회하기

```
$ docker ps # 실행되고 있는 컨테이너가 없다.
$ docker ps -a # 확인해보니 컨테이너가 종료되어 있다.
```

Docker의 컨테이너는 내부적으로 필요한 명령을 다 수행하면 컨테이너가 저절로 종료된다.

## 5. 컨테이너 내부로 들어가서 jdk가 잘 깔렸는지 확인해보기

Dockerfile

```
FROM openjdk:17-jdk
ENTRYPOINT ["/bin/bash", "-c", "sleep 500"]
```

```
$ docker build -t my-jdk17-server . # 이미지 빌드
$ docker run -d my-jdk17-server # 컨테이너 실행
$ docker ps # 실행 중인 컨테이너 조회
$ docker exec -it [컨테이너 ID] bash # 컨테이너 접속

$ java -version # JDK 설치되어 있는 지 확인
```

## 🔗 Node 베이스 이미지로 컨테이너 띄워보기

## 1. Dockerfile 만들기

```
FROM node

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"]
```

## 2. 이미지 만들고 컨테이너 띄우기

```
$ docker build -t my-node-server . # 이미지 생성
$ docker run -d my-node-server # 이미지를 기반으로 컨테이너 생성
$ docker ps # 실행 중인 컨테이너 조회
$ docker exec -it [컨테이너 ID] bash # 컨테이너 접속

$ node -v # Node 설치되어 있는 지 확인
```

```
jaeseong ~/Documents/Develop/jscode/node-practice ▶ docker exec -it 5bd bash
root@5bdcf15d2fa7:/# node -v
v22.2.0
```

### 3. 종료된 컨테이너에 들어가서 디버깅하고 싶을 때

프로그래밍을 할 때 중간중간 잘 작동하는 지 확인하는 습관은 굉장히 중요하다. 어떤 명령어를 입력하고 난 뒤에 명령어가 정상적으로 수행됐는 지 어떻게 확인할 수 있는 지 방법을 찾아봐야 한다.

이 습관이 몸에 익으면 어떤 명령어를 수행하더라도 그 명령어가 어떻게 작동하는 지 파헤칠 수 있게 된다. 또한 어떤 명령어를 실행시킨 뒤에 에러가 생기더라도 금방 발견할 수 있어서 디버깅도 훨씬 수월하다.

Docker를 사용하면 대부분의 코드가 컨테이너 내부에서 작동한다. 그러다보니 어떤 과정으로 처리됐는 지, 잘 처리는 됐는 지를 직접적으로 눈에 보이지 않는다. 이 때문에 Docker 학습에 어려움을 겪는다.

이를 해결하기 위해 우리는 2가지 방법을 이미 익혔다.

- `'docker logs'`를 활용해 컨테이너 로그 확인하기
- `'docker exec -it'`를 활용해 컨테이너 내부에 직접 들어가보기

위 방법 중 `docker exec -it`은 실행 중인 컨테이너에만 쓸 수 있는 명령어이다. 종료된 컨테이너는 아래와 같은 에러가 발생한다.

```
jaeseong ~/Documents/Develop/jscode/node-practice docker exec -it 393 bash
Error response from daemon: container 3938f02306f741f204adc34be377217e57456dee70a0437e3e088c469b3a9a73 is not running
```

하지만 이미지를 만들면서 컨테이너를 실행시켜보면, 컨테이너의 특성상 명령어 처리가 다 끝나는대로 컨테이너가 종료된다. 그러다보니 내부적으로 어떻게 컨테이너가 형성됐는 지 디버깅을 하는데 어려움을 겪는다.

#### ▶ 어떻게 해야 할까?

Dockerfile

```
FROM openjdk:17-jdk
```

```
...
```

```
ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 500초 동안 시스템을 일시정지 시키는 명령어
```

위 명령어를 추가함으로써 컨테이너가 바로 종료되는 걸 막을 수 있다. 그런 뒤에 `docker exec -it`를 활용해 컨테이너 내부에 직접 들어가서 디버깅을 하면 된다.

## 4. COPY : 파일 복사(이동)

### ✓ 의미

`COPY`는 **호스트 컴퓨터**에 있는 파일을 복사해서 **컨테이너**로 전달한다.

### ✓ 사용법

```
# 문법
COPY [호스트 컴퓨터에 있는 복사할 파일의 경로] [컨테이너에서 파일이 위치할 경로]

# 예시
COPY app.txt /app.txt
```

### 🔗 파일 복사해보기

1. app.txt 파일 만들기
2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행

Dockerfile

```
FROM ubuntu

COPY app.txt /app.txt

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 디버깅용 코드
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```

```
jaeseong ~/Documents/Develop/jscode/node-practice ➤ docker exec -it df bash
root@df75d7867a5d:/# ls
app.txt  bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

🔗 폴더 안에 있는 모든 파일들 복사

1. `my-app` 디렉터리 만들기, `my-app` 디렉터리 안에 파일 만들기
2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행

Dockerfile

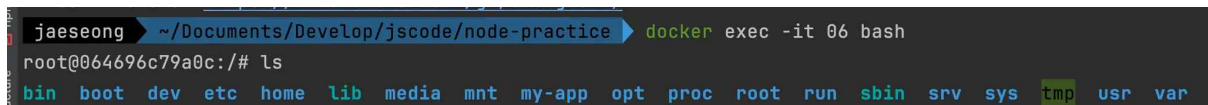
```
FROM ubuntu

COPY my-app /my-app/

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 디버깅용 코드
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```



```
jaeseong ~/Documents/Develop/jscode/node-practice $ docker exec -it 06 bash
root@064696c79a0c:/# ls
bin boot dev etc home lib media mnt my-app opt proc root run sbin srv sys tmp usr var
```

🔗 와일드 카드 사용해보기

1. `app.txt`, `readme.txt` 파일 2개 만들기
2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행

Dockerfile

```
FROM ubuntu

COPY *.txt /text-files/

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 디버깅용 코드
```

**주의)** /text-files라고 적으면 안 되고 /text-files/라고 적어야 text-files라는 디렉토리 안에 파일들이 정상적으로 복사된다

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```

### 🔗 `.dockerignore` 사용해보기

특정 파일 또는 폴더만 `COPY`를 하고 싶지 않을 수 있다. 그럴 때 `.dockerignore`를 활용한다.

#### 1. `.dockerignore` 파일 만들기

.dockerignore

```
readme.txt
```

#### 2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행

Dockerfile

```
FROM ubuntu

COPY ./ /

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 디버깅용 코드
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```

## 5. ENTRYPOINT : 컨테이너가 시작할 때 실행되는 명령어

### ✓ 의미

`ENTRYPOINT`는 컨테이너가 생성되고 최초로 실행할 때 수행되는 명령어를 뜻한다. 쉽게 설명하자면

`ENTRYPOINT`에는 미니 컴퓨터의 전원을 키고나서 실행시키고 싶은 명령어를 적으면 된다.

### ✓ 사용법

```
# 문법
ENTRYPOINT [명령문...]

# 예시
ENTRYPOINT ["node", "dist/main.js"]
```

### 🔗 예제

Dockerfile

```
FROM ubuntu

ENTRYPOINT ["/bin/bash", "-c", "echo hello"]
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker ps -a
$ docker logs [Container ID]
```

```
jaeseong ~/Documents/Develop/jscode/node-practice docker logs 022
hello
```



## 6. [실습] 백엔드 프로젝트(Spring Boot) 프로젝트를 Docker로 실행시키기

### ✓ 백엔드 프로젝트(Spring Boot) 프로젝트를 Docker로 실행시키기

#### 1. 프로젝트 셋팅

start.spring.io

<https://start.spring.io/>

- Java 17 버전을 선택하자. 아래 과정을 Java 17 버전을 기준으로 진행할 예정이다.

#### 2. 간단한 코드 작성

AppController

```
# java

@RestController
public class AppController {
    @GetMapping("/")
    public String home() {
        return "Hello, World!";
    }
}
```

### 3. Dockerfile 작성하기

Dockerfile

```
FROM openjdk:17-jdk

COPY build/libs/*SNAPSHOT.jar app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]
```

### 4. Spring Boot 프로젝트 빌드하기

```
$ ./gradlew clean build
```

### 5. Dockerfile을 바탕으로 이미지 빌드하기

```
$ docker build -t hello-server .
```

### 6. 이미지가 잘 생성됐는 지 확인하기

```
$ docker image ls
```

### 7. 생성한 이미지를 컨테이너로 실행시켜보기

```
$ docker run -d -p 8080:8080 hello-server
```

### 8. 컨테이너 잘 실행되고 있는 지 확인하기

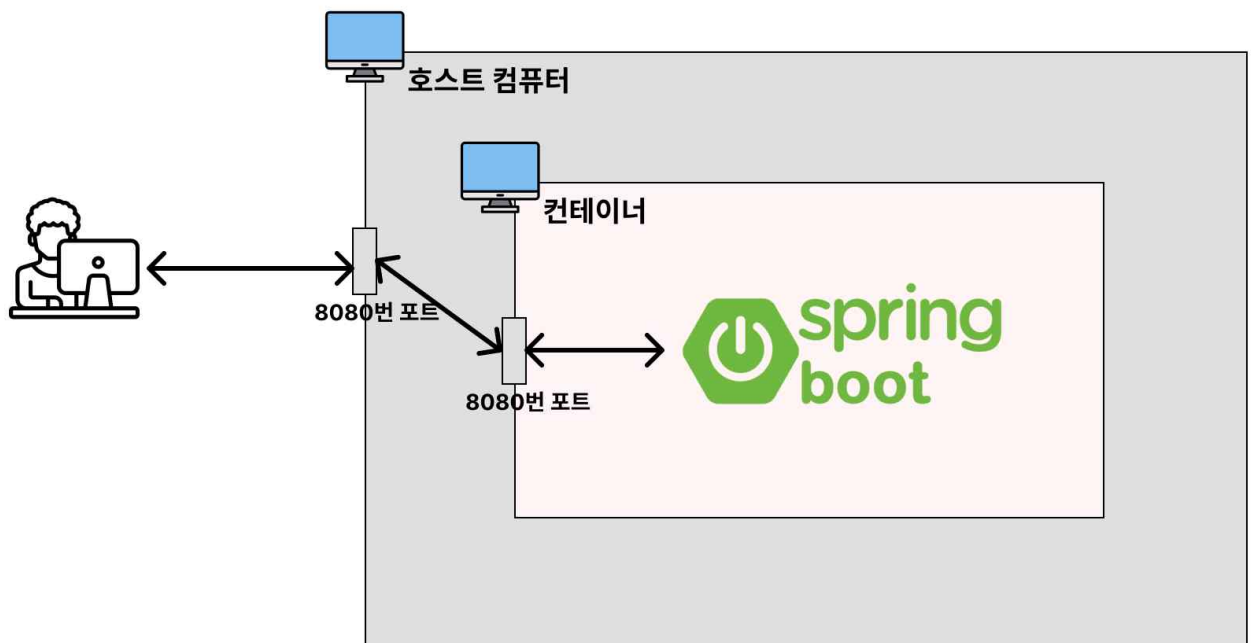
```
$ docker ps
```

### 9. localhost:8080으로 들어가보기

## 10 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

```
$ docker stop {컨테이너 ID}
$ docker rm {컨테이너 ID}
$ docker image rm {이미지 ID}
```

### ✓ 그림으로 이해하기



## 7. RUN : 이미지를 생성하는 과정에서 사용할 명령문 실행

### ✓ 의미

**RUN**은 이미지 생성 과정에서 명령어를 실행시켜야 할 때 사용한다

### ✓ 사용법

```
# 문법
RUN [명령문]

# 예시
RUN npm install
```

### ✓ RUN vs ENTRYPOINT

**RUN** 명령어와 **ENTRYPOINT** 명령어가 헷갈릴 때가 있다. 둘 다 같이 명령어를 실행시키기 때문이다. 하지만 엄연히 둘의 사용 용도는 다르다. **RUN**은 '이미지 생성 과정'에서 필요한 명령어를 실행시킬 때 사용하고, **ENTRYPOINT**는 생성된 이미지를 기반으로 컨테이너를 생성한 직후에 명령어를 실행시킬 때 사용한다

### 🔗 예제

미니 컴퓨터 환경이 ubuntu로 구성되었으면 좋겠고 git이 깔려있으면 좋겠다고 가정하자. 이런 환경을 구성하기 위해 `Dockerfile`을 활용해 ubuntu, git이 깔려있는 이미지를 만들면 된다.

#### 1. Dockerfile 작성하기

Dockerfile

```
FROM ubuntu

RUN apt update && apt install -y git

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"]
```

## 2. 이미지 빌드 및 컨테이너 실행

```
$ docker build -t my-server .  
$ docker run -d my-server  
$ docker exec -it [Container ID] bash  
  
$ git -v # 컨테이너 내에 git이 잘 설치됐는 지 확인
```

## 8. WORKDIR : 작업 디렉토리 지정

### ✔ 의미

**WORKDIR**으로 작업 디렉토리를 전환하면 그 이후에 등장하는 모든 **RUN**, **CMD**, **ENTRYPOINT**, **COPY**, **ADD** 명령문은 해당 디렉토리를 기준으로 실행된다. 작업 디렉토리를 굳이 지정해주는 이유는 컨테이너 내부의 폴더를 깔끔하게 관리하기 위해서이다. 컨테이너도 미니 컴퓨터와 같기 때문에 **Dockerfile**을 통해 생성되는 파일들을 특정 폴더에 정리해두는 것이 추후에 관리가 쉽다. 만약 **WORKDIR**을 쓰지 않으면 컨테이너 내부에 존재하는 기존 파일들과 뒤섞여버린다.

### ✔ 사용법

```
# 문법  
WORKDIR [작업 디렉토리로 사용할 절대 경로]  
  
# 예시  
WORKDIR /usr/src/app
```

## 🔗 예제

1. `app.txt`, `src`, `config.json` 파일 만들기
2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행
  - ▶ WORKDIR을 안 썼을 때 파일이 어떻게 구성되는 지 먼저 확인해보자.

Dockerfile

```
FROM ubuntu

COPY ./ ./

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] # 디버깅용 코드
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```

- ▶ **WORKDIR**을 썼을 때 파일이 어떻게 구성되는 지 확인해보자.

Dockerfile

```
FROM ubuntu

WORKDIR /my-dir

COPY ./ ./

ENTRYPOINT ["/bin/bash", "-c", "sleep 500"]
```

```
$ docker build -t my-server .
$ docker run -d my-server
$ docker exec -it [Container ID] bash

$ ls
```

## 9. EXPOSE : 컨테이너 내부에서 사용 중인 포트를 문서화하기

### ✓ 의미

EXPOSE는 컨테이너 내부에서 어떤 포트에 프로그램이 실행되는 지를 문서화하는 역할만 한다. `docker -p 8080:8080 ...` 와 같은 명령어의 `-p` 옵션과 같은 역할은 일체 하지 않는다. 쉽게 표현하자면 EXPOSE 명령어는 쓰나 안 쓰나 작동하는 방식에는 영향을 미치지 않는다.

### ✓ 사용법

```
# 문법
EXPOSE [포트 번호]

# 예시
EXPOSE 3000
```

## [실습] 웹 프론트엔드 프로젝트(HTML, CSS, Nginx)를 Docker로 배포하기

### ✓ 웹 프론트엔드 프로젝트(HTML, CSS, Nginx)를 Docker로 배포하기

#### 1. HTML, CSS 파일 만들기

index.html

```
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>My Web Page</h1>
</body>
</html>
```

**주의)** Nginx의 기본 설정에 의하면 메인 페이지(첫 페이지)의 파일명을 **index.html** 이라고 지어야 한다.

style.css

```
* {
  color: blue;
}
```

#### 2. Dockerfile 작성하기

Dockerfile

```
FROM nginx
COPY ./ /usr/share/nginx/html
```

#### 3. Dockerfile을 바탕으로 이미지 빌드하기

```
$ docker build -t my-web-server .
```



#### 4. 이미지가 잘 생성됐는 지 확인하기

```
$ docker image ls
```

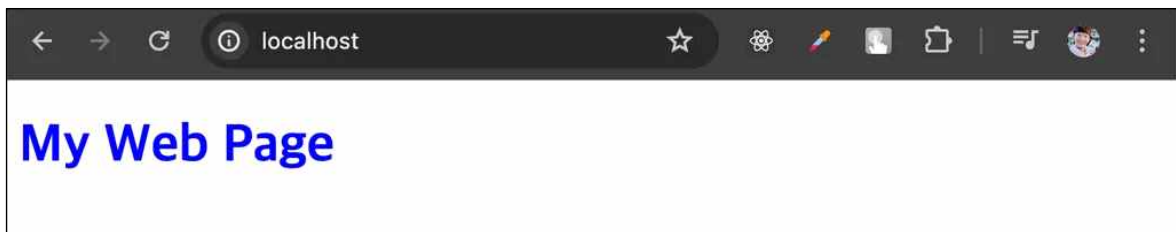
#### 5. 생성한 이미지를 컨테이너로 실행시켜보기

```
$ docker run -d -p 80:80 my-web-server
```

#### 6. 컨테이너 잘 실행되고 있는 지 확인하기

```
$ docker ps
```

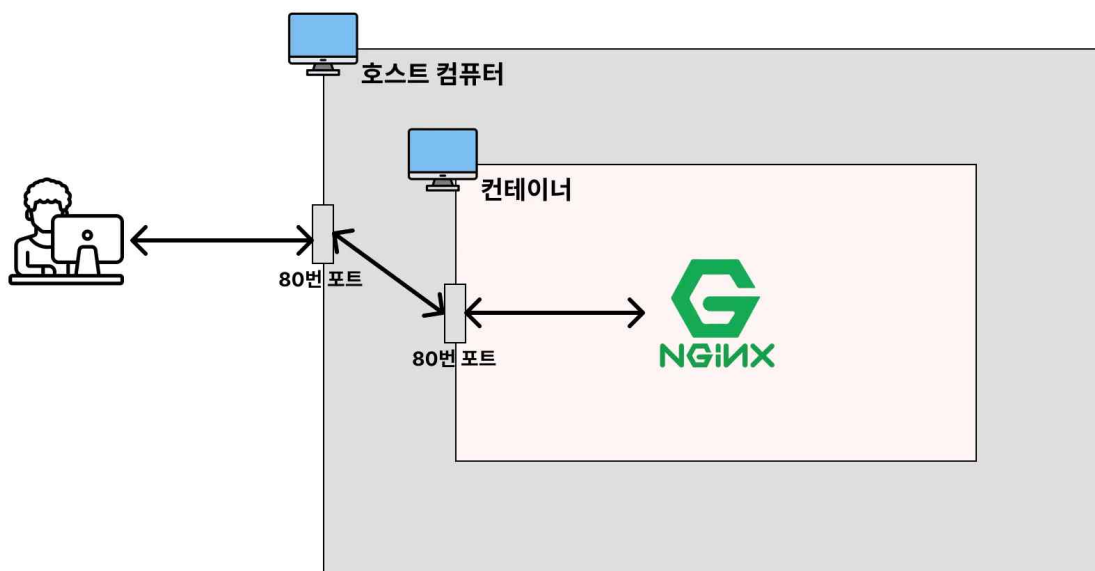
#### 7. localhost:80으로 들어가보기



#### 9. 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

```
$ docker stop {컨테이너 ID}
$ docker rm {컨테이너 ID}
$ docker image rm {이미지 ID}
```

#### ✓ 그림으로 이해하기



## [실습] 웹 프론트엔드 프로젝트(Next.js)를 Docker로 배포하기

### ✓ 웹 프론트엔드 프로젝트(Next.js)를 Docker로 배포하기

#### 1. Next.js 프로젝트 만들기

```
$ npx create-next-app@latest
```

#### 2. Dockerfile 작성하기

Dockerfile

```
FROM node:20-alpine

WORKDIR /app

COPY . .

RUN npm install

RUN npm run build

EXPOSE 3000

ENTRYPOINT [ "npm", "run", "start" ]
```

**alpine** : 불필요한 프로그램을 포함하지 않고 이미지 크기를 최소화한 버전. 실제 배포 할 때도 되도록이면 **alpine** 버전을 사용한다.

#### 3. .dockerignore 작성하기

.dockerignore

```
node_modules
```

이미지를 생성할 때 **npm install**을 통해 처음부터 깔끔하게 필요한 의존성만 설치한다. 따라서 호스트 컴퓨터에 있는 **node\_modules**는 컨테이너로 복사해갈 필요가 없다.

#### 4. Dockerfile을 바탕으로 이미지 빌드하기

```
$ docker build -t my-web-server .
```

#### 5. 이미지가 잘 생성됐는 지 확인하기

```
$ docker image ls
```

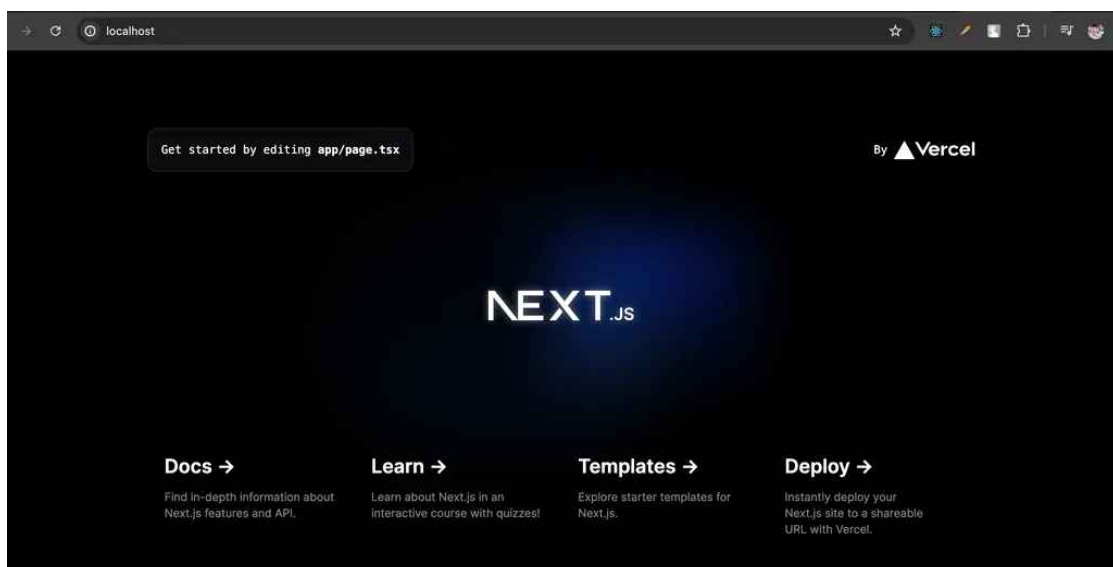
#### 6. 생성한 이미지를 컨테이너로 실행시켜보기

```
$ docker run -d -p 80:3000 my-web-server
```

#### 7. 컨테이너 잘 실행되고 있는 지 확인하기

```
$ docker ps
```

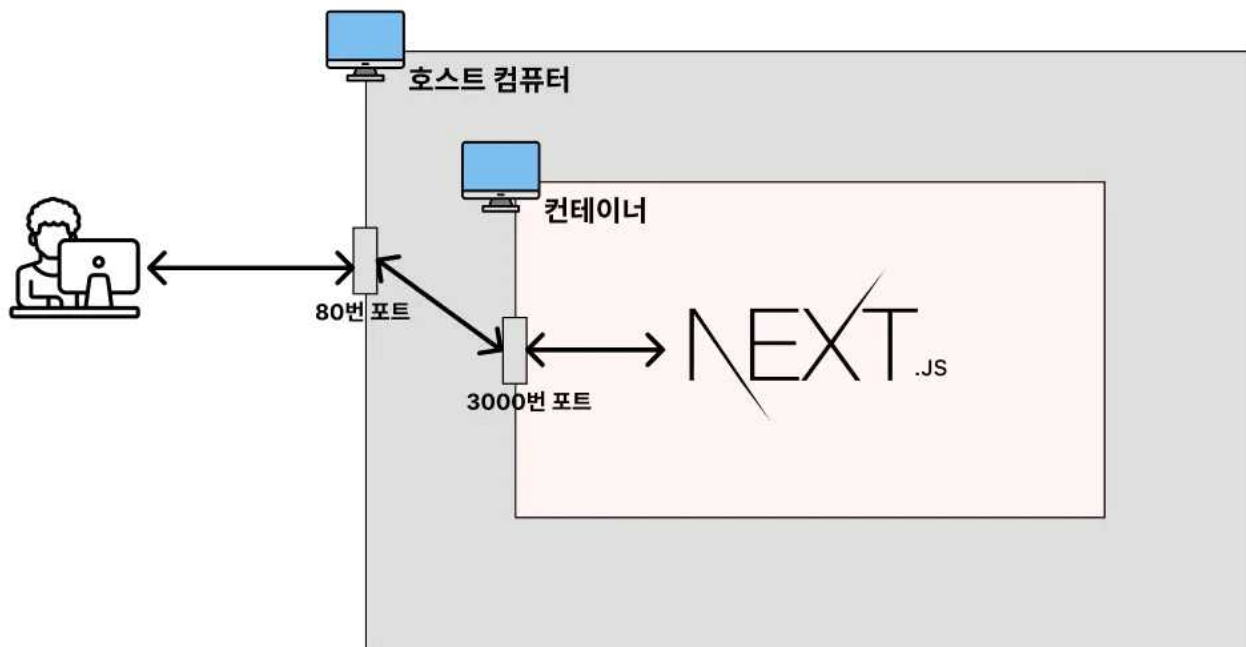
#### 8. 컨테이너 잘 실행되고 있는 지 확인하기



## 9. 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

```
$ docker stop {컨테이너 ID}
$ docker rm {컨테이너 ID}
$ docker image rm {이미지 ID}
```

### ✓ 그림으로 이해하기



## [실습] 백엔드 프로젝트(Nest.js)를 Docker로 실행시키기

### ✓ 백엔드 프로젝트(Nest.js)를 Docker로 실행시키기

#### 1. Nest.js 프로젝트 만들기

```
# Nest CLI 설치
$ npm i -g @nestjs/cli

# nest new {프로젝트명}
$ nest new my-server
```

#### 2. Dockerfile 작성하기

Dockerfile

```
FROM node

WORKDIR /app
COPY . .

RUN npm install

RUN npm run build

EXPOSE 3000

ENTRYPOINT [ "node", "dist/main.js" ]
```

**심화)** Docker 이미지 생성 시 캐시를 활용해서 최적화할 수 있는 방법이 있다. 입문자한테는 불필요한 내용이기 때문에 별도로 설명하지 않았다. 관심 있는 분들은 아래 링크를 참고하자.

참고 사이트: <https://tinyurl.com/27e2omwf>

### 3. .dockerignore 작성하기

.dockerignore

```
node_modules
```

이미지를 생성할 때 `npm install`을 통해 처음부터 깔끔하게 필요한 의존성만 설치한다. 따라서 호스트 컴퓨터에 있는 `node_modules`는 컨테이너로 복사해갈 필요가 없다.

### 4. Dockerfile을 바탕으로 이미지 빌드하기

```
$ docker build -t my-server .
```

### 5. 이미지가 잘 생성됐는 지 확인하기

```
$ docker build -t my-server .
```

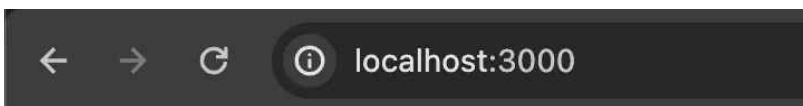
### 6. 생성한 이미지를 컨테이너로 실행시켜보기

```
$ docker run -d -p 3000:3000 my-server
```

### 7. 컨테이너 잘 실행되고 있는 지 확인하기

```
$ docker run -d -p 3000:3000 my-server
```

### 8. localhost:3000으로 들어가보기



Hello World!

## 9. 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

```
$ $ docker stop {컨테이너 ID}
$ docker rm {컨테이너 ID}
$ docker image rm {이미지 ID}
```

### ✓ 그림으로 이해하기

