# **04.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 Dcoker 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.

1. **FROM : 베이스 이미지 생성**

### **✅ 의미**

FROM은 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.

조금 더 쉽게 설명해보겠다.

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 어떤게 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.

1. **[실습] FROM : 베이스 이미지 생성**

### **🎯 JDK 17 베이스 이미지로 컨테이너 띄워보기**

1. Dockerfile 만들기

작업 폴더 만들기

| $ mkdir docker-practice  $ cd docker-practice |
| --- |

[openjdk - Official Image | Docker Hub](https://hub.docker.com/_/openjdk) Dockerfile 파일 만들기(에디터 프로그램을 사용해도 됨)

<https://www.vim.org/download.php#pc>

| $ vim Dockerfile |
| --- |

| # JDK 17  FROM openjdk:17-jdk |
| --- |

1. Dockerfile을 기반으로 이미지 만들기  
   > Dockerfile로 이미지(Image) 생성하는 문법(닷을 빼먹지 말것)

| $ docker build -t my-jdk17-server **.** |
| --- |



1. 이미지를 기반으로 컨테이너 띄우기

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

1. 컨테이너 조회하기

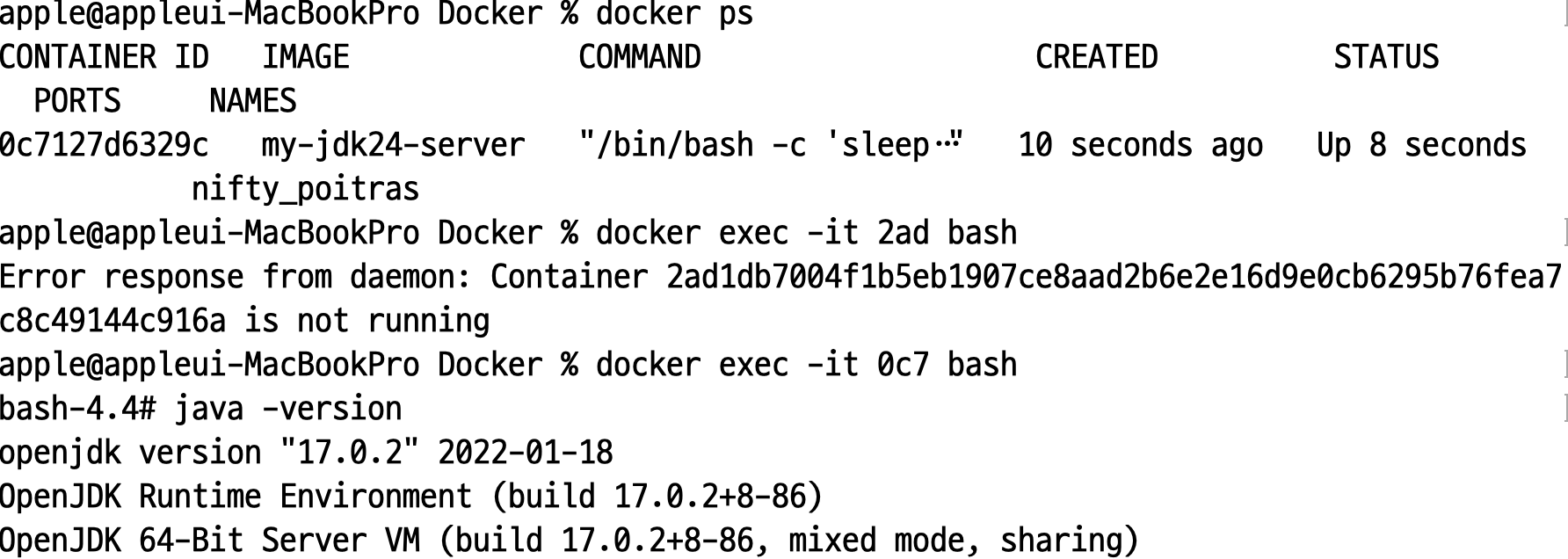
| $ docker ps # 실행되고 있는 컨테이너가 없다.  $ docker ps -a # 확인해보니 컨테이너가 종료되어 있다. |
| --- |

  
 Docker의 컨테이너는 내부적으로 필요한 명령을 다 수행하면 컨테이너가 저절로 종료된다.

1. 컨테이너 내부로 들어가서 jdk가 잘 깔렸는지 확인해보기  
     
    [종료된 컨테이너에 들어가서 디버깅하고 싶을 때](https://docs.google.com/document/d/1BHpR5sSIdL-Evjr071kR1mJQ3DXCcOPSPXOV45HDqrU/edit?usp=drive_link) Dockerfile

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

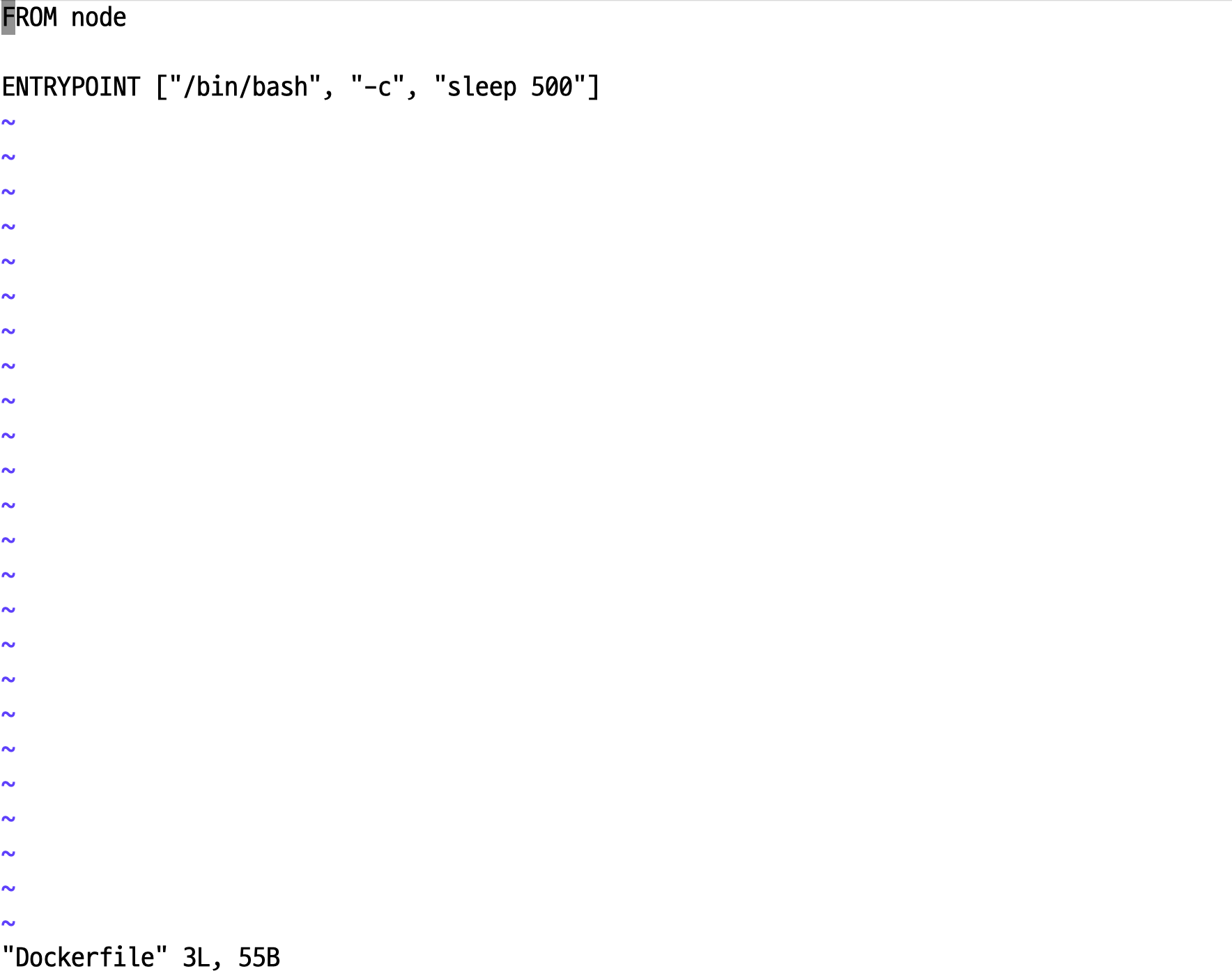
| $ docker build -t my-jdk17-server . # 이미지 빌드  $ docker run -d my-jdk17-server # 컨테이너 실행  $ docker ps # 실행 중인 컨테이너 조회  $ docker exec -it [컨테이너 ID] bash # 컨테이너 접속  $ java -version # JDK 설치되어 있는 지 확인 |
| --- |



### **🎯 Node 베이스 이미지로 컨테이너 띄워보기**

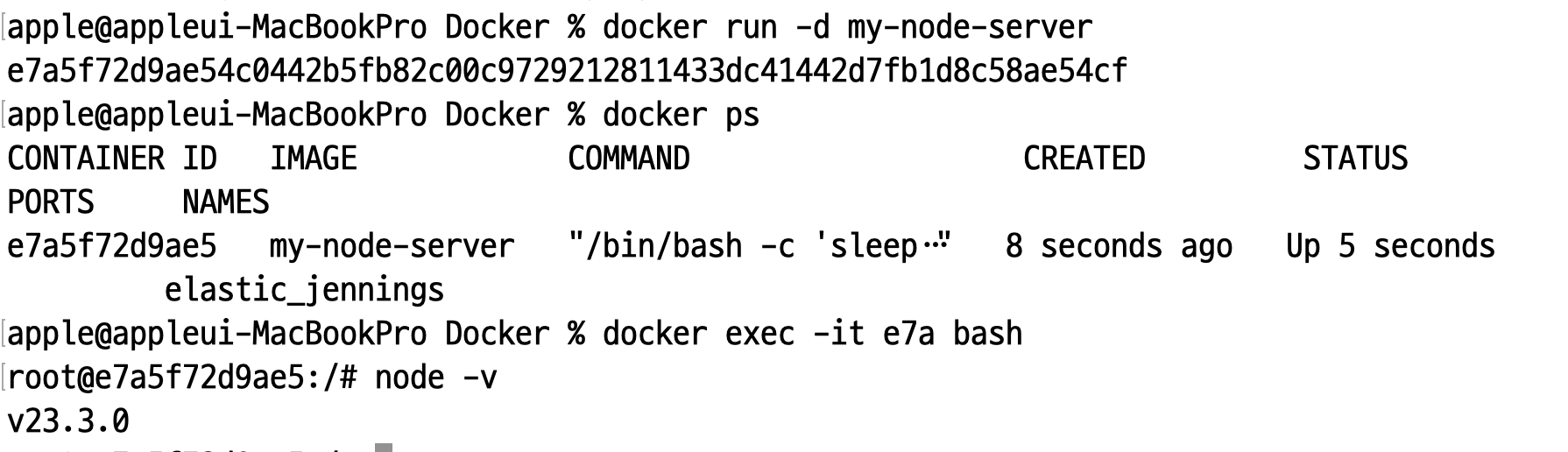
1. Dockerfile 만들기

| FROM node  ENTRYPOINT ["/bin/bash", "-c", "sleep 500"] |
| --- |



1. 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 |
| --- |

****

1. **COPY : 파일 복사(이동)**

### **✅ 의미**

**COPY는 호스트 컴퓨터에 있는 파일을 복사해서 컨테이너로 전달한다.**

### **✅ 사용법**

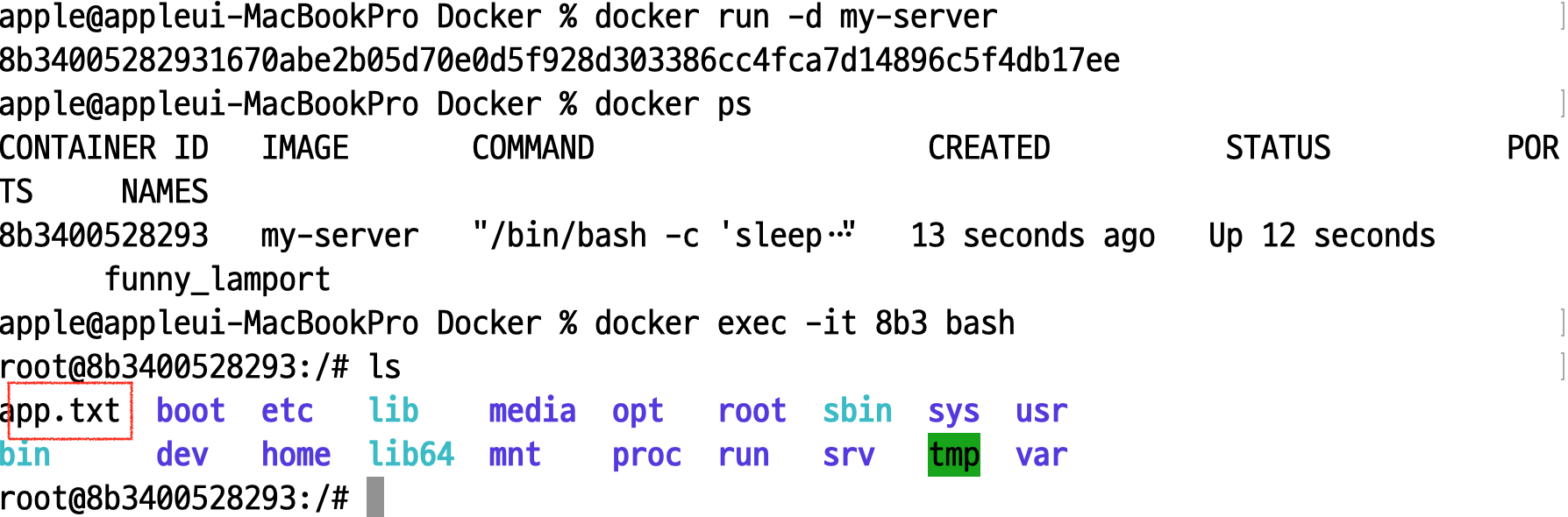
| # 문법  COPY [호스트 컴퓨터에 있는 복사할 파일의 경로] [컨테이너에서 파일이 위치할 경로]  # 예시  COPY app.txt /app.txt |
| --- |

### **🎯 파일 복사해보기**

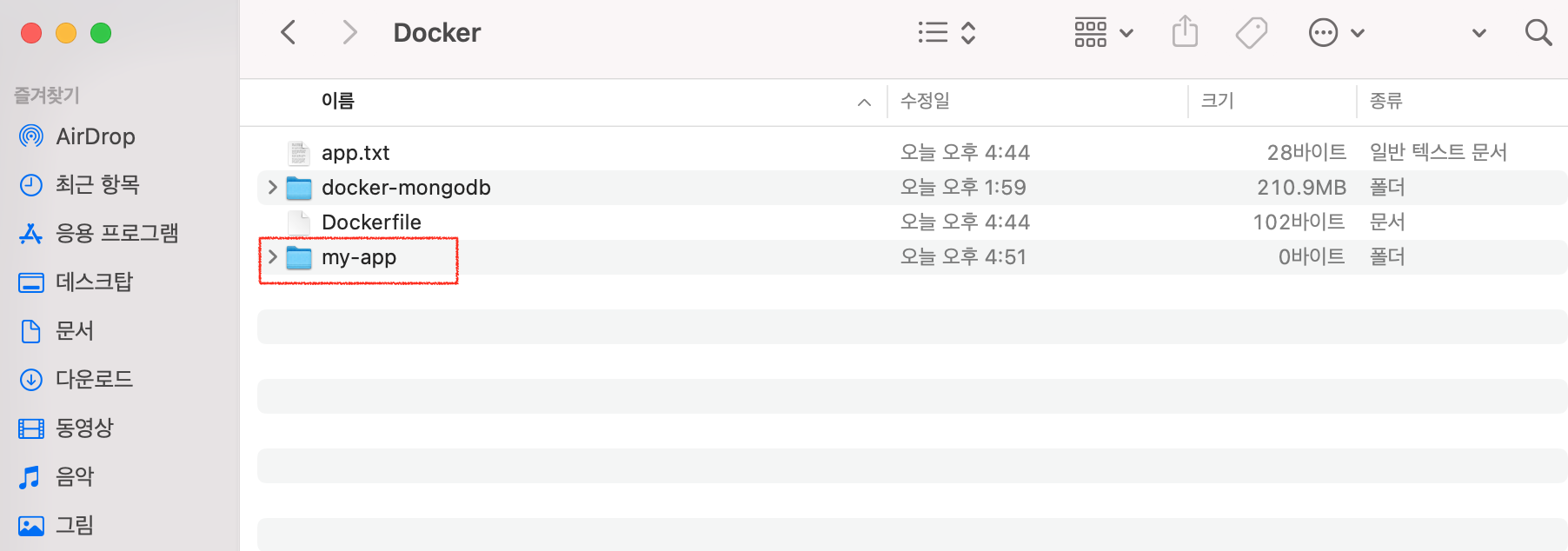
1. app.txt 파일 만들기 “Hello World!”
2. Dockerfile 만들어서 이미지 생성 및 컨테이너 실행  
    Dockerfile

| FROM ubuntu  COPY app.txt /app.txt # 앞에 우리가 만든 app.txt 를 컨테이너에 복사해라  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 |
| --- |



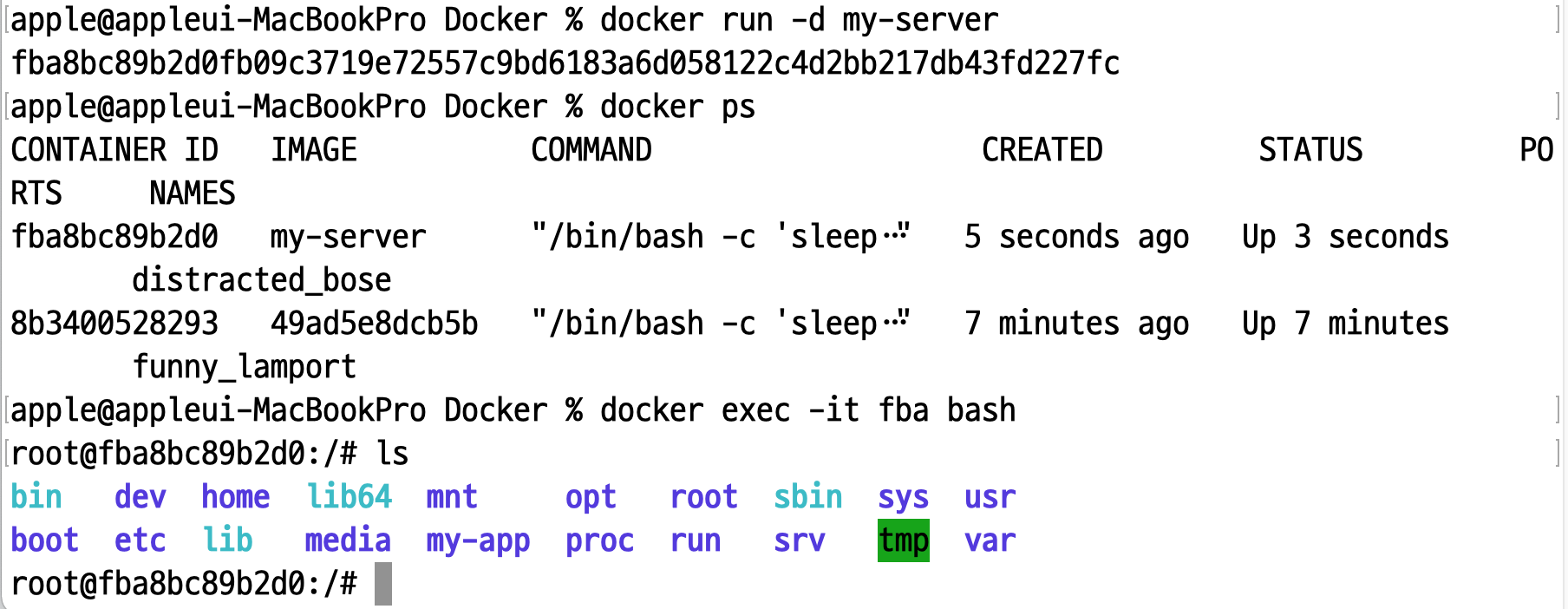
### **🎯 폴더 안에 있는 모든 파일들 복사**

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

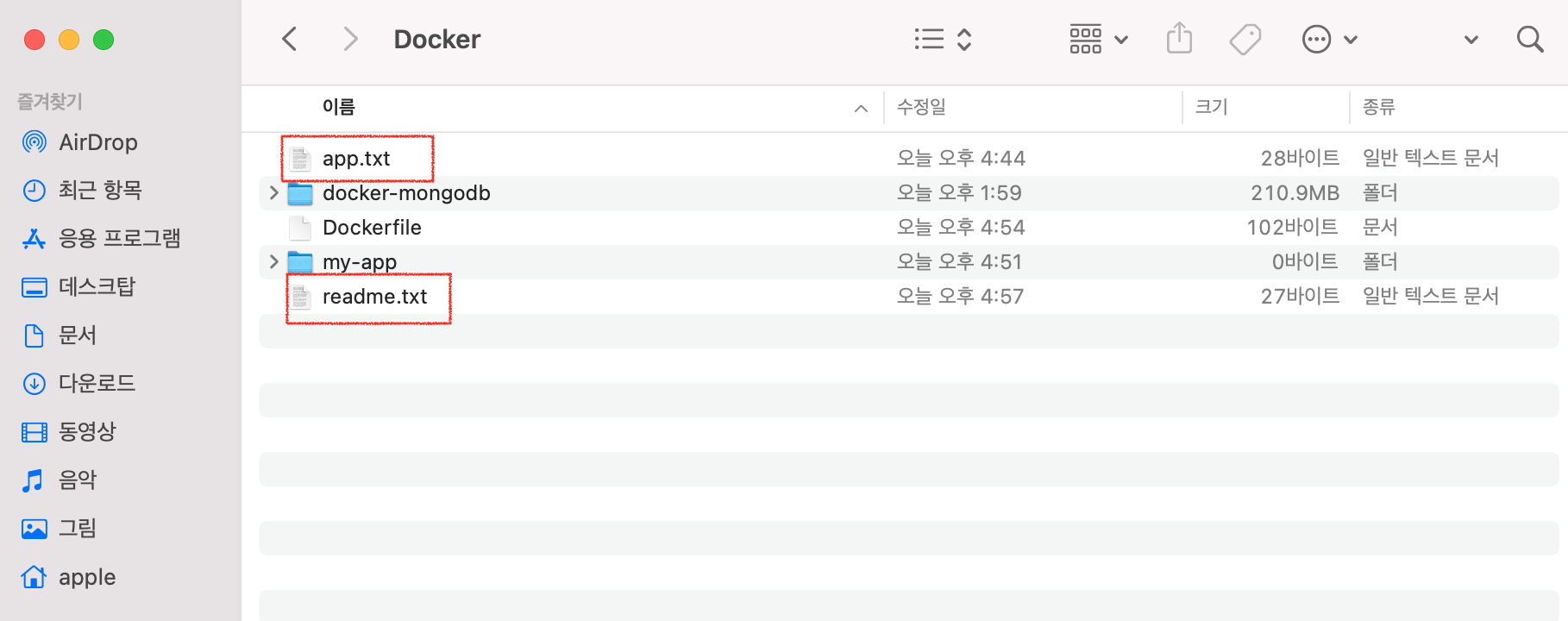
Dockerfile

| 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 |
| --- |



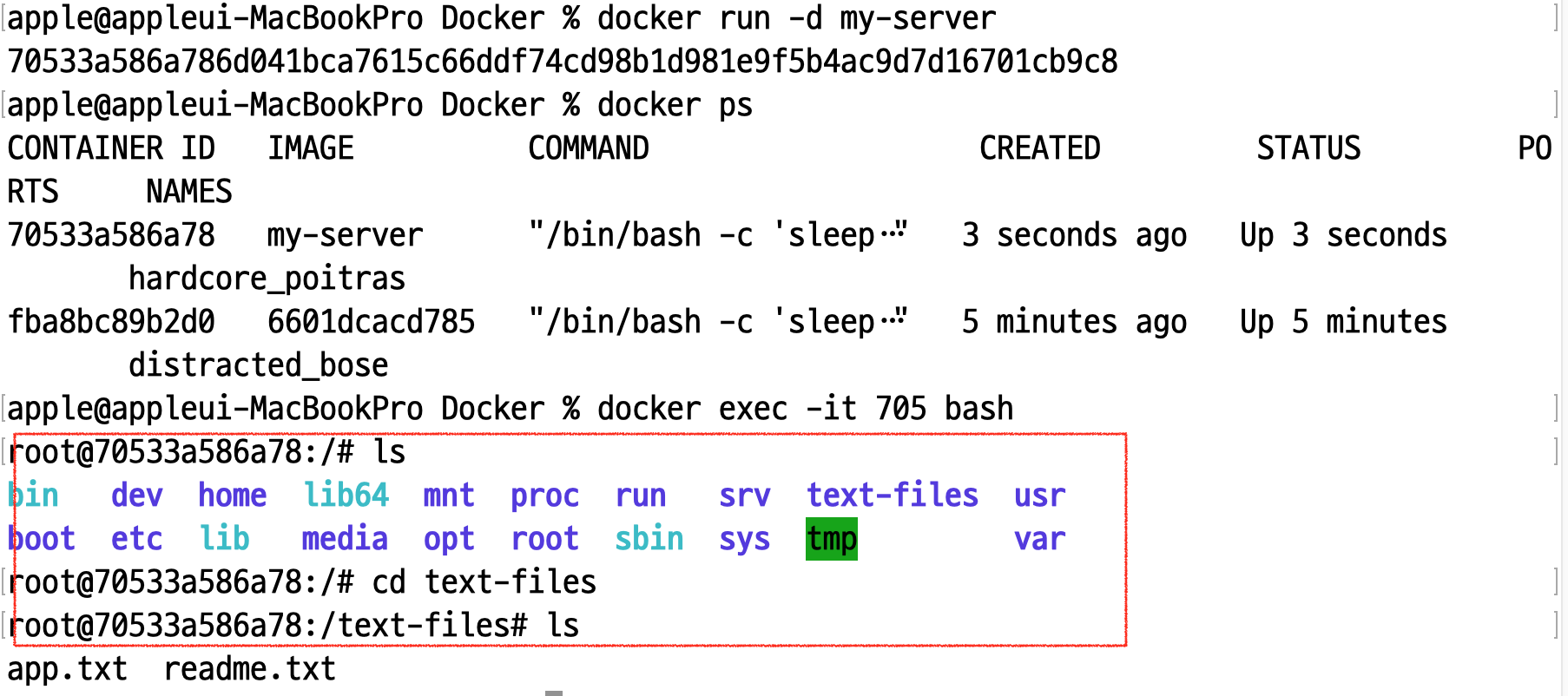
### **🎯 와일드 카드 사용해보기**

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 COPY를 하고 싶지 않을 수 있다. 그럴 때 .dockerignore를 활용한다.

1. .dockerignore 파일 만들기  
    .dockerignore

| readme.txt # 이파일을 제외 시켜라 |
| --- |

****

1. 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 |
| --- |

****

1. **ENTRYPOINT : 컨테이너가 시작할 때 실행되는 명령어**

### **✅ 의미**

ENTRYPOINT는 컨테이너가 생성되고 최초로 실행할 때 수행되는 명령어를 뜻한다. 쉽게 설명하자면 ENTRYPOINT에는 미니 컴퓨터의 전원을 켜고 나서 실행시키고 싶은 명령어를 적으면 된다.

### **✅ 사용법**

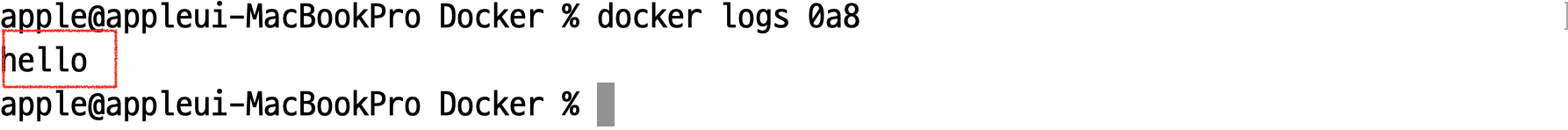
| # 문법  ENTRYPOINT [명령문...]  # 예시  ENTRYPOINT ["node", "dist/main.js"] |
| --- |

### **🎯 예제**

Dockerfile

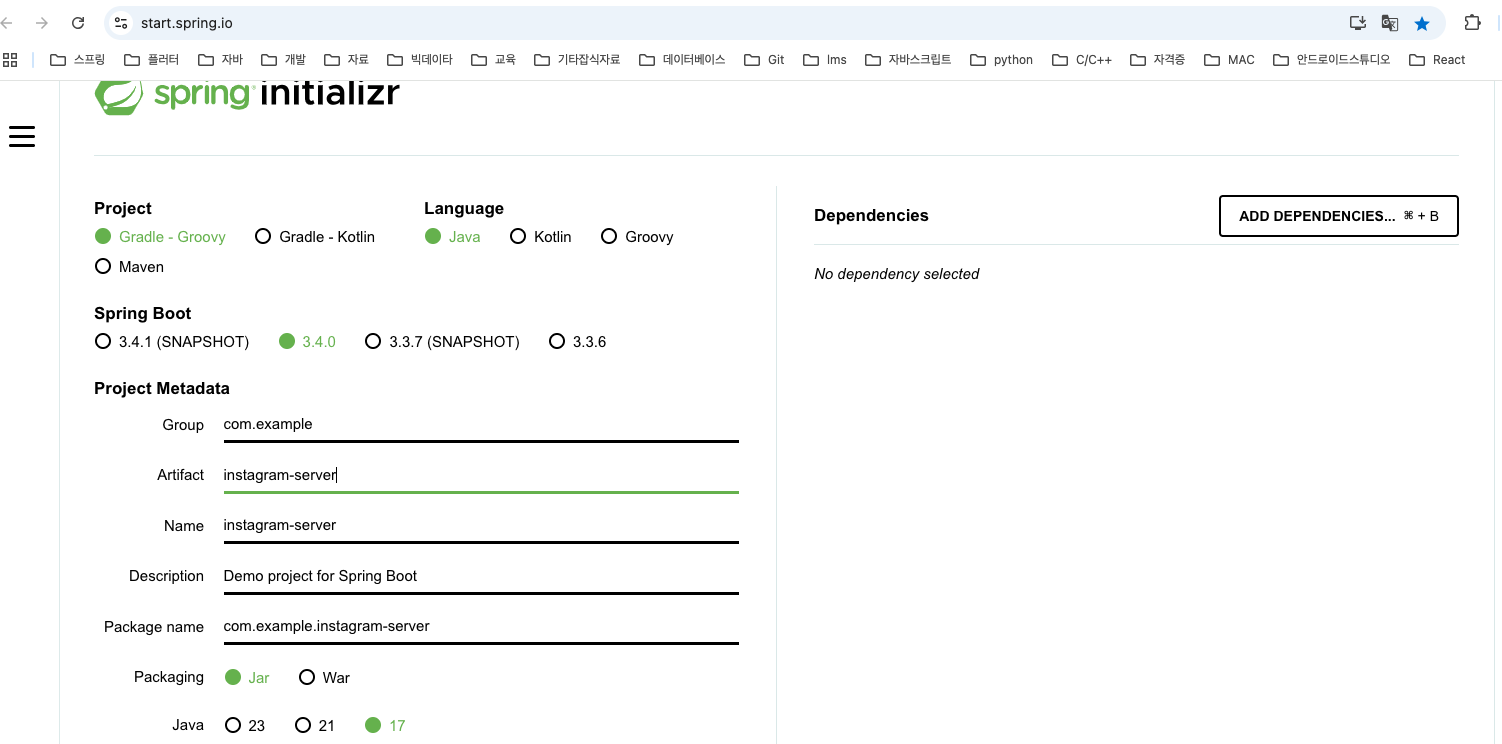
| FROM ubuntu  ENTRYPOINT ["/bin/bash", "-c", "echo hello"] |
| --- |

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



1. **[실습] 백엔드 프로젝트(Spring Boot) 프로젝트를 Docker로 실행시키기**

### **✅ 백엔드 프로젝트(Spring Boot) 프로젝트를 Docker로 실행시키기**

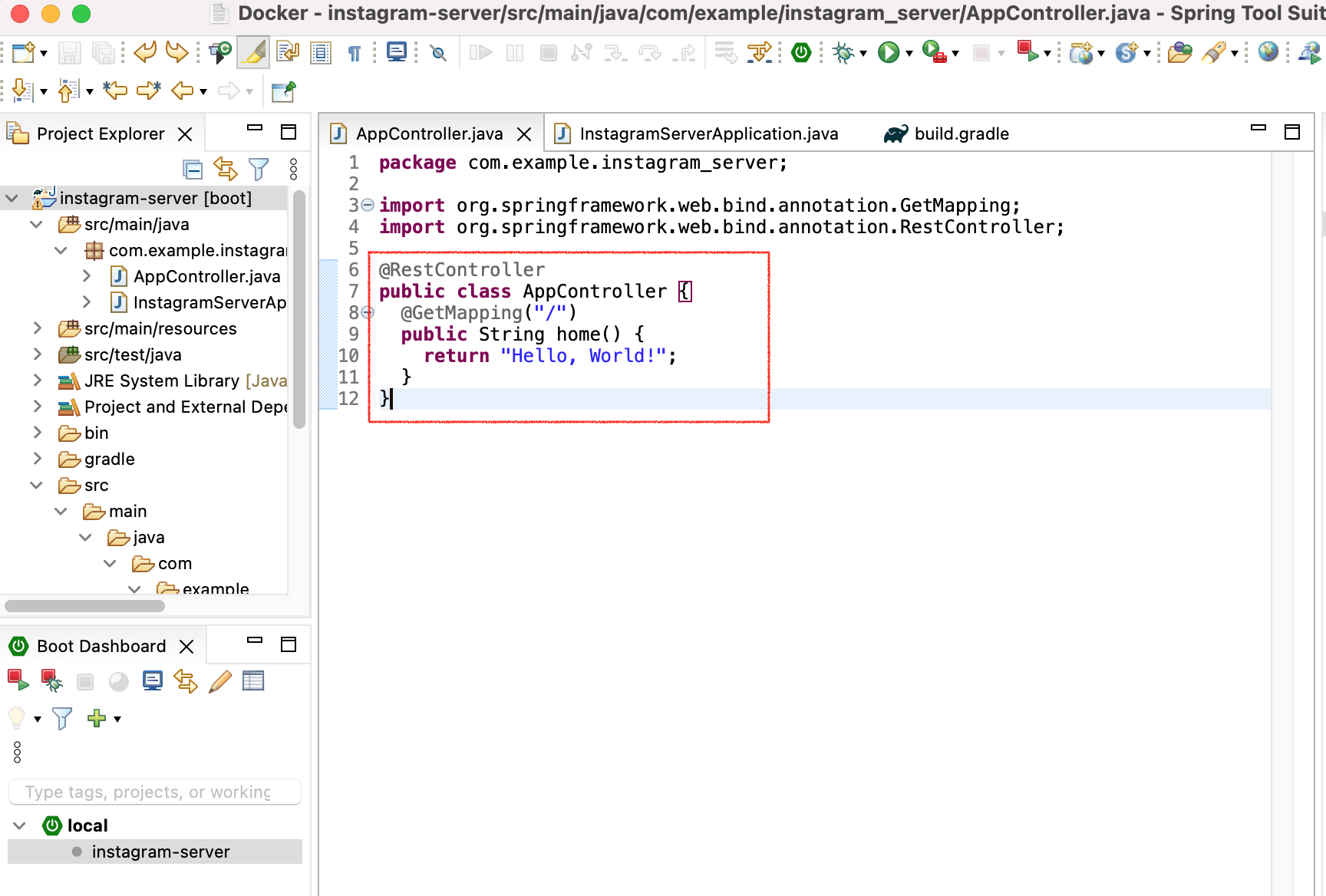
1. 프로젝트 셋팅  
    [start.spring.io](https://start.spring.io/) 
   * Java 17 버전을 선택하자. 아래 과정을 Java 17 버전을 기준으로 진행할 예정이다.
2. 간단한 코드 작성

의존성 추가

implementation 'org.springframework.boot:spring-boot-starter-web:3.4.0'

AppController.java

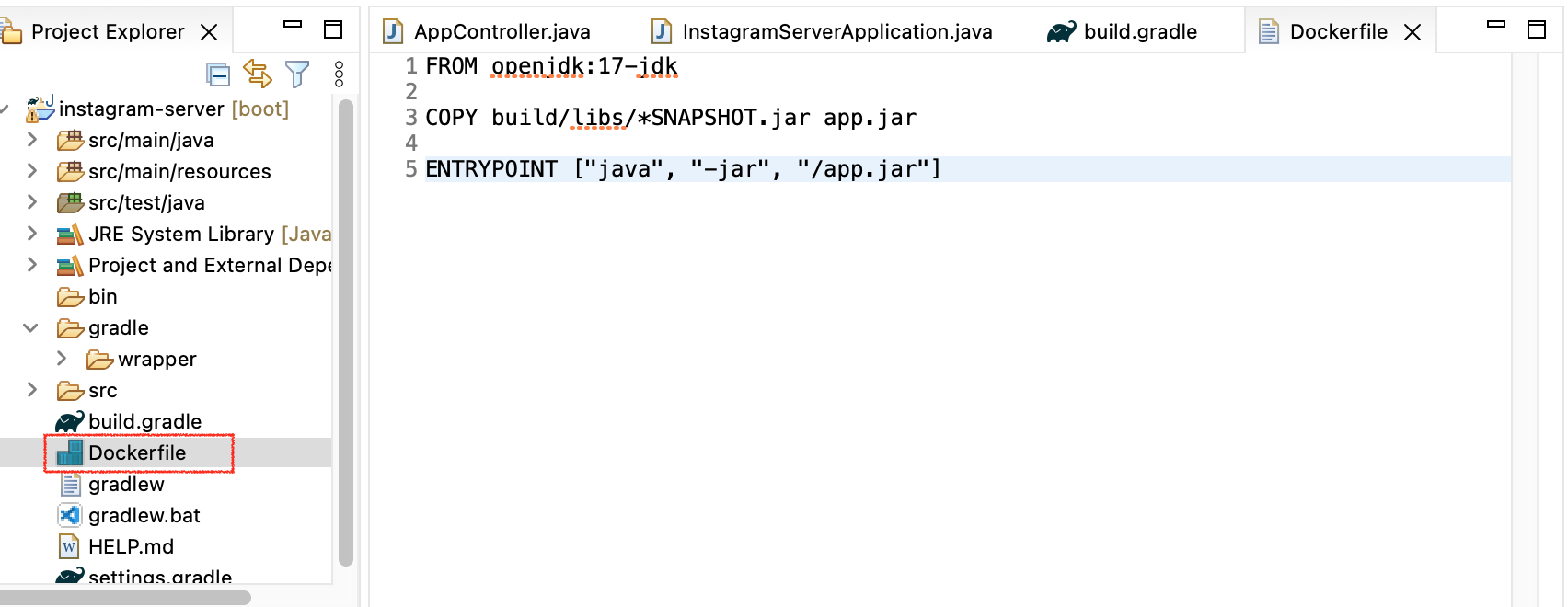
| @RestController  public class AppController {  @GetMapping("/")  public String home() {  return "Hello, World!";  }  } |
| --- |



1. Dockerfile 작성하기

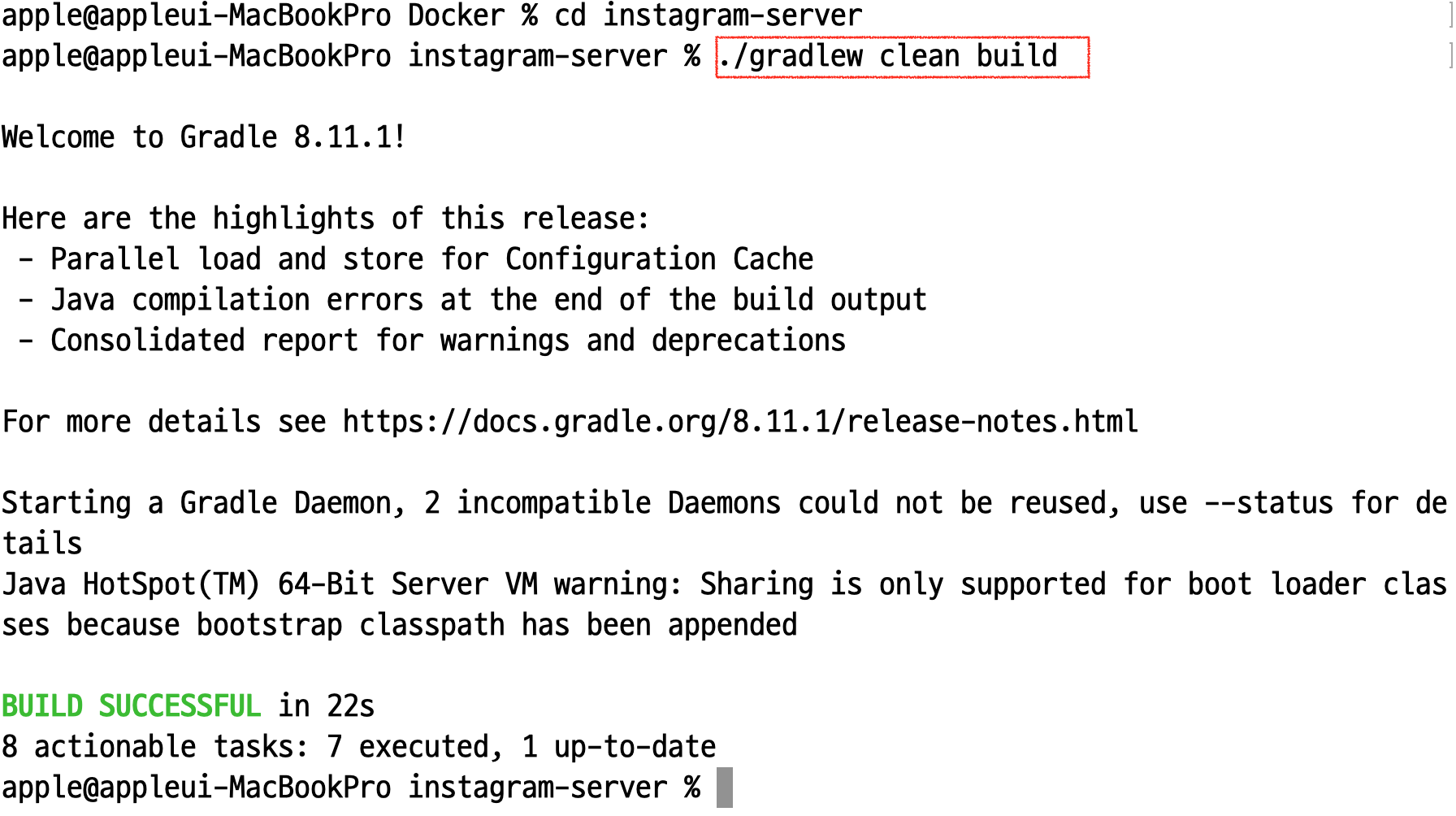
Dockerfile

| FROM openjdk:17-jdk  COPY build/libs/\*SNAPSHOT.jar app.jar  ENTRYPOINT ["java", "-jar", "/app.jar"] |
| --- |



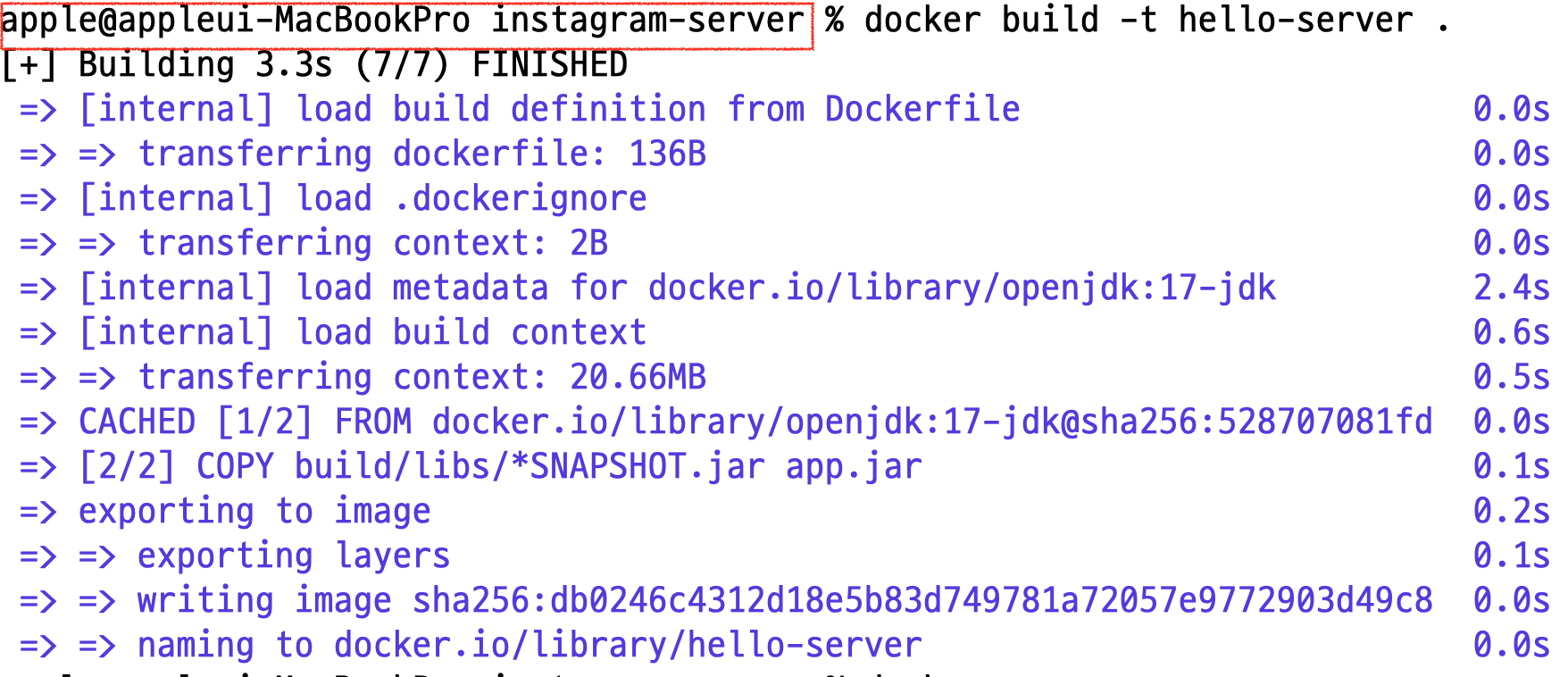
1. Spring Boot 프로젝트 빌드하기(스프링 터미널에서 하기)

| $ ./gradlew clean build |
| --- |



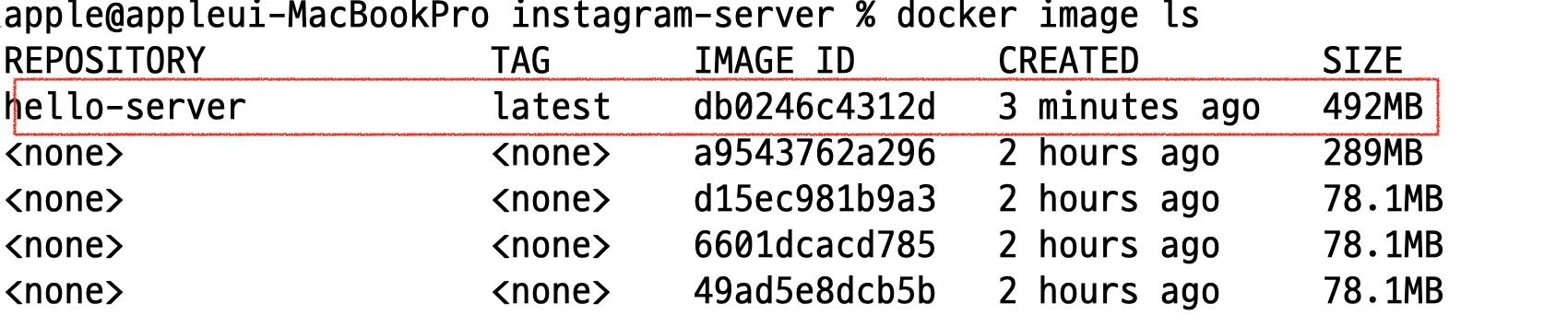
1. Dockerfile을 바탕으로 이미지 빌드하기

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



1. 이미지가 잘 생성 됐는지 확인하기

| $ docker image ls |
| --- |

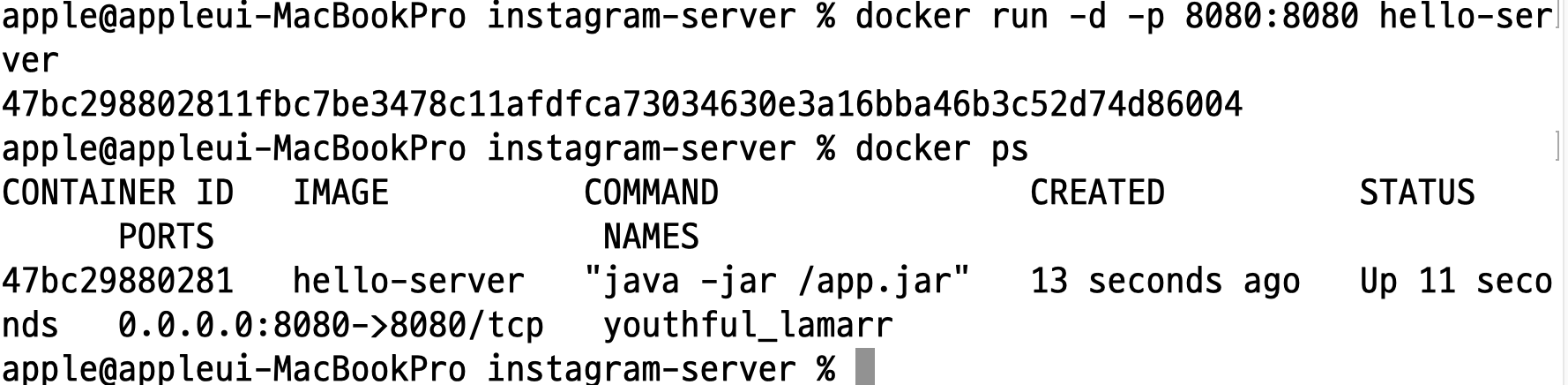


1. 생성한 이미지를 컨테이너로 실행시켜보기

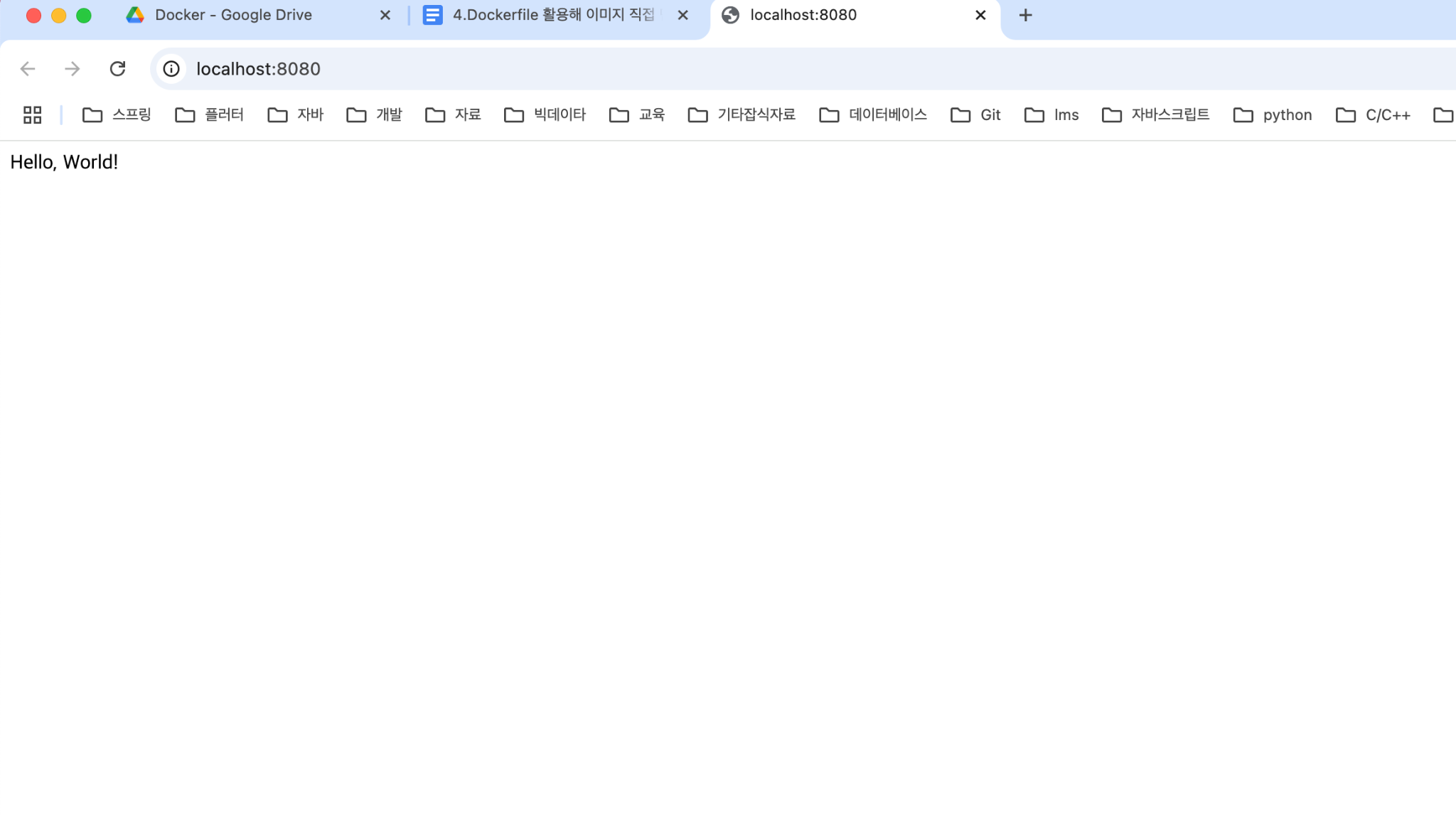
| $ docker run -d -p 8080:8080 hello-server |
| --- |

1. 컨테이너 잘 실행되고 있는 지 확인하기

| $ docker ps |
| --- |



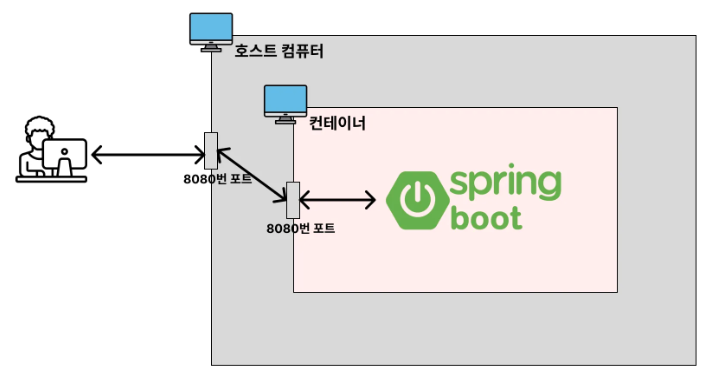
1. l[**ocalhost:8080**](http://localhost:8080)으로 들어가보기



1. 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

| $ docker stop {컨테이너 ID}  $ docker rm {컨테이너 ID}  $ docker image rm {이미지 ID} |
| --- |

### ✅ 그림으로 이해하기

****

1. **RUN : 이미지를 생성하는 과정에서 사용할 명령문 실행**

#### **✅ 의미**

**RUN은 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. RUN은 ‘이미지 생성 과정’에서 필요한 명령어를 실행시킬 때 사용하고, ENTRYPOINT는 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 Dockerfile을 활용해 ubuntu, git이 깔려있는 이미지를 만들면 된다.**

1. **Dockerfile 작성하기**

**Dockerfile**

| **FROM ubuntu # 우분투 환경에**  **RUN apt update && apt install -y git # 우분투 기본 이미지에 깃을 설치**  **ENTRYPOINT ["/bin/bash", "-c", "sleep 500"]** |
| --- |

1. **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** |
| --- |

****

1. **WORKDIR : 작업 디렉토리를 지정**

#### **✅ 의미**

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

#### **✅ 사용법**

| **# 문법**  **WORKDIR [작업 디렉토리로 사용할 절대 경로]**  **# 예시**  **WORKDIR /usr/src/app** |
| --- |

작업 디렉토리 만들기

| $ mkdir docker-workdir  $ cd docker-workdir |
| --- |

**​🎯 예제**

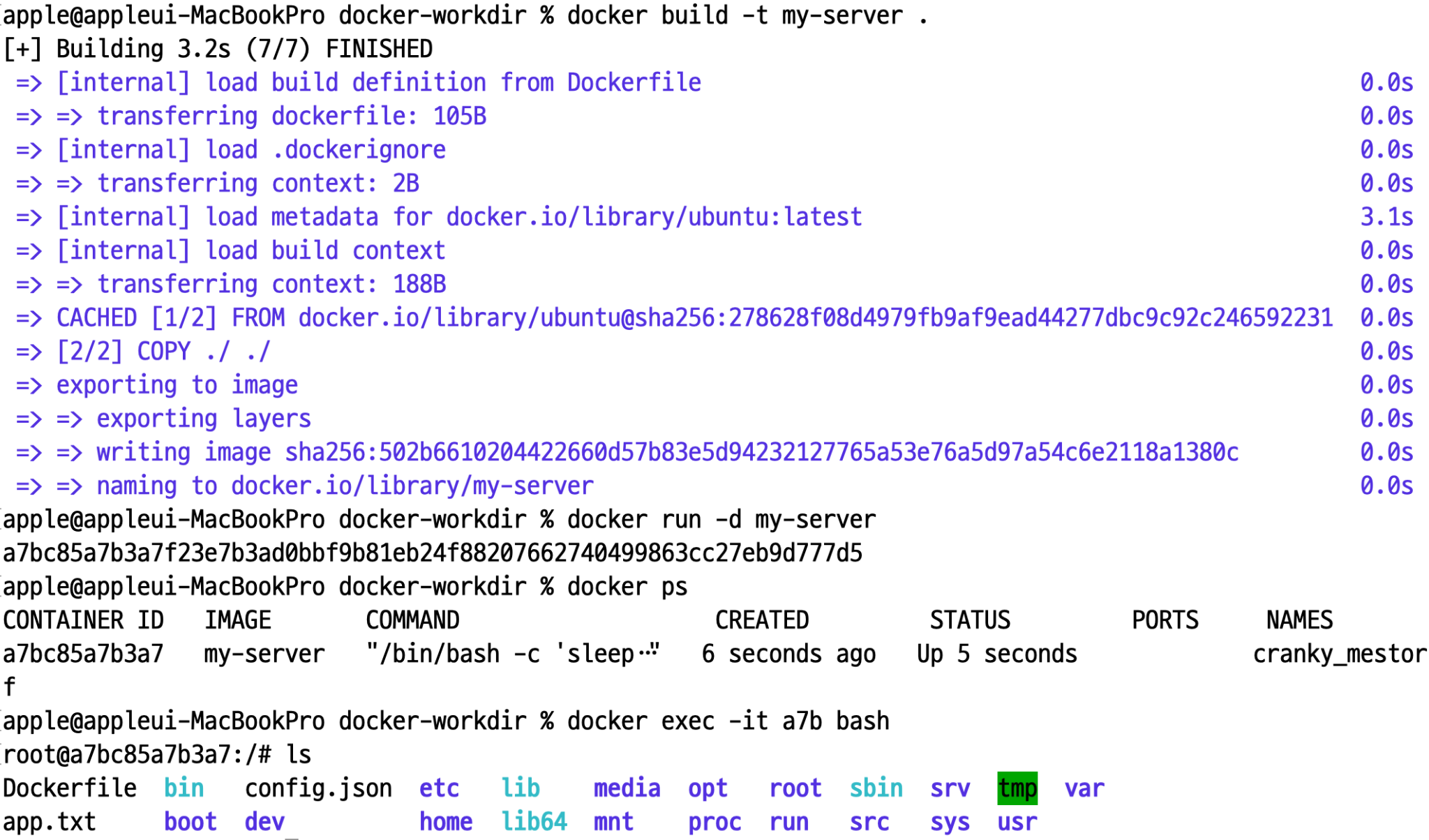
1. **app.txt, src, config.json 파일 만들기**
2. **Dockerfile 만들어서 이미지 생성 및 컨테이너 실행**

**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** |
| --- |

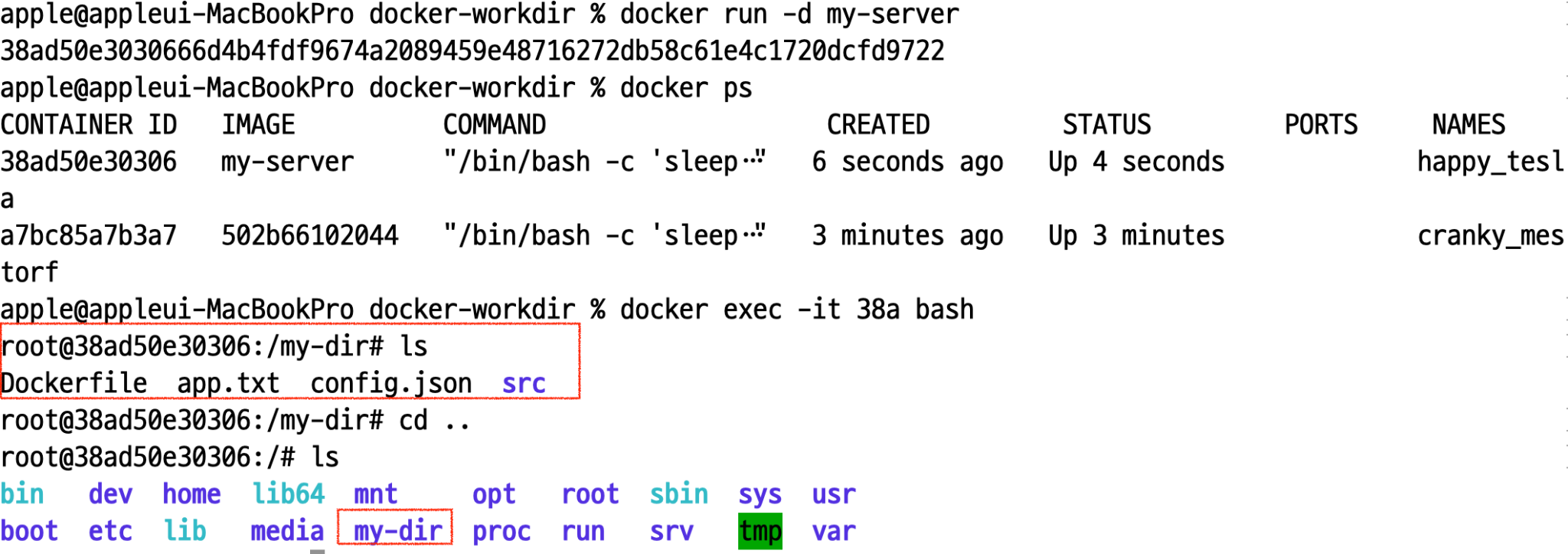
****

**WORKDIR을 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** |
| --- |

****

1. **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** |
| --- |

1. **[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 |
| --- |

1. **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. [**Docker Layer와 Cache**](https://medium.com/@hee98.09.14/docker-layer%EC%99%80-cache-574c12a1e9f7)

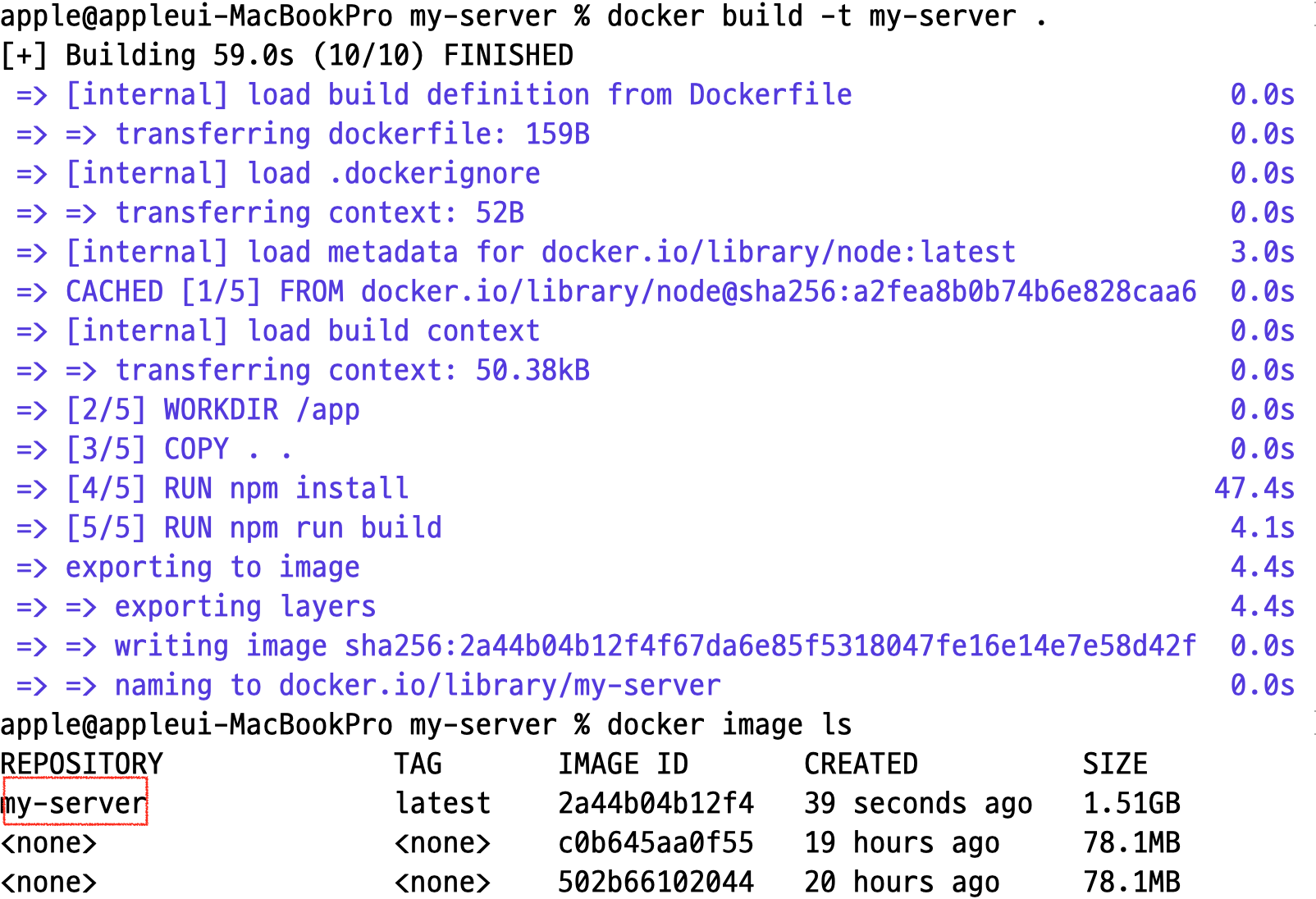
1. **.dockerignore 작성하기** .dockerignore

| node\_modules |
| --- |

이미지를 생성할 때 npm install을 통해 처음부터 깔끔하게 필요한 의존성만 설치한다. 따라서 호스트 컴퓨터에 있는 node\_modules는 컨테이너로 복사해갈 필요가 없다.

1. **Dockerfile을 바탕으로 이미지 빌드하기**

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

****

1. **이미지가 잘 생성 됐는지 확인하기**

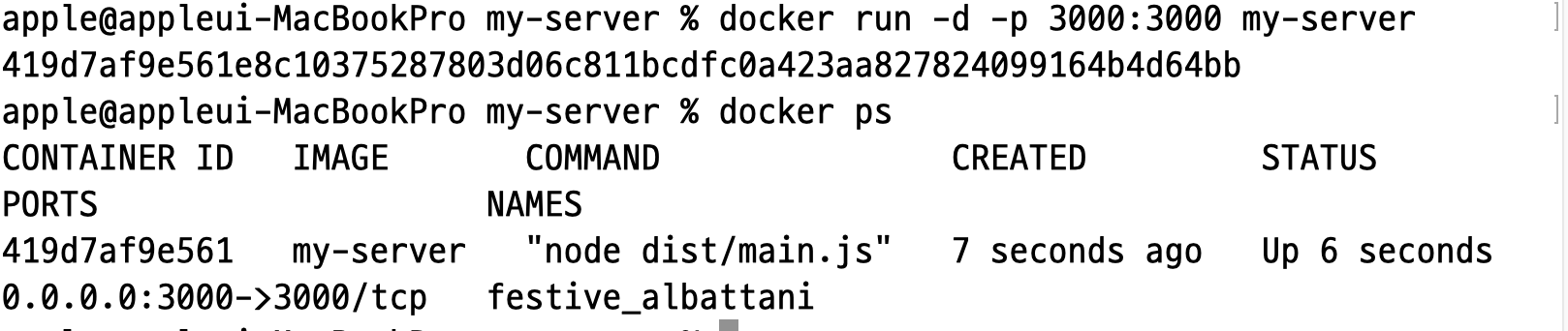
| $ docker image ls |
| --- |

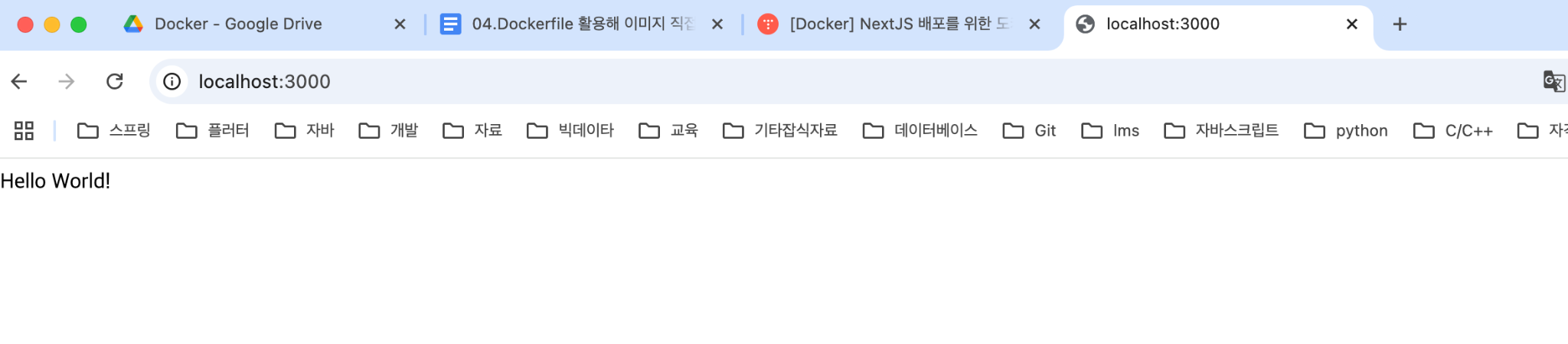
1. **생성한 이미지를 컨테이너로 실행시켜보기**

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

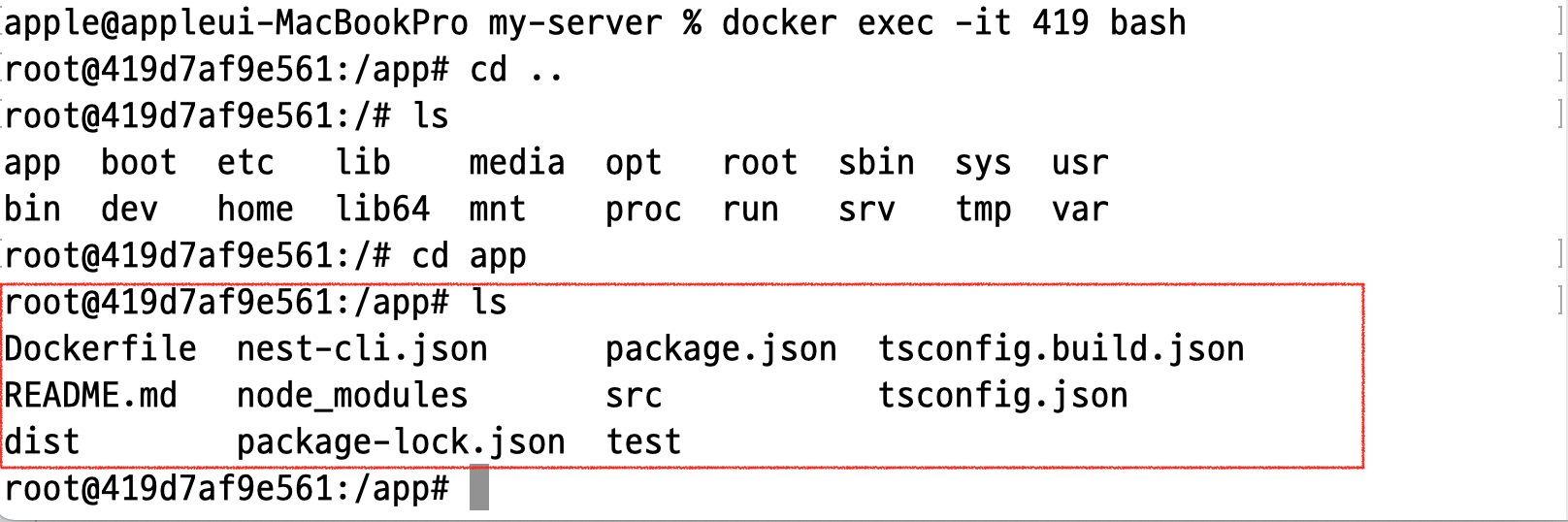
1. **컨테이너 잘 실행되고 있는 지 확인하기**

| $ docker ps |
| --- |

****

1. **localhost:3000으로 들어가보기  
     
   **
2. **파일 복사 한 것 확인하기**

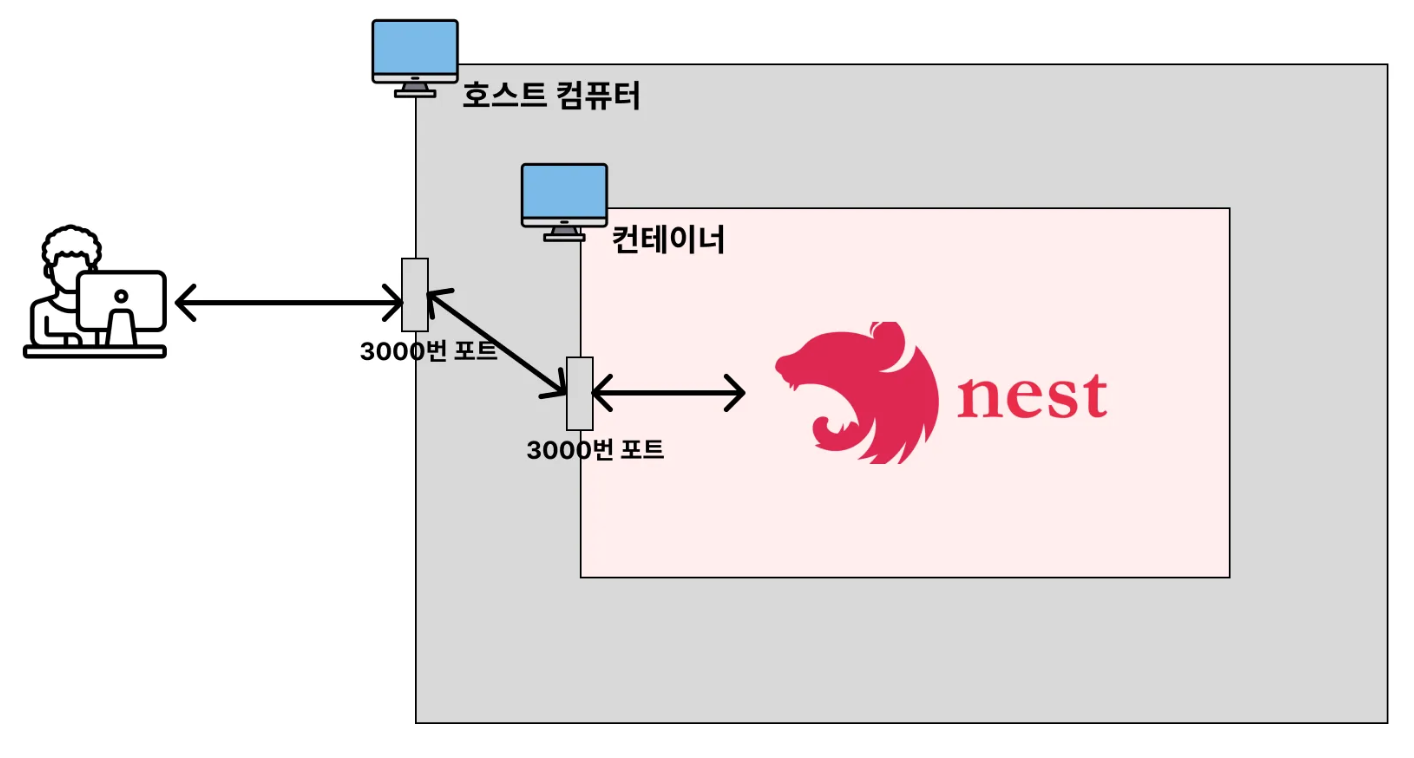
| $ docker exec -it 419 bash |
| --- |

****

1. **실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기**

| $ docker stop {컨테이너 ID}  $ docker rm {컨테이너 ID}  $ docker image rm {이미지 ID} |
| --- |

### **✅ 그림으로 이해하기**

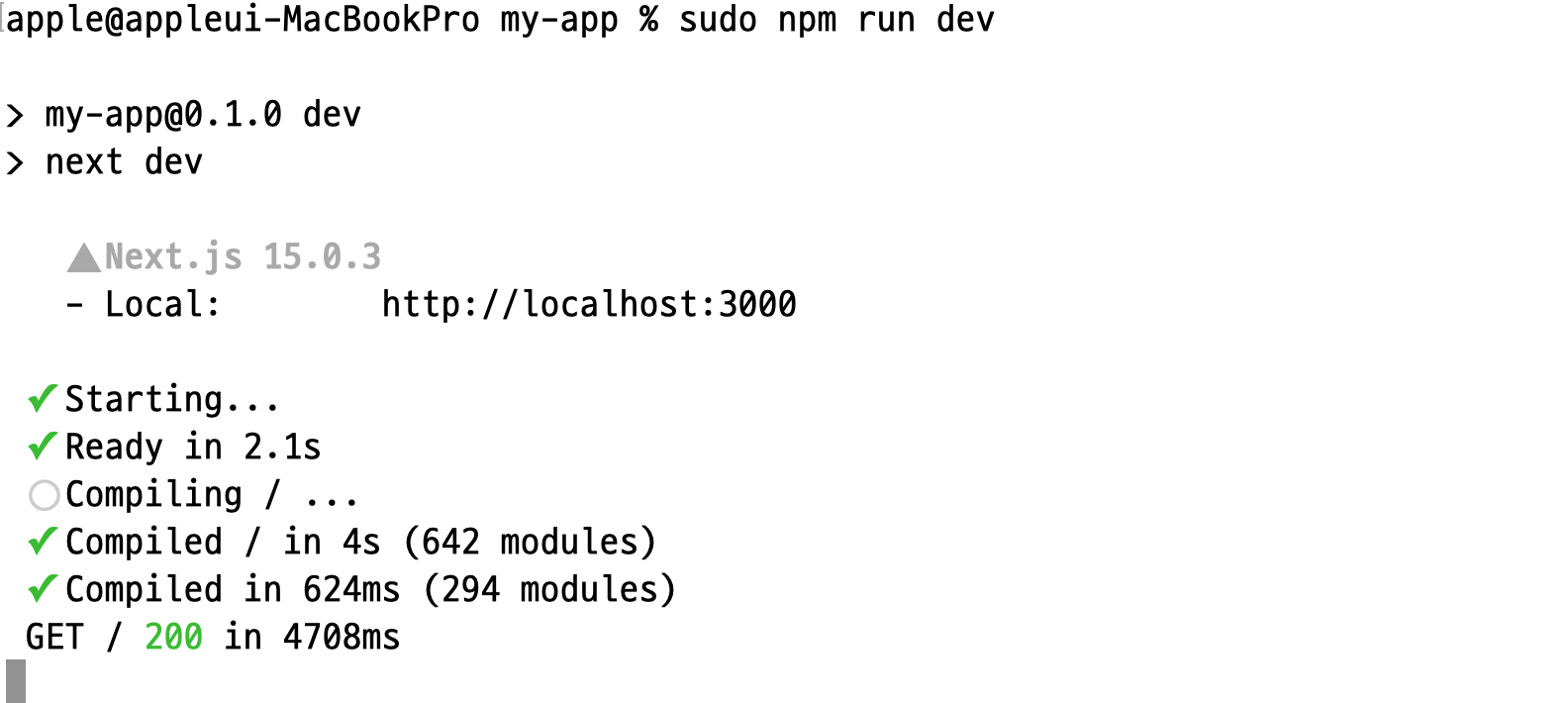
****

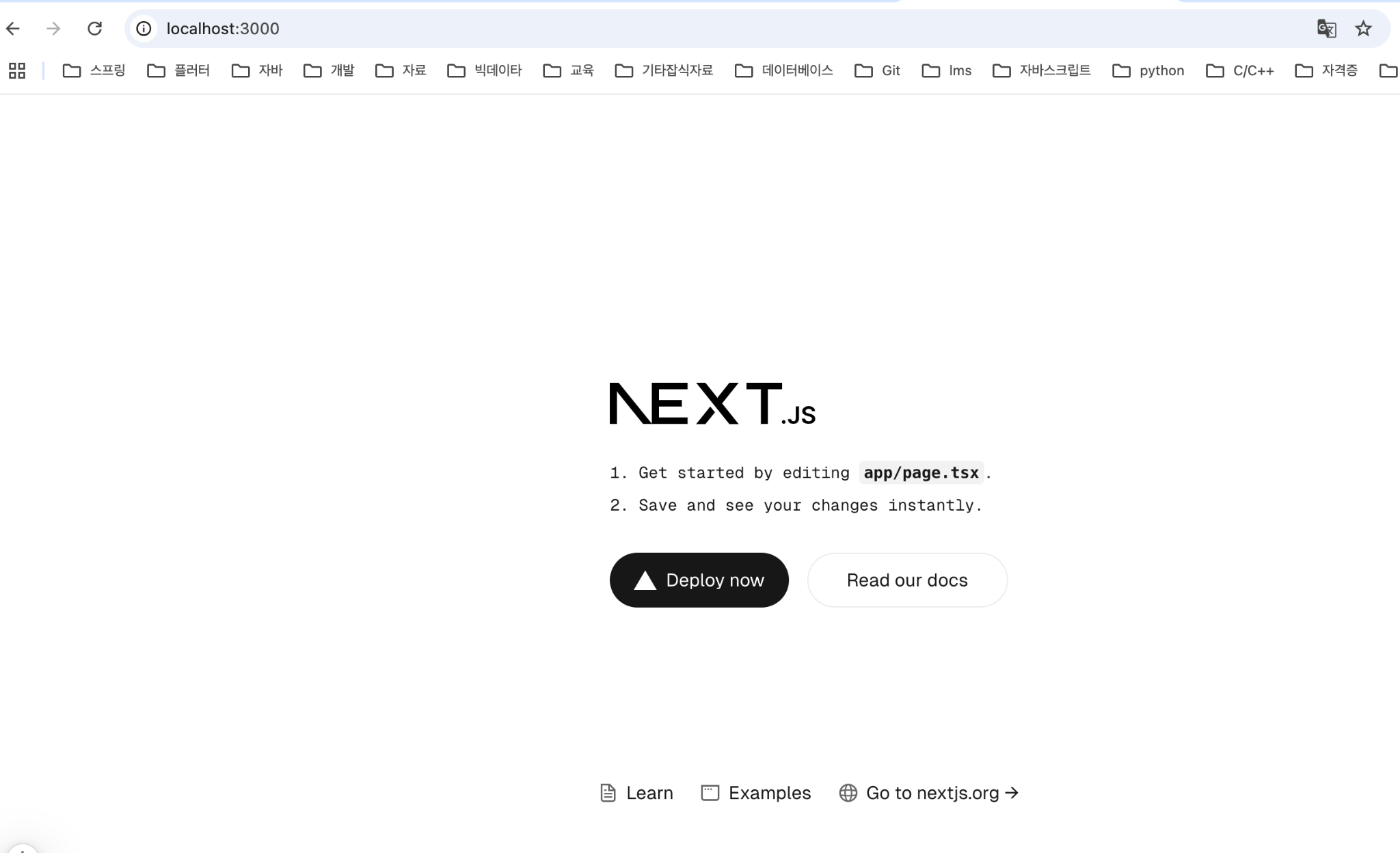
1. **[실습] 웹 프론트엔드 프로젝트(Next.js)를 Docker로 배포하기**

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

1. **Next.js 프로젝트 만들기**

| $ npx create-next-app@latest  $npm run dev # 실행해 보기 안되면 앞에 sudo 넣기 |
| --- |

****

****

1. **Dockerfile 작성하기**맥북에서 파일이 안 만들어지면 권한 설정 해 줄 것

| $ sudo chmod -R 777 /Users/apple/DevData/Docker/my-app |
| --- |

**Dockerfile**

| FROM node:20-alpine  WORKDIR /app  COPY . .  RUN npm install  RUN npm run build  EXPOSE 3000  ENTRYPOINT [ "npm", "run", "start" ] |
| --- |

* + **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.** [**Docker Layer와 Cache**](https://medium.com/@hee98.09.14/docker-layer%EC%99%80-cache-574c12a1e9f7)
  + **alpine : 불필요한 프로그램을 포함하지 않고 이미지 크기를 최소화한 버전. 실제 배포 할 때도 되도록이면 alpine 버전을 사용한다.**

1. **dockerignore 작성하기** .dockerignore

| node\_modules |
| --- |

**이미지를 생성할 때 npm install을 통해 처음부터 깔끔하게 필요한 의존성만 설치한다. 따라서 호스트 컴퓨터에 있는 node\_modules는 컨테이너로 복사해갈 필요가 없다.**

1. **Dockerfile을 바탕으로 이미지 빌드하기**

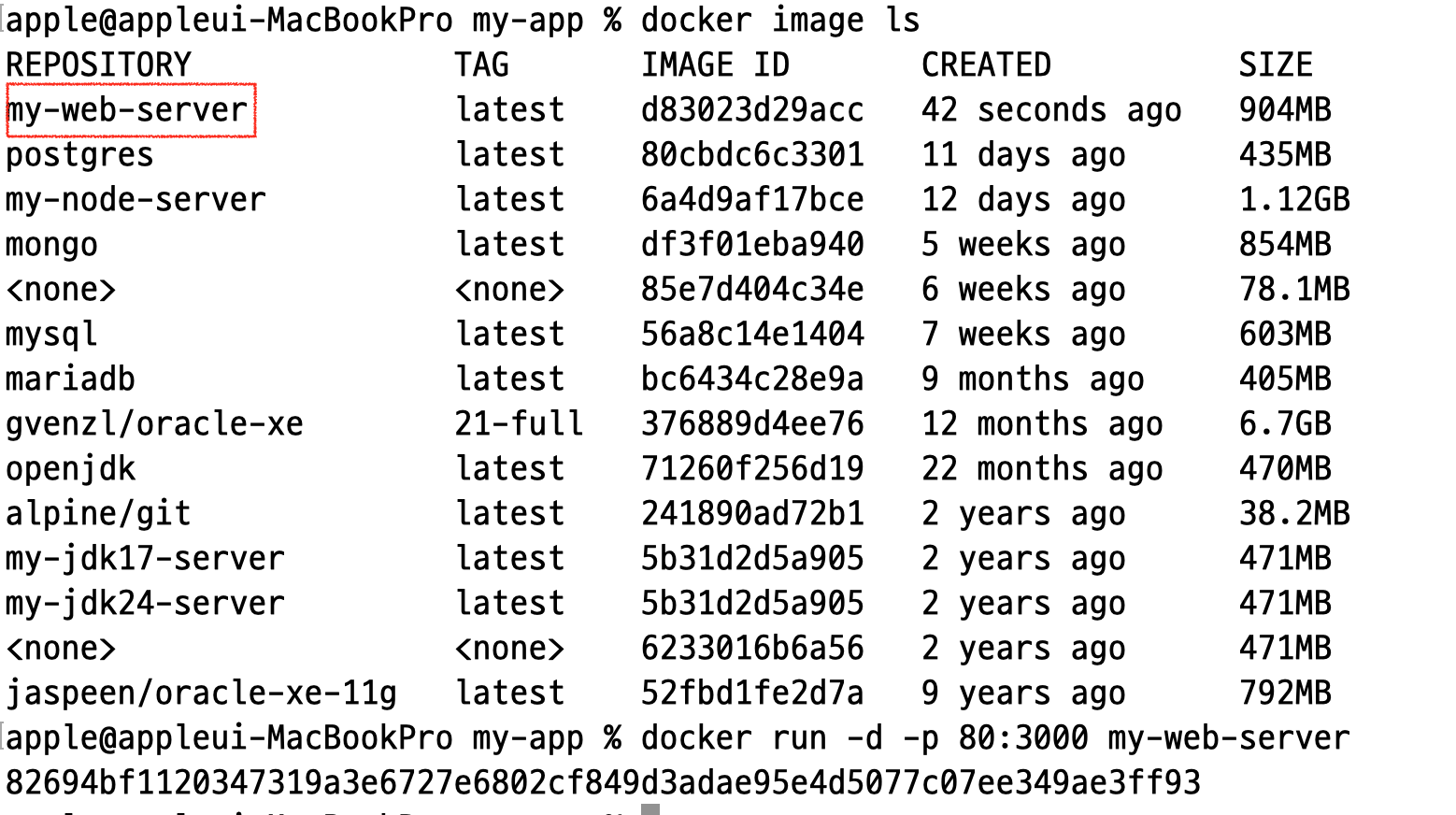
| $ docker build -t my-web-server . |
| --- |

1. **이미지가 잘 생성됐는 지 확인하기**

| $ docker image ls |
| --- |

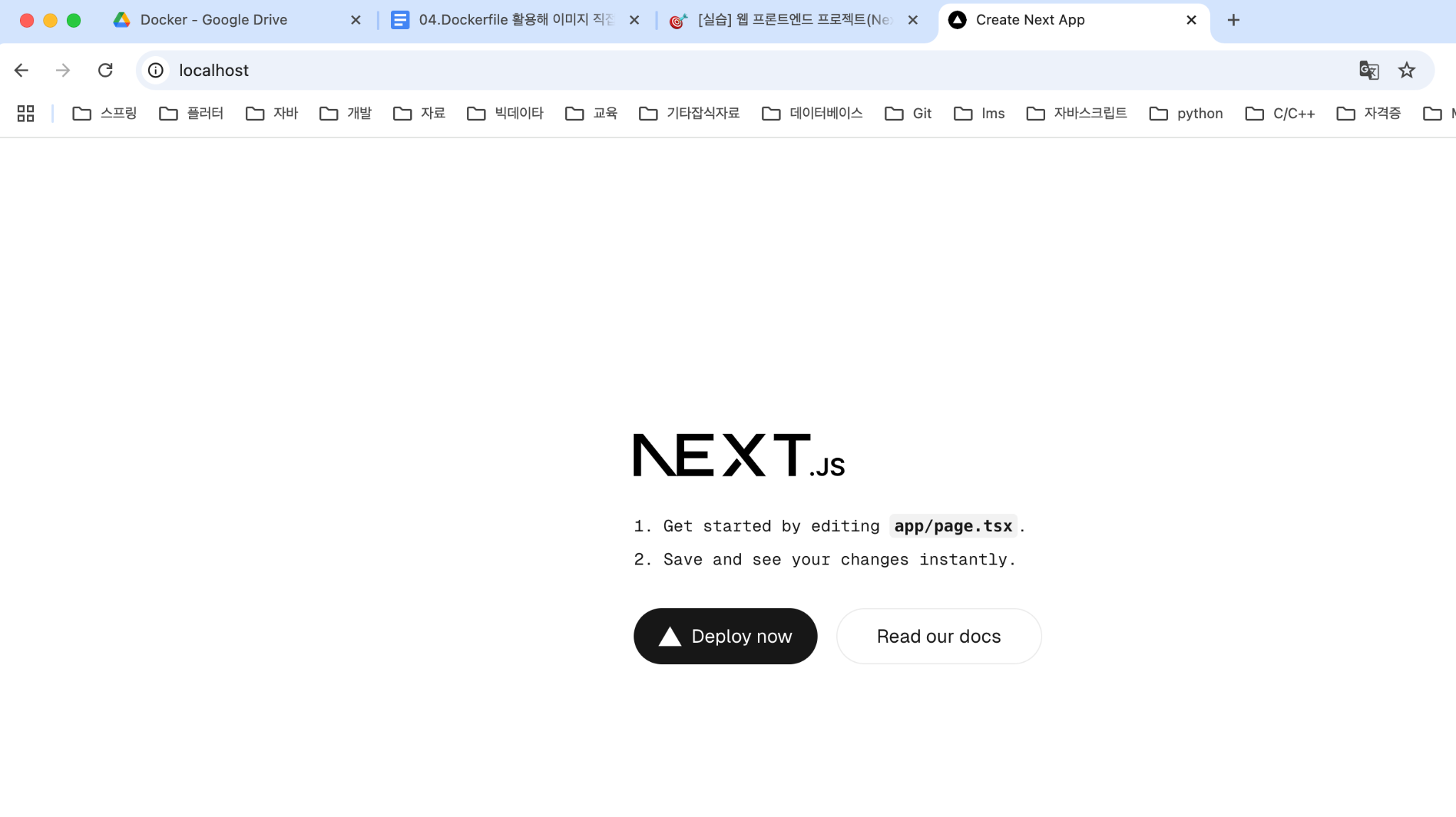
1. **생성한 이미지를 컨테이너로 실행시켜보기**

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

****

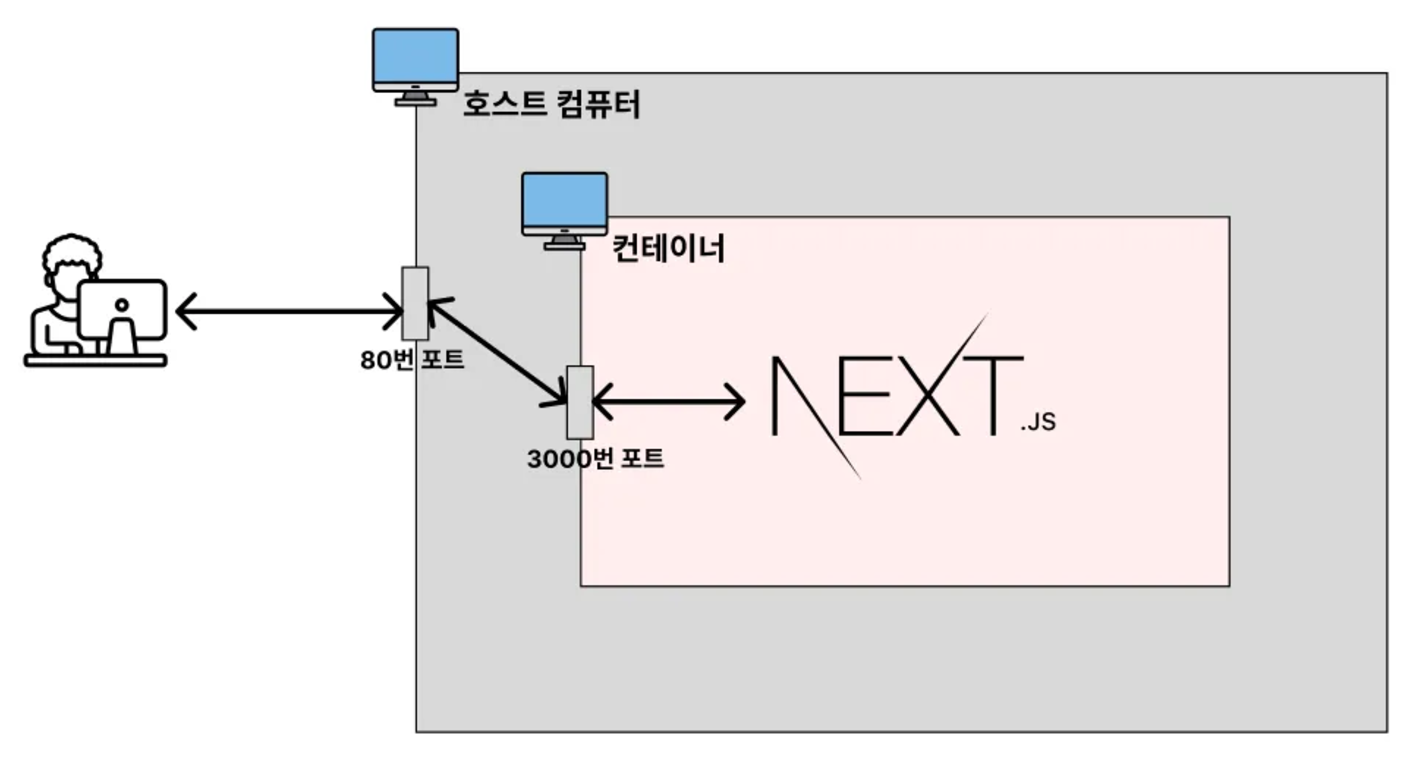
1. **컨테이너 잘 실행되고 있는 지 확인하기**

| $ docker ps |
| --- |

1. **localhost:80으로 들어가보기  
     
   **
2. **실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기**

| $ docker stop {컨테이너 ID}  $ docker rm {컨테이너 ID}  $ docker image rm {이미지 ID} |
| --- |

### **✅ 그림으로 이해하기**

****

1. **[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>  <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.html이라고 지어야 한다.  
  
 style.css

| \* {  color: blue;  } |
| --- |

1. **Dockerfile 작성하기**  
     
    [nginx - Official Image | Docker Hub](https://hub.docker.com/_/nginx) Dockerfile

| FROM nginx  COPY ./ /usr/share/nginx/html |
| --- |

1. **Dockerfile을 바탕으로 이미지 빌드하기**

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

1. **이미지가 잘 생성 됐는지 확인하기**

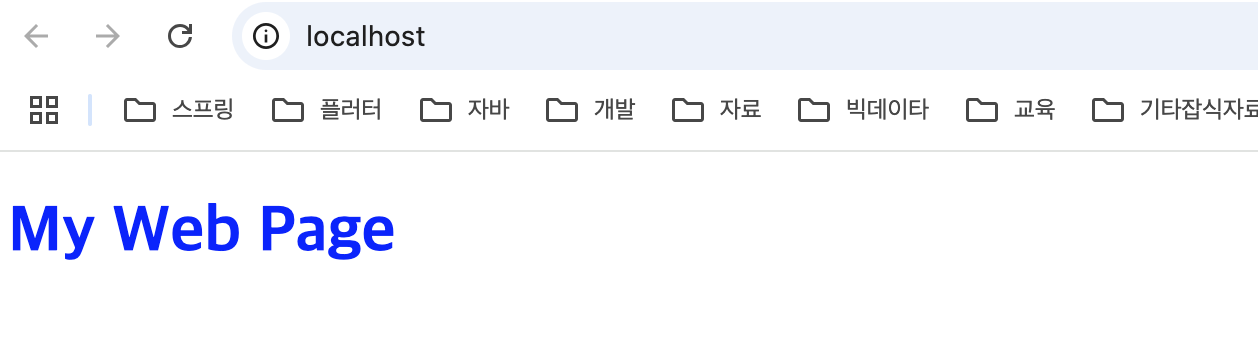
| $ docker image ls |
| --- |

1. 생성한 이미지를 컨테이너로 실행시켜보기

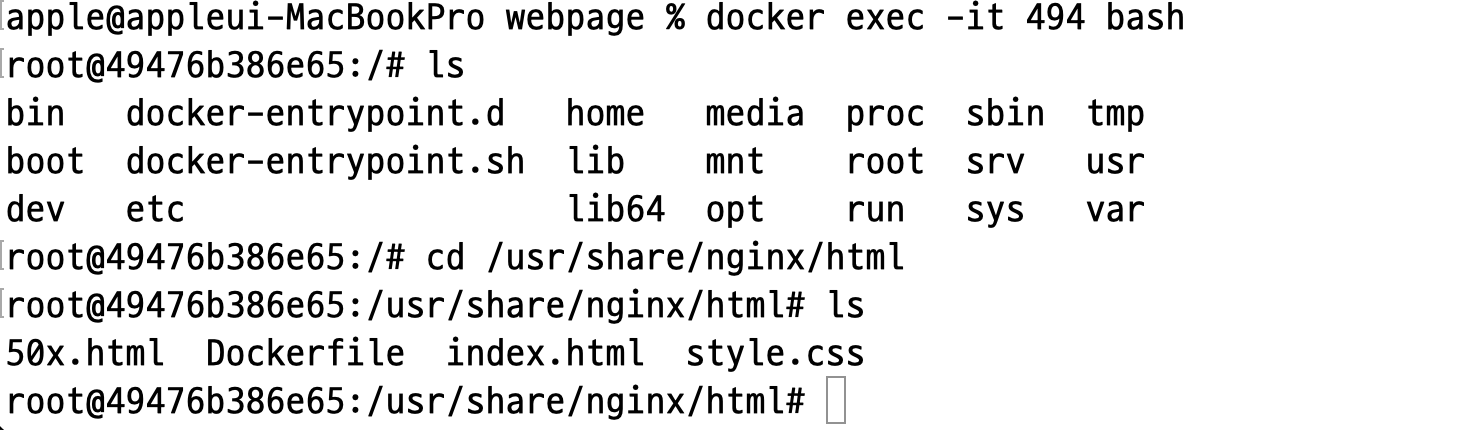
| $ docker run -d -p 80:80 my-web-server |
| --- |

1. 컨테이너 잘 실행되고 있는 지 확인하기

| $ docker ps |
| --- |

1. [**localhost:80**](http://localhost:80)**으로 들어가보기**  
   폴더 확인하기

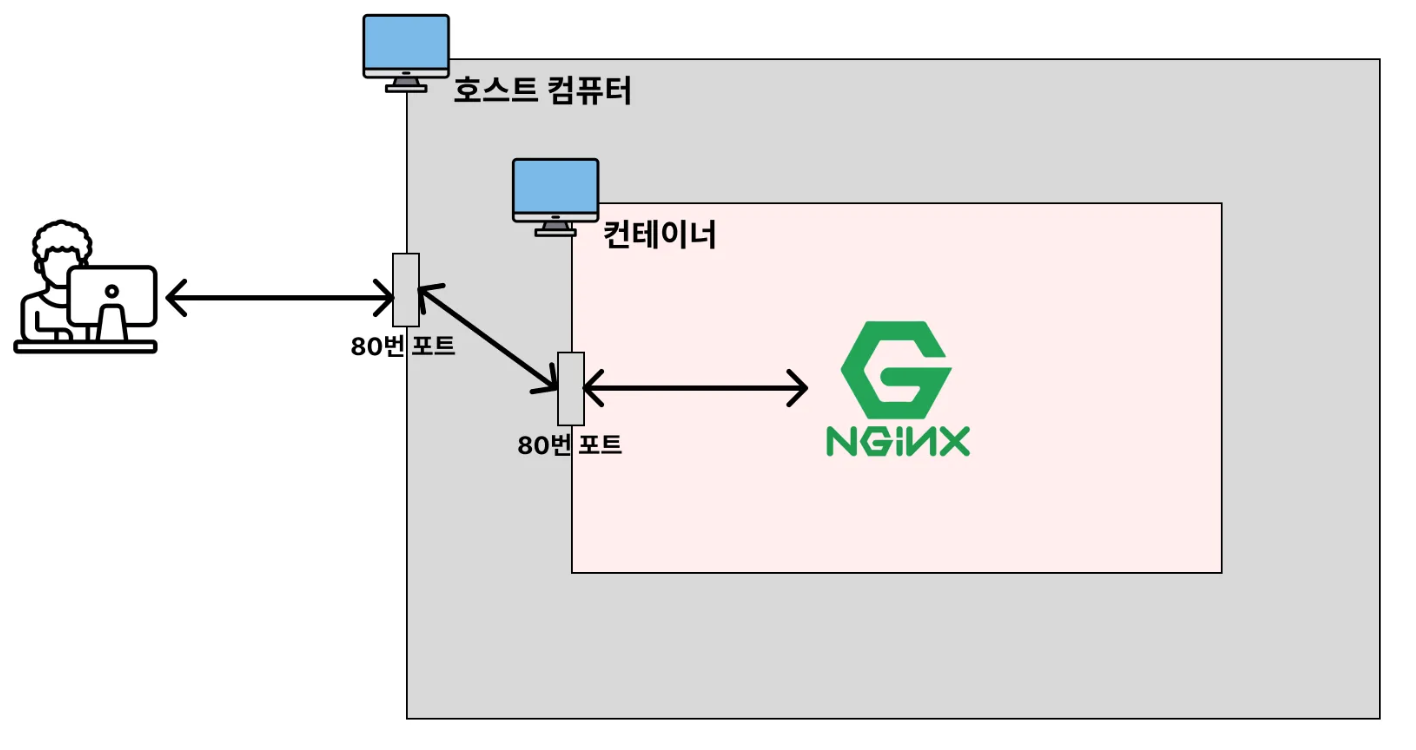
| $ docker exec -it 콘테이너ID bash  #/ cd /usr/share/nginx/html |
| --- |



1. 실행시킨 컨테이너 중지 / 삭제하기, 이미지 삭제하기

| $ docker stop {컨테이너 ID}  $ docker rm {컨테이너 ID}  $ docker image rm {이미지 ID} |
| --- |

### **✅ 그림으로 이해하기**

****