도커를 이용한 간단한 Node.js 어플 만들기

섹션 설명

이번 4강에서는

- Node js 공식 홈페이지에서 도커를 이용하여 Node.js를 이용하는 예시 부분을 사용하여 도커를 실전에 도입하는 연습을 해보겠습니다.

↓

https://nodejs.org/fr/docs/guides/nodejs-docker-webapp/

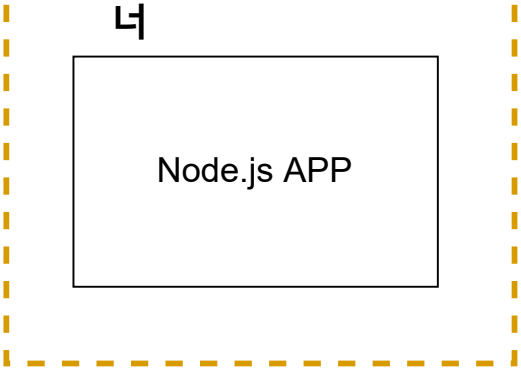
이번 강에서 가장 중점적으로 봐야 할 것은!

- 아래 보이는 Dockerfile을 어떤 식으로 작성하는지 중점적으로 다루겠습니다.

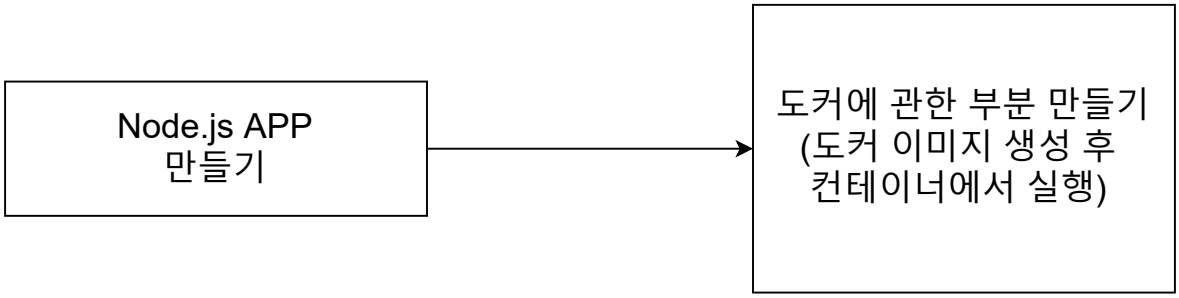
```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

이번 강의 전체적인 흐름

도커 컨테이



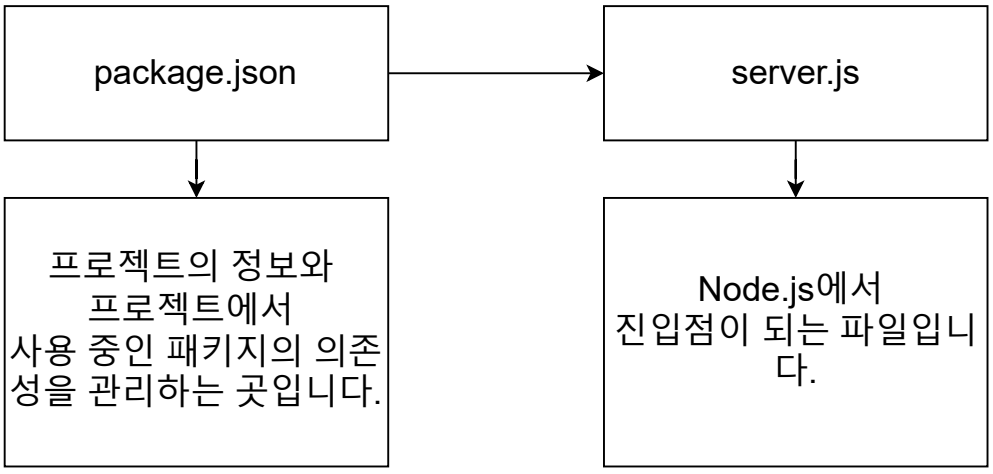
도커를 이용해서 노드 애플리케이션을 만드는



Node.js 앱 만들기

도커 강의이지만 Node.js로 실습을 하다 보니
Node.js에 사용 방법에 대한 설명을 조금씩 하면서
넘어가게 됩니다.

Node.js 앱을 만드는데 필요한 두 가지 파일



package.json 만들기

NPM

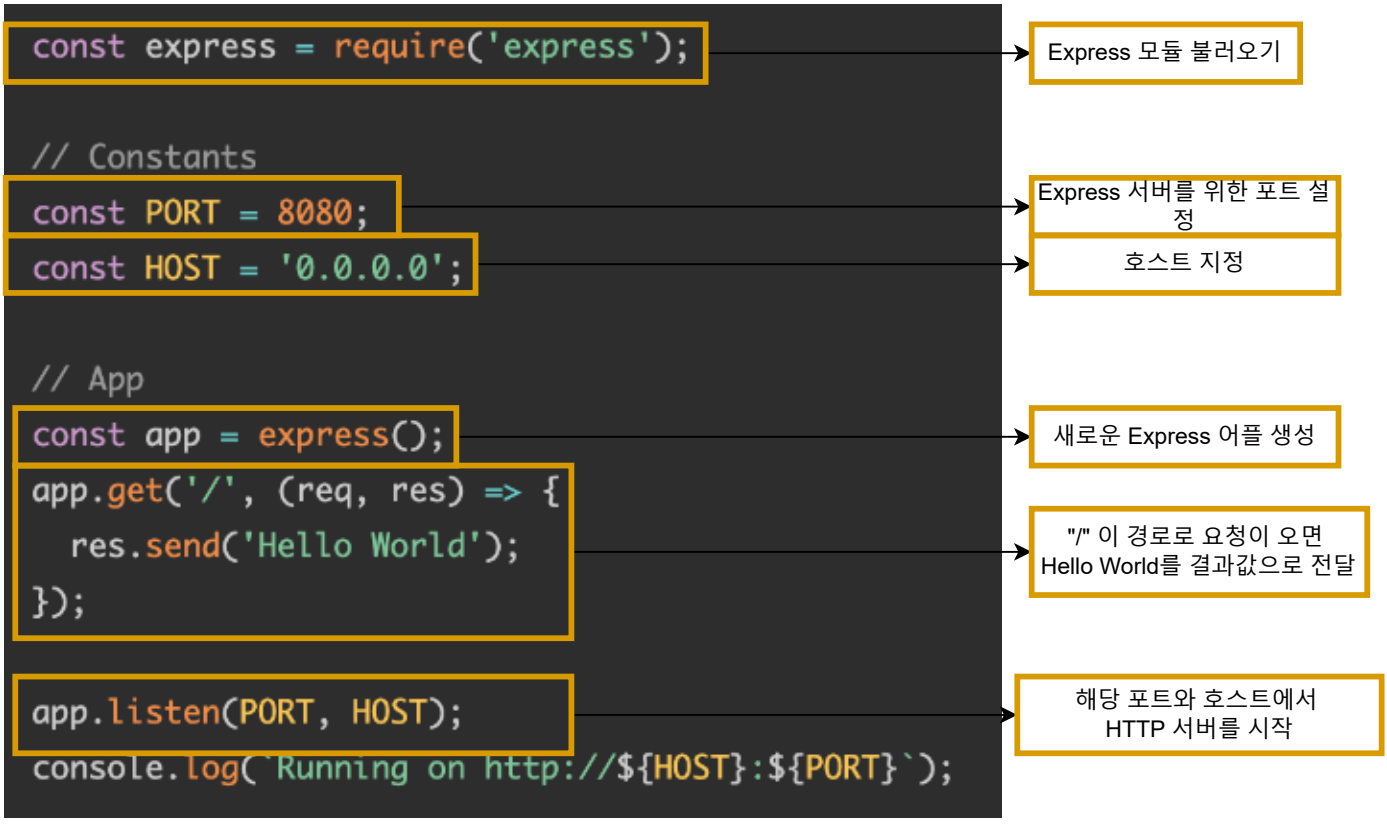
INIT

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
```

Javascript와 jQuery의 관계처럼
Node.js의 API를 단순화하고 새로운
기능을 추가해서 Node.js를 더 쉽고 유
용하게 사용할 수 있게 해 줍니다.

```
}
```

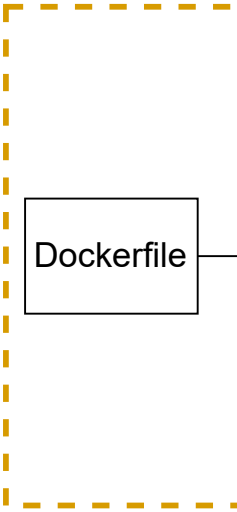
server.js(시작점) 만들기



이렇게 해서 기본적인 Node.js 애플리케이션을 완성했습니다.
이제 이 Node.js 앱을 도커 환경에서 실행하기 위해서
도커와 관련된 부분을 만들로 가보겠습니다.

Dockerfile 작성하기

Nodejs 앱을 도커 환경에서 실행하려면 먼저 이미지를 생성하고
그 이미지를 이용해서 컨테이너를 실행한 후
그 컨테이너 안에서 Nodejs 앱을 실행해야 하는데요.
그래서 그 이미지를 먼저 생성하려면
Dockerfile을 먼저 작성을 해야 합니다.
그래서 이번 시간에는 Dockefile을 작성해볼 건데요.



먼저 전에 Dockerfile을 만들었던 내용을 복습해보겠습니다.

```
# 베이스 이미지를 명시해준다.
FROM baseImage

# 추가적으로 필요한 파일들을 다운로드 받는다.
RUN command

# 컨테이너 시작시 실행 될 명령어를 명시해준다.
CMD [ "executable" ]
```

FROM RUN CMD 등은 도커 서버에게 무엇을 하라고 알려주는 것입니다.

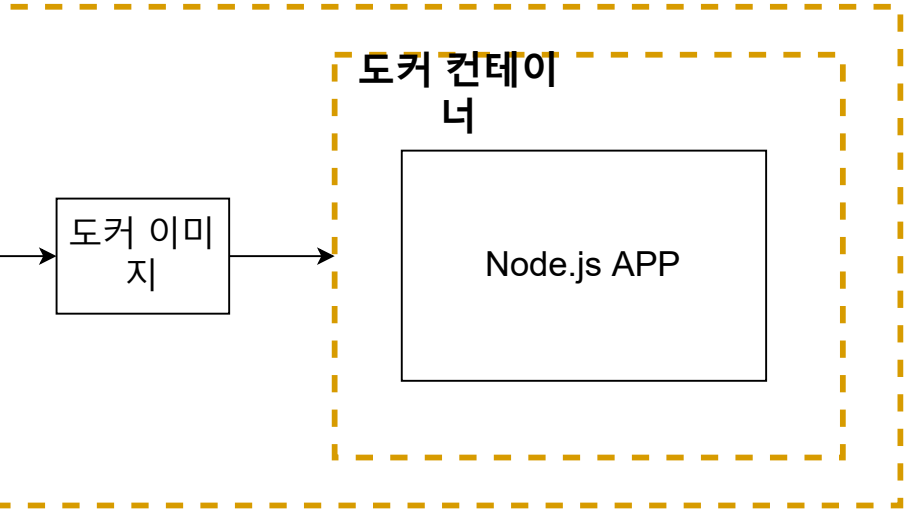
FROM
이미지 생성 시 기반이 되는 되는 이미지
<이미지 이름>:<태그> 형식으로 작성
태그를 안 붙이면 자동적으로 가장 최신
ex) ubuntu:14.04

RUN
도커 이미지가 생성되기 전에 수행할 쉘

CMD
컨테이너가 시작되었을 때 실행할 실행
입니다.
해당 명령어는 DockerFile 내 1회만 쓸

**이런 식으로 베이스 이미지를 먼저 명시해주고,
더 필요한 프로그램을 다운로드할 수 있게 명시해주고,
마지막으로 시작 시 실행할 명령어를 명시해줬었던 게
dockerfile이었습니다.**

```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./
```



이 레이어입니다.

것으로 다운 받음

명령어

파일 또는 셸 스크립트

수 있습니다

```
RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

하지만 이번에 작성할 Dockerfile을 보면
저번보다 훨씬 더 복잡해 보입니다.
그래서 여기서 하나하나가 어떤 일을 하는지
하나하나 만들면서 자세하게 이해해 보겠습니다.

먼저 저번에 했던 것처럼 가장 근본이 되는 것부터 작성해주겠습니다.

```
# 베이스 이미지를 명시해준다.
FROM baseImage

# 추가적으로 필요한 파일들을 다운로드 받는다.
RUN command

# 컨테이너 시작시 실행 될 명령어를 명시해준다.
CMD [ "executable" ]
```

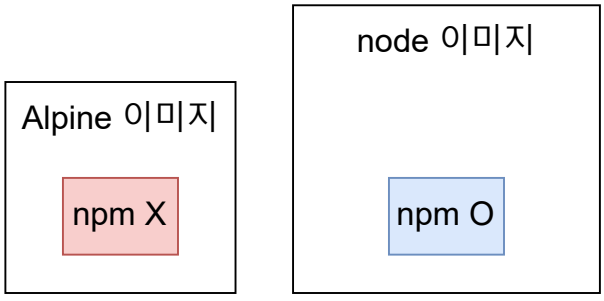
FROM	node:10
RUN	npm install
CMD	"node", "server.js"

왜 저번에는 alpine 베이스 이미지를 썼는데 이번엔 node 이미지를 쓰나요?

- 먼저 한번 베이스 이미지를 alpine으로 해서 build를 해보겠습니다. !!!

=> 그렇게 해보면 npm not found 라는 에러가 나옵니다.
그 이유는 alpine 이미지는 가장 최소한의 경량화된 파일들이 들어있기에
npm을 위한 파일이 들어있지 않아서 RUN 부분에 npm install을 할 수가 없습니다.
알파인 이미지의 사이즈는 5MB 정도밖에 안됩니다.

npm install은 무엇인가요?


Alpine 이미지
npm X
node 이미지
npm O

- npm은 Node.js로 만들어진 모듈을 웹에서 받아서 설치하고 관리해주는 프로그램입니다

- npm install은 package.json에 적혀있는 종속성들을 웹에서
  자동으로 다운 받아서 설치해주는 명령어입니다.

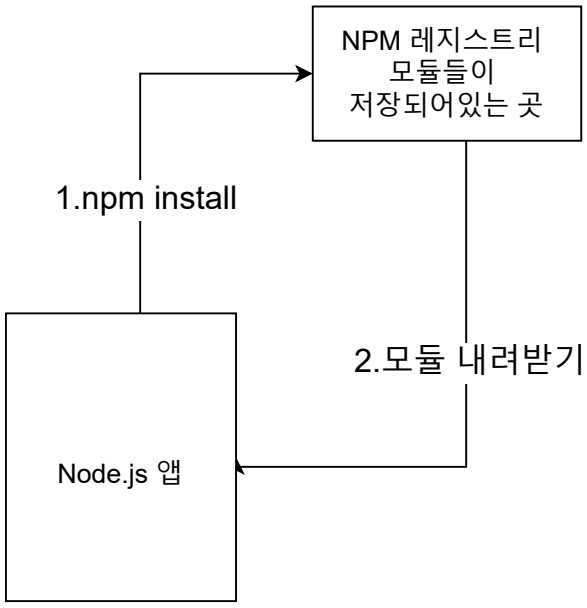
- 결론적으로는 현재 노드 JS앱을 만들 때 필요한 모듈들을 다운받아 설치하는 역할을 합니다.

"node" "server"는 무엇인가요?

-노드 웹 서버를 작동시키려면 node + 엔트리 파일 이름을 입력해야 한다.

이렇게 해서 dockerfile을 다 작성한 거 같은데...
docker build. 를 해서 dockerfile에 작성한 내용을
docker 서버에 보내줘서 이미지를 생성해보겠습니다.

package.json이 없다는 에러가 나옵니다.
분명히 현재 디렉터리 안에 package.json이 있는데 말이죠...
왜 이러한 에러가 나는지에 대해서 살펴보겠습니다.

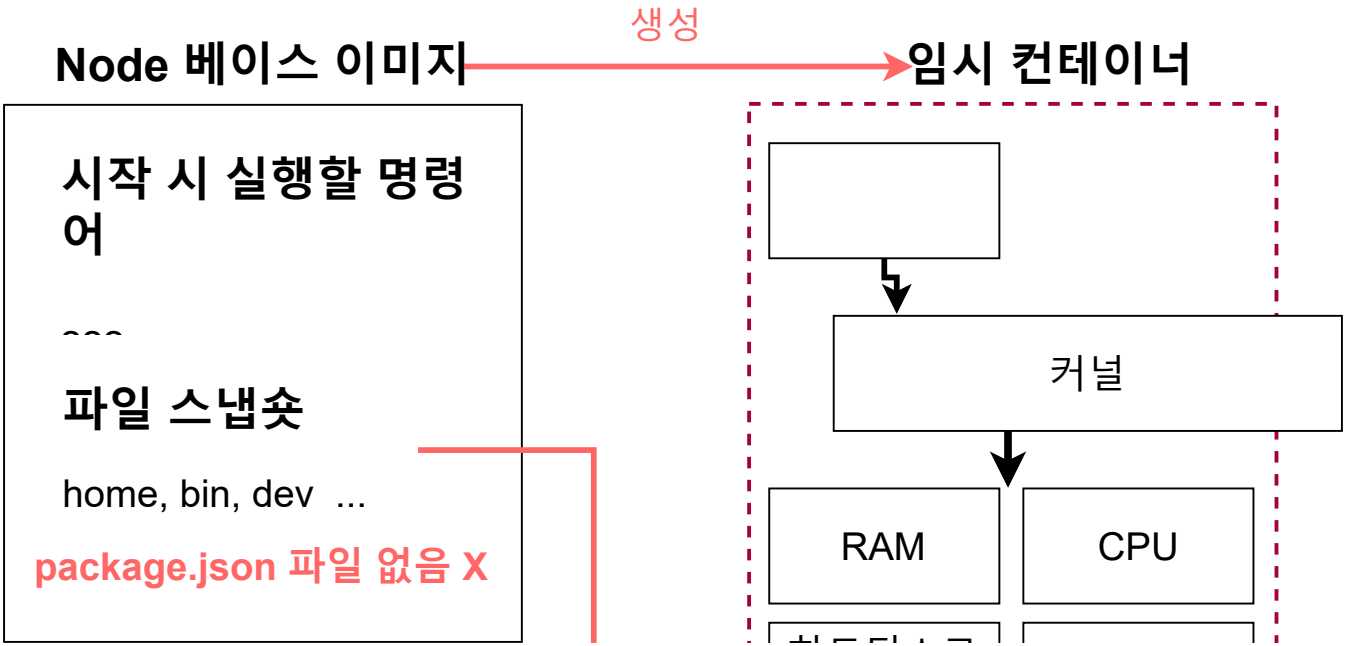

NPM 레지스트리
모듈들이
저장되어있는 곳
1.npm install
2.모듈 내려받기
Node.js 앱

Package.json이 없다고 나오는 이유(dockerfile copy)

우리가 만들 dockerfile 완성본을 보면 COPY라는 게 있습니다
이제 이 Copy가 왜 필요한지에 대해
package.json 문제를 해결해보면서 알아보겠습니다.

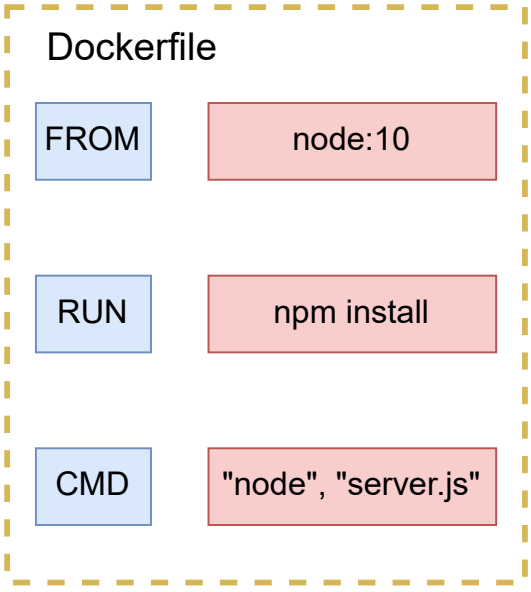
```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

이미지를 빌드 할때 왜 Package.json 파일이 없다고 나올까?

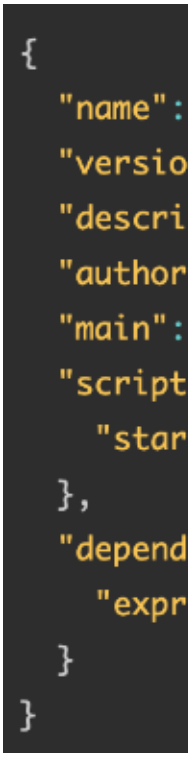


도커 파일을 Build를 할 때
Node 베이스 이미지로
임시 컨테이너를 생성합니다.
그리고 그 임시 컨테이너로 이미지를 만듭니다
하지만 그 임시 컨테이너에는 package.json이
Node 이미지 파일 스냅숏에
포함되어 있지 않습니다.
그래서 package.json은 컨테이너 밖에
있는 상황이 됩니다.

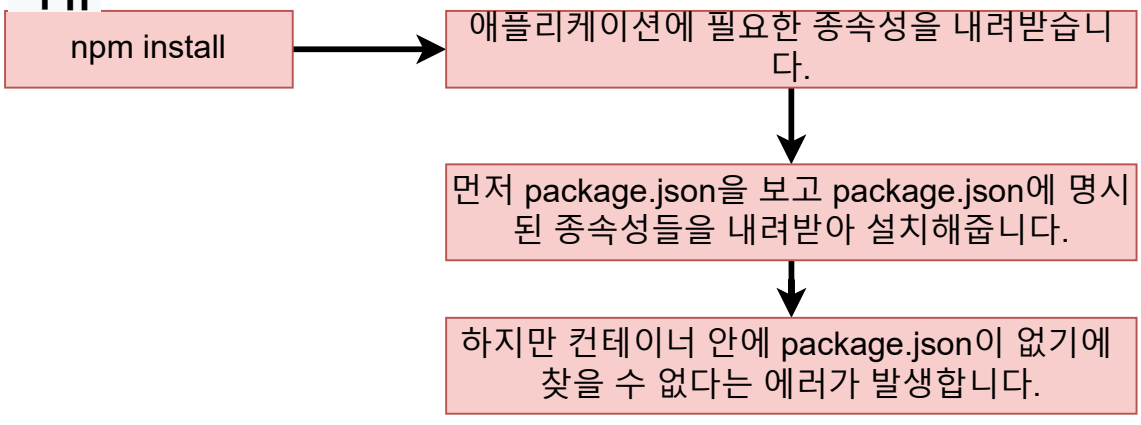
다.
이



이렇게 package.json 이 컨테이너 안에 있지 않은 상황에서 RUN npm install을 하면 컨테이너에서 package.json을 찾을 수 없다 에러가 발생합니다.



에러가 발생하는 이유



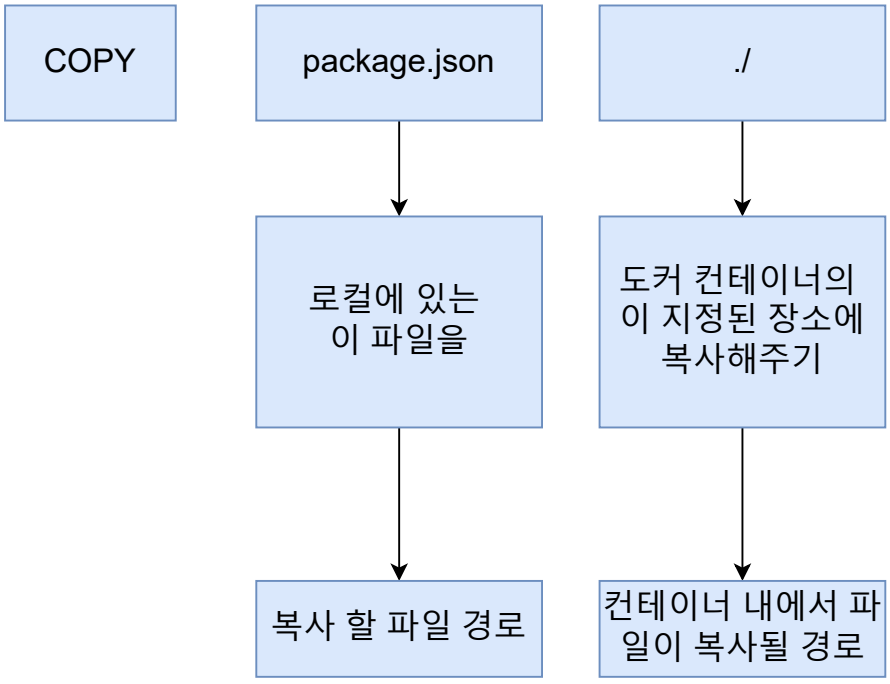
지

는

```
 "docker_web_app",
n": "1.0.0",
ption": "Node.js on Docker",
": "First Last <first.last@example.com>",
 "server.js",
s": {
t": "node server.js"

encies": {
ess": "^4.16.1"
```

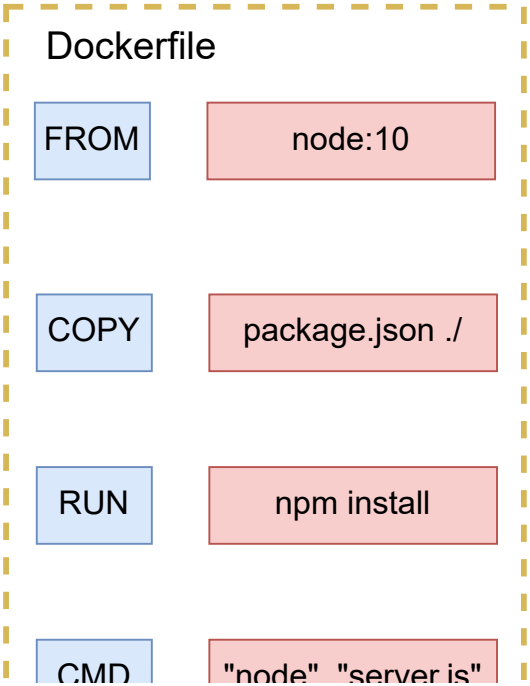
이러한 이유 때문에 COPY를 이용해서
Package.json을 컨테이너 안으로 넣어주어야 한다!!!

```
# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./
```



COPY를 포함한 Dockerfile

이렇게 해서 package.json을 컨테이너 안으로 넣어 줍니다.




```
FROM node:10

COPY package.json ./

RUN npm install

CMD [ "node", "server.js" ]
```

CMD

node , server.js

docker build -t <docker account>/<앱 이름> .

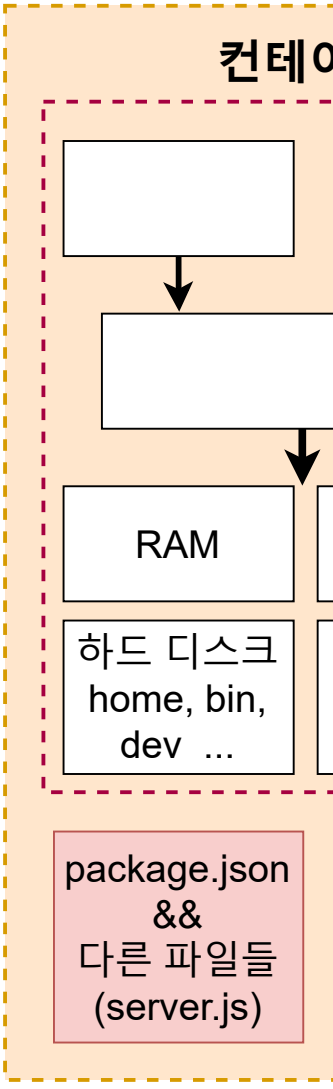
```
jaewon@Jaewonui-MacBookPro node-app-docker % docker build -t  johnahn/node-app .
Sending build context to Docker daemon  4.096kB
Step 1/4 : FROM node:10
 ---> 4d698635068f
Step 2/4 : COPY package.json ./
 ---> 46ce4908e1d8
Step 3/4 : RUN npm install
 ---> Running in e7c62f6b2321
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN docker_web_app@1.0.0 No repository field.
npm WARN docker_web_app@1.0.0 No license field.

added 50 packages from 37 contributors and audited 50 packages in 1.671s
found 0 vulnerabilities

Removing intermediate container e7c62f6b2321
 ---> 00756abb244d
Step 4/4 : CMD [ "node", "server.js" ]
 ---> Running in 1a0c35325a47
Removing intermediate container 1a0c35325a47
 ---> f10e3b4500f1
Successfully built f10e3b4500f1
Successfully tagged johnahn/node-app:latest
```

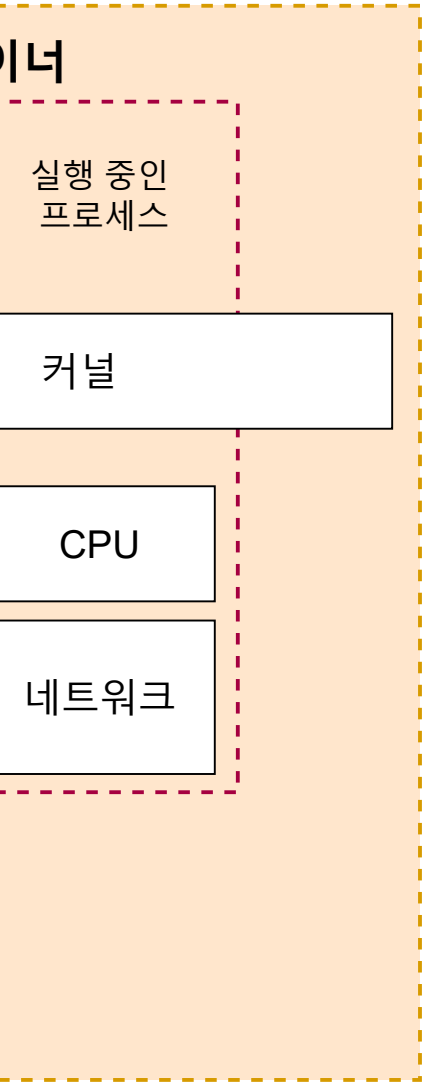
dockerfile로 이미지 만들기 성공!
그러면 실제로 컨테이너에서 실행을 해보자!

docker run johnahn/node-app (이미지 이름)

```
Error: Cannot find module '/server.js'
    at Function.Module._resolveFilename (internal/modules/
```

또 에러!!! package.json만 컨테이너에
복사해야 하는 것이 아닌 sever.js도 복사해줘야 함!



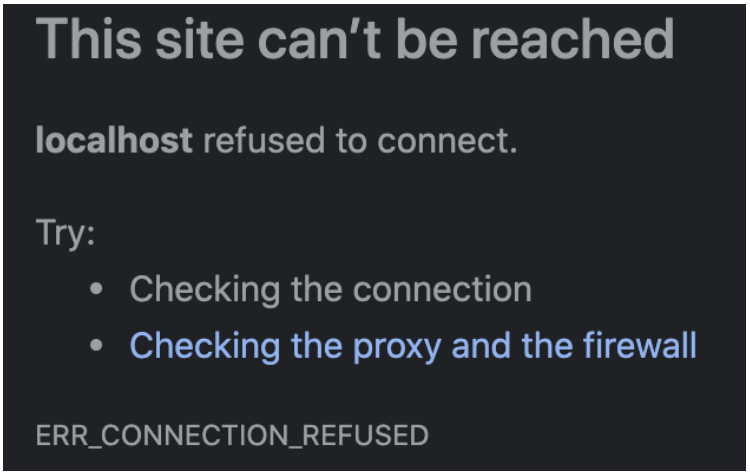

이너
실행 중인
프로세스
커널
CPU
네트워크

수정후 다시
docker build <docker account>/<앱 이름>.

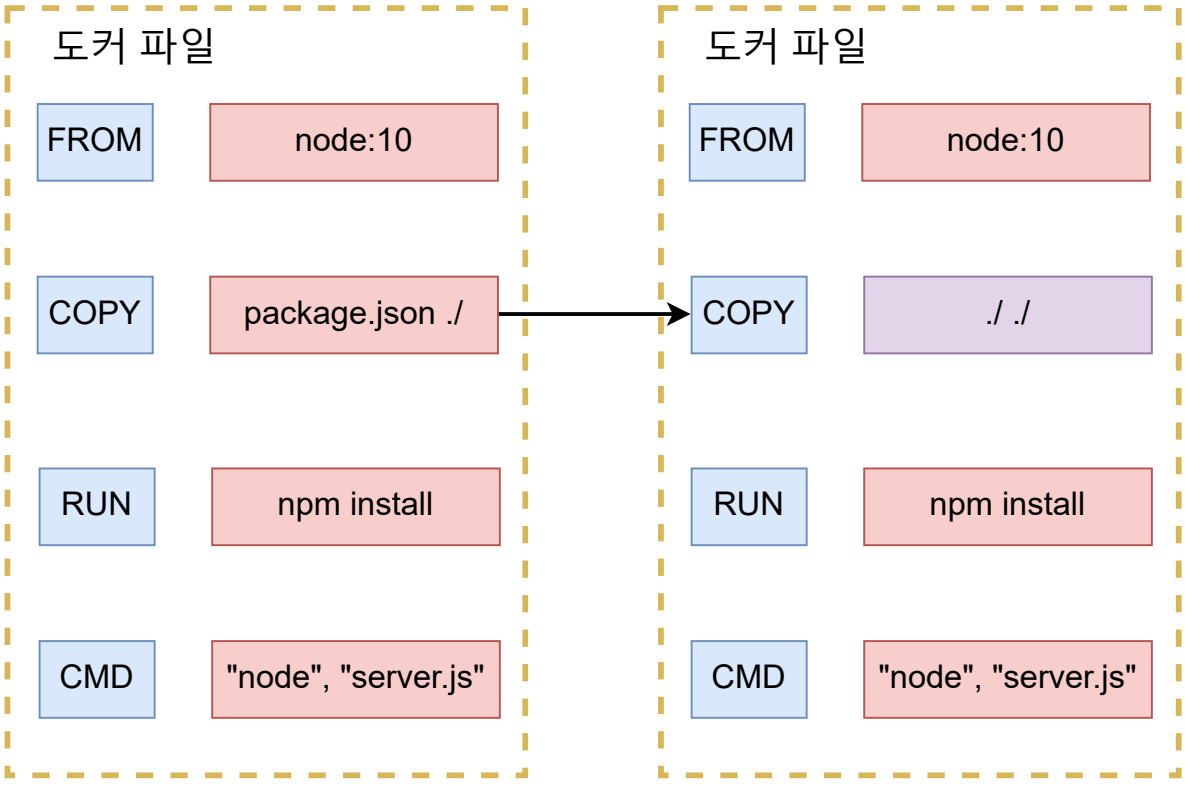
새롭게 만든 이미지로
다시 컨테이너 만들어서 앱을 실행
docker run johnahn/node-app (이미지 이름)

```
jaewon@Jaewonui-MacBookPro node-app-docker % docker run johnahn/node-app
Running on port 8080
```

port를 8080으로 지정해줬었으니
localhost:8080에서 앱이 실행 중입니다.
브라우저에 가서 애플리케이션이
잘 작동되고 있는지 한번 확인해보겠습니다.

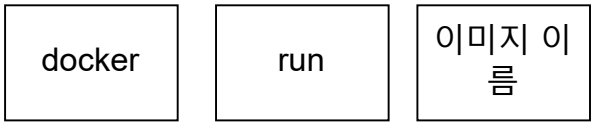


??? 무슨 일일까요? 다음 강에서 확인해보겠습니다!


도커 파일
FROM
node:10
COPY
package.json ./
RUN
npm install
CMD
"node", "server.js"
도커 파일
FROM
node:10
COPY
./ ./
RUN
npm install
CMD
"node", "server.js"

생성한 이미지로 애플리케이션 실행 시 접근이 안 되는 이유 (포트 맵핑)

현재까지 컨테이너를 실행하기 위해 사용한 명령어

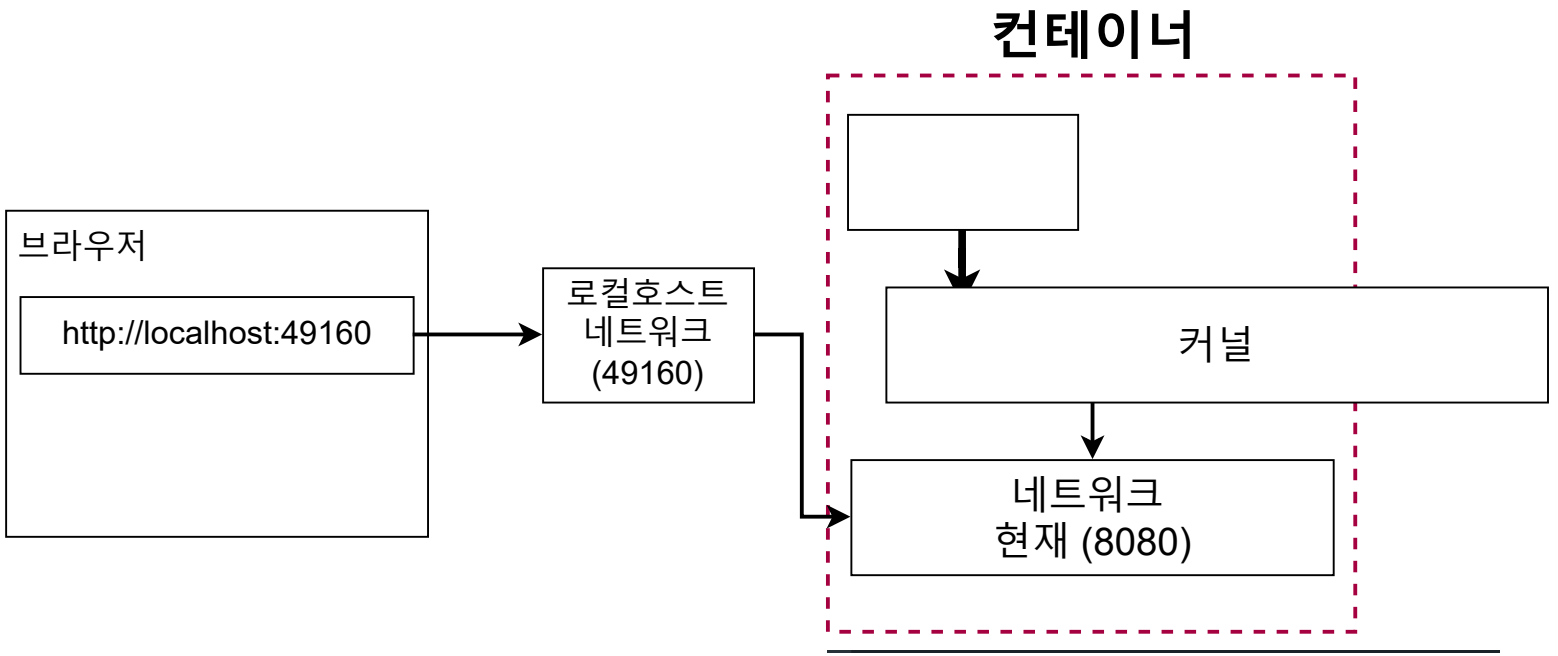


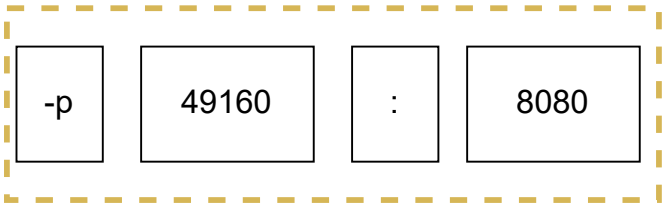
앞으로 컨테이너를 실행하기 위해 사용 할 명령어

새롭게 추가된 부분

새롭게 추가된 부분은 무엇을 위한 부분인가?

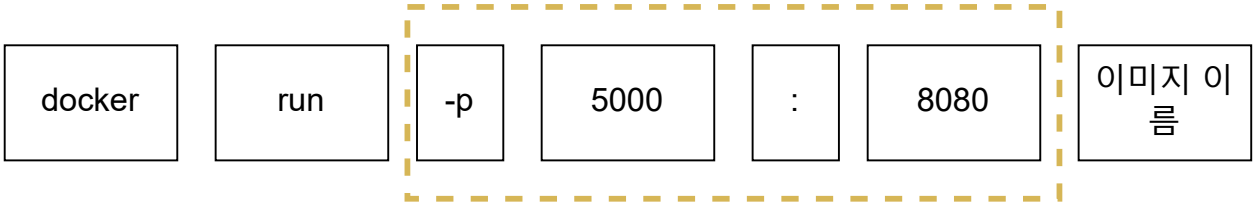
우리가 이미지를 만들 때 로컬에 있던 파일(package.json)등을 컨테이너에 복사해줘야 했었다.
그것과 비슷하게 네트워크도 로컬 네트워크에 있던 것을 컨테이너 내부에 있는 네트워크에 연결을 시켜주어야 한다.



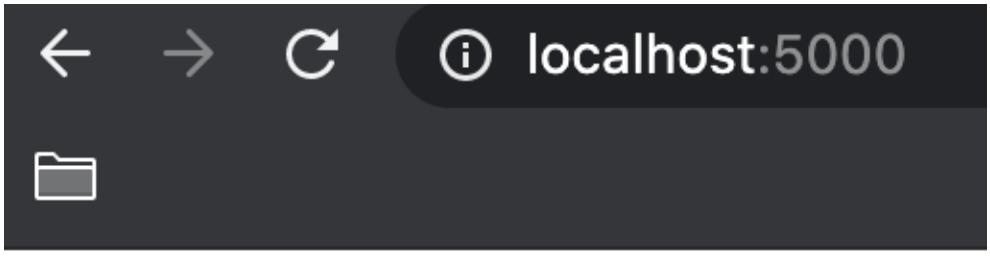



```
// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(8080);
```

이 명령어를 이용해서 다시 실행



5000번 포트를 이용해서 애플리케이션에 접속하기



Hello World

WORKING DIRECTORY 명시해주기

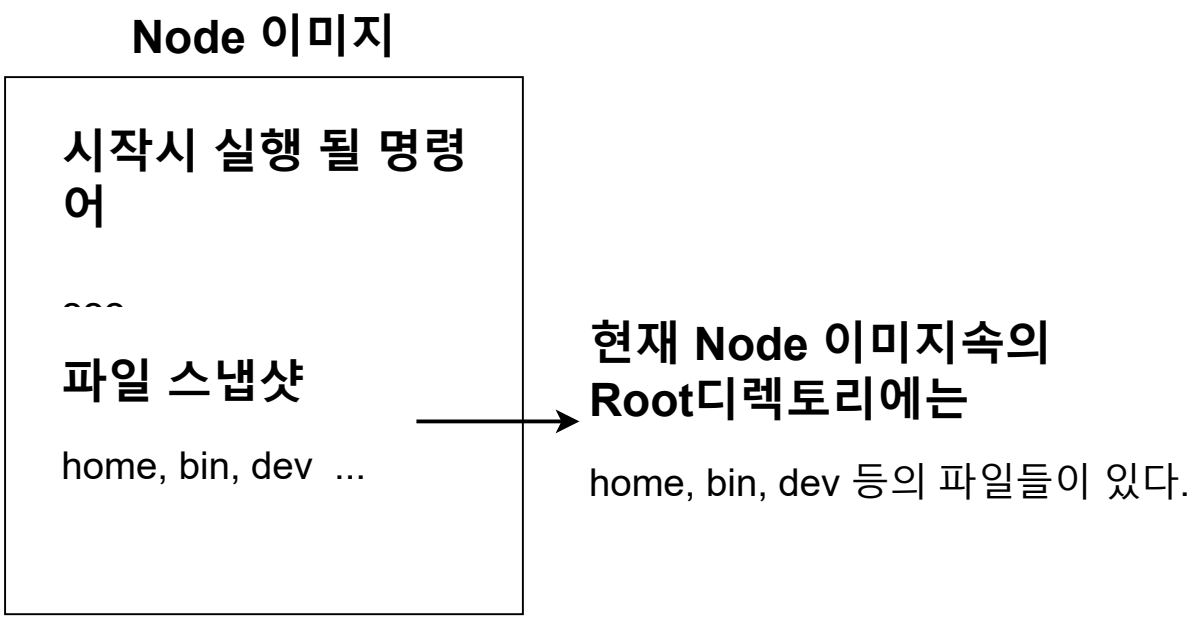
도커 파일에 이 WORKDIR이라는 부분을 추가해주어야 합니다.
하지만 이 부분은 무엇을 위해서 추가해주어야 할까요?

```
# Create app directory
WORKDIR /usr/src/app
```

우선 정의를 보면...

이미지 안에서 애플리케이션 소스 코드를 가지고 있을
디렉터리를 생성하는 것입니다.
그리고 이 디렉터리가 애플리케이션에 working directory가 됩니다.

그런데 왜 따로 working 디렉터리가 있어야 하나요?



현재 만들어 놓은 이미지의 Root 디렉토리를 봐보자.

docker run -it <이미지 이름> sh

터미널 안속에서...

ls 명령어 입력

```
jaewon@Jaewonui-MacBookPro node-app-docker % docker run -it johnahn/node-app sh
# ls
Dockerfile  dev   lib    mnt           package-lock.json  root  server.js  tmp
bin         etc   lib64  node_modules  package.json       run   srv        usr
boot        home  media  opt           proc               sbin  sys        var
```

여기 파일 중에서 COPY를 해서 컨테이너 안으로 들어온것들

Dockerfile | package.json | package-lock.json | node_module

이렇게 workdir를 지정하지 않고 그냥 COPY할때 생기는 문제점

1. 혹시 이 중에서 원래 이미지에 있던 파일과 이름이 같다면?
ex) 베이스 이미지에 이미 home이라는 폴더가 있고
COPY를 함으로써 새로 추가되는 폴더 중에 home이라는 폴더가 있다면 중복이 되므로 원래 있던 폴더가 덮어 써져 버린다.

2. 그리고 모든 파일이 한 디렉토리에 들어가 버리니 너무 정리 정돈이 안되어있다.

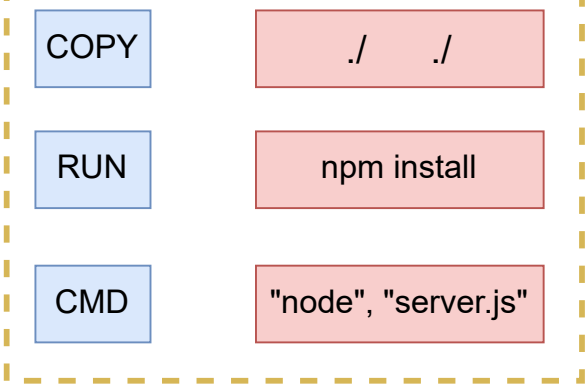
그래서 모든 어플리케이션을 위한 소스들은 WORK 디렉토리를 따로 만들어서 보관한다.

만드는 방법

Dockerfile

FROM	node:10
WORKDIR	/usr/src/app

```
FROM node:10

WORKDIR /usr/src/app
```




```
COPY ./ ./

RUN npm install

CMD [ "node", "server.js" ]
```

새로운 dockerfile로 이미지 다시 생성 후
컨테이너에서 실행
docker build <dockerhub 아이디/앱이름> .

docker run -it <이미지 이름> sh

워크 디렉토리에 잘 파일이 들어가 있는지 확인.

명령어 ls 입
력
WORKDIR 설정 후 터미널에 들어오면
기본적으로 work 디렉터리에서 시작하게
됨.

```
jaewon@Jaewonui-MacBookPro node-app-docker % docker run -it  johnahn/node-app sh
# ls
Dockerfile  node_modules  package-lock.json  package.json  server.js
```

Root 디렉토리도 확
인
명령어 cd /

```
Dockerfile  node_modules  package-lock.json  package.json  server.js
# cd /
# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

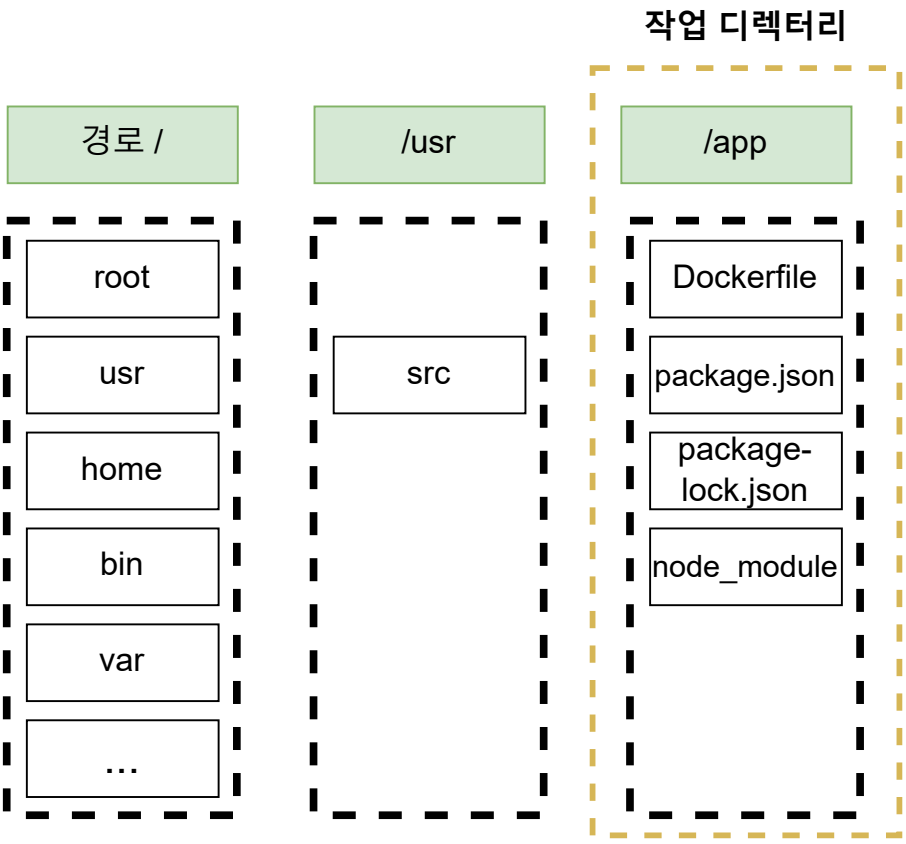
현재 상황은 이렇게 되어 있습니
다.

애플리케이션과 관련된
은
워크 디렉터리에 들어갑
그래서 WORKDIR
/usr/src/app
이렇게 지정을 해줍니다

소스들

니다.

.

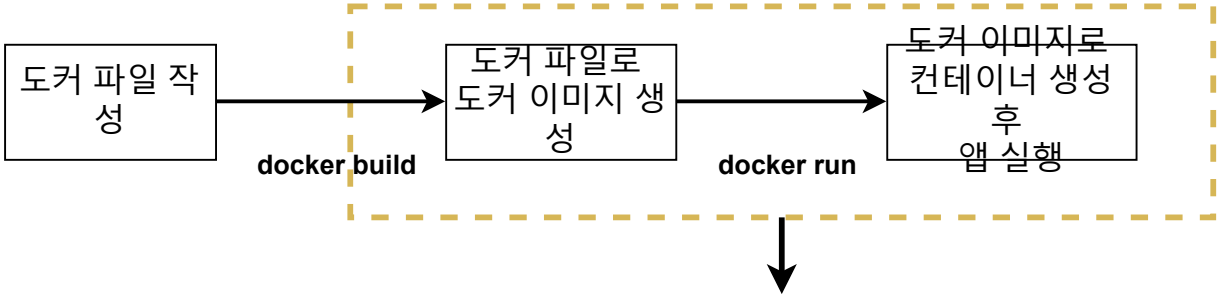

작업 디렉터리
경로 /
/usr
/app
root
usr
home
bin
var
...
src
Dockerfile
package.json
package-lock.json
node_module

어플리케이션 소스 변경으로 다시 빌드하는 것에 대한 문제점

**어플리케이션을 만들다 보면 소스 코드를 계속 변경시켜줘야 하며
그에 따라서 변경된 부분을 확인하면서 개발을 해나가야 합니다.
그래서 도커를 이용해서 어떻게 실시간으로
소스가 반영되게 하는지 알아보겠습니다.**

현재까지 만든 어플을 소스 변경 시 변경된 소스를 반영시켜주기 위해서는...

현재까지 도커 컨테이너로 어플을 실행한 순서를 보면...



소스코드를 계속 변경시켜 변경된 부분을
애플리케이션에 반영시키려면
표시된 이 구간 전체를 (이미지 생성부터)
다시 해줘야 합니다.

실제로 이러한 방식으로 해보기

먼저 server.js에 코드 변경

```
// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World');
});
```

→

```
// App
const app = express();
app.get('/', (req, res) => {
  res.send('안녕하세요');
});
```

**소스를 바꿨는데도 화면은 변하지 않습니다.
그래서 화면에 반영이 될 수 있게 하려면...**

바뀐 소스 파일로 이미지 재생성

docker build <도커 아이디>/앱 이름 .

새로 생성된 이미지로 앱 실행

docker run -d -p 5000:8080 <이미지 이름>

실제로 이러한 방식으로 할 때 비효율 적인 점들

- COPY ./ ./ 이 부분으로 인해서
소스를 변화시킨 부분은 server.js 뿐인데
모든 node_module에 있는 종속성들까지
다시 다운받아주어야 합니다.

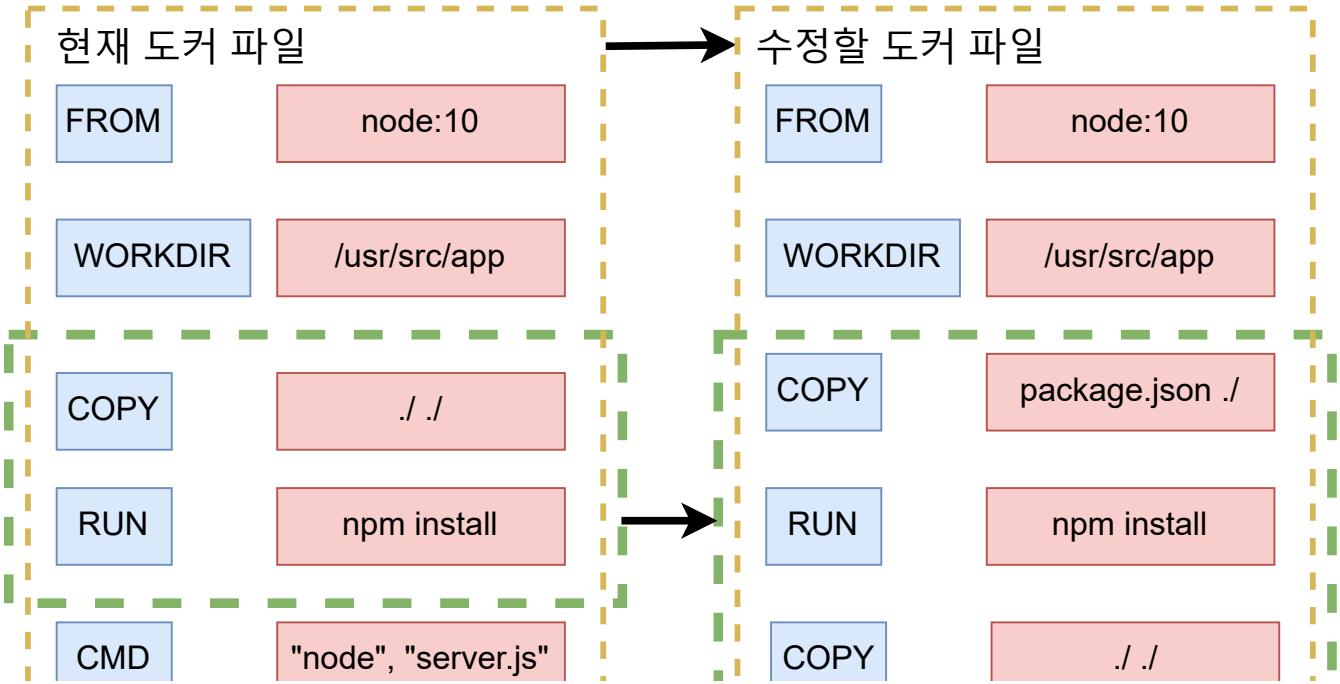
- 그리고 소스 하나 변경시켰을 뿐인데 이미지를 다시 생성하고
다시 컨테이너를 실행시켜주어야 됩니다.

그러면 이러한 비효율적인 부분을 어떻게 하면 해결 할까요?

어플리케이션 소스 변경으로 재 빌드시 효율적으로 하는 법

우선 이미 완성된 Dockerfile을 살펴보겠습니다.

```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```



위에 도표를 보면 완성본 Dockerfile에는
RUN 위에 COPY package.json ./ 이 하나가 추가되고
원래의 COPY가 RUN 아래로 내려갔습니다.

그 이유는 무엇일까요?

npm install 할 때 불 필요한 다운로드를 피하기 위해서입니다.
원래 모듈을 다시 받는 것은 모듈에 변화가 생겨야만 다시 받아야 하는데
소스 코드에 조금의 변화만 생겨도 모듈 전체를 다시 받는 문제점이 있습니다.

새롭게 server.js에 소스를 바꿨을때

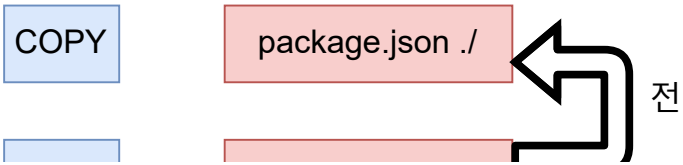
```
Step 1/5 : FROM node:10
 ---> 4d698635068f
Step 2/5 : WORKDIR /usr/src/app
 ---> Using cache
 ---> dbe9a0e09744
Step 3/5 : COPY ./ ./
 ---> 5e0c30136200
Step 4/5 : RUN npm install
 ---> Running in 0f1311c3b5cf

Removing intermediate container 0f1311c3b5cf
 ---> d8c6b7881a32
Step 5/5 : CMD [ "node", "server.js" ]
 ---> Running in da33e83dc3c9
Removing intermediate container da33e83dc3c9
 ---> c73481618620
Successfully built c73481618620
```

그 후 바로 한번 더 build 했을때

```
Step 1/5 : FROM node:10
 ---> 4d698635068f
Step 2/5 : WORKDIR /usr/src/app
 ---> Using cache
 ---> dbe9a0e09744
Step 3/5 : COPY ./ ./
 ---> Using cache
 ---> 5e0c30136200
Step 4/5 : RUN npm install
 ---> Using cache
 ---> d8c6b7881a32
Step 5/5 : CMD [ "node", "server.js" ]
 ---> Using cache
 ---> c73481618620
Successfully built c73481618620
```

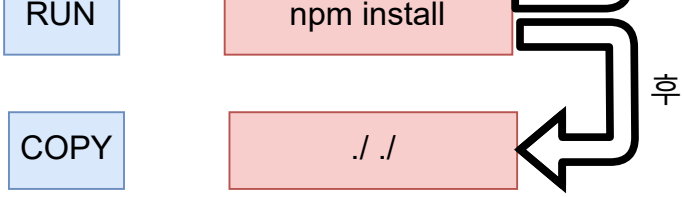
결국은 RUN npm install 전 단계에서
COPY 할 때 조금이라도 바뀐것이 있다면
npm install이 다시 실행된다.

그러기에 RUN npm install 전 단계에서
COPY 할 때는 오직 module에 관한 것만 해준다.
그리고 RUN npm install 이후에
다시 모든 파일들을 COPY 해준다



Server.js의 소스가 변했는데 변한 부분만 COPY 하는 게 아닌
npm install를 실행하여 변하지도 않은 모듈들을
다시 다운로드하는 게 문제였다.

그럼.... 이미지를 build 할 때
어떤 때는 npm install을 실행하고
어떤 때는 cache를 사용한다.
그 기준은 무엇일까?



이렇게 해줘서 모듈은 모듈에 변화가 생길 때만 다시 다운로드하여주며, 소스 코드에 변화가 생길 때 모듈을 다시 받는 현상을 없애 줬습니다.

Docker Volume에 대하여

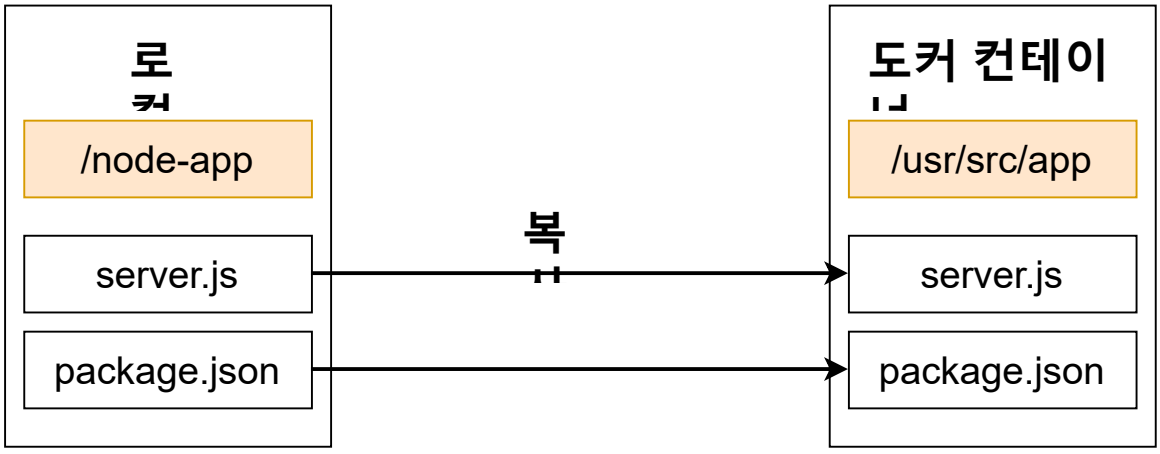
이제는 npm install 전에 package.json만 따로 변경을 해줘서 쓸 때 없이 모듈을 다시 받지는 않아도 되게 됐

하지만 아직도 소스를 변경할 때마다 변경된 소스 부분은 COPY 한 후 이미지를 다시 빌드를 해주고 컨테이너를 다시 실행해줘야지 변경된 소스가 화면에 반영이 됩니다.
이러한 작업은 너무나 시간 소요가 크며 이미지도 너무나 많이 빌드하게 됩니다.

그럼 Docker Volume은 무엇인가요 ?

현재까지 컨테이너에 소스 코드를 COPY로 넣어줬습니다.

지금까지 이용한 방식(COPY)



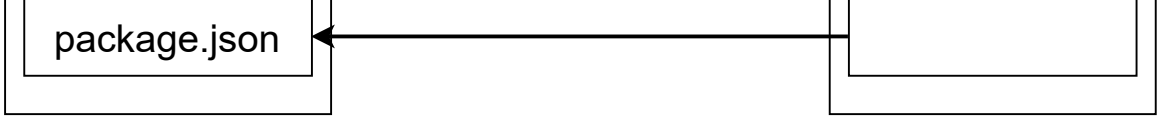
도커 볼륨을 이용한 방

호스트 디렉터리에 있는 파일을

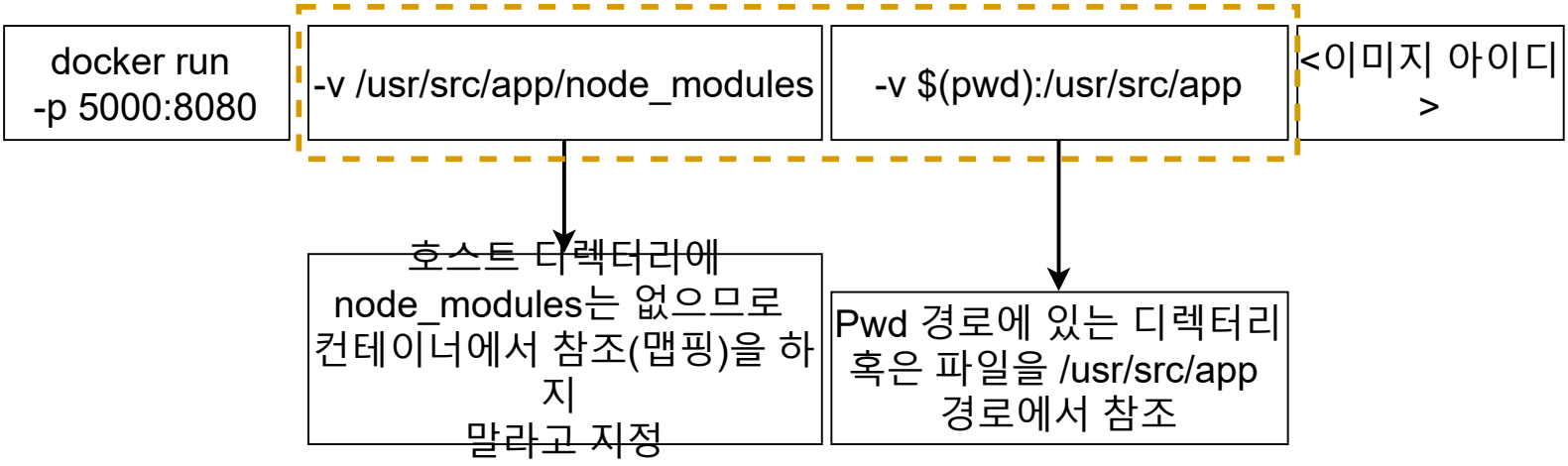
도커 컨테이너에 복사하는 것이고

볼륨은 도커 컨테이너에서

호스트 디렉터리에 있는 파일



Volume 사용해서 어플리케이션 실행하는 법



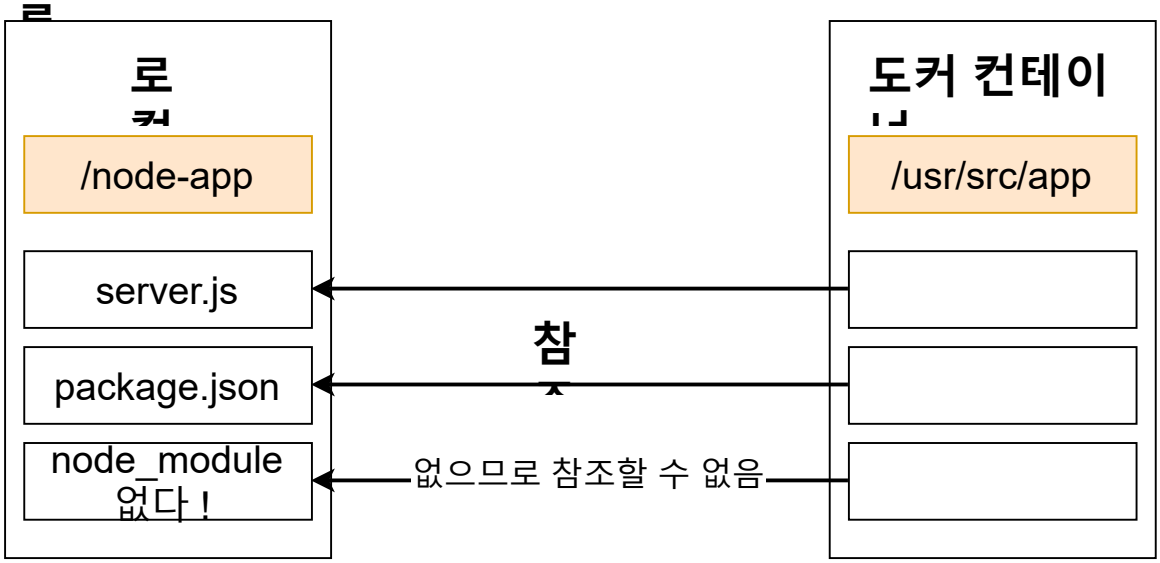
PWD (print working directory)

현재 작업 중인 디렉터리의 이름을 출력하는 데 쓰인다

```
jaewon@Jaewonui-MacBookPro node-app-docker % pwd
/Users/jaewon/Desktop/node-app-docker
```

```
CONTAINER ID   IMAGE              COMMAND                  CREATED         STATUS         PORTS                                       NAMES
c26a1d5babe7   johnahn/node-app   "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:5000->8080/tcp, :::5000->8080/tcp   cranky_hofstadter
```

도커 볼륨




doc

이렇게 Volume까지 설정했다면 이제는

소스를 변경해서 바로 변경된 것이 반영이 되는지를 확인

이렇게 volume을 이용해서 키면 빌드할 때 소스를 바꾸더라도

cker run -p 5000:8080	-v /usr/src/app/node_modules	-v $(pwd):/usr/src/app	<이미지 아이디>

vs

이렇게 COPY로만 만든 이미지를 사용해서
어플리케이션을 켜면 소스 코드를 바꿔서 해도 바꾼 소스가 적용되어있지 않는다.

```
docker run -p 5000:8080 johnahn/node-app
```