Dockerfile

Dockerfile

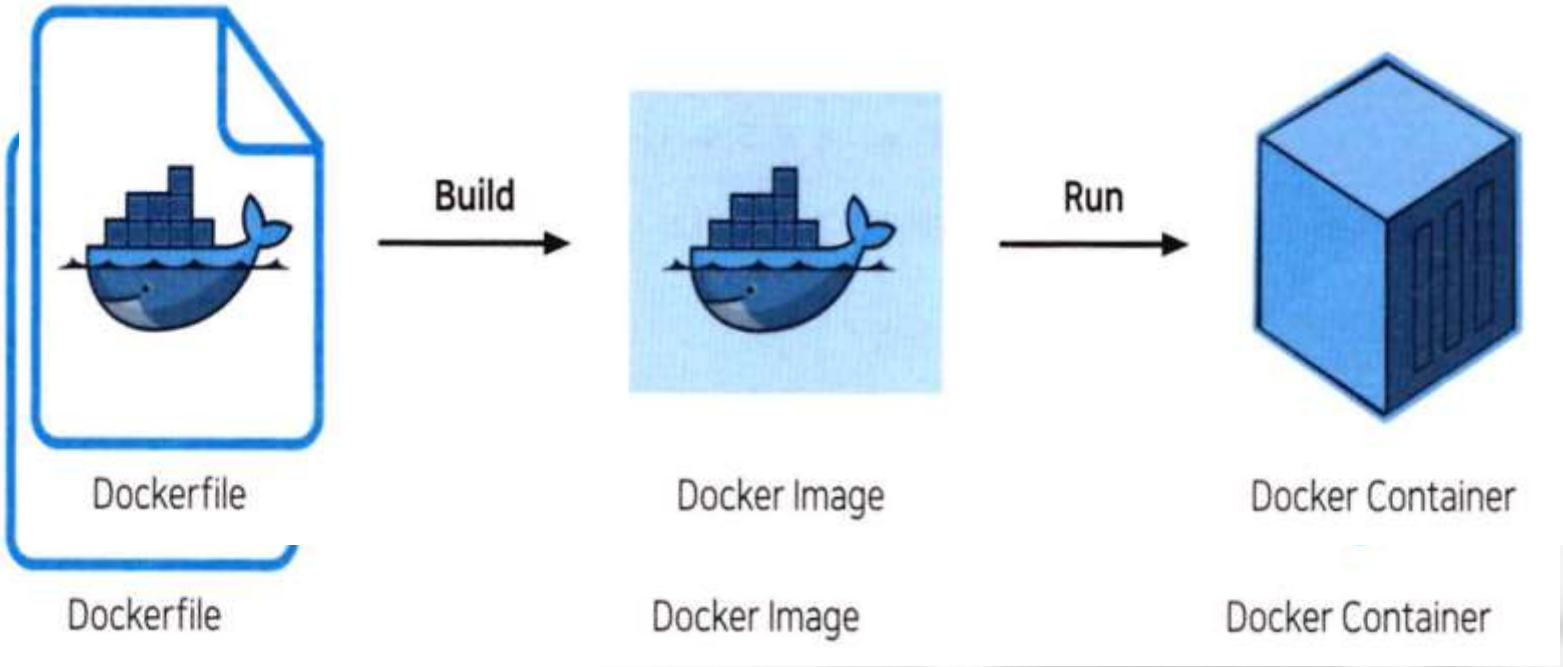
- Dockerfile
  - ✓ 코드형 인프라(Infrastructure as Code, IaC)
    - 수동으로 배포했을 때 문제점
      - 커맨드 기반의 인프라 구성 시 사용자 실수 등의 인적 오류 가능성이 높은데 APM(Apache + PHP + MySQL)을 구축하는 경우 설치 순서와 상호 연관성 등을 고려하여 각종 라이브러리와 함께 복잡한 명령어를 고민해야 함
      - 각종 환경 설정 정보와 설치 프로그램을 요구사항에 맞게 살펴봐야 하는데 만일 설치 이후에 잘못된 설정이 있다면 수정해야 하고 때로는 재설치도 불가피

Dockerfile

- Dockerfile
  - ✓ 코드형 인프라(Infrastructure as Code, IaC)
    - 개요
      - 수동 프로세스가 아닌 코드를 통해 인프라를 관리하고 프로비저닝 하는 것
      - IaC를 사용하면 인프라 사양을 담은 구성 파일이 생성되므로 구성을 편집하고 배포하기가 쉬워지고 동일한 환경을 프로비저닝 하도록 보장
      - IaC는 구성 사양을 코드화 하고 문서화함으로써 구성 관리를 지원하며 구성 변경 사항을 문서화하지 않고 임시로 변경하는 일을 막을 수 있음
      - 버전 제어는 IaC의 중요한 부분으로 다른 소프트웨어 소스 코드 파일과 마찬가지로 구성 파일도 버전 제어가 필요
      - 코드로 인프라를 배포한다는 것은 인프라를 모듈식 구성 요소로 분할하고 자동화를 통해 다양한 방식으로 결합할 수 있다는 의미
      - IaC로 인프라 프로비저닝을 자동화하면 애플리케이션을 개발하거나 배포할 때마다 개발자가 직접 서버, 운영 체제, 스토리지, 기타 인프라 구성 요소를 수동으로 프로비저닝하고 관리할 필요가 없어지는데 인프라를 코드화하여 템플릿을 만들고 프로비저닝 할 때 이 템플릿을 사용하면 됨
      - 코드형 인프라 개발은 탄력성, 확장성, 반복성을 부여하여 눈송이 서버가 아닌 동일한 환경을 보유한 서버를 수십에서 수백 대까지 운영 관리하게 해주고 민첩성, 비용 효율적인 구성, 위험 감소와 같은 장점을 가짐
      - IaC 지원 도구로 도커, 앤서블, 쿠버네티스 등이 있음

Dockerfile

- Dockerfile
  - ✓ 개요
    - Docker Image를 생성하기(Application을 Packaging) 위한 스크립트(설정 파일)



Dockerfile

- Dockerfile
  - ✓ Dockerfile을 사용했을 때의 장점
    - 이미지가 어떻게 만들어졌는지를 기록
    - 배포에 용이
    - 컨테이너(이미지)가 특정 행동을 수행하도록 할 수 있음

Dockerfile

- Dockerfile 작성 명령어
  - ✓ FROM
    - 필수 항목으로 image의 바탕이 될 베이스 image를 지정
    - hub.docker.com에서 제공하는 공식 이미지를 권장
    - 태그를 선택할 때 작은 크기의 이미지(slim)와 리눅스 배포판 인 알파인(alpine) 이미지를 권장하지만 모든 애플리케이션이 동일하지는 않음
    - 태그를 넣지 않으면 latest로 지정
    - 사용 예
      - FROM ubuntu:20.04
      - FROM python:3.9-slim-buste

Dockerfile

- Dockerfile 작성 명령어
  - ✓ MAINTAINER
    - 이미지를 빌드한 작성자 이름 과 이메일을 작성 – MAINTAINER adam.park itstudy@kakao.com
  - ✓ LABEL
    - 이미지 작성 목적으로 버전, 타이틀, 설명, 라이선스 정보를 작성
    - 여러 개 작성 가능
    - 사용 예
      - LABEL version = ‘1.0’
      - LABEL description = 'web service application using Nginx'

Dockerfile

- Dockerfile 작성 명령어
  - ✓ RUN
    - 설정된 기본 이미지에 패키지 업데이트, 각종 패키지 설치, 명령 실행 등을 작성
    - 여러 개 작성 가능
    - 권장 사항
      - 다단계 빌드 사용 권장 - 각 이미지 별로 개별 Dockerfile로 빌드
      - RUN 명령어의 개별 명령 수를 최소화하기 위해 여러 설치 명령을 연결하면 이미지의 레이어 수 감소
      - autoremove, autoclean, rm -rf/var/lib/apt/lists/*을 사용하면 저장되어 있는 apt 캐시가 삭제되므로 이미지 크기가 감소

Dockerfile

- ❖ Dockerfile 작성 명령어
  - ✓ RUN
    - ● 사용 예
      - ◆ exec 방식

```
RUN [“/bin/bash”, ”-c”, ‘’apt update”]
RUN ["/bln/bash", "-c”, "apt -y install nginx git vim curl’’]
```

      - ◆ shell 방식

```
RUN apt update && apt install -y nginx ₩
git ₩
vim ₩
curl && ₩
apt clean -y && ₩
apt autoremove -y && ₩
rm -rfv /tmp/* /var/lib/apt/lists/* /var/tmp/*
```

Dockerfile

- Dockerfile 작성 명령어
  - ✓ CMD
    - 생성된 이미지를 컨테이너로 실행할 때 실행되는 명령이고 ENTRYPOINT 명령문으로 지정된 커맨드에 디폴트로 넘길 파라미터를 지정할 때 사용
    - 여러 개의 CMD를 작성해도 마지막 하나만 처리
    - 일반적으로 이미지의 컨테이너 실행 시 애플리케이션 데몬이 실행되도록 하는 경우 유용
    - 사용 예

      CMD[“/usr/sbin/apachectl”, “-D“, “FOREGROUND”’]

      CMD apachectl –D FOREGROUND

      CMD ["nginx”, “-g”, “daemon off;"]

      CMD ["python", ’’app.py”]

Dockerfile

- Dockerfile 작성 명령어
  - ✓ ENTRYPOINT
    - CMD 와 마찬가지로 생성된 이미지가 컨테이너로 실행될 때 사용되지만 컨테이너가 실행될 때 명령어 및 인자 값을 전달하여 실행한다는 점이 다름
    - 여러 개의 CMD를 사용하는 경우 ENTRYPOINT 명령문과 함께 사용
    - ENTRYPOINT는 커맨드를 지정하고 CMD는 기본 명령을 지정하면 탄력적으로 이미지를 실행할 수 있음
    - python 명령을 기본으로 runapp.py 코드를 실행하는 경우

      ENTRYPOINT [“python”]

      CMD [“runapp.py”]
    - 동일 환경에 entrypoint.sh 셸 스크립트를 이미지에 넣고(ADD) 실행 권한 설정(RUN) 후 컨테이너 실행 시 entrypoint.sh를 실행(ENTRYPOINT)

      ADD ./entrypoint.sh /entrypoint.sh

      RUN chmod +x /entrypoint.sh

      ENTRYPOINT [“/bin/bash”, “/entrypoint.sh”]
    - ENTRYPOINT는 Docker 컨테이너 실행 시 항상 수행해야 하는 명령어를 지정(웹 서버나 데이터베이스 등의 데몬 실행)하는데 이용하고 CMD는 Docker 컨테이너 실행 시 다양한 명령어를 지정하는 경우 유용

Dockerfile

- Dockerfile 작성 명령어
  - ✓ COPY
    - 호스트 환경의 파일, 디렉토리를 이미지 안에 복사하는 경우 작성
    - 단순한 복사 작업만 지원
    - 빌드 작업 디렉토리 외부의 파일은 COPY 할 수 없음
      - COPY index.html /usr/share/nginx/html
      - COPY ./runapp.py /
      - COPY . /app - 작업 영역 전체를 COPY하므로 비 효율적

Dockerfile

- Dockerfile 작성 명령어
  - ✓ ADD
    - 호스트 환경의 파일, 디렉토리를 이미지 안에 복사하는 경우 뿐 만 아니라 URL 주소에서 직접 다운로드하여 이미지에 넣을 수도 있고 압축 파일(tar, tar.gz)인 경우에는 지정한 경로에 압축을 풀어서 추가
    - 빌드 작업 디렉토리 외부의 파일은 ADD 할 수 없고 디렉토리 추가 시에는 /로 끝
      - ADD index.html /usr/share/nginx/html
      - ADD http://example.com/view/customer.tar.gz /workspace/data/
      - ADD website.tar.gz /var/www/html

Dockerfile

- Dockerfile 작성 명령어
  - ✓ ENV
    - 이미지 안에 각종 환경 변수를 지정하는 경우 작성
    - 애플리케이션 사용을 쉽게 하기 위해 사전에 구성되어야 하는 환경 변수들이 있는 경우 사용
    - 자바 홈 디렉토리, 특정 실행 파일의 경로를 보장하기 위해 절대 경로 지정을 위한 PATH 설정, 프로그램 버전 등을 사전에 설정
    - 반복된 표현이 사용되는 경우에도 환경 변수 설정을 권장
    - Dockerfile에서 ENV를 설정하면 RUN, WORKDIR 등에서 환경 변수를 사용해 반복을 피할 수 있음
      - ENV JAVA_HOME /usr/lib/jvm/java-8-oracle
      - ENV PATH /usr/local/nginx/bin:$PATH
      - ENV Python 3.9
      - ENV NODE_VERSION V15.1.0
      - RUN curl -SLO "http://nodejs.org/dist/$N0DE_VERSI0N/node-$NODE_VERSI0N-linux-x64.tar.gz" && tar -xzf "node-$NODE_VERSION-linux-x64.tar.gz" -C /usr/local ••"strip-cc*nponents=l ₩ && rm "node-$NODE_VERSI0N-linux-x64.tar.gz"

Dockerfile

- Dockerfile 작성 명령어
  - ✓ EXPOSE
    - 컨테이너가 호스트 네트워크를 통해 들어오는 트래픽을 리스닝(listening)하는 포트와 프로토콜을 지정하기 위해 작성
    - nginx 나 apache는 기본 포트로 HTTP 80번과 HTTPS 443번 포트를 사용
    - 이미지 내에 애플리케이션이 사용하는 포트를 사전에 확인하고 호스트와 연결되도록 구성하는 경우에 설정하고 docker run 사용 시 -p 옵션을 통해 포트포워딩을 해서 사용
      - EXPOSE 80 또는 EXPOSE 80/tcp
      - EXPOSE 443
      - EXPOSE 8080/udp

Dockerfile

- Dockerfile 작성 명령어
  - ✓ VOLUME
    - 볼륨을 이미지 빌드에 미리 설정하는 경우 작성
    - Docker 컨테이너에서 사용된 파일과 디렉토리는 컨테이너 삭제와 함께 사라지기 때문에 사용자 데이터의 보존과 지속성을 위해 볼륨 사용을 권장
    - VOLUME으로 지정된 컨테이너의 경로는 볼륨의 기본 경로 /var/lib/docker와 자동으로 연결
      - VOLUME /var/log
      - VOLUME /var/www/html
      - VOLUME /etc/nginx
      - HOST OS의 Volume 기본 경로와 container 내부의 /project 연결 - VOLUME ["/project"]

Dockerfile

- Dockerfile 작성 명령어
  - ✓ USER
    - 컨테이너의 기본 사용자는 root
    - 애플리케이션이 권한없이 서비스를 실행할 수 있다면 USER를 통해 다른 사용자로 변경하여 사용

      RUN ["useradd", "adam"]

      USER adam

      RUN ["/bin/bash", ''-c", "date"]

      또는

      RUN groupadd -r mongodb && useradd –no-log-init –r –g mongodb

      mongodb

Dockerfile

- Dockerfile 작성 명령어
  - ✓ WORKDIR
    - 컨테이너상에서 작업할 경로(디렉토리) 전환을 위해 작성
    - WORKDIR을 설정하면 RUN, CMD, ENTRYPOINT, COPY, ADD 명령문은 해당 디렉토리를 기준으로 실행
    - 지정한 경로가 없으면 자동 생성되고 컨테이너 실행 이후 컨테이너에 접속(docker exec -it my_container bash)하면 지정한 경로로 연결
      - WORKDIR /workspace
      - WORKDIR /usr/share/nginx/html
      - WORKDIR /go/src/app

Dockerfile

- Dockerfile 작성 명령어
  - ✓ ARG
    - docker build 시점에서 변수의 값을 전달하기 위해 --build-arg=인자를 정의하여 사용
    - 비밀키,계정 비밀번호 같은 민감한 정보 사용 시 이미지에 직접 작성하면 노출될 위험이 있음
      - Dockerfile에 ARG 변수를 정의 - ARG db_name
      - docker build 시 변수의 값을 저장하면 이미지 내부로 인자가 전달- docker build –build-arg db_name=adam_db .
      - 입력받은 변수의 값을 명령에 사용 - CMD db_start.sh -h 127.0.0.1 -d ${db_name}

Dockerfile

- Dockerfile 작성 명령어
  - ✓ ONBUILD
    - 처음 이미지 빌드에 포함하지만 실행되지 않고 해당 이미지가 다른 이미지의 기본 이미지로 사용되는 경우 실행될 명령을 지정할 때 작성
    - ONBUILD 명령은 부모 Dockerfile이 자식 Dockerfile에 명령을 전달하고자 할 때 사용
    - 1차 개발에서 환경을 만들어주고 2차 개발에서 ONBUILD에 지정된 소스를 실행
    - 1차 Dockerfile 빌드 시 ONBUILD 포함 - ONBUILD ADD websource.tar.gz /usr/share/nginx/html/
    - 2차 Dockerfile에 1차에서 생성된 이미지를 지정하면 ONBUILD에 지정된 ADD 명령이 실행되어 새로운 이미지로 생성
  - ✓ STOPSIGNAL
    - docker stop 명령은 컨테이너에게 SIGTERM을 보내 정지시킴
    - 다른 시그널을 넣고자 하는 경우 작성
      - STOPSIGNAL SIGKILL # 시그널 번호 또는 이름
  - ✓ SHELL
    - Dockerfile 내부에서 사용할 기본 셸을 지정하는 경우 작성
    - 기본값으로 /bin/sh 가 지정
      - SHELL [''/bin/bash", "-c"]
      - RUN echo "Docker world!"

Dockerfile

- Dockerfile 작성 명령어
  - ✓ HEALTHCHECK
    - 컨테이너의 프로세스 상태를 체크하고자 하는 경우에 작성
    - HEALTHCHECK는 하나의 명령만이 유효하고 여러 개가 지정된 경우 마지막에 선언된 HEALTHCHECK 가 적용
    - HEALTHCHECK 옵션
      - --interval=(초): 헬스 체크 간격으로 기본값은 30s
      - --timeout=(초): 타임 아웃으로 기본값을 30s
      - --retries=N: 타임 아웃 횟수로 기본값은 3
    - HEALTHCHECK 상태 코드
      - 0: success: 컨테이너 정상 작동
      - 1: unhealthy: 컨테이너가 올바르게 작동하지 않는 상태
      - 2: strating: 예약된 코드
        - docker container inspect [컨테이너명] 또는 docker ps에서 확인
    - 1분마다 CMD에 있는 명령을 실행하여 3초 이상이 소요되면 한 번의 실패로 간주하고 5번의 타임 아웃이 발생하면 컨테이너의 상태를 unhealthy로 변경

      HEALTHCHECK ---interval=1m --timeout=3s --retries=5 CMD curl -f http://localhost || exit 1

Dockerfile

- Dockerfile 명령어 작성 방법
  - ✓ 일반적으로 시작은 FROM 명령부터 작성하는데 그 다음 명령부터는 순서가 없지만 명령 순서는 빌드 캐시의 무효화와 연관되므로 변경 빈도수가 적은 명령을 먼저 배치하는 것을 권장
  - ✓ Docker 허브에는 수백만 개의 이미지가 존재하는데 무조건 OS 이미지를 선택하고 설치하는 것보다는 원하는 애플리케이션 패키지가 이미 설치된 이미지 선택을 권장하는데 이미지 관리의 유지보수 측면에서 시간 절약에 도움이 됨

Dockerfile

- Dockerfile 스크립트로 이미지 생성
  - ✓ 이미지 생성 명령어

    docker build [옵션] 생성할_이미지_이름[:태그] Dockerfile_파일_디렉토리경로 | URL | 압축 파일

    - 옵션
      - -t:(tag): 이미지명: 태그를 지정하는 경우
      - -f(file): Dockerfile이 아닌 다른 파일명을 사용하는 경우

        -f Dockerfile_nginx
      - 동시에 여러 저장소를 생성하려면 -t를 반복해서 사용

        docker build -t mypyapp:2.0 -t mypyapp:init.
    - 이미지명:태그
      - 생성할 이미지 이름 과 태그를 지정
      - 태그는 버전 관리 차원

        my-nginx-image:1.19_v1.0
    - 경로
      - 디렉토리 단위 개발을 권장
      - 현재 경로에 Dockerfile이 있다면 .을 사용

Dockerfile

- Dockerfile 스크립트로 이미지 생성
  - ✓ 이미지 생성 명령어

    docker build [옵션] 생성할_이미지_이름[:태그] Dockerfile_파일_디렉토리경로 | URL | 압축 파일

    - URL
      - Dockerfile이 포함된 깃허브 URL을 제공하는 경우

        docker build -t phpserver:2.0 github.com/brayanlee/docker-phpserver
    - 압축 파일
      - 압축 파일 내에 Dockerfile이 포함된 경우

        docker birild -f app1/Dockerfile http://server/appl.tar.gz

Dockerfile

- Dockerfile 스크립트로 이미지 생성
  - ✓ 실습: 로컬 컴퓨터의 index.html을 welcome file로 사용하는 httpd 이미지 생성
    - Dockerfile 이 저장될 디렉토리 생성: apa_folder
    - 디렉토리에 이미지에 포함될 index.html 파일을 생성

```
<html>
<meta charset="utf-8"/>
<body>
<div> 안녕하세요! </div>
</body>
</html>
```

Dockerfile

- Dockerfile 스크립트로 이미지 생성
  - ✓ 실습: 로컬 컴퓨터의 index.html을 welcome file로 사용하는 httpd 이미지 생성
    - Dockerfile을 디렉토리에 생성하고 작성

      FROM httpd

      COPY index.html /usr/local/apache2/htdocs/
    - 빌드: docker build -t apacheex /Users/adam/Documents/apa_folder
    - 이미지 확인: docker image ls

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
apacheex	latest	0f17cf1a23fa	46 hours ago	145MB

Dockerfile

- Dockerfile 스크립트로 이미지 생성
  - ✓ 실습: 로컬 컴퓨터의 index.html을 welcome file로 사용하는 httpd 이미지 생성
    - 컨테이너 생성: docker run -d --name apacheex -p 8080:80 apacheex
    - 확인: curl localhost:8080

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● 요구 사항
      - ◆ 운영체제: Ubuntu 18.04
      - ◆ 웹 서버: Apache2
      - ◆ Language: PHP
      - ◆ 테스트: PHP 웹 페이지 구동

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - 서버 직접 구축
      - sudo apt update
      - sudo apt install -y apache2
      - sudo service apache2 start
      - sudo service apache2 status
      - curl localhost:80 으로 테스트(없으면 설치)

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● 서버 직접 구축
      - ◆ sudo mv /var/www/html/index.html /var/www/html/index.html.org
      - ◆ sudo apt install vim
      - ◆ sudo vim /var/www/html/index.html
      - ◆ i를 누르고 작성

        <h1> Hello Docker Application</h1>

        - ▪ ESC를 누르고 :wq!
      - ◆ curl localhost:80 으로 테스트(없으면 설치)

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● 서버 직접 구축
      - ◆ sudo apt -y install php
      - ◆ sudo vim /var/www/html/index.php

```
<?php
    phpinfo();
?>
```

      - ◆ sudo service apache2 restart
      - ◆ 브라우저에서 localhost:80/index.php로 접속

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - 서버 직접 구축
      - 호스트 운영체제에 이와 같은 설치와 설정을 통해 환경을 구성하고 개발 팀에서 제공하는 웹 소스를 이용해 애플리케이션 서비스를 수행
      - 이러한 인프라 구축 작업은 규모가 클수록 인프라 구성 관리에 대한 부담도 늘어남
      - 클라우드 시스템과 여러 가지 가상화 기술의 등장으로 인프라 구축 방법이 크게 변경되면서 구축된 인프라를 변경하지 않고 파기한 뒤 새로 구축하는 것이 쉬워져 지금까지 큰 부담이었던 인프라 변경 이력 관리도 필요가 없어졌으며 현재 가동 중인 인프라 상태만 관리하면 되는 환경으로 변하고 있음

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - Ubuntu 환경의 컨테이너를 실행

```
docker run -it --name myweb -p 8095:80 ubuntu:14.04 bash
```

    - 컨테이너 환경에서 동일한 작업 수행

```
apt update
apt install -y apache2
service apache2 start
```

    - 확인

```
curl localhost:8095
```

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - 웹 페이지 갱신을 위해서 기본 페이지 변경

    mv /var/www/html/index.html /var/www/html/index.html.org
    - 기본 페이지 작성

    vim /var/www/html/index.html

    <h1> Hello Docker Applicaiton </h1>
    - 확인

    curl localhost:8095

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - PHP 설치

apt install –y php5

- 기본 페이지 작성

vi /var/www/html/index.php

```
<?php
  phpinfo();
?>
```

- 서비스 재시작

service apache2 restart

- 확인

curl localhost:8095/index.php

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - 현재 컨테이너를 이미지로 변경

```
docker ps
```

```
CONTAINER ID   IMAGE          COMMAND   CREATED          STATUS
PORTS                                       NAMES
041912fec331   ubuntu:14.04   "bash"    24 minutes ago   Up 2 minutes
0.0.0.0:8095->80/tcp, [::]:8095->80/tcp     myweb
```

```
docker commit myweb myphpapp:1.0
```

```
sha256:a0959f202e4b35f021678b01061ac52719a3eef2d7d08682a3d9fd4ea1dd23db
```

- 이미지를 컨테이너로 실행

```
docker run -dit -p 8096:80 --name=phpapp myphpapp:1.0
```

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - 확인 - 에러

curl localhost:8096/index.php

curl: (56) Recv failure: Connection reset by peer

- 컨테이너로 진입해서 apache 시작

docker exec -it phpapp bash

root@4b856079ef7d:/# service apache2 start

- 확인 - 성공

curl localhost:8096/index.php

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - 베이스 이미지를 만들기는 했지만 매번 아파치 서비스를 수동으로 시작해야 된다면 서버에 설치해서 사용하는 것과 다르지 않음
    - 해결 방법은 생성한 베이스 이미지가 컨테이너로 실행될 때 자동으로 서비스를 시작하게 하는 것으로 이를 위해서 Dockerfile의 CMD를 활용
    - 디렉토리 생성

mkdir phpapp

cd phpapp

- Dockerfile 작성

```
FROM myphpapp:1.0
MAINTAINER adam <itstudy@kakao.com>
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Dockerfile

- Dockerfile 파일이 필요한 이유
  - ✓ 컨테이너 실행 후 내부에서 직접 환경 구성 등을 통해 베이스 이미지를 생성하는 작업
    - 이미지 확인: docker images | grep myphpapp

    WARNING: This output is designed for human readability. For machine-readable output, please use --format.

    myphpapp:1.0 a0959f202e4b 354MB 88.6MB U

    myphpapp:2.0 d51b12d61cf3 354MB 88.6MB

    - 컨테이너 생성: docker run -dit -p 8097:80 --name=phpappv2 myphpapp:2.0
    - 확인 - 성공

    curl localhost:8097

    - 이렇게 생성한 베이스 이미지를 도커 허브에 공유하거나 깃허브에 코드를 올려 공유할 수도 있지만 이번 작업도 사용자 의존성을 가지며 명령어 실행으로 이미지를 생성(docker commit)하고 자동 서비스 시작이 되지 않는 문제점을 해결하고자 Dockerfile로 다시 빌드하는 번거로움이 있음
    - 인프라 구성 정보를 코드로 관리해 두면 애플리케이션 개발 시 버전 관리 차원의 소스 코드 관리가 가능해져 변경 이력을 관리할 수 있음
    - 소스 코드를 통해 전체 구성을 한눈에 파악할 수 있고 사용자 의존성을 배제할 수 있는 것이 인프라 구성을 코드로 관리하는 것이 Dockerfile을 사용하는 이유

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● Dockerfile 이용
      - ◆ Dockerfile 작성

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:20.04

# Dockerfile 작성자에 대한 정보를 기록
MAINTAINER "adam <itstudy@kakao.com>"

# 생성하는 이미지에 대한 설명을 작성
LABEL title "IaC, PHP application"

#필요한 패키지를 설치
RUN apt update
RUN apt install -y tzdata
RUN apt -y install php
RUN apt -y install apache2
RUN apt -y install git
RUN apt -y install curl
RUN apt -y install ssh
RUN apt -y install wget
```

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● Dockerfile 이용
      - ◆ Dockerfile 작성

```
# 아파치2의 환경 변수를 지정
# 아래 설정 값은 아파치2의 기본값
ENV APACHE2_RUN_USER www-data
ENV APACHE2_RUN_GROUP www-data
ENV APACHE2_LOG_DIR /var/log/apache2
ENV APACHE2_WEB_DIR /var/www/html
ENV APACHE2_PID_FILE /var/run/apache2/apache2.pid

#기본 웹 페이지를 생성
RUN echo 'Hello, Docker Application.' > /var/www/html/index.html

# 테스트 PHP 웹 페이지를 생성
RUN echo '<?php phpinfo(); ?>' > /var/www/html/index.php

# 80번 포트를 노출
EXPOSE 80

# RUN, CMD, ENTRYPOINT의 명령어가 실행되는 디렉토리를 설정
WORKDIR /var/www/html
```

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● Dockerfile 이용
      - ◆ Dockerfile 작성

```
# 이미지가 컨테이너로 실행될 때 아파치 서비스를 자동으로 실행
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Dockerfile

- ❖ Dockerfile 파일이 필요한 이유
  - ✓ Ubuntu에서 PHP 웹 페이지 연동
    - ● Dockerfile 이용
      - ◆ 이미지 빌드

        docker build -t myphpapp:2.0 .
      - ◆ 컨테이너 실행

        docker run -dit -p 8098:80 --name=phpapp3 myphpapp:3.0

Dockerfile

- 이미지 빌드 과정
  - ✓ 이미지 빌드가 필요한 이유
    - 도커 허브로부터 다운로드한 이미지는 불변
    - 빌드가 완료된 이미지는 그 내용을 수정할 수 없기 때문에 이미지로부터 컨테이너를 생성하여 변경 사항을 추가하고 다시 docker commit 명령을 통해 새로운 이미지를 생성하는 방법으로 변경할 수 있지만 이 작업은 변경 사항을 추가하여 새로운 이미지를 만든 것이지 기존 이미지를 수정한 것은 아님
    - 필요한 애플리케이션을 컨테이너로 서비스하려면 이미지가 필요한데 만족할 만한 이미지를 도커 허브가 모두 보유하고 있는 것은 아니기 때문에 애플리케이션 요구사항을 만족할 수 있는 인프라 환경을 직접 구성하려면 서비스에 필요한 인프라 설계 요구서와 여러 환경 변수 등을 고려한 작업 시트를 작성하여 Dockerfile을 생성해야 함

Dockerfile

- 이미지 빌드 과정
  - ✓ Dockerfile 작성 Life Cycle
    - Dockerfile은 인프라 구성을 위해 필요한 명령을 담은 일반 텍스트 문서
    - Dockerfile은 docker build 명령을 통해 작성한 명령을 순서대로 읽으며 자동으로 이미지를 빌드
    - 이 때 주의할 점은 빌드는 사용자와의 대화식 처리가 아닌 자동화된 빌드라는 것

Dockerfile

- ❖ 이미지 빌드 과정
  - ✓ Dockerfile 작성 Life Cycle
    - ● 자동화된 빌드
      - ◆ Dockerfile 작성

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:20.04


#필요한 패키지를 설치
RUN apt install python
```

      - ◆ 이미지 빌드

```
docker build -t myapp:1.0 .
```

      - ◆ 이미지 빌드 에러

```
ERROR: failed to solve: process "/bin/sh -c apt install python" did not
    complete successfully: exit code: 100
```

        - ▪ Python 패키지를 찾지 못하여 설치되지 못한다는 에러
        - ▪ Ubuntu 기반의 이미지를 생성하는 경우 반드시 apt update(또는 apt update)를 포함하여야 함

Dockerfile

- ❖ 이미지 빌드 과정
  - ✓ Dockerfile 작성 Life Cycle
    - ● 자동화된 빌드
      - ◆ Dockerfile 수정

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:20.04
#필요한 패키지를 설치
RUN apt update
RUN apt install python
```

      - ◆ 이미지 빌드

```
docker build -t myapp:1.0 .
```

      - ◆ 이미지 빌드 에러

```
ERROR: failed to solve: process "/bin/sh -c apt install python" did not
    complete successfully: exit code: 1
```

        - ▪ 패키지 설치 시 -y 옵션을 사용하지 않으면 출력 메시지처럼 강제 종료(Abort)가 되는데 Dockerfile은 사용자의 개입이 없는 자동화된 빌드이기 때문

Dockerfile

- ❖ 이미지 빌드 과정
  - ✓ Dockerfile 작성 Life Cycle
    - ● 자동화된 빌드
      - ◆ Dockerfile 수정

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:20.04

#필요한 패키지를 설치
RUN apt update
RUN apt -y install python
```

      - ◆ 이미지 빌드

```
docker build -t myapp:1.0 .
```

      - ◆ 이미지 빌드 성공

```
What's Next?
  View summary of image vulnerabilities and recommendations → docker
    scout quickview
```

Dockerfile

- 이미지 빌드 과정
  - ✓ Dockerfile 작성 Life Cycle
    - docker build를 실행하는 현재 작업 중인 디렉토리를 빌드 컨텍스트라고 부르는데 Dockerfile은 새로운 빈 디렉토리에서 생성하여 빌드하는 것을 권장
    - 이미지 빌드가 시작되면 Dockerfile 위치와 상관없이 현재 디렉토리에 있는 모든 파일과 디렉토리의 콘텐츠는 Docker 데몬에 빌드 컨텍스트로 전달되는데 이때 현재 디렉토리에 있는 파일과 디렉토리 중 빌드 컨텍스트에서 제외 대상이 있다면 .dockerignore 파일에 작성
    - 이미지 빌드도 문법적 오류나 오타가 포함되면 빌드 과정에서 에러가 출력되는데 Docker 데몬은 Dockerfile 빌드를 시작하면 Dockerfile에 포함된 모든 내용을 읽어와 유효성 검사를 수행하여 오류를 내보냄
    - -f 옵션으로 현재 디렉토리가 아닌 다른 경로 지정 가능
      - mkdir appimage && mv Dockerfile ./appimage
      - docker build -t mypyapp:1.0 -f ./appimage/Dockerfile .

Dockerfile

- 이미지 빌드 과정
  - ✓ Docker 이미지는 Dockerfile의 명령어 단위로 실행할 때마다 읽기 전용 레이어를 생성하여 최종 이미지로 생성
  - ✓ Dockerfile 작성

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:20.04

#필요한 패키지를 설치
RUN apt update
RUN apt install -y nginx
RUN apt install -y curl

RUN echo 'Docker Container Application' > /var/www/html/index.html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile

- 이미지 빌드 과정
  - ✓ 빌드: docker build -t webapp:1.0 .

```
[+] Building 72.8s (10/10) FINISHED                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 542B                                          0.0s
 => [internal] load .dockerignore                                             0.0s
 => => transferring context: 2B                                               0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest              3.0s
 => [auth] library/ubuntu:pull token for registry-1.docker.io                 0.0s
 => [1/5] FROM docker.io/library/ubuntu:latest@sha256:9b8dec3bf938bc80fb      10.2s
 => => resolve docker.io/library/ubuntu:latest@sha256:9b8dec3bf938bc80fbe      0.0s
 => => sha256:9b8dec3bf938bc80fbe758d856e96fdfab5f56c39d4 1.13kB / 1.13kB     0.0s
 => => sha256:f58b48967ecc767fc238144ffdb7eb668cefcc8438de8f8 424B / 424B     0.0s
 => => sha256:bf9e5b7213b4e0d99cddc039011cc60bfd76ed5ef63 2.32kB / 2.32kB
    0.0s
 => => sha256:c391327a0f1df5790bcb00ac010a1a3924c9e4387 27.35MB / 27.35MB
    9.2s
 => => extracting sha256:c391327a0f1df5790bcb00ac010a1a3924c9e4387ce36b29
    0.8s
```

Dockerfile

- 이미지 빌드 과정
  - ✓ 빌드: docker build -t webapp:1.0 .

```
=> [2/5] RUN apt update                                          34.8s
=> [3/5] RUN apt install -y nginx                                15.8s
=> [4/5] RUN apt install -y curl                                 8.5s
=> [5/5] RUN echo 'Docker Container Application' > /var/www/html/index.h  0.3s
=> exporting to image                                            0.1s
=> => exporting layers                                           0.1s
=> => writing image sha256:d9a9b6f69e4e0f98e6e81eb318ef19a74782b7ad284fc
   0.0s
=> => naming to docker.io/library/webapp:1.0                     0.0s
```

Dockerfile

- 이미지 빌드 과정
  - ✓ 동일한 파일로 다시 빌드: docker build -t webapp:2.0 .

```
[+] Building 1.0s (9/9) FINISHED                                docker:desktop-linux
 => [internal] load build definition from Dockerfile                              0.0s
 => => transferring dockerfile: 542B                                              0.0s
 => [internal] load .dockerignore                                                 0.0s
 => => transferring context: 2B                                                   0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest                  0.9s
 => [1/5] FROM docker.io/library/ubuntu:latest@sha256:9b8dec3bf938bc80fbe          0.0s
 => CACHED [2/5] RUN apt update                                                   0.0s
 => CACHED [3/5] RUN apt install -y nginx                                         0.0s
 => CACHED [4/5] RUN apt install -y curl                                          0.0s
 => CACHED [5/5] RUN echo 'Docker Container Application' > /var/www/html/         0.0s
 => exporting to image                                                            0.0s
 => => exporting layers                                                           0.0s
 => => writing image sha256:d9a9b6f69e4e0f98e6e81eb318ef19a74782b7ad284fc
    0.0s
 => => naming to docker.io/library/webapp:2.0                                     0.0s
```

Dockerfile

- 이미지 빌드 과정
  - ✓ docker build는 빌드 속도 향상을 위해 실행 중간에 있는 이미지 캐시를 사용
  - ✓ 기본적으로 첫 번째 빌드 과정에서 단계별로 캐시를 생성하고 이 빌드 캐시는 동일한 이미지 작업으로 제한
  - ✓ 빌드 과정에서 캐시를 사용하지 않는 경우 --no-cache를 지정하여 빌드
  - ✓ 다른 이미지로부터 만들어진 빌드 캐시를 사용해 빌드하려면 --cache-from 옵션을 통해 수행할 수 있음
  - ✓ Docker 18.09 버전에 Buildkit이 추가되어 이미지 빌드에 향상된 기능을 제공하기 시작했고 20.10 버전부터 안정화 되어서 기본 옵션으로 설정됨
  - ✓ Buildkit 기능
    - 빌드 과정을 병렬 처리하여 더 빠른 빌드를 제공
    - 사용되지 않는 빌드 단계를 찾아 비활성화
    - 비밀번호 등의 민감한 데이터가 포함되는 경우 secret 구축이 가능
    - 빌드 중 빌드 정보에 따라 변경된 파일만 전송
    - 자동 빌드 시 빌드 캐시의 우선 순위를 정할 수 있음

Application Image

- Python Django
  - ✓ 가상 환경 생성
    - sudo apt install python3.12-venv
    - mkdir django_workspace
    - cd django_workspace
    - sudo apt install python3-pip
    - pip3 install virtualenv --break-system-packages
    - python3 -m venv ./{your venv name}
    - 가상 환경 활성화
      - Linux, Mac: source {your venv name}/bin/activate
      - Windows: {your venv name}/Scripts/activate

Application Image

- Python Django
  - ✓ Django 설치 및 실행
    - (venv)$ pip install django
    - (venv)$ django-admin startproject dockerdjango
    - (venv)$ cd dockerdjango
    - (venv)$ python manage.py runserver
    - 브라우저에 접속해서 localhost:8000 으로 접속
  - ✓ 라이브러리의 의존성을 파일에 내보내기
    - pip freeze > requirements.txt

Application Image

- Python Django
  - ✓ Dockerfile 작성

```
FROM python

RUN apt update ₩
    && apt install -y --no-install-recommends ₩
                && rm -rf /var/lib/apt/lists/*

WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . .

EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

  - ✓ 이미지 빌드 후 실행
    - docker build -t dockerdjango .
    - docker run --name dockerdjango -p 8000:8000 -d dockerdjango:latest

Application Image

- Python FastAPI
  - ✓ 가상 환경 생성
    - mkdir fastapi_workspace
    - cd fastapi_workspace
    - python3 -m venv ./{your venv name}
    - 가상 환경 활성화
      - Linux, Mac: source {your venv name}/bin/activate
      - Windows: {your venv name}/Scripts/activate
  - ✓ 패키지 설치
    - pip install fastapi uvicorn

Application Image

- Python FastAPI
  - ✓ 소스 코드 작성 – main.py

```
from fastapi import FastAPI

# 1. FastAPI 인스턴스 생성
app = FastAPI()

# 2. 경로 연산자 데코레이터 정의
@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

Application Image

- Python FastAPI
  - ✓ Dockerfile 작성

```
# 1. 베이스 이미지 설정 (파이썬 3.12 슬림 버전)
FROM python:3.12-slim
# 2. 작업 디렉토리 설정
WORKDIR /app
# 3. 종속성 설치를 위해 requirements.txt 복사
# (현재 디렉토리에 requirements.txt가 있어야 합니다)
COPY requirements.txt .
# 4. 패키지 설치
RUN pip install --no-cache-dir -r requirements.txt
# 5. 나머지 소스 코드 복사
COPY . .
# 6. 컨테이너가 사용할 포트 명시
EXPOSE 8000
# 7. 서버 실행 (호스트 0.0.0.0으로 설정해야 외부 접속 가능)
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Application Image

- Python FastAPI
  - ✓ dockerignore 생성 (선택 사항이나 권장) 가상환경 폴더나 캐시 파일이 도커 이미지에 포함되지 않도록 막아줌

    myvenv/

    __pycache__/

    .pytest_cache/

    .env
  - ✓ 이미지 빌드

    docker build -t my-fastapi-app .
  - ✓ 컨테이너 실행

    docker run -d -p 8000:8000 my-fastapi-app
  - ✓ requirements.txt를 먼저 복사하고 패키지를 설치하는 이유는 도커 캐시(Layer Caching) 때문으로 소스 코드만 수정했을 경우 시간이 오래 걸리는 pip install 단계를 건너뛰고 바로 코드만 복사해서 빌드 속도를 획기적으로 줄일 수 있음

Application Image

❖ Spring Boot 프로젝트(Gradle) 이미지 생성 및 실행

✓ JDK 설치

- Spring Boot 3.x 버전은 Java 17 이상 필수

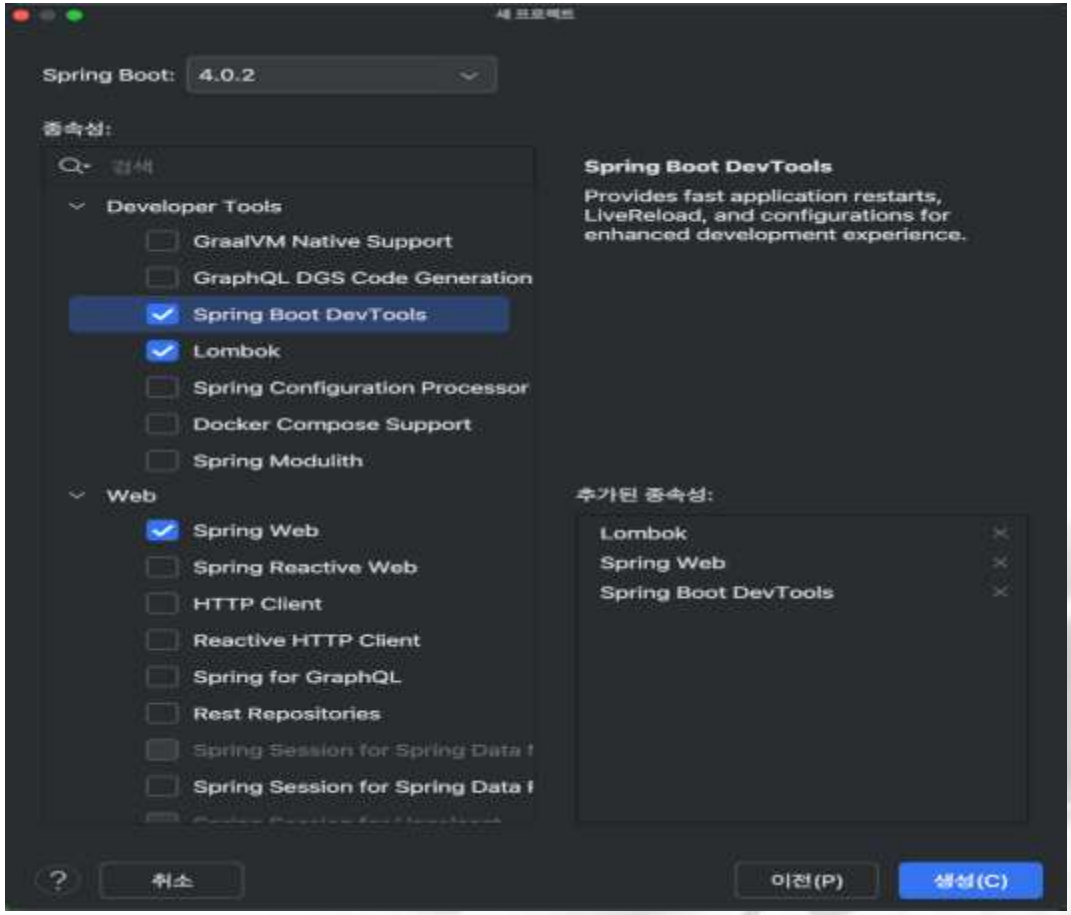
```
sudo apt update
sudo apt install openjdk-17-jdk -y
```

- 설치 확인

```
java -version
```

Application Image

- Spring Boot 프로젝트(Gradle) 이미지 생성 및 실행
  - ✓ IntelliJ 에서 스프링 부트 프로젝트(Spring Initializer) 생성



Application Image

- Spring Boot 프로젝트(Gradle) 이미지 생성 및 실행
  - ✓ 기본 패키지 안에 클래스를 생성하고 작성

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FrontController {
    @GetMapping("/")
    public String home(){
        return "Hello SpringBoot Application";
    }
}
```

  - ✓ 프로젝트 실행
  - ✓ 브라우저에서 localhost:8080 으로 접속해서 확인

Application Image

- Spring Boot 프로젝트(Gradle) 이미지 생성 및 실행
  - ✓ 리눅스에서 다운로드
    - sudo apt install git
    - git clone https://github.com/itstudy001/docker_spring.git
  - ✓ 실행
    - cd docker_spring
    - ./gradlew bootRun
  - ✓ 터미널에서 build

    ./gradlew clean build

Application Image

- Spring Boot 프로젝트(Gradle) 이미지 생성 및 실행
  - ✓ 프로젝트의 root 디렉토리에 Dockerfile 생성 및 작성

```
FROM amazoncorretto:17
CMD ["./mvnw", "clean", "package"]
ARG JAR_FILE=target/*.jar
COPY ./build/libs/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

  - ✓ 이미지 빌드 및 생성: docker build -f Dockerfile -t docker-example:0.0.1 .
  - ✓ 이미지 확인: docker images
  - ✓ 컨테이너 생성 및 실행: docker run -p 8080:8080 docker-example:0.0.1

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ 노드 설치 및 버전 확인
    - sudo apt update
    - sudo apt install nodejs npm
    - node –v
  - ✓ 이미 설치된 경우 최신 버전으로 업데이트
    - npm 캐시 삭제: sudo npm cache clean -f
    - n 모듈 설치: sudo npm install -g n
    - 최신 LTS 버전으로 업데이트: sudo n lts
  - ✓ 노드 프로젝트 생성
    - 디렉토리를 생성
      - mkdir node_workspace
      - cd node_workspace
    - 프로젝트 초기화: npm init
  - ✓ 2개의 패키지 설치(express, nodemon)

    npm install express nodemon

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ package.json 수정

```
{
  "name": "nodeapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ app.js 파일을 생성하고 작성

```
const express = require('express');
const app = express();

app.set('port', process.env.PORT || 3000);

app.get('/', (req, res) => {
    res.send('Hello, Express')
})

app.listen(app.get('port'), () => {
    console.log(app.get('port'), '번 포트에서 대기 중')
})
```

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ 애플리케이션 실행 - node app.js
  - ✓ 브라우저에서 localhost:3000 을 입력하고 확인

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ Dockerfile 작성

```
# 1. 베이스 이미지 설정 (LTS 버전의 슬림형 이미지)
FROM node:20-slim

# 2. 작업 디렉토리 생성
WORKDIR /usr/src/app

# 3. 의존성 파일만 먼저 복사 (캐시 활용)
COPY package*.json ./

# 4. 패키지 설치
RUN npm install

# 5. 소스 코드 복사
COPY . .

# 6. 서비스 포트 노출
EXPOSE 3000

# 7. 앱 실행
CMD [ "node", "app.js" ]
```

Application Image

- Node 프로젝트 이미지 생성 및 실행
  - ✓ node:alpine vs node:slim: alpine은 용량이 매우 작지만(약 5MB), 일부 라이브러리(C++ 컴파일 필요 시) 설치가 까다로울 수 있으므로 slim은 호환성과 용량 사이의 가장 적절한 타협점
  - ✓ npm ci: 운영 환경(Production) 빌드 시에는 npm install 대신 npm ci를 사용하는 것이 좋은데 package-lock.json에 명시된 버전과 완전히 일치하도록 설치하기 때문

Application Image

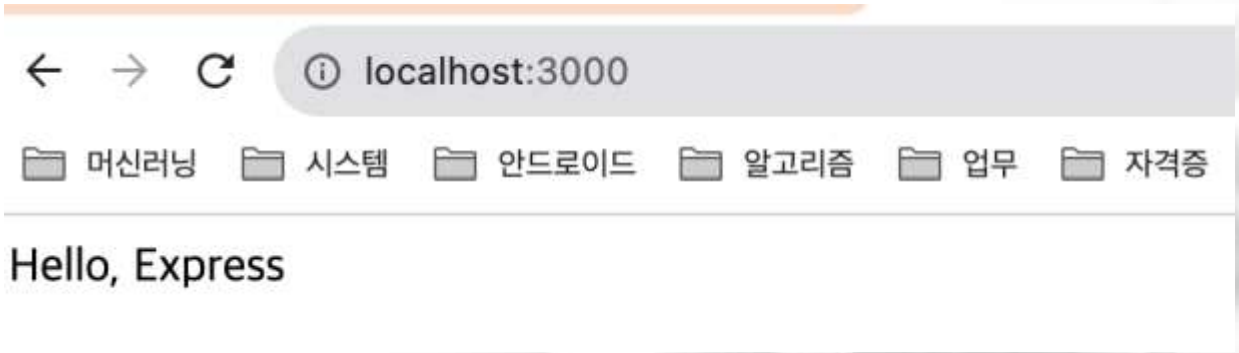
- Node 프로젝트 이미지 생성 및 실행
  - ✓ .dockerignore 파일 작성: 이미지 빌드 시 불필요한 파일이 포함되어 이미지 용량이 커지는 것을 막아줌 - Dockerfile과 같은 위치에 생성

    node_modules

    npm-debug.log

    .git

    .env
  - ✓ 이미지 빌드: docker build -t nodeapp .
  - ✓ 컨테이너 생성 및 실행: docker run -d -p 3000:3000 --name=dockernode nodeapp
  - ✓ 브라우저에서 localhost:3000 을 입력하고 확인



Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ golang 설치
    - 기존에 설치된 go가 있다면 삭제

      sudo rm -rf /usr/local/go
    - Go 다운로드 (버전 숫자는 변경될 수 있음)
      - x86_64인 경우 (Standard Intel/AMD)

        curl -OL https://golang.org/dl/go1.22.0.linux-amd64.tar.gz

        sudo tar -C /usr/local -xzf go1.22.0.linux-amd64.tar.gz
      - aarch64 (또는 arm64)인 경우

        curl -OL https://golang.org/dl/go1.22.0.linux-arm64.tar.gz

        sudo tar -C /usr/local -xzf go1.22.0.linux-arm64.tar.gz

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ golang 설치
    - 어느 디렉토리에서나 go 명령어를 사용할 수 있도록 환경 변수를 설정
      - ◆ ~/.profile 파일 끝에 경로 추가

      echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.profile

      echo 'export PATH=$PATH:$(go env GOPATH)/bin' >> ~/.profile

      - ◆ 변경사항 적용

      source ~/.profile
    - 확인

      go version

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 생성 및 실행
    - 디렉토리 생성

      mkdir golang

      cd golang
    - 프로젝트 초기화

      go mod init my-go-app

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 생성 및 실행
    - main.go 작성

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
)
```

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 생성 및 실행
    - main.go 작성

```
func main() {
    // 1. 기본 경로 ("/") 핸들러
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(map[string]string{"message": "Hello from Go!"})
    })

    // 2. 파라미터 확인용 경로 ("/health")
    http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "OK")
    })

    fmt.Println("서버가 8080 포트에서 시작되었습니다...")

    // 3. 서버 실행 (0.0.0.0으로 해야 도커 외부에서 접속 가능)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        panic(err)
    }
}
```

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 생성 및 실행
    - 실행

      go run main.go
    - # 다른 터미널에서 확인

      curl http://localhost:8080

      # 결과: {"message":"Hello from Go!"}

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 이미지 생성 및 컨테이너 실행
    - Dockerfile 작성

```
# 1. Go 환경이 포함된 베이스 이미지 사용
FROM golang:1.23-alpine

# 2. 컨테이너 내부 작업 디렉토리 설정
WORKDIR /app

# 3. 소스 코드 복사 (go.mod, go.sum 포함)
COPY . .

# 4. 의존성 다운로드 및 빌드
RUN go mod download
RUN go build -o main .

# 5. 실행할 포트 노출
EXPOSE 8080

# 6. 바이너리 실행
CMD ["./main"]
```

Application Image

- golang 프로젝트 이미지 생성 및 실행
  - ✓ go 프로그램 이미지 생성 및 컨테이너 실행
    - 이미지 빌드

      docker build -t my-go-app-single .
    - 컨테이너 실행

      docker run -p 8080:8080 my-go-app-single

Application Image

- ADD 명령어의 자동 압축 해제 기능 활용
  - ✓ 웹 소스 코드 다운로드: git clone https://github.com/brayanlee/webapp.git
  - ✓ 디렉토리 이동: cd webapp
  - ✓ 디렉토리 생성 및 이동: mkdir dockerfiles
  - ✓ dockerfiles 디렉토리에 Dockerfile 작성

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:latest

#필요한 패키지를 설치
RUN apt update
RUN apt install -y apache2
RUN apt install -y curl
RUN apt install -y vim

ADD webapp.tar.gz /var/www/html

WORKDIR /var/www/html

EXPOSE 80

CMD /usr/sbin/apachectl -D FOREGROUND
```

Application Image

- ADD 명령어의 자동 압축 해제 기능 활용
  - ✓ 이미지 빌드: docker build -t webapp:5.0 -f ./dockerfiles/Dockerfile .
  - ✓ 실행: docker run -dit -p 8008:80 --name=webapp5 webapp:5.0
  - ✓ 확인: 브라우저에서 localhost:8008 로 확인

Application Image

- 이미지 용량 절감
  - ✓ apt를 이용한 패키지 업데이트와 설치 시 남게 되는 캐시를 제거하여 생성되는 이미지의 용량이 감소됨을 확인
  - ✓ 캐시 삭제 관련 명령: apt clean, apt autoremove, rm -rfv ~
  - ✓ index.html 파일을 생성

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Docker Container App</title>
    <style>body {margin-top: 40px; background-color: #333;}</style>
  </head>
  <body>
    <div style=color:white;text-allgn:center>
      <hl> Docker Container Web Application. </hl> <h2> Great Works! </h2>
      <p> Application is now good running on a container in Docker. </p>
    </div>
  </body>
</html>
```

Application Image

- 이미지 용량 절감
  - ✓ Dockerfile을 작성

```
# 베이스 이미지로 Ubuntu를 지정
FROM ubuntu:latest

#필요한 패키지를 설치
RUN apt update
RUN apt install -y apache2 -qq --no-install-recommends
RUN apt clean -y
RUN apt autoremove -y
RUN rm -rfv /var/lib/apt/lists/* /tmp/* /var/tmp/*

WORKDIR /var/www/html

ADD index.html .

EXPOSE 80

CMD apachectl -D FOREGROUND
```

  - ✓ 이미지 빌드

```
docker build -t webapp:6.0  .
```

Application Image

- 이미지 용량 절감
  - ✓ apt clean: 설치에 사용한 패키지 라이브러리, 임시 파일, 오래된 파일을 삭제하는데 autoclean을 사용하면 더 이상 설치되어 있지 않은 패키지들의 .deb까지 삭제
  - ✓ apt autoremove: 다른 패키지들의 종속성을 충족시키기 위해 자동으로 설치된 패키지를 삭제
  - ✓ rm -rfv /tmp/* /var/lib/apt/lists/* /var/tmp/*: apt와 연관된 캐시 파일을 모두 삭제하는데 이미지의 시스템 소프트웨어를 다시 업데이트할 계획이 있다면 삭제하지 않는 것이 좋음

Application Image

- ❖ 다단계 빌드
  - ✓ 개요
    - 빌드를 여러 단계로 나누어 빌드하는 것
    - 각 빌드 단계는 FROM 명령어로 시작되며 필요한 경우 빌드 단계에 AS 파라미터를 이용해 이름을 붙일 수 있음
    - 빌드가 여러 단계로 나뉘어 있다고는 하지만 최종 산출물은 마지막 단계의 내용물을 담은 도커 이미지
    - 각 빌드 단계는 독립적으로 실행되지만 앞선 단계에서 만들어진 디렉터리나 파일을 복사할 수 있음
    - 멀티 스테이지 Dockerfile 스크립트를 통해 빌드 과정을 세밀하게 조정하며 최종 산출물인 이미지를 가능한 한 작게 유지할 수 있음
    - 비단 컴파일러에만 국한된 내용은 아님
    - 어떤 도구든지 그 도구가 사용되는 단계만으로 도구의 포함 여부를 국한시킬 있음
    - 최종 산출물인 이미지에 불필요한 도구는 빼버릴 수 있는 것
    - 좋은 예가 curl인데 curl은 인터넷을 통해 필요한 파일을 내려받을 수 있는 중요한 도구이지만 파일 다운로드를 빌드 초기 단계에 모아 놓는다면 최종 이미지에는 curl을 포함시키지 않아도 됨
    - 이런 방법으로 이미지 크기를 줄여서 애플리케이션의 시작 시간을 단축할 수 있으며 애플리케이션의 의존 모듈 자체를 줄여 취약점을 이용한 외부 공격의 가능성도 최대한 차단할 수 있음

Application Image

- ❖ 다단계 빌드
  - ✓ node 다단계 빌드
    - Dockerfile 수정

```
# --- 빌드 단계 ---
FROM node:20-slim AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# --- 실행 단계 ---
FROM node:20-alpine
WORKDIR /app
# 빌드 단계에서 생성된 필요한 파일만 복사
COPY --from=builder /app ./

EXPOSE 3000
# 컨테이너 실행 시 권한 제한을 위해 node 사용자 사용 (보안 권장)
USER node
CMD [ "node", "app.js" ]
```

Application Image

- 다단계 빌드
  - ✓ node 다단계 빌드
    - 이미지 빌드: docker build -t my-node-app .
    - 컨테이너 실행: docker run -d -p 3000:3000 --name node-server my-node-app

Application Image

- ❖ 다단계 빌드
  - ✓ Java Gradle 다단계 빌드
    - ● Dockerfile 작성

```
# --- 1단계: 빌드 스테이지 ---
FROM eclipse-temurin:17-jdk-jammy AS build
WORKDIR /app

# Gradle 래퍼와 설정 파일들을 먼저 복사 (캐시 활용)
COPY gradlew .
COPY gradle gradle
# 실행 권한 부여 (이 줄을 추가하세요)
RUN chmod +x gradlew
COPY build.gradle .
COPY settings.gradle .

# 종속성 먼저 다운로드 (코드 수정 시 빌드 속도 향상)
RUN ./gradlew dependencies --no-daemon

# 소스 코드 복사 및 빌드
COPY src src
RUN ./gradlew clean build -x test --no-daemon
```

Application Image

- ❖ 다단계 빌드
  - ✓ Java Gradle 다단계 빌드
    - ● Dockerfile 작성

```
# --- 2단계: 실행 스테이지 ---
FROM eclipse-temurin:17-jre-jammy
WORKDIR /app

# 빌드 스테이지에서 생성된 jar 파일만 복사
# (빌드 결과물 파일명은 보통 프로젝트명-버전.jar 형태입니다)
COPY --from=build /app/build/libs/*.jar app.jar

# 컨테이너가 사용할 포트 (기본 8080)
EXPOSE 8080

# 앱 실행
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Application Image

- ❖ 다단계 빌드
  - ✓ Java Gradle 다단계 빌드
    - ● Dockerfile 설정 내용
      - ◆ eclipse-temurin: 안정적이고 가벼운 OpenJDK 배포판
      - ◆ -x test: 빌드 시 테스트 과정을 제외하여 빌드 속도를 높임(실제 운영 환경에서는 테스트를 통과한 후 빌드하는 것이 좋음)
      - ◆ JRE 이미지 사용: 실행 단계에서는 컴파일 도구가 포함된 JDK 대신 실행 전용인 JRE 이미지를 사용하여 이미지 크기를 수백 MB 이상 줄일 수 있음
      - ◆ ENTRYPOINT: CMD와 비슷하지만, 컨테이너가 시작될 때 반드시 실행되어야 하는 명령어를 지정할 때 사용

Application Image

- Python FastAPI
  - ✓ 실행

    uvicorn main:app --reload
  - ✓ 확인

    curl http://127.0.0.1:8000

    {"Hello":"World"}
  - ✓ 라이브러리의 의존성을 파일에 내보내기

    pip freeze > requirements.txt

Application Image

- 다단계 빌드
  - ✓ Java Gradle 다단계 빌드
    - 이미지 빌드: docker build -t my-spring-app .
    - 컨테이너 실행: docker run -d -p 8080:8080 --name spring-server my-spring-app

Application Image

- ❖ 다단계 빌드
  - ✓ golang 다단계 빌드
    - Dockerfile 작성

```
# --- 1단계: 빌드 스테이지 (Go 컴파일러 포함) ---
FROM golang:1.23-alpine AS builder

# 작업 디렉토리 설정
WORKDIR /app

# 의존성 파일 복사 및 다운로드 (캐시 활용)
COPY go.mod go.sum* ./
RUN go mod download

# 전체 소스 코드 복사 및 빌드
COPY . .
# CGO_ENABLED=0: 정적 라이브러리 포함 (어떤 환경에서도 실행되도록)
# GOOS=linux: 대상 OS 지정
RUN CGO_ENABLED=0 GOOS=linux go build -o main .
```

Application Image

- ❖ 다단계 빌드
  - ✓ golang 다단계 빌드
    - Dockerfile 작성

```
# --- 2단계: 실행 스테이지 (최소 크기 이미지) ---
FROM alpine:latest
# 보안을 위해 필요한 인증서(HTTPS 통신용) 등 설치
RUN apk --no-cache add ca-certificates

WORKDIR /root/

# 빌드 스테이지에서 생성된 'main' 바이너리만 복사
COPY --from=builder /app/main .

# 실행 권한 부여 및 포트 노출
EXPOSE 8080

# 앱 실행
CMD ["./main"]
```

Application Image

- 다단계 빌드
  - ✓ golang 다단계 빌드
    - 이미지 빌드: docker build -t my-go-app .
    - 컨테이너 실행: docker run -d -p 8081:8080 my-go-app

Application Image

- ❖ 다단계 빌드
  - ✓ Fast API 다단계 빌드
    - ● Dockerfile 수정

```
# --- 1단계: 빌드 스테이지 (Builder) ---
FROM python:3.12-slim AS builder

# 작업 디렉토리 설정
WORKDIR /app

# 환경변수 설정 (파이썬이 .pyc 파일을 생성하지 않도록 설정)
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# 가상환경 생성 및 경로 설정
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

# 종속성 설치 (캐시 활용을 위해 requirements.txt만 먼저 복사)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

Application Image

- ❖ 다단계 빌드
  - ✓ Fast API 다단계 빌드
    - Dockerfile 수정

```
# --- 2단계: 실행 스테이지 (Runner) ---
FROM python:3.12-slim
WORKDIR /app
# 빌드 스테이지에서 생성된 가상환경(/opt/venv)만 복사
COPY --from=builder /opt/venv /opt/venv
# 가상환경의 바이너리를 사용하도록 환경변수 설정
ENV PATH="/opt/venv/bin:$PATH"
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
# 소스 코드 복사
COPY . .
# 보안을 위해 root가 아닌 일반 사용자 생성 및 전환
RUN adduser --disabled-password --gecos "" appuser
USER appuser

# 포트 노출
EXPOSE 8000

# 앱 실행
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Application Image

- 다단계 빌드
  - ✓ Fast API 다단계 빌드
    - 이미지 빌드: docker build -t fastapi-multi-stage .
    - 이미지 확인: docker images | grep fastapi
    - 컨테이너 생성: docker run -d -p 8000:8000 fastapi-multi-stage

Application Image

- Dockerfile 작성 시 유의 사항
  - ✓ 이미지 크기 최소화 (Lightweight)
    - 이미지가 무거우면 배포 속도가 느려지고 저장 공간을 많이 차지함
    - 경량 베이스 이미지 사용: ubuntu 전체 이미지보다는 alpine이나 slim 버전의 이미지를 사용하는 것이 좋음 (예: python:3.9-slim)
    - 레이어 수 줄이기
      - Dockerfile 명령어의 수와 도커 이미지 레이어 수는 동일한데 레이어 수가 많을수록 이미지를 생성하는 빌드 시간은 길어질 것이고 파일 용량도 커짐
      - RUN 명령어를 여러 번 쓰기보다 &&를 사용해 하나로 합치면 이미지 레이어 수가 줄어 용량이 최적화됨
    - 불필요한 파일 삭제: 패키지 설치 후 캐시 파일(rm -rf /var/lib/apt/lists/*) 등을 즉시 삭제
  - ✓ 빌드 캐시(Build Cache) 활용
    - 도커는 빌드 시 변경되지 않은 레이어를 재사용하므로 이 효율을 극대화
    - 순서가 중요: 자주 바뀌는 소스 코드 복사(COPY . .)는 나중에 잘 바뀌지 않는 의존성 설치(RUN npm install)는 앞부분에 배치
    - .dockerignore 파일 작성: .git, 로그 파일, 로컬 노드 모듈 등 빌드에 필요 없는 파일이 이미지에 포함되지 않도록 설정

Application Image

- Dockerfile 작성 시 유의 사항
  - ✓ 보안 및 관리 설정
    - Root 계정 사용 지양: 보안을 위해 USER 명령어를 사용하여 별도의 일반 사용자 계정으로 프로세스를 실행하는 것이 안전
    - 명확한 태그 사용: latest 태그는 버전 관리가 어려우므로 v1.0.1 처럼 명확한 버전 태그를 지정
    - 멀티 스테이지 빌드 (Multi-stage Build): 빌드 도구(컴파일러 등)는 빌드 단계에서만 쓰고 최종 이미지에는 실행 파일만 남겨 보안과 용량을 동시에 최적화
  - ✓ 도커의 철학! 하나의 애플리케이션은 하나의 컨테이너에
    - 하나의 컨테이너에 2개 이상의 애플리케이션을 설정하게 되면 애플리케이션의 결합성이 높고 확장성을 저해
    - 하나의 컨테이너에 하나의 애플리케이션 동작은 컨테이너 간의 독립성을 보장함과 동시에 애플리케이션 버전 관리, 소스 코드 모듈화 등에서 이점이 있음
    - 모놀리식 구성보다는 결합 해제된 애플리케이션(decouple applications) 설계나 마이크로서비스 지향적 설계를 고려해야 장애가 발생해도 애플리케이션 자체가 유지될 수 있음
  - ✓ IaC 환경 개발은 디렉터리 단위로
    - Dockerfile로 이미지 빌드 시 현재 위치로부터 하위 경로의 모든 디렉터리와 파일을 도커 컨텍스트context에 저장한 뒤 작업이 진행하는데 이를 빌드 컨텍스트라고 하는데 이미지 빌드와 상관없는 파일이 포함되지 않도록 별도의 디렉터리를 생성한 뒤 독립된 환경에서 빌드가 이루어져야 빌드 성능에 도움이 됨

Application Image

- Dockerfile 작성 시 유의 사항
  - ✓ 서버리스 환경으로 개발
    - Dockerfile을 통해 미리 설정된 환경을 서버리스 애플리케이션 환경이라고 하고 이러한 환경은 사용자가 서버를 프로비저닝, 확장 및 관리할 필요가 없음
    - 거의 모든 유형의 애플리케이션 또는 백엔드 서비스를 위해 서버리스 애플리케이션을 구축할 수 있으며 애플리케이션을 고가용성으로 실행하고 확장하는 데 필요한 모든 것이 자동으로 처리됨
    - 서버리스 애플리케이션 구축은 개발자가 클라우드에서든 온프레미스에서든 서버 및 런타임의 관리 및 운영에 필요한 업무량을 줄이고 핵심 개발에 집중할 수 있다는 것을 의미하는데 이렇게 오버헤드가 줄어들면 개발자는 확장성과 안정성을 갖춘 좋은 제품을 개발하는 데 시간과 열정을 쏟을 수 있음

Application Image

- React 프로젝트 다단계 빌드
  - ✓ 프로젝트 생성: pm create vite@latest 내-프로젝트-이름 -- --template react
  - ✓ 확인: curl localhost:5173/

Application Image

- React 프로젝트 다단계 빌드
  - ✓ Dockerfile 작성

```
# 1단계: 빌드 (Node.js 환경)
FROM node:20-alpine AS build-stage
WORKDIR /app

# 캐싱을 위해 package.json 먼저 복사
COPY package*.json ./
RUN npm install

# 전체 소스 복사 및 빌드
COPY . .
RUN npm run build

# 2단계: 실행 (Nginx 환경)
FROM nginx:stable-alpine
# Vite의 기본 빌드 폴더인 dist를 Nginx의 정적 파일 폴더로 복사
COPY --from=build-stage /app/dist /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Application Image

- React 프로젝트 다단계 빌드
  - ✓ .dockerignore 파일 작성

    node_modules

    dist

    .git

    .DS_Store
  - ✓ 이미지 빌드: docker build -t my-react-app .
  - ✓ 컨테이너 실행: docker run -d -p 8080:80 --name react-container my-react-app