4장 쿠버네티스를 이루는 컨테이너 도우미, 도커

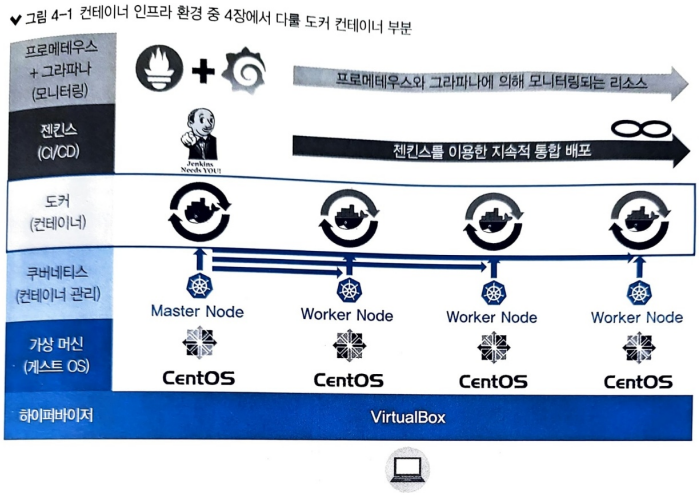
💡**질문!**

볼륨을 사용했을 때 컨테이너에 존재하는 파일을 그래도 보존할 수 있고 변경해서 사용할 수 있는 것 말고 더 효과적인 기능은 없나?

▼ 답변

1. **포터빌리티(Portability)**: 볼륨은 Docker의 데이터 관리를 위한 추상화 계층을 제공하므로 호스트 시스템에 대한 상세한 정보를 알 필요가 없습니다. 이는 도커 엔진이 실행되는 모든 환경에서 동일하게 작동하므로 코드의 포터빌리티를 증가시킵니다.
2. **드라이버 지원**: 볼륨은 스토리지 백엔드에 따라 다양한 드라이버를
3. 사용할 수 있습니다. 예를 들어, 클라우드 기반 스토리지 또는 네트워크 파일 시스템(NFS) 같은 스토리지 백엔드를 이용하고 싶을 수 있습니다. 이러한 옵션은 바인드 마운트에서는 사용할 수 없습니다.
    a. NFS는 Network File System의 약자로, 여러 컴퓨터가 네트워크를 통해 같은 파일을 공유하고 접근할 수 있게 해주는 분산 파일 시스템
4. **백업과 마이그레이션**: 볼륨은 도커가 관리하기 때문에, 볼륨 데이터의 백업이나 마이그레이션을 수행하는 것이 더 쉽습니다. Docker CLI 또는 API를 통해 볼륨을 생성하고 삭제하고, 데이터를 백업하고 복원할 수 있습니다.
5. **보안**: 볼륨은 Docker가 관리하므로, 간접적으로 파일시스템의 권한과 소유권을 관리할 수 있습니다. 이는 바인드 마운트보다 보안적으로 우수하다고 할 수 있습니다. 또한, 바인드 마운트는 호스트 시스템의 임의의 디렉토리에 액세스할 수 있으므로 잠재적으로 위험할 수 있습니다.

4.1 도커를 알아야 하는 이유



4.1.1 파드, 컨테이너, 도커, 쿠버네티스의 관계

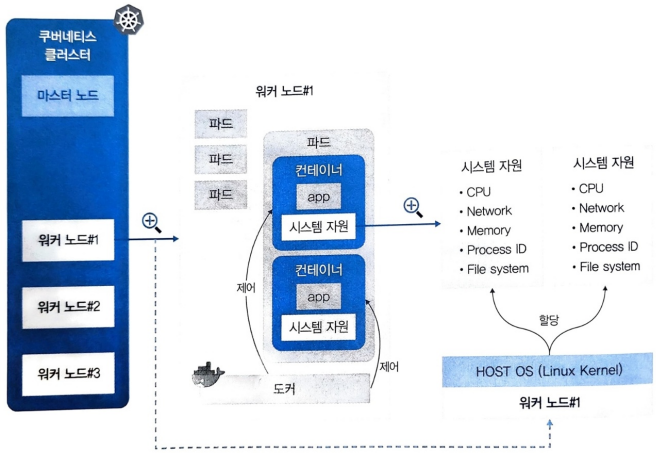
파드

- 워커 노드라는 노드 단위로 관리
- (워커 노드 + 마스터 노드) 모여서 ⇒ 쿠버네티스 클러스터
- 파드는 1개 이상의 컨테이너로 구성됨
- 쿠버네티스로부터 IP를 받아 컨테이너가 외부로 통신할 수 있는 경로 제공
- 컨테이너들의 정상작동 확인, 네트워크/저장공간 서로 공유하도록 함
- 따라서, 컨테이너들은 마치 하나의 호스트에 존재하는 것처럼 작동 가능
- 쿠버네티스 마스터 —돌봄—> 쿠버네티스 워커 노드 —돌봄→ 파드 —돌봄—> 컨테이너

그림 4-2 쿠버네티스 클러스터의 구조

컨테이너

그림 4-3 쿠버네티스, 파드 그리고 컨테이너 간의 연결 구조



- 하나의 os 안에서 커널을 공유하며 개별적인 실행 환경을 제공하는 격리된 공간
  - *개별적인 실행 환경 : 시스템 자원 (CPU, NW, Mem)을 독자적으로 사용하도록 할당된 환경
  - *개별적인 실행환경에서 실행되는 프로세스를 구분하는 ID도 컨테이너 안에 격리돼 관리됨
- 내부에서 실행되는 애플리케이션들은 서로 영향을 미치지 않고 **독립적으로** 작동 가능

도커의 등장

- 각 컨테이너가 독립적으로 작동하므로 여러 컨테이너를 효과적으로 다룰 방법이 필요해졌음
- 오래전부터 Unix/Linux는 하나의 호스트 OS안에서 자원을 분리해 할당하고 실행되는 프로세스를 격리해 관리하는 방법을 제공함
- But, 파일 시스템을 설정하고 자원 공간을 관리하는 등의 복잡한 과정을 직접 수행해야 해서 일부 전문가만 사용할 수 있다는 단점이 있음
- 따라서, 도커 등장
  - 이러한 복잡한 과정을 쉽게 만들어주는 도구
  - 컨테이너를 사용하는 방법을 명령어로 정리한 것
  - 사용하면 사용자가 따로 신경쓰지 않아도 컨테이너를 생성할 시 개별적인 실행환경을 분리하고 자원 할당함

4.1.2 다양한 컨테이너 관리 도구

대표적인 컨테이너 관리 도구

- 컨테이너디(Containerd)
- 크라이오(CRI-O)
- 카타 컨테이너(Kata Containers)
- **도커(Docker)**
  - 컨테이너 관리 도구
  - 컨테이너를 실행하는 데 필요한 이미지를 만들거나 공유하는 등의 다양한 기능 제공
  - 사용자가 명령어를 입력하는 CLI와 명령을 받아들이는 도커 데몬으로 구성되어 있음
  - 도커, 쿠버네티스를 함께 설치할 경우 쿠버네티스는 컨테이너 오케스트레이션을 위해 도커에 포함된 컨테이너디를 활용
  - NW를 통한 호출로 작동하므로 구조적으로 다소 복잡한 편이지만 이를 모두 도커에서 관리하기 때문에 사용자 입장에서는 신경쓰지 않아도 됨
  -

4.2 도커로 컨테이너 다루기

🚩**목표**

- 도커로 컨테이너를 다루는 기본 명령들을 실습해보자!
- 도커 이미지를 내려받아 컨테이너로 실행하고 도커 이미지와 컨테이너를 삭제하는 방법까지!
- 이미지 찾기 → 실행하기 → 디렉터리와 연결하기 → 삭제하기

컨테이너 이미지와 컨테이너의 관계 정리

- 베이그런트 이미지 : 이미지 자체로 사용할 수 X, 베이그런트를 실행할 때 추가해야만 사용가능
- 컨테이너 이미지 :
  그대로는 사용할 수 없고 **도커와 같은 CRI로** 불러들여야 **컨테이너가 실제로 작동함**
  (즉, 실행 파일과 실행된 파일관계)
- 따라서, 컨테이너 삭제 시 내려받은 이미지와 이미 실행된 컨테이너를 모두 삭제해야 디스크의 용량을 온전히 확보할 수 있음

4.2.1 컨테이너 이미지 알아보기

🚩목표

- 먼저 컨테이너를 만들 이미지가 있어야 한다
- 이미지를 검색해서 내려받고 구조를 살펴보자!

이미지 검색하고 내려받기

- 이미지는 레지스트리(registry)라고 하는 저장소에 모여있다.
  - 레지스트리는 도커 허브처럼 공개된 유명 레지스트리일 수도 있고
    내부 구축된 레지스트리일 수도 있다.
  - 이미지는 레지스트리 웹 사이트에서 직접 검색해도 되고
    슈퍼푸티 명령 창에서 쿠버네티스 마스터 노드에 접속해 검색할 수도 있음
  - 별도의 레지스트리를 지정하지 않으면 기본으로 도커 허브에서 이미지를 찾음
- `docker search <검색어>`
  - 특정한 이름(검색어)을 포함하는 이미지가 있는지 찾음
  - 이미지는 애플리케이션, 미들웨어 등 고유한 목적에 맞게 패키지되어 있음
  - 예시 - docker search nginx

  ▼ 표시되는 각 열의 의미

  - INDEX : 이미지가 저장된 레지스트리의 이름
  - NAME : 검색된 이미지 이름.
    공식 이미지 제외한 나머지는 '레지스트리 주소/저장소 소유자/이미지 이름' 형태임
  - DESCROPTION : 이미지에 대한 설명
  - STAR : 해당 이미지를 내려받은 사용자에게 받은 평가 횟수
  - OFFICIAL : [OK] 공식
  - AUTOMATED : [OK] 도커 허브에서 자체 제공 이미지 빌드 자동화기능을 활용해 생성한 이미지
- `docker pull nginx`
  - 이미지 내려받기

  ▼ 이미지를 내려받을 때 사용하는 태그, 레이어, 이미지의 고유 식별 값

- **태그**
  - 아무런 조건을 주지않고 이미지 이름만으로 pull을 수행하면 기본으로 latest 태그가 적용됨.
  - 가장 최신 이미지를 의미함
- **레이어**
  - d121f8d1c412와 같이 pull을 수행해 내려받은 레이어.
  - 하나의 이미지는 여러 개의 레이어로 이루어져 있어서 레이어마다 Pull complete 메시지가 발생
- **다이제스트**
  - 이미지의 고유 식별자.
  - 이미지에 포함된 내용과 이미지 생성 환경을 식별 가능.
  - 식별자는 해시 함수로 생쇵되며 이미지가 동일한지 검증하는 데 사용
  - 이름이나 태그는 이미지를 생성할 때 임의로 지정하므로 이름이나 태그가 같아고 해서 같은 이미지라고 할 수 없음
  - But, 다이제스트는 고유한 값이므로 다이제스트가 같은 이미지는 이름이나 태그가 다르더라도 같은 이미지임
- 상태
  - 이미지를 내려받은 레지스트리, 이미지, 태그 등의 상태 정보를 확인할 수 있음
  - 형식 - '레지스트리 이름/이미지 이름:태그'
  - 내려받은 이미지는 docker.io 레지스트리에서 왔으며, 이미지의 이름은 nginx, 태그는 앞서 설명한 것처럼 별도의 태그를 지정하지 않았으므로 latest

이미지 태그

태그

- 이름이 동일한 이미지에 추가하는 식별자
- 이름이 동일해도 도커 이미지의 버전이나 플랫폼이 다를 수 있으므로 이를 구분하는 데 사용
- 이미지를 내려받기/컨테이너 구동 시에는 이미지 이름만 사용.
  태그 명시하지 않으면 latest 태그 기본으로 사용.
- 이미지 태그와 관련된 정보는 해당 이미지의 도커 허브 메뉴 중 Tags 탭에서 확인 가능
- 안정화 버전 사용 - `docker pull nginx:stable`

이미지의 레이어 구조

이미지

- 애플리케이션과 각종 파일을 담고 있다는 점에서 ZIP 같은 압축 파일에 가까움
- But, 압축 파일은 압축한 파일의 개수에 따라 전체 용량 증가
  이미지는 같은 내용일 경우 여러 이미지에 동일한 레이어를 공유하므로 **전체 용량 감소**

⏩실습 : 이미지의 레이어 구조 확인하기

```
docker pull nginx:stable
docker images nginx
docker history nginx:stable
docker history nginx:latest
```

ADD file:e74 ~ 부분이 stable과 latest가 같기 때문에 레이어를 공유 ⇒ 효율적인 디스크 용량 사용

4.2.2 컨테이너 실행하기

⏩실습: 컨테이너 단순히 실행하기

```
docker run -d --restart always nginx
```

- 결과값으로 컨테이너를 식별할 수 있는 고유한 ID인 16진수 문자열이 나온다.
- **컨테이너 생성** 명령 형식 `docker run [옵션] <사용할 이미지 이름>[:태그 | @다이제스트]`

▼ 여기서 사용된 옵션

- **-d(--detach)**
  - 컨테이너를 백그라운드에서 구동한다는 의미
  - 봅션을 생략하고 ctrl+c를 누르면 애플리케이션 뿐만아니라 컨테이너도 함께 중단됨
- **--restart always**
  - 컨테이너의 재시작과 관련된 정책을 의미하는 옵션
  - 도커 서비스가 중지되는 경우 컨테이너도 작동이 중지되는데 VM을 중지한 후 다시 실행해도 자동으로 컨테이너가 기존 상태를 이어갈 수 있게 사용

```
docker ps
```

- **생성한 컨테이너 상태를 확인**

▼ 조회 결과에서 각 열의 의미

- **CONTAINER ID :** 컨테이너를 식별하기 위한 고유 ID
- **IMAGE :** 컨테이너를 만드는 데 사용한 이미지, 여기서는 nginx
- **COMMAND**
  - 컨테이너가 생성될 때 내부에서 작동할 프로그램을 실행하는 명령어

    - 여기서는 /docker-entrypoint.sh이다. 해당 셸이 nginx 이미지로 컨테이너가 생성될 때 nginx 프로그램을 호출해 서비스를 하도록 해줌
  - **CREATED :** 컨테이너가 생성된 시각 표시
  - **STATUS :** 컨테이너 작동 시작한 시각 표시
  - **PORTS :** 컨테이너가 사용하는 프로토콜 표시
  - **NAMES :** 컨테이너 이름 표시

```
docker ps -f id=cec7
```

- **컨테이너를 지정해 검색**
- docker ps에 -f <필터링 대상> 옵션을 주면 검색 결과를 필터링할 수 있음
- id=cec7 : CONTAINER ID에 cec7이라는 문자열이 포함된 컨테이너만을 출력

▼ 🔎 **자주 사용하는 필터링 키**

id : 컨테이너 아이디

name : 컨테이너 이름

label : 컨테이너 레이블

exited : 컨테이너가 종료됐을 때 반환하는 숫자 코드

status : 컨테이너의 작동 상태

ancestor : 컨테이너가 사용하는 이미지

```
curl 127.0.0.1
```

- 생성된 nginx 컨테이너는 마스터 노드 내부에 존재하므로 curl 명령으로 컨테이너가 제공하는 nginx 웹 페이지 정보를 가지고 오게 함
- 오류가 발생함 ⇒ why? 앞에서 컨테이너의 PORTS의 80/tcp는 컨테이너 내부에서 TCP 프로토콜의 80번 포트를 사용한다는 의미였으나, curl 127.0.0.1로 전달한 요청은 로컬호스트(127.0.0.1)의 80번 포트로 전달만 될 뿐 컨테이너까지는 도달하지 못함. **즉, 호스트에 도달한 후 컨테이너로 도달하기 위한 추가 경로 설정이 되어있지 않은 것**

⏩ 실습: 추가로 경로를 설정해 정상적으로 컨테이너 실행하기

```
docker un -d -p 8080:80 --name nginx-exposed --restart always nginx
```

- docker run에 -p 8080:80 옵션을 추가해 **새로운 컨테이너(nginx-exposed)를 실행함**
- -p는 외부에서 호스트로 보낸 요청을 컨테이너 내부로 전달하는 옵션
  `-p <요청 받을 호스트 포트>:<연결할 컨테이너 포트>`

```
docker ps -f name=nginx-exposed
```

- **컨테이너가 제대로 작동하는지 확인**
- 이름을 필터링해서 확인해보자
- 달라진 점
  - **0.0.0.0:8080→80/tcp :** 0.0.0의 8080번 포트로 들어오는 요청을 컨테이너 내부의 80번 포트로 전달한다는 의미
  - **nginx-exposed :** 현재 작동중인 컨테이너의 이름

```
웹 브라우저에서 192.168.1.10:8080을 입력해
가상 머신을 호스팅하는 PC나 노트북에서 컨테이너로 접근할 수 있는지 확인
```

- 현재 nginx 내부에는 따로 작성한 파일이 없으므로 기본 페이지를 보여줌
- 사용자가 원하는 페이지 출력 위해서는 별도로 작성해야함
- 컨테이너 내부에서 웹 페이지 파일을 변경할 수 있찌만, 컨테이너를 다시 생성하게 되면 매번 웹 페이지 파일을 전송해야함 ⇒ 영속적으로 웹 페이지 파일을 사용하기 위해서는 특정 디렉터리와 컨테이너 내부의 디렉터리를 연결하는 것이 효과적!

4.2.3 컨테이너 내부 파일 변경하기

도커는 컨테이너 내부에서 컨테이너 외부의 파일을 사용할 수 있는 방법 크게 4가지 제공

- **docker cp**
  - docker cp <호스트 경로> <컨테이너 이름>:<컨테이너 내부 경로>
  - 호스트에 위치한 파일을 구동중인 컨테이너 내부에 복사
  - 임시로 필요한 파일이 있을 때 단편적으로 전송하기 위해 사용
- **Dockerfile ADD**
  - 이미지는 Dockerfile을 기반으로 만들어지는데, Dockerfile에 ADD라는 구문으로 컨테이너 내부로 복사할 파일을 지정하면 이미지 빌드 시 지정한 파일이 이미지 내부로 복사 됨
  - But, 사용자가 원하는 파일을 선택해 사용할 수 없다는 약점
- **바인드 마운트**
  - 호스트의 파일 시스템과 컨테이너 내부를 연결해 어느 한쪽에서 작업한 내용이 양쪽에 동시에 반영되는 방법
  - 새로운 컨테이너를 구동할 때도 호스트와 연결할 파일이나 디렉터리의 경로만 지정하면 다른 컨테이너에 있는 파일을 새로 생성한 컨테이너와 연결할 수 있음

    - 컨테이너가 바뀌어도 없어지면 안되는 자료는 이 방법으로 보존
- **볼륨**
    - 호스트의 파일 시스템과 컨테이너 내부를 연결하는 것은 바인드 마운트와 동일
    - But, 얘는 호스트의 특정 디렉터리가 아닌 도커가 관리하는 볼륨을 컨테이너와 연결
    - 볼륨 : 쿠버네티스에서 살펴본 볼륨 구조
    - 도커가 관리하는 볼륨 공간을 NFS와 같은 공유 디렉터리에 생성한다면 다른 호스트 에서도 도커가 관리하는 볼륨을 사용할 수 있음

⏩실습 : 바인드 마운트로 호스트와 컨테이너 연결하기

현재 정상적으로 노출된 nginx 컨테이너 구조 살펴보자

처음 접속할 때 노출되는 페이지 /usr/share/nginx/html/index.html

따라서 우리가 수정해야 하는 파일은 index.html

이러한 경로 설정은 /etc/nginx/nginx.conf에 존재

```
mkdir -p /root/html
```

- 컨테이너 내부에 연결할 /root/html/ 디렉터리를 호그트에 생성

```
docker run -d -p 8081:80 \ -v /root/html:/usr/share/nginx/html --restart always --name nginx-bind-mounts nginx
```

- 컨테이너 구동하며 컨테이너의 디렉터리와 호스트의 디렉터리를 연결한다
- 호스트의 포트 번호는 8081로 지정
- *바인드 마운트의 특성 : 호스트 디렉터리의 내용을 그대로 컨테이너 디렉터리에 덮어씀
  ⇒ 컨테이너 디렉터리에 어떠한 내용이 있어도 해당 내용은 삭제됨

```
docker ps -f name=nginx-bind-mounts
```

- 컨테이너를 조회해 STATUS가 정상(Up n minutes)인지 확인

```
ls /root/html
```

- 컨테이너 내부와 연결된 /root/html/ 디렉터리를 확인
- 이 디렉터리는 사용자가 호스트에 생성한 빈 디렉터리인데, 조회하면 여전히 비어있음
- 빈 디렉터리가 컨테이너와 연결됐으므로 현재 컨테이너의 nginx는 초기화면으로 보여줄 파일 X

```
웹 브라우저에 192.168.1.10:8081 연결
```

- nginx의 기본 화면이 아닌 오류 화면이 보임
- 사용자가 nginx에 접속하면 index.html를 읽어 화면에 표시하도록 설계되어 있음
- 따라서 바인드 마운트 설정에 따라 호스트 디렉터리의 /root/html에 있는 index.html을 노출하려 하지만 없으므로 오류화면 출력

```
cp ~/_Bokkt_k8sInfra/ch4/4.2.3/index-BindMount.html /root/html/index.html
ls /root/html
```

- 호스트 리렉터리와 컨테이너 디렉터리를 연겨할 때 사용할 index.html을 복사한다.

```
웹 브라우저에 192.168.1.10:8081 연결
```

- index.html이 표시되는 지 확인

⏩실습 : 볼륨으로 호스트와 컨테이너 연결하기

```
docker volume create nginx-volume
```

- 볼륨을 생성
- nginx-volume은 생성할 볼륨의 이름

```
docker volume inspect nginx-volume
```

- 생성된 볼륨 조회
- Mountpoint 디렉터리(/var/lib/docker/volumes/nginx-volume/_data)가 볼륨 디렉터리임을 확인 할 수 있음

```
ls /var/lib/docker/volumes/nginx-volume/_data
```

- 디렉터리가 비어있음

```
docker run -d -v nginx-volume:/usr/share/nginx/html \ -p 8082:80 --restart always --name nginx-volume nginx
```

- 호스트와 컨테이너의 디렉터리를 연결할 컨테이너를 구동
- 기존 컨테이너는 설정을 바꿀 수 없으므로 새로운 컨테이너를 구동

```
ls /var/lib/docker/volumes/nginx-volume/_data
cp ~/_Book_k8sInfra/ch4/4.2.3/index-Volume.html /var/lib/docker/volumes/nginx-volume/_data/index.html
```

4.2.4 사용하지 않는 컨테이너 정리하기

⏩ 실습: 컨테이너 정지하기

```
docker ps -f ancestor=nginx                       # 생성된 컨테이너 조회
docker stop tender_snyder                         # 컨테이너 정지
docker stop f530                                  # ID로 정지
docker ps -q -f ancestor=nginx                    # 컨테이너 ID 출력
docker stop $(docker ps -q -f ancestor=nginx)     # nginx 사용하는 모든 컨테이너 정지
docker ps -f ancestor=nginx                       # 모든 컨테이너 정지됐는지 확인
docker ps -a -f ancestor=nginx                    # 모든 컨테이너 목록 조회
```

⏩ 실습 : 컨테이너와 이미지 삭제하기

```
docker rm $(docker ps -aq -f ancestor=nginx)      # 컨테이너 삭제
docker ps -a -f ancestor=nginx                    # 삭제 확인
docker rmi $(docker images -q nginx)              # 이미지 삭제
```

4.3 4가지 방법으로 컨테이너 이미지 만들기

🚩목적

- 소스 코드로 자바 실행 파일을 빌드하고
  이를 다시 도커 빌드를 사용해 컨테이너 이미지로 만들자!

- 컨테이너 이미지 생성 방법
  기본적인 빌드 → 용량 줄이기 → 컨테이너 내부 빌드 → 멀티 스테이지

4.3.1 기본 방법으로 빌드하기

- 스프링 부트를 이용해 만든 자바 소스 코드로 이미지를 빌드 하자
- 기본적인 컨테이너 빌드 과정
  자바 소스 빌드 → 도커파일 작성 → 도커파일 빌드 → 빌드 완료

⏩ 실습 : 기본 방법으로 빌드하기

```
cd ~/_Book_k8sInfra/ch4/4.3.1
ls
```

- 기본적인 컨테이너 빌드 도구와 파일이 있는 빌드 디렉터리로 이동해 파일 확인

```
yum install java-1.8.0-openjdk-devel -y
```

- 자바 개발 도구 JDK를 설치

```
chmod 700 mvnw
./mvnw clean package
ls target
```

- 자바 빌드 시 메이븐을 사용하므로 실행

```
docker build -t basic-img .                          # 컨테이너 이미지 빌드
docker images basic-img                              # 생성된 이미지 확인
docker build -t basic-img:1.0 -t basic-img:2.0 .     # 1.0, 2.0 태그의 이미지 생성
docker images basic-img                              # 생성된 이미지 확인
sed -i 's/Application/Development/' Docker file      # Dockerfile 일부 변경
docker build -t basic-img:3.0 .                      # 다시 빌드
docker images basic-img                              # 생성된 이미지 확인
docker run -d -p 60431:80 --name basic-run --restart always basic img    # 컨테이너 실행
docker ps -f name=basic-run                          # 컨테이너 상태 출력
curl 127.0.0.1:60431                                 # 응답 오는지 확인
docker rm -f basic-run                               # 컨테이너 삭제
docker rmi -f $(docker images -q basic-img)          # 빌드한 컨테이너 이미지 모두 삭제
docker build -t basic-img .                          # 컨테이너 이미지 다시 빌드 (다음 절 비교용)
docker images basic-img                              # 용량 확인
```

4.3.2 컨테이너 용량 줄이기

- 컨테이너 용량을 줄여 빌드하는 과정
  도커파일 작성 → 도커파일 빌드 → 빌드 완료

⏩ 실습 : 컨테이너 용량 줄여 빌드하기

```
cd ~/_Book_k8sInfra/ch4/4.3.2
ls
```

- 컨테이너 용량을 줄여 빌드하는 과정을 담고있는 디렉터리로 이동해 어떤 파일이 있는지 살펴보자.

```
cat build-in-host.sh                                    # 추가한 파일 확인
cat Dockerfile                                          # Dockerfile 변경 확인
chmod 700 mvnw                                          # 메이븐에 실행권한 부여
./build-in-host.sh                                      # 경량화 이미지 빌드
docker images | head -n 3                               # 앞절의 이미지와 용량 비교
docker run -d -p 60432:80 --name optimal-run --restart always optimal-img  # 컨테이너 작동 확인
docker rm -f optimal-run                                # 빌드한 컨테이너 삭제
```

4.3.3 컨테이너 내부에서 컨테이너 빌드하기

- 컨테이너 내부에서 컨테이너를 빌드하는 과정
  도커파일 작성 → 도커파일 빌드 → 빌드 완료

```
cd ~/_Book_k8sInfra/ch4/4.3.3/
ls
```

- openjdk 이미지에서 자바 소스를 빌드하는 내용이 있는 디렉터리로 이동해 어떤 파일이 있는지 살펴보자.
- Dockerfile하나만 있음. 빌드 과정 자체를 openjdk 이미지 에서 진행하므로 나머지는 필요없음

```
cat Dockerfile
```

- 특별한 내용 없음
- 이미지 내부에 소스코드를 내려받으려고 깃을 사용함
- 내려받은 소스코드를 이미지 내부에서 실행하기 위해 RUN을 추가
- 이미지 내부에서 파일의 위치만 옮기면 되므로 COPY가 아닌 mv를 사용함

```
git clone https://github.com/iac-source/inbuilder.git
```

- 이미지 빌드전, 이미지 내부에 내려받은 inbuilder 저장소의 구조 확인

```
docker build -t nohost-img .              # 컨테이너 이미지 빌드
docker images | head -n 4                 # 기존 이미지들과 비교 (용량)
docker run -d -p 60433:80 --name nohost-run --restart always nohost-img  # 컨테이너 실행
curl 127.0.0.1:60433                      # 컨테이너 작동 확인
docker rm -f nohost-run                   # 컨테이너 삭제
```

- openjdk 이미지를 기초 이미지로 컨테이너 내부에서 자바 소스를 빌드한 결과, 가장 큰 컨테이너 이미지를 얻었음
- 따라서 openjdk로 컨테이너 내부에서 컨테이너 빌드? 좋지 않은 방법
- But, Dockerfile 하나만 빌드하면 컨테이너가 바로 생서오디는 편리함을 포기할 수 없음 → 뒷 장

4.3.2 최적화해 컨테이너 빌드하기

- 멀티 스테이지 빌드 (Multi-Stage Build) 방법
  - 최종 이미지의 용량을 줄일 수 있고 호스트에 어떠한 빌드 도구도 설치할 필요가 없음
  - 멀티 스테이지를 이용한 컨테이너 빌드 과정
    도커파일 작성 → 도커파일 빌드 → 빌드 완료

⏩ 실습 : 멀티 스테이지 빌드

```
kubectl get nodes -o wide              # 현재 사용하는 도커 버전 확인
cd C:\HashCorp\_Book_k8sInfra-main\ch3\3.1.3         # 디렉터리 이동
vagrant destroy -f                                   # 기존 VM 제거
cd C:\HashCorp\_Book_k8sInfra-main\ch3\4.3.4\k8s-SingleMaster-18.9_9_w_auto-compl # 디렉터리 이동
vagrant up                  # 멀티 스테이지 지원하는 도커 포함 쿠버네티스 클러스터 환경 구성
```

```
슈퍼푸티로 m-k8s 노드 접속
```

```
kubectl get nodes -o wide             # 도커 버전 확인
cd ~/_Book_k8sInfra/ch4/4.3.4/
ls                                    # Dockerfile 확인
cat Dockerfile                        # 멀티 스테이지 방식으로 작성된 Dockerfile
docker build -t multistage-img .      # 컨테이너 이미지 빌드
docker images | head -n -3            # optimal-img와 용량 같음
docker rmi $(docker images -f dangling=true -q)   # 댕글링 이미지 삭제
docker run -d -p 60434:80 --name multistage-run --restart always multistage-img  # 컨테이너 실행
curl 127.0.0.1:60434                  # 컨테이너 작동 확인
```

```
docker rm -f multistage-run           # 컨테이너 삭제
cd ~                                  # 홈 디렉터리 이동
```

*댕글링 이미지 : 이름이 없는 이미지