Docker란 무엇인가?

Docker는 애플리케이션을 개발, 패키징 및 실행하기 위한 오픈 플랫폼.
Docker를 사용하면 애플리케이션을 인프라에서 분리하여 소프트웨어를 빠르게 제공할 수 있다.
Docker를 사용하면 애플리케이션을 관리하는 것과 같은 방식으로 인프라를 관리할 수 있다.
Docker의 코드 패키징, 테스트 및 배포 방법론을 활용하면 코드를 작성하고 프로덕션에서 실행하는 사이의 지연 시간을 크게 줄일 수 있다.

Docker 플랫폼

Docker는 컨테이너라고 하는 격리된 환경에서 애플리케이션을 패키징하고 실행할 수 있다.
격리 및 보안을 통해 주어진 호스트에서 여러 컨테이너를 동시에 실행할 수 있다.
컨테이너는 가볍고 애플리케이션을 실행하는 데 필요한 모든 것을 포함하므로 호스트에 설치된 것에 의존할 필요가 없다.
작업하는 동안 컨테이너를 공유할 수 있으며, 공유하는 모든 사람이 동일한 방식으로 작동하는 동일한 컨테이너를 받도록 할 수 있다.

Docker는 컨테이너의 수명 주기를 관리하기 위한 도구와 플랫폼을 제공

- 컨테이너를 사용하여 애플리케이션과 지원 구성요소를 개발.
- 컨테이너는 애플리케이션을 배포하고 테스트하는 단위가 된다.
- 준비가 되면 애플리케이션을 컨테이너 또는 오케스트레이션된 서비스로 프로덕션 환경에 배포한다. 프로덕션 환경이 로컬 데이터 센터, 클라우드 공급자 또는 두 가지의 하이브리드이든 동일하게 작동

Docker 기능

1. 빠르고 일관된 애플리케이션 제공

Docker는 개발자가 애플리케이션과 서비스를 제공하는 로컬 컨테이너를 사용하여 표준화된 환경에서 작업할 수 있도록 하여 개발 라이프사이클을 간소화한다. 컨테이너는 지속적인 통합 및 지속적인 배포(CI/CD) 워크플로에 적합

다음의 예제 시나리오를 생각해 보자.

- 개발자는 Docker 컨테이너를 사용하여 로컬에서 코드를 작성하고 동료와 작업을 공유
- 그들은 Docker를 사용하여 애플리케이션을 테스트 환경으로 푸시하고 자동 및 수동 테스트를 실행
- 개발자가 버그를 발견하면 개발 환경에서 버그를 수정한 후 테스트 및 검증을 위해 테스트 환경에 다시 배포할 수 있다.
- 테스트가 완료되면 업데이트된 이미지를 프로덕션 환경에 푸시하는 것만큼 간단하게 고객에게 수정 사항을 전달할 수 있다.

2. 반응형 배포 및 확장

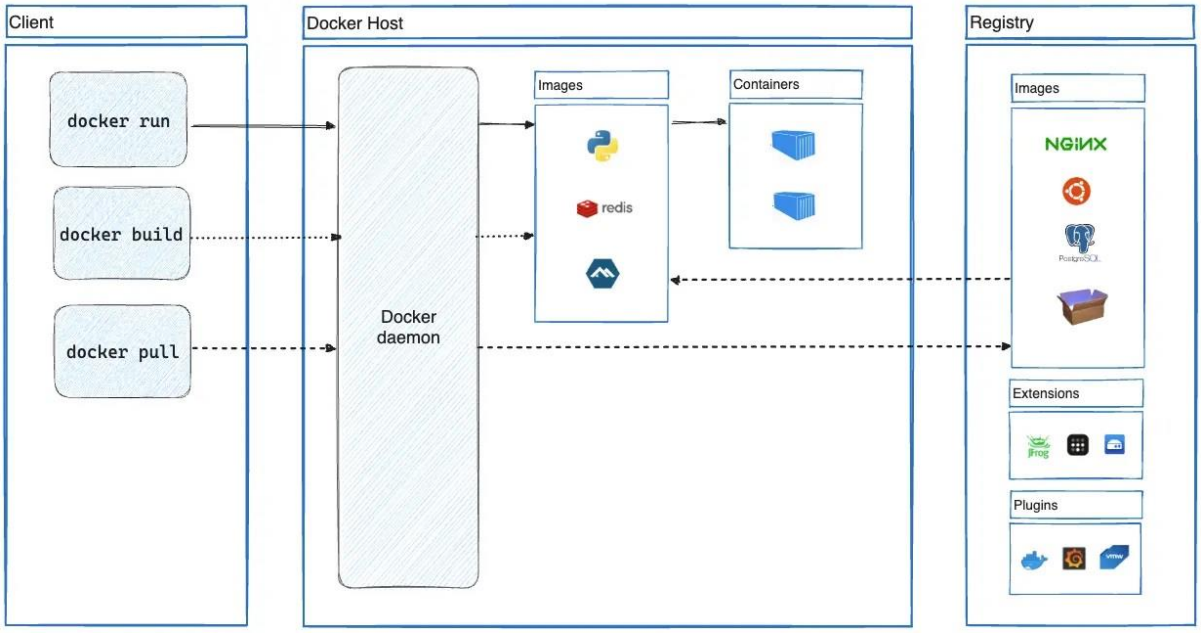
Docker의 컨테이너 기반 플랫폼은 매우 이동성이 뛰어난 워크로드를 제공한다. Docker 컨테이너는 개발자의 로컬 노트북, 데이터 센터의 물리적 또는 가상 머신, 클라우드 공급자 또는 혼합된 환경에서 실행할 수 있다.
Docker의 이동성과 가벼운 특성 덕분에 작업 부하를 동적으로 관리하고, 비즈니스 요구에 따라 거의 실시간으로 애플리케이션과 서비스를 확장하거나 해체하는 것이 쉽다.

3. 동일한 하드웨어에서 더 많은 작업 실행

Docker는 가볍고 빠르다. 하이퍼바이저 기반 가상 머신에서 비용 효율적인 실행 방법을 제공하므로 서버 용량을 더 많이 사용하여 비즈니스 목표를 달성할 수 있다. Docker는 고밀도 환경과 더 적은 리소스로 더 많은 작업을 수행해야 하는 소규모 및 중규모 배포에 적합하다.

도커 아키텍처

Docker는 클라이언트-서버 아키텍처를 사용한다. Docker 클라이언트는 Docker 데몬과 통신하며, 이 데몬은 Docker 컨테이너를 빌드, 실행 및 배포하는 작업을 수행. Docker 클라이언트와 데몬은 동일한 시스템에서 실행될 수도 있고 Docker 클라이언트를 원격 Docker 데몬에 연결할 수도 있다. Docker 클라이언트와 데몬은 REST API, UNIX 소켓 또는 네트워크 인터페이스를 사용하여 통신한다. 또 다른 Docker 클라이언트는 Docker Compose로, 컨테이너 세트로 구성된 애플리케이션으로 작업할 수 있다.



- Docker 데몬

Docker 데몬(dockerd)은 Docker API 요청을 수신하고 이미지, 컨테이너, 네트워크, 볼륨과 같은 Docker 객체를 관리. 데몬은 다른 데몬과 통신하여 Docker 서비스를 관리할 수도 있다.

- Docker 클라이언트

Docker 클라이언트(docker)는 많은 Docker 사용자가 Docker와 상호 작용하는 주요 방법. [docker run]와 같은 명령을 사용하여 도킹된 이미지 실행할 수 있다.

- 도커 데스크톱

Docker Desktop은 컨테이너화된 애플리케이션과 마이크로서비스를 빌드하고 공유할 수 있는 Mac, Windows 또는 Linux 환경을 위한 설치하기 쉬운 애플리케이션. Docker Desktop에는 Docker 데몬(dockerd), Docker 클라이언트(docker), Docker Compose, Docker Content Trust, Kubernetes 및 Credential Helper가 포함되어 있다.

- Docker 레지스트리

Docker 레지스트리는 Docker 이미지를 저장. Docker Hub는 누구나 사용할 수 있는 공개 레지스트리이며 Docker는 기본적으로 Docker Hub에서 이미지를 찾는다. 개인 레지스트리를 직접 실행할 수도 있다.
docker pull 명령을 사용하면 레지스트리에서 필요한 이미지를 가져온다. docker push 명령을 사용하면 Docker는 레지스트리에 이미지를 푸시한다.

- Docker 객체

Docker에서 사용되는 이미지, 컨테이너, 네트워크, 볼륨, 플러그인 및 기타 객체를 말한다.

- 이미지

이미지는 Docker 컨테이너를 만드는 지침이 포함된 읽기 전용 템플릿이다. 다른 이미지를 기반으로 하며 정의를 더 추가하는 형태로 만들 수도 있다. 예를 들어, ubuntu 이미지를 기반으로 한 Apache 웹 서버와 애플리케이션, 그리고 애플리케이션을 실행하는 데 필요한 구성 세부 정보를 설치하는 이미지를 빌드할 수 있다.
자신의 이미지를 만들거나 다른 사람이 만들어 레지스트리에 게시한 이미지를 사용할 수도 있다. 자신의 이미지를 빌드하려면 이미지를 만들고 실행하는 데 필요한 단계를 정의하는 간단한 구문이 있는 Dockerfile을 만든다. Dockerfile의 각 명령어는 이미지에 레이어를 만든다. Dockerfile을 변경하고 이미지를 다시 빌드하면 변경된 레이어만 다시 빌드된다. 이것이 다른 가상화 기술과 비교했을 때 이미지를 매우 가볍고 작고 빠르게 만드는 요인 중 하나다.

- 컨테이너

컨테이너는 이미지의 실행 가능한 인스턴스. Docker API 또는 CLI를 사용하여 컨테이너를 생성, 시작, 중지, 이동 또는 삭제할 수 있다. 컨테이너를 하나 이상의 네트워크에 연결하거나, 스토리지를 연결하거나, 현재 상태를 기반으로 새 이미지를 생성할 수도 있다.
기본적으로 컨테이너는 다른 컨테이너와 호스트 머신에서 비교적 잘 격리된다. 컨테이너의 네트워크, 스토리지 또는 기타 기본 하위 시스템이 다른 컨테이너나 호스트 머신에서 얼마나 격리되는지 제어할 수 있다.
컨테이너는 이미지와 컨테이너를 만들거나 시작할 때 제공하는 모든 구성 옵션으로 정의된다. 컨테이너가 제거되면 영구 저장소에 저장되지 않은 상태의 모든 변경 사항이 사라진다.

1. 컨테이너

개발 앱에 필요한 언어 API나 데이터베이스 등은 팀원들이 모두 같은 버전으로 설치해야 통합 시 문제가 발생하지 않는다. 이를 관리하기 위한 시간과 작업이 필요한데 이를 해결할 수 있는 쉬운 방법 중 하나가 컨테이너다.

컨테이너란 무엇인가?
컨테이너는 앱의 각 구성 요소에 대한 격리된 프로세스다. 각 구성 요소는 자체 격리된 환경에서 실행되며, 머신의 다른 모든 것과 완전히 격리된다.

컨테이너 특징

- 독립형
  각 컨테이너는 호스트 머신에 미리 설치된 종속성에 의존하지 않고도 작동하는 데 필요한 모든 것을 갖추고 있다.
- 격리됨
  컨테이너는 격리되어 실행되므로 호스트와 다른 컨테이너에 미치는 영향이 최소화되어 애플리케이션의 보안이 강화된다.
- 독립적
  각 컨테이너는 독립적으로 관리된다. 컨테이너 하나를 삭제해도 다른 컨테이너에는 영향을 미치지 않는다.
- 휴대 가능
  컨테이너는 어디에서나 실행될 수 있다. 개발 머신에서 실행되는 컨테이너는 데이터 센터나 클라우드의 어디에서나 같은 방식으로 작동한다.

컨테이너 대 가상 머신(VM)
VM은 자체 커널, 하드웨어 드라이버, 프로그램 및 애플리케이션을 갖춘 전체 운영 체제다. 단일 애플리케이션만 분리하기 위해 VM을 사용하는 것은 오버헤드가 크다.

컨테이너는 실행에 필요한 모든 파일이 있는 고립된 프로세스일 뿐이다. 여러 컨테이너를 실행하면 모두 동일한 커널을 공유하여 더 적은 인프라에서 더 많은 애플리케이션을 실행할 수 있다.

VM과 컨테이너를 함께 사용
컨테이너와 VM이 함께 사용되는 것을 자주 볼 수 있다. 예를 들어, 클라우드 환경에서 프로비저닝된 머신은 일반적으로 VM입니다. 그러나 하나의 애플리케이션을 실행하기 위해 하나의 머신을 프로비저닝하는 대신 컨테이너 런타임이 있는 VM은 여러 컨테이너화된 애플리케이션을 실행하여 리소스 활용도를 높이고 비용을 절감할 수 있다.

실습: CLI 사용

다음은 도커를 설치하면 기본 제공되는 컨테이너 welcome-to-docker를 실행하는 명령이다.

docker run -d -p 8080:80 docker/welcome-to-docker

현재 실행중인 컨테이너 목록을 보려면 다음 명령을 실행한다.

```
docker ps
```

다음과 같은 출력이 표시된다.

```
CONTAINER ID   IMAGE                    COMMAND                  CREATED
STATUS          PORTS                    NAMES
a1f7a4bb3a27   docker/welcome-to-docker   "/docker-entrypoint...."   11 seconds ago   Up 11
seconds   0.0.0.0:8080->80/tcp          gracious_keldysh
```

이 명령은 실행 중인 컨테이너 만 보여준다. 중지된 컨테이너를 보려면 모든 컨테이너를 나열하는 플래그를 추가한다.

```
docker ps -a
```

컨테이너 중지
컨테이너 docker/welcome-to-docker는 중지할 때까지 계속 실행된다.
docker stop 명령을 사용하여 컨테이너를 중지할 수 있다.

1. docker ps 명령으로 컨테이너의 ID 확인
2. docker stop <the-container-id>

2. 이미지

컨테이너를 고립된 프로세스로 보면 , 파일과 구성은 어디서 가져오나요? 어떻게 그 환경을 공유하나요?

컨테이너 이미지는 컨테이너를 실행하는 데 필요한 모든 파일, 바이너리, 라이브러리 및 구성을 포함하는 표준화된 패키지다.

PostgreSQL 이미지의 경우 해당 이미지는 데이터베이스 바이너리, 구성 파일 및 기타 종속성을 패키징한다. Python 웹 앱의 경우 Python 런타임, 앱 코드 및 모든 종속성을 포함한다.

이미지에는 두 가지 중요한 원칙이 있다.

1. 이미지는 변경할 수 없다. 이미지가 생성되면 수정할 수 없다. 새 이미지를 만들거나 그 위에 변경 사항을 추가할 수만 있다.
2. 컨테이너 이미지는 레이어로 구성된다. 각 레이어는 파일을 추가, 제거 또는 수정하는 파일 시스템 변경 세트를 나타낸다.

이 두 가지 원칙을 사용하면 기존 이미지를 확장하거나 추가할 수 있다. 예를 들어 Python 앱을 빌드하는 경우 Python 이미지에서 시작 하여 추가 레이어를 추가하여 앱의 종속성을 설치하고 코드를 추가할 수 있다. 이렇게 하면 Python 자체가 아닌 앱에 집중할 수 있다.

이미지 찾기

Docker Hub는 이미지를 저장하고 배포하기 위한 기본 글로벌 마켓플레이스.
개발자가 만든 100,000개 이상의 이미지가 있으며 로컬에서 실행할 수 있다. Docker Hub 이미지를 검색하여 Docker Desktop에서 직접 실행할 수 있다.
Docker Hub는 Docker Trusted Content라고 알려진 다양한 Docker 지원 및 보증 이미지를 제공한다. 허브에는 다음 것들이 포함된다.

- Docker 공식 이미지 - Docker 저장소의 큐레이션된 세트로, 대부분 사용자의 시작점 역할을 하며 Docker Hub에서 가장 안전한 이미지 중 하나
- Docker Verified Publishers - Docker에서 검증한 상업 게시자의 고품질 이미지
- Docker가 후원하는 오픈 소스 - Docker의 오픈 소스 프로그램을 통해 Docker가 후원하는 오픈 소스 프로젝트에서 게시 및 유지 관리하는 이미지

이미지 검색 및 다운로드

도커 허브에 올려있는 이미지를 검색할 수 있다.
명령어: docker search

```
docker search docker/welcome-to-docker
```

실행결과

```
NAME                                DESCRIPTION                                     STARS
OFFICIAL
docker/welcome-to-docker            Docker image for new users getting started w...  20
```

이 출력은 Docker Hub에서 사용할 수 있는 관련 이미지에 대한 정보를 보여준다.

1. [docker pull] 명령으로 이미지 다운로드
2. docker pull docker/welcome-to-docker

실행결과

Using default tag: latest
latest: Pulling from docker/welcome-to-docker
579b34f0a95b: Download complete
d11a451e6399: Download complete
1c2214f9937c: Download complete
b42a2f288f4d: Download complete
54b19e12c655: Download complete
1fb28e078240: Download complete
94be7e780731: Download complete
89578ce72c35: Download complete
Digest: sha256:eedaff45e3c78538087bdd9dc7afafac7e110061bbdd836af4104b10f10ab693
Status: Downloaded newer image for docker/welcome-to-docker:latest
docker.io/docker/welcome-to-docker:latest

각 줄은 이미지의 다른 다운로드된 레이어를 나타낸다. 각 레이어는 파일 시스템 변경 사항의 집합이며 이미지의 기능을 제공한다.

이미지 목록 확인

```
docker image ls
```

실행결과
REPOSITORY TAG IMAGE ID CREATED SIZE
docker/welcome-to-docker latest eedaff45e3c7 4 months ago 29.7MB

이 명령은 현재 시스템에서 사용할 수 있는 Docker 이미지 목록을 보여준다.

이미지 크기
여기에 표시된 이미지 크기는 레이어의 다운로드 크기가 아닌, 압축되지 않은 이미지 크기를 나타낸다.
다음 명령을 사용하여 이미지의 레이어를 나열한다.

```
docker image history docker/welcome-to-docker
```

실행결과
IMAGE CREATED CREATED BY SIZE COMMENT
648f93a1ba7d 4 months ago COPY /app/build /usr/share/nginx/html # buil... 1.6MB buildkit.dockerfile.v0

```
<missing>      5 months ago    /bin/sh -c #(nop)   CMD ["nginx" "-g" "daemon...     0B
<missing>      5 months ago    /bin/sh -c #(nop)   STOPSIGNAL SIGQUIT                0B
<missing>      5 months ago    /bin/sh -c #(nop)   EXPOSE 80                         0B
<missing>      5 months ago    /bin/sh -c #(nop)   ENTRYPOINT ["/docker-entr...     0B
<missing>      5 months ago    /bin/sh -c #(nop) COPY file:9e3b2b63db9f8fc7...     4.62kB
<missing>      5 months ago    /bin/sh -c #(nop) COPY file:57846632accc8975...     3.02kB
<missing>      5 months ago    /bin/sh -c #(nop) COPY file:3b1b9915b7dd898a...     298B
<missing>      5 months ago    /bin/sh -c #(nop) COPY file:caec368f5a54f70a...     2.12kB
<missing>      5 months ago    /bin/sh -c #(nop) COPY file:01e75c6dd0ce317d...     1.62kB
<missing>      5 months ago    /bin/sh -c set -x        && addgroup -g 101 -S ...     9.7MB
<missing>      5 months ago    /bin/sh -c #(nop)   ENV PKG_RELEASE=1                 0B
<missing>      5 months ago    /bin/sh -c #(nop)   ENV NGINX_VERSION=1.25.3          0B
<missing>      5 months ago    /bin/sh -c #(nop)   LABEL maintainer=NGINX Do...     0B
<missing>      5 months ago    /bin/sh -c #(nop)   CMD ["/bin/sh"]                  0B
<missing>      5 months ago    /bin/sh -c #(nop) ADD file:ff3112828967e8004...     7.66MB
```

이 출력에서는 모든 레이어, 크기, 레이어를 만드는데 사용된 명령을 보여준다.

전체 명령 보기
명령에 [--no-trunc] 플래그를 추가하면 전체 명령이 표시된다.

3. 레지스트리(Registry)

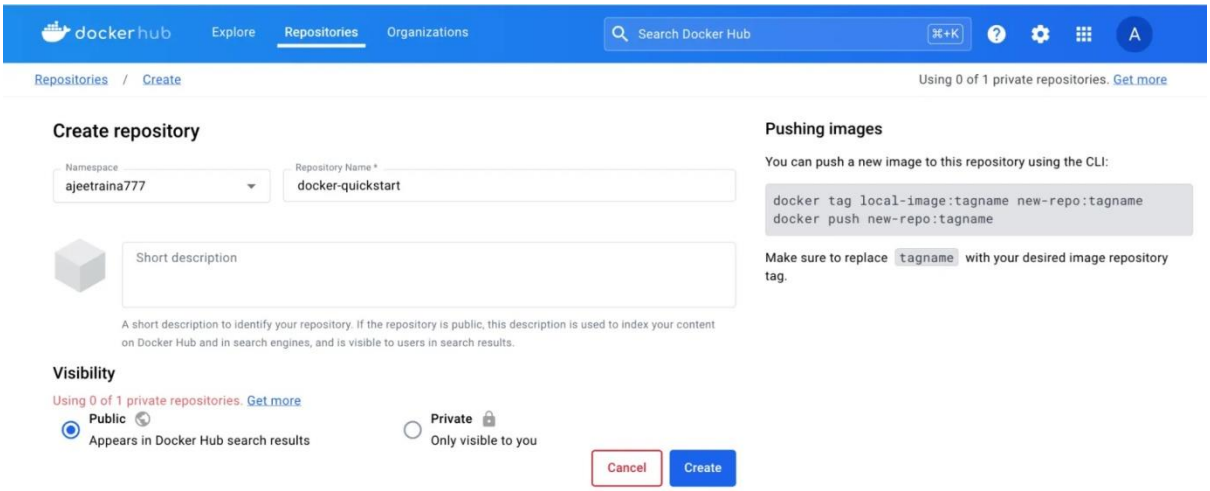
컨테이너 이미지를 컴퓨터 시스템에 저장할 수 있지만, 팀원들과 공유하거나 다른 컴퓨터에서 사용하고 싶다면 이미지 레지스트리가 필요하다.
이미지 레지스트리는 컨테이너 이미지를 저장하고 공유하기 위한 중앙 집중 영역이다. 공개 또는 비공개일 수 있고 Docker Hub 는 누구나 사용할 수 있는 공개 그리고 기본 레지스트리다.

Docker Hub가 인기 있는 옵션이지만, 오늘날 Amazon Elastic Container Registry(ECR) , Azure Container Registry(ACR) , Google Container Registry(GCR) 등 다른 많은 컨테이너 레지스트리를 사용할 수 있다 . 로컬 시스템이나 조직 내부에서 개인 레지스트리를 실행할 수도 있다. 예를 들어 Harbor, JFrog Artifactory, GitLab Container Registry 등이 있다.

레지스트리(Registry) 대 리포지토리(Repository)
레지스트리와 리포지토리는 같은 의미일까?
서로 관련이 있지만 완전히 같은 것은 아니다.
레지스트리는 컨테이너 이미지를 저장하고 관리하는 영역인 반면, 리포지토리는 레지스트리 내의 관련 컨테이너 이미지 모음이다. 프로젝트를 기준으로 이미지를 구성하는 폴더라고 생각하면 된다. 각 리포지토리에는 하나 이상의 컨테이너 이미지가 들어 있다.

Docker 이미지를 빌드하고 Docker Hub 저장소에 푸시해보자.

1. 무료 Docker 계정에 가입
2. Docker Hub 에 로그인
3. 오른쪽 상단에 있는 **저장소 만들기** 버튼을 선택
4. 네임스페이스(대부분 사용자 이름)를 선택하고 저장소 이름을 입력



5. **공개 여부를 공개** 로 설정
6. 저장소를 만들려면 **'만들기'** 버튼을 선택

생성된 저장소는 비어 있다. 이미지를 푸시해보자.

1. 푸시할 이미지 빌드
   docker build -t *USERNAME*/app_name .

   주의
   명령어 끝에 점(.)을 포함해야 한다 [docker build .] 이것은 Docker에게 Dockerfile을 어디에서 찾아야 하는지 알려준다.
2. 새로 생성된 Docker 이미지 확인
   docker images

   실행결과

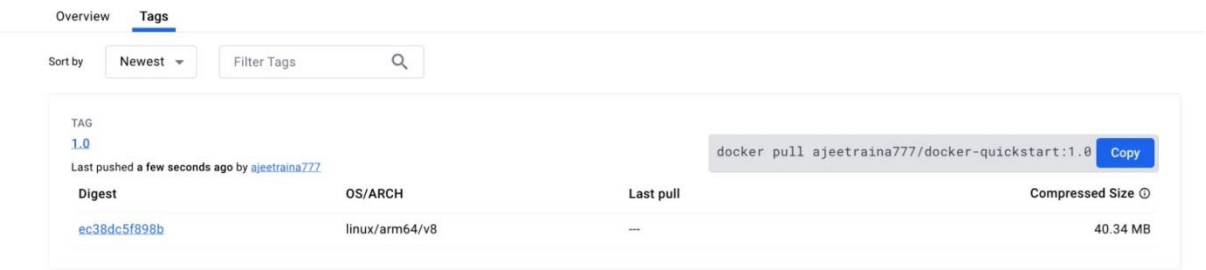
REPOSITORY TAG IMAGE ID CREATED
SIZE
USERNAME/app_name latest 476de364f70e 2 minutes ago 170MB

3. 생성한 이미지 실행하여 테스트함
   docker run -d -p 8080:8080 *USERNAME/*app_name

4. [docker tag] 명령을 사용하여 Docker 이미지에 태그를 지정할 수 있다. Docker 태그를 사용하면 이미지에 레이블을 지정하고 버전을 지정할 수 있다.
   docker tag *USERNAME/* app_name *YOUR USERNAME/* app_name:1.0

5. 마지막으로 [docker push] 명령을 사용하여 새로 빌드한 이미지를 Docker Hub 저장소에 푸시한다 .
   docker push *USERNAME/* app_name:1.0

6. Docker Hub를 열고 저장소로 이동. **태그** 섹션으로 이동하여 새로 푸시된 이미지를 확인한다.



4. 컴포즈(Docker Compose)

app 컨테이너, db 컨테이너 등 여러 컨테이너를 연결하여 실행할 수 있다. 이 경우 네트워크를 관리해야 하고, 컨테이너를 해당 네트워크에 연결하는 데 필요한 모든 플래그 등을 관리해야 한다.

Docker Compose를 사용하면 모든 컨테이너와 해당 구성을 단일 YAML 파일에 정의할 수 있다. 이 파일을 코드 리포지토리에 포함하면 리포지토리를 복제하는 모든 사람이 단일 명령으로 시작하고 실행할 수 있다.

Compose는 선언적 도구라서 정의하고 실행하면 된다. 항상 모든 것을 처음부터 다시 만들 필요가 없다. 변경한 경우 [docker compose up] 명령을 다시 실행하면 Compose가 파일의 변경 사항을 조정하고 적용한다.

Dockerfile과 Compose 파일
Dockerfile은 컨테이너 이미지를 빌드하는 지침을 제공하는 반면 Compose 파일은 실행 중인 컨

테이너를 정의한다. Compose 파일은 특정 서비스에 사용할 이미지를 빌드하기 위해 Dockerfile을 참조하는 경우가 많다.

실습

Docker Compose를 사용하여 다중 컨테이너 애플리케이션을 실행
다음은 Node.js와 MySQL을 데이터베이스 서버로 사용하여 빌드하는 컴포즈 설정 파일의 예다

```
services:
app:
image: node:18-alpine
command: sh -c "yarn install && yarn run dev"
ports:
- 127.0.0.1:3000:3000
working_dir: /app
volumes:
- ./:/app
environment:
MYSQL_HOST: mysql
MYSQL_USER: root
MYSQL_PASSWORD: secret
MYSQL_DB: todos

mysql:
image: mysql:8.0
volumes:
- todo-mysql-data:/var/lib/mysql
environment:
MYSQL_ROOT_PASSWORD: secret
MYSQL_DATABASE: todos

volumes:
todo-mysql-data:
```

애플리케이션을 구성하는 모든 서비스와 해당 구성을 정의. 각 서비스는 이미지, 포트, 볼륨, 네트워크 및 기능에 필요한 기타 설정을 지정한다. 이제 컴포즈를 실행한다.

```
docker compose up -d --build
```

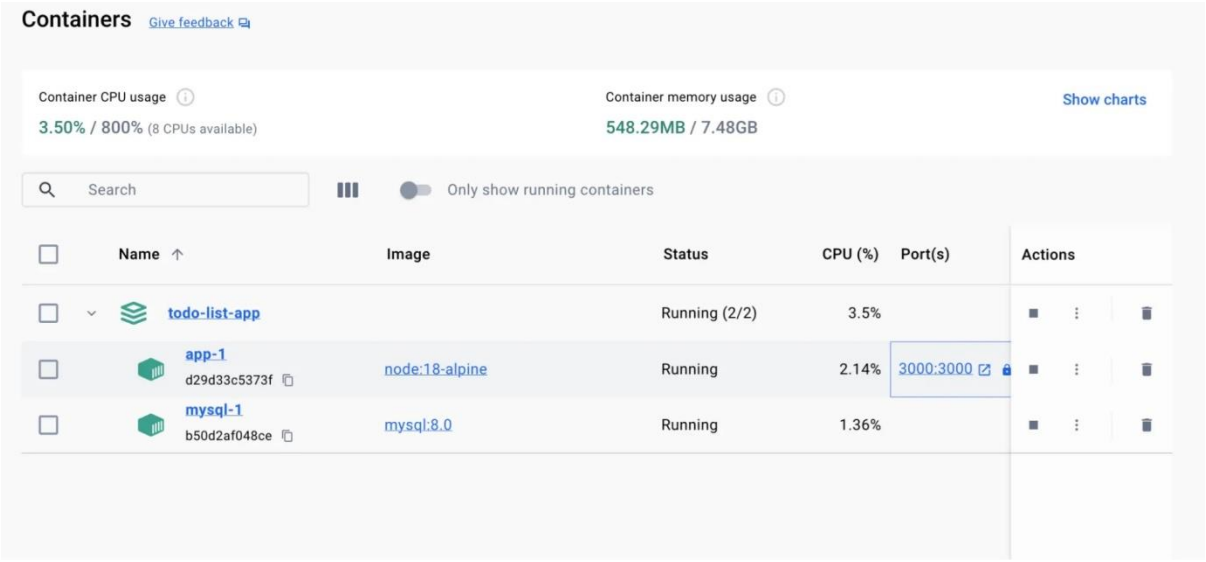
실행결과

[+] Running 4/4
✔ app 3 layers [⣿⣿⣿] 0B/0B Pulled 7.1s
✔ e6f4e57cc59e Download complete 0.9s
✔ df998480d81d Download complete 1.0s
✔ 31e174fedd23 Download complete 2.5s
[+] Running 2/4
⠇ Network todo-list-app_default Created 0.3s
⠇ Volume "todo-list-app_todo-mysql-data" Created 0.3s
✔ Container todo-list-app-app-1 Started 0.3s
✔ Container todo-list-app-mysql-1 Started 0.3s

다음은 컴포즈 과정에서 작업한 내용이다.

- Docker Hub에서 두 개의 컨테이너 이미지(node 및 MySQL)가 다운로드
- 애플리케이션을 위한 네트워크 생성
- 데이터베이스 파일을 유지하기 위해 볼륨이 생성됨
- 두 개의 컨테이너가 모든 필수 구성으로 시작됨

Docker Desktop GUI 컨테이너 구성을 확인할 수 있다.



컴포즈 삭제

이 애플리케이션은 Docker Compose를 사용하여 시작되었으므로 작업이 끝나면 쉽게 해체할 수 있다.

```
docker compose down
```

실행결과

[+] Running 2/2

✔ Container todo-list-app-mysql-1 Removed 2.9s

✔ Container todo-list-app-app-1 Removed 0.1s

✔ Network todo-list-app_default Removed 0.1s

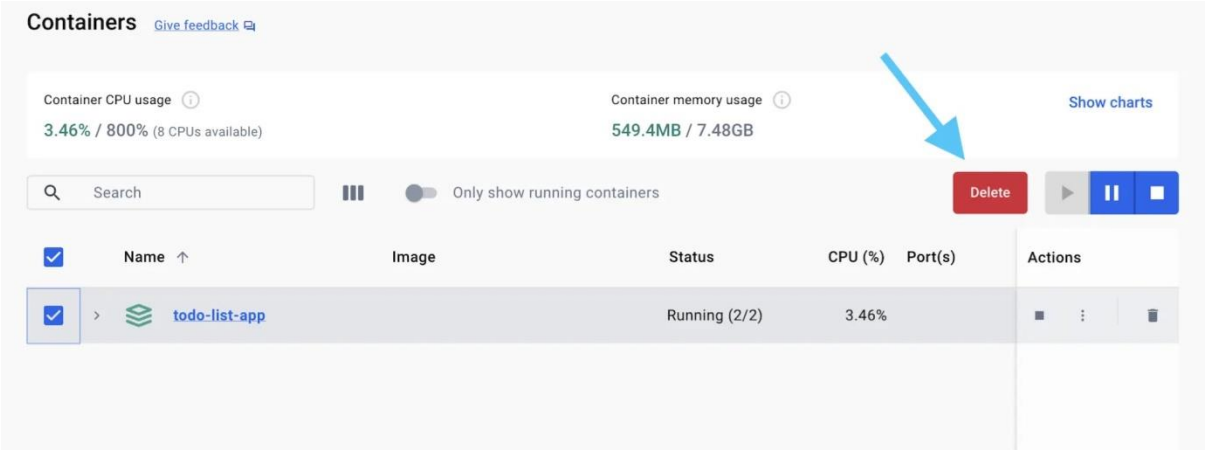
볼륨 지속성

기본적으로 Compose 스택을 해체할 때 볼륨은 자동으로 제거되지 않는다. 스택을 다시 시작했을 때 데이터가 다시 필요할 수 있기 때문이다.

볼륨 제거하려면 다음 명령을 사용한다.

```
docker compose down --volumes
```

또는 Docker Desktop GUI를 사용하여 애플리케이션 스택을 선택하고 **삭제** 버튼을 선택하여 컨테이너를 제거할 수 있다.



Compose 스택에 GUI 사용

GUI에서 Compose 앱의 컨테이너를 제거하면 컨테이너만 제거됩니다. 그렇게 하려면 네트워크와 볼륨을 수동으로 제거해야 한다.

5. 이미지 레이어

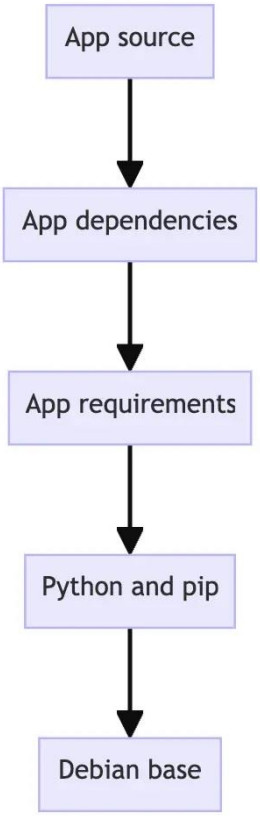
컨테이너 이미지는 레이어로 구성된다. 그리고 이러한 각 레이어는 일단 생성되면 변경할 수 없다.

이미지 레이어

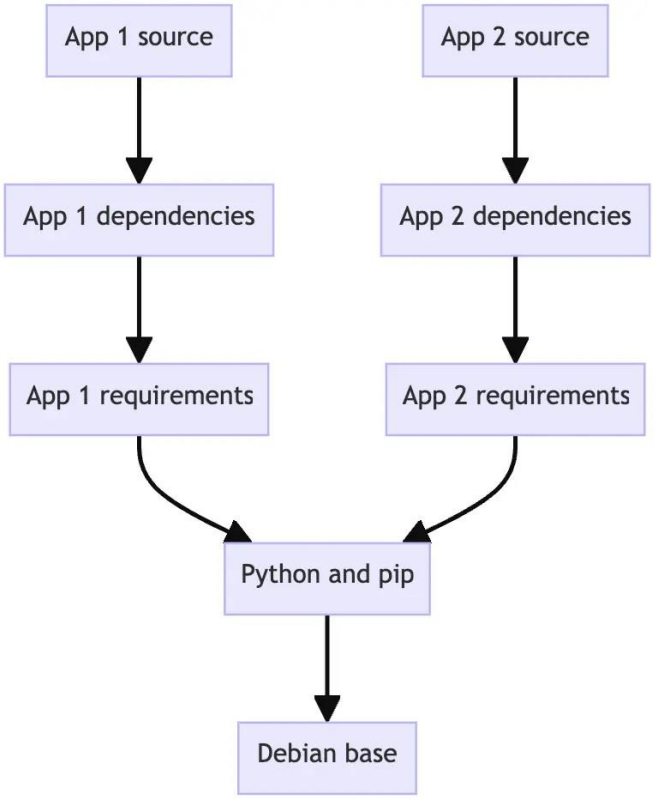
이미지의 각 레이어에는 파일 시스템 변경 사항(추가, 삭제 또는 수정) 세트가 포함되어 있다. 이

론적 이미지를 살펴보면 다음과 같다.

1. 첫 번째 레이어는 기본 명령과 apt와 같은 패키지 관리자를 추가
2. 두 번째 레이어에서는 종속성 관리를 위한 Python 런타임과 pip를 설치
3. 세 번째 레이어는 애플리케이션의 특정 requirements.txt 파일을 복사
4. 네 번째 계층은 해당 애플리케이션의 특정 종속성을 설치
5. 다섯 번째 레이어는 애플리케이션의 실제 소스 코드를 복사



이는 레이어를 이미지 간에 재사용할 수 있기 때문에 유용하다. 예를 들어, 다른 Python 애플리케이션을 만들고 싶다면 레이어링 덕분에 동일한 Python 기반을 활용할 수 있다. 이렇게 하면 빌드가 더 빨라지고 이미지를 배포하는 데 필요한 저장 공간과 대역폭이 줄어든다. 이미지 레이어링은 다음과 같다.



레이어를 사용하면 다른 레이어의 기본 레이어를 재사용하여 이미지를 확장하고, 애플리케이션에 필요한 데이터만 추가할 수 있다.

레이어링
레이어링은 콘텐츠 주소 지정 스토리지와 유니온 파일 시스템을 통해 가능하다.
작동 방식은 다음과 같다.

1. 각 레이어가 다운로드된 후에는 호스트 파일 시스템의 자체 디렉토리에 압축이 풀린다.
2. 이미지에서 컨테이너를 실행하면 각 레이어가 서로 쌓여서 새롭고 통합된 뷰가 생성되는 유니온 파일 시스템이 생성된다.
3. 컨테이너가 시작되면 루트 디렉토리가 통합 디렉토리로 설정된다.

유니온 파일 시스템이 생성되면 이미지 레이어 외에도 실행 중인 컨테이너를 위해 디렉토리가 생성된다. 이를 통해 컨테이너는 파일 시스템을 변경하는 동시에 원래 이미지 레이어는 그대로 유지할 수 있다. 이를 통해 동일한 기본 이미지에서 여러 컨테이너를 실행할 수 있다.

실습
[docker container commit] 명령을 사용하여 수동으로 새 이미지 레이어를 만든다. 보통 Dockerfile을 사용하므로 이런 방식으로 이미지를 만드는 일은 거의 없을 것이다. 하지만 모든 것이 어떻게 작동하는지 이해하기가 더 쉽다.

기본 이미지 생성

```
$ docker run --name=base-container -ti ubuntu
```

이미지가 다운로드되고 컨테이너가 시작되면 새 셸 프롬프트가 표시된다. 이것은 컨테이너 내부에서 실행 중이다. 다음과 비슷하게 표시된다(컨테이너 ID는 다를 수 있음):

root@d8c5ca119fcd:/#

컨테이너 내부에서 다음 명령으로 Node.js를 설치

```
$ apt update && apt install -y nodejs
```

이 명령을 실행하면 컨테이너 내부에 Node를 다운로드하고 설치한다. 유니온 파일 시스템의 맥락에서 이러한 파일 시스템 변경은 이 컨테이너에 고유한 디렉토리내에서 발생한다.

다음 명령을 실행하여 Node가 설치되었는지 확인

```
$ node -e 'console.log("Hello world!")'
```

그러면 콘솔에 "Hello world!"가 출력된다.

이제 Node가 설치되었으므로 변경 사항을 새 이미지 레이어로 저장할 준비가 되었다. 여기서 새 컨테이너를 시작하거나 새 이미지를 빌드할 수 있다. 이를 위해 [docker container commit] 명령을 사용한다.

```
$ docker container commit -m "Add node" base-container node-base
```

이미지 레이어 확인: [docker image history]

```
$ docker image history node-base
```

실행결과

```
IMAGE          CREATED         CREATED BY                                      SIZE
COMMENT
d5c1fca2cdc4   10 seconds ago  /bin/bash                                       126MB
Add node
2b7cc08dcdbb   5 weeks ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B
<missing>      5 weeks ago     /bin/sh -c #(nop) ADD file:07cdbabf782942af0...  69.2MB
<missing>      5 weeks ago     /bin/sh -c #(nop)  LABEL org.opencontainers....  0B
<missing>      5 weeks ago     /bin/sh -c #(nop)  LABEL org.opencontainers....  0B
<missing>      5 weeks ago     /bin/sh -c #(nop)  ARG LAUNCHPAD_BUILD_ARCH      0B
<missing>      5 weeks ago     /bin/sh -c #(nop)  ARG RELEASE                   0B
```

맨 위 줄에 있는 " Add node" 주석을 확인할 수 있다. 이 레이어에는 방금 만든 Node.js 설치가

들어 있다.

이미지에 Node가 설치되었는지 확인하려면 이 새 이미지를 사용하여 새 컨테이너를 시작할 수 있다.

```
$ docker run node-base node -e "console.log('Hello again')"
```

그러면 터미널에 "Hello again"이라는 출력이 표시되어 Node가 설치되고 작동 중임을 알 수 있다.

이제 기본 이미지 생성이 끝났으므로 해당 컨테이너를 제거할 수 있다.

```
$ docker rm -f base-container
```

기본 이미지 정의

기본 이미지는 다른 이미지를 빌드하기 위한 기초. 모든 이미지를 기본 이미지로 사용할 수 있다. node-base는 아무 동작도 하지 않지만 다른 빌드에 사용할 수 있다.

앱 이미지 구축

이제 기본 이미지가 있으므로 해당 이미지를 확장하여 추가 이미지를 빌드할 수 있다.
새로 생성된 노드 기반 이미지를 사용하여 새 컨테이너를 시작

```
$ docker run --name=app-container -ti node-base
```

이 컨테이너 내부에서 다음 명령을 실행하여 Node 프로그램을 만든다.

```
$ echo 'console.log("Hello from an app")' > app.js
```

app.js 실행

```
$ node app.js
```

이 컨테이너의 변경 사항을 새 이미지로 저장

```
$ docker container commit -c "CMD node app.js" -m "Add app" app-container sample-app
```

이 명령은 <sample-app>이라는 이름의 새 이미지를 생성할 뿐만 아니라 컨테이너를 시작할 때 기본 명령을 설정하기 위해 이미지에 명령어 [node app.js]를 추가했다.

다음 명령으로 업데이트된 레이어를 확인

```
$ docker image history sample-app
```

실행결과

```
IMAGE               CREATED                  CREATED BY
SIZE        COMMENT
```

```
c1502e2ec875     About a minute ago     /bin/bash                                          33B
Add app
5310da79c50a     4 minutes ago          /bin/bash
126MB        Add node
2b7cc08dcdbb     5 weeks ago            /bin/sh -c #(nop)   CMD ["/bin/bash"]              0B
<missing>        5 weeks ago            /bin/sh -c #(nop) ADD file:07cdbabf782942af0...
69.2MB
<missing>        5 weeks ago            /bin/sh -c #(nop)   LABEL org.opencontainers....    0B
<missing>        5 weeks ago            /bin/sh -c #(nop)   LABEL org.opencontainers....    0B
<missing>        5 weeks ago            /bin/sh -c #(nop)   ARG LAUNCHPAD_BUILD_ARCH
0B
<missing>        5 weeks ago            /bin/sh -c #(nop)   ARG RELEASE                    0B
```

맨 위 레이어 주석에는 " Add app "가 있고 다음 레이어에는 " Add node "가 있다.

마지막으로, 완전히 새로운 이미지를 사용하여 새 컨테이너를 시작. 기본 명령을 지정했으므로 다음 명령을 사용할 수 있다.

```
$ docker run sample-app
```

Node 프로그램에서 나오는 인사말이 터미널에 표시되는 것을 볼 수 있다.

이제 컨테이너 작업이 끝났으므로 다음 명령을 사용하여 컨테이너를 제거할 수 있다.
$ docker rm -f app-container

6. Dockerfile 작성

Dockerfile은 컨테이너 이미지를 만드는 데 사용되는 텍스트 기반 문서. 실행할 명령, 복사할 파일, 시작 명령 등 이미지 빌더에 대한 지침을 제공한다.
다음 Dockerfile은 실행 가능한 Python 애플리케이션을 생성하는 예

```
FROM python:3.12
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
```

```
COPY src ./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Dockerfile 작성법

- FROM <image>- 빌드가 확장할 기본 이미지를 지정
- WORKDIR <path>- 파일이 복사되고 명령이 실행되는 이미지의 "작업 디렉토리" 경로 지정
- COPY <host-path> <image-path>- 빌더에게 호스트에서 파일을 복사하여 컨테이너 이미지에 넣으라고 지시
- RUN <command>- 빌더에게 지정된 명령을 실행하라고 지시
- ENV <name> <value>- 실행 중인 컨테이너가 사용할 환경 변수를 설정
- EXPOSE <port-number>- 이미지가 노출하고자 하는 포트를 이미지에 설정
- USER <user-or-uid>- 이후의 모든 명령어에 대한 기본 사용자를 설정
- CMD ["<command>", "<arg1>"]- 이 이미지를 사용하는 컨테이너가 실행할 기본 명령을 설정

실습

이전 예에서 보았듯이 Dockerfile은 일반적으로 다음 단계를 따른다.

1. 기본 이미지 결정
2. 애플리케이션 종속성 설치
3. 관련 소스 코드 및/또는 바이너리를 복사
4. 최종 이미지 구성

이 간단한 예에서는 Node.js 애플리케이션을 빌드하는 Dockerfile을 작성한다.

Dockerfile 생성

```
# 기본 이미지 정의
FROM node:20-alpine

#작업 디렉토리 정의
```

```
#향후 명령이 실행되는 위치와 디렉토리 파일이 컨테이너 이미지 내부에 복사되는 위치가 지정됨
WORKDIR /app

#프로젝트에 있는 모든 파일을 컨테이너 이미지로 복사
COPY . .

#패키지 관리자 yarn을 사용하여 앱의 종속성을 설치
RUN yarn install --production

#실행할 기본 명령을 지정
CMD ["node", "./src/index.js"]
```

7. 이미지 빌드, 태그지정, 배포

- 이미지 구축 - Dockerfile 기반으로 이미지를 구축하는 프로세스
- 이미지 태그 지정 - 이미지에 이름을 지정하는 프로세스로, 이를 통해 이미지를 어디에 배포할 수 있는지도 결정된다.
- 이미지 배포 - 컨테이너 레지스트리를 사용하여 새로 생성된 이미지를 배포하거나 공유하는 프로세스
-

이미지 빌드

이미지는 Dockerfile을 사용하여 빌드되는 경우가 가장 많다.

빌드 명령> docker build .

명령의 마지막 .은 빌드 컨텍스트에 대한 경로를 의미한다. 이 위치에서 빌더는 Dockerfile 및 기타 참조 파일을 찾는다. 빌드를 실행하면 빌더가 필요한 경우 기본 이미지를 가져온 다음 Dockerfile에 지정된 절차를 실행한다.

[docker build .] 명령을 사용하면 이미지에 이름 없이 빌드되지만 결과물은 이미지의 ID를 제공한다. 예를 들어 위 명령은 다음과 같은 출력을 생성할 수 있다.

```
$ docker build .
[+] Building 3.5s (11/11) FINISHED
docker:desktop-linux
=> [internal] load build definition from Dockerfile   0.0s
=> => transferring dockerfile: 308B       0.0s
=> [internal] load metadata for docker.io/library/python:3.12     0.0s
```

=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [1/6] FROM docker.io/library/python:3.12 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 123B 0.0s
=> [2/6] WORKDIR /usr/local/app 0.0s
=> [3/6] RUN useradd app 0.1s
=> [4/6] COPY ./requirements.txt ./requirements.txt 0.0s
=> [5/6] RUN pip install --no-cache-dir --upgrade -r requirements.txt 3.2s
=> [6/6] COPY ./app ./app 0.0s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image
sha256:9924dfd9350407b3df01d1a0e1033b1e543523ce7d5d5e2c83a724480ebe8f00 0.0s

이미지 이름은 없지만 id로 컨테이너를 시작할 수 있다.
docker run sha256:9924dfd9350407b3df01d1a0e1033b1e543523ce7d5d5e2c83a724480ebe8f00

하지만 이렇게 긴 id는 기억하기 힘들다. 태그가 유용한 이유가 바로 여기에 있다.

이미지 태그 지정
이미지에 태그를 지정하는 것은 이미지에 기억하기 쉬운 이름을 제공하는 방법이다. 그러나 이미지 이름에는 구조가 있다. 전체 이미지 이름은 다음과 같은 구조를 갖는다.

[HOST[:PORT_NUMBER]/]PATH[:TAG]

- HOST: 이미지가 있는 위치의 레지스트리 호스트 이름. 선택적임. 호스트가 지정되지 않으면 Docker 공개 레지스트리인 docker.io가 기본적으로 사용된다.
- PORT_NUMBER: 호스트 이름이 제공된 경우 레지스트리 포트 번호
- PATH: 이미지 경로. Docker Hub의 경우 형식은 다음과 같다
  [NAMESPACE/]REPOSITORY. 여기서 namespace는 사용자 또는 조직의 이름.
  namespace가 지정되지 않으면 Docker 공식 네임스페이스인 library가 사용된다.
- TAG: 일반적으로 이미지의 다른 버전이나 변형을 식별하는 데 사용되는 사용자 지정 식별자. 태그가 지정되지 않으면 latest기본적으로 사용된다.

이미지 이름 예>

- docker.io/library/nginx:latest
  docker.io는 레지스트리
  library는 네임스페이스
  nginx는 이미지 리포지토리

latest는 태그

- docker.io/docker/welcome-to-docker:latest
  docker.io는 레지스트리
  docker는 네임스페이스
  welcome-to-docker는 이미지 리포지토리
  latest는 태그
- ghcr.io/dockersamples/example-voting-app-vote:pr-311:
  ghcr.io는 레지스트리. GitHub Container Registry
  dockersamples 는 네임스페이스
  example-voting-app-vote는 이미지 리포지토리
  pr-311는 태그

이미지 빌드 시 태그 지정

```
docker build -t my-username/my-image .
```

이미 만든 이미지에 태그 지정

```
docker image tag my-username/my-image another-username/another-image:v1
```

이미지 배포

이미지를 빌드하고 태그를 지정했으면 이제 레지스트리에 올리자.

```
docker push my-username/my-image
```

이 명령을 실행하면 이미지의 모든 레이어가 레지스트리에 푸시된다.
만약 인증이 필요하다는 메시지가 뜬다면 [docker login]을 실행한다.

실습

작업한 소스를 준비하고 Dockerfile을 작성한다. 그리고 다음 명령으로 이미지를 빌드한다.

```
$ docker build -t USERNAME/concepts-build-image-demo .
```

빌드가 완료되면 다음 명령으로 이미지 목록 확인.

```
$ docker image ls
```

실행결과

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mobywhale/concepts-build-image-demo	latest	746c7e06537f	24 seconds ago	354MB

[docker image history] 명령을 사용하면 실제로 이미지가 생성된 방식을 볼 수 있다 .

```
$ docker image history mobywhale/concepts-build-image-demo
```

실행결과

```
IMAGE          CREATED             CREATED BY                                      SIZE      COMMENT
f279389d5f01   8 seconds ago       CMD ["node" "./src/index.js"]                   0B        buildkit.dockerfile.v0
<missing>      8 seconds ago       EXPOSE map[3000/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      8 seconds ago       WORKDIR /app                                    8.19kB    buildkit.dockerfile.v0
<missing>      4 days ago          /bin/sh -c #(nop)  CMD ["node"]                 0B
<missing>      4 days ago          /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry…   0B
<missing>      4 days ago          /bin/sh -c #(nop) COPY file:4d192565a7220e13…   20.5kB
<missing>      4 days ago          /bin/sh -c apk add --no-cache --virtual .bui…   7.92MB
<missing>      4 days ago          /bin/sh -c #(nop)  ENV YARN_VERSION=1.22.19     0B
<missing>      4 days ago          /bin/sh -c addgroup -g 1000 node        && addu…   126MB
<missing>      4 days ago          /bin/sh -c #(nop)  ENV NODE_VERSION=20.12.0     0B
<missing>      2 months ago        /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      2 months ago        /bin/sh -c #(nop) ADD file:d0764a717d1e9d0af…   8.42MB
```

이 결과는 이미지의 레이어를 보여주며, 추가한 레이어와 기본 이미지에서 상속받은 레이어도 보여준다.

이제 이미지를 배포한다

```
$ docker push USERNAME/concepts-build-image-demo
```

8. 멀티스테이지 빌드

기존 빌드에서는 모든 빌드 명령이 순서대로 실행되고 단일 빌드 컨테이너에서 종속성 다운로드, 코드 컴파일, 애플리케이션 패키징이 이루어진다. 모든 레이어가 최종 이미지로 끝난다. 이 접근 방식은 효과가 있지만 불필요하게 부피가 큰 이미지가 생성되고 보안 위험이 증가한다. 그래서 멀티스테이지 빌드가 필요하다.

멀티스테이지 빌드는 Dockerfile에 각기 다른 목적을 갖는 여러 단계로 작성한다. 이는 여러 다른 환경에서 동시에 서로 다른 빌드 부분을 실행할 수 있는 기능을 제공하고, 빌드 환경을 최종 런타임 환경과 분리해서 이미지 크기를 줄일 수 있습니다. 이는 특히 빌드 종속성이 큰 애플리케이션에 유용하다.

모든 유형의 애플리케이션에는 다단계 빌드가 권장된다.

- JavaScript, Ruby 또는 Python과 같은 인터프리트 언어의 경우, 한 단계에서 코드를 빌드하고 축소하고, 제품화에 적합한 파일을 더 작은 런타임 이미지에 복사할 수 있다. 이렇게

하면 배포를 위해 이미지가 최적화된다.

- C, Go 또는 Rust와 같은 컴파일된 언어의 경우, 멀티스테이지 빌드를 사용하면 한 스테이지에서 컴파일하고 컴파일된 바이너리를 최종 런타임 이미지에 복사할 수 있다. 최종 이미지에 전체 컴파일러를 번들할 필요가 없다.

실습

스프링 앱을 이미지로 빌드해보자

1) 단일 스테이지 Dockerfile

```
# 기본 이미지 정의
FROM eclipse-temurin:21.0.2_13-jdk-jammy

# 작업 디렉토리를 정의.
# 명령이 실행되는 위치와 디렉토리 파일이 컨테이너 이미지 내부에 복사되는 위치
WORKDIR /app

# Maven wrapper script 파일 복사
COPY .mvn/ .mvn

# pom.xml 파일을 현재 작업디렉토리(/app)에 복사
COPY mvnw pom.xml ./

# 컨테이너에서 [./mvnw dependency:go-offline] 명령 실행
# Maven wrapper가 프로젝트 종속성을 다운로드.
RUN ./mvnw dependency:go-offline

# src 디렉토리를 작업 디렉토리 아래에 복사
COPY src ./src

# 컨테이너가 시작될 때 실행할 명령어 정의
# 스프링 프로젝트의 런타임 설정을 기반으로 Maven wrapper 실행
CMD ["./mvnw", "spring-boot:run"]
```

Docker 이미지를 빌드

```
docker build -t spring-helloworld .
```

Docker 이미지의 크기 확인

```
docker images
```

실행결과

```
REPOSITORY              TAG       IMAGE ID       CREATED              SIZE
spring-helloworld       latest    ff708d5ee194   3 minutes ago        880MB
```

이미지 크기가 880MB임을 확인할 수 있다. 여기에는 전체 JDK, Maven 툴체인 등이 포함되어 있다. 제품화에서는 최종 이미지에 필요하지 않다.

앱 실행

이제 이미지가 빌드되었으니 컨테이너를 실행할 차례입니다.

```
docker run -d -p 8080:8080 spring-helloworld
```

실행결과

```
[INFO] --- spring-boot:3.3.4:run (default-cli) @ spring-boot-docker ---
[INFO] Attaching agents: []

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.3.4)

2024-09-29T23:54:07.157Z  INFO 159 --- [spring-boot-docker] [           main] c.e.s.SpringBootDockerApplication     : Starting SpringBootDockerApplication using Java 21.0.2 with PID 159 (/app/target/classes started by root in /app)
 ....
```

웹 브라우저 http://localhost:8080 로 실행할 수 있다

2) 멀티 스테이지 Dockerfile

Dockerfile이 두 단계로 분할됨

```
FROM eclipse-temurin:21.0.2_13-jdk-jammy AS builder
WORKDIR /opt/app
COPY .mvn/ .mvn
COPY mvnw pom.xml ./
RUN ./mvnw dependency:go-offline
COPY ./src ./src
```

```
RUN ./mvnw clean install

FROM eclipse-temurin:21.0.2_13-jre-jammy AS final
WORKDIR /opt/app
EXPOSE 8080
COPY --from=builder /opt/app/target/*.jar /opt/app/*.jar
ENTRYPOINT ["java", "-jar", "/opt/app/*.jar"]
```

첫 번째 단계는 이전 Dockerfile과 동일하게 유지되며 애플리케이션을 빌드하기 위한 Java Development Kit(JDK) 환경을 제공한다. 이 단계는 builder라는 이름이 지정된다.

두 번째 단계는 final 이라는 이름의 새로운 단계로 애플리케이션을 실행하는 데 필요한 Java Runtime Environment(JRE)만 포함하는 더 얇은 eclipse-temurin:21.0.2_13-jre-jammy 이미지를 사용한다. 이 이미지는 컴파일된 애플리케이션(JAR 파일)을 실행하기에 충분한 Java Runtime Environment(JRE)를 제공한다.

멀티 스테이지 빌드를 사용하면 Docker 빌드는 컴파일, 패키징 및 단위 테스트에 하나의 기본 이미지를 사용하고 애플리케이션 런타임에 별도의 이미지를 사용한다. 결과적으로 최종 이미지는 개발 또는 디버깅 도구를 포함하지 않으므로 크기가 더 작다. 빌드 환경을 최종 런타임 환경과 분리하면 이미지 크기를 크게 줄이고 최종 이미지의 보안을 강화할 수 있다.

이제 이미지를 다시 빌드하고 실행해보자.

```
docker build -t spring-helloworld-builder .
```

이 명령은 현재 디렉토리에 있는 Dockerfile 파일의 최종 단계를 사용하여 spring-helloworld-builder로 명명된 Docker 이미지를 빌드한다.

생성된 이미지 크기 확인

```
docker images
```

실행결과

```
spring-helloworld-builder latest      c5c76cb815c0    24 minutes ago      428MB
spring-helloworld         latest      ff708d5ee194    About an hour ago   880MB
```

최종 이미지의 크기는 880MB였던 원본 빌드 크기에 비해 428MB에 불과하다.

각 단계를 최적화하고 필요한 것만 포함함으로써 동일한 기능을 달성하면서도 전체 이미지 크기를 크게 줄일 수 있었다. 이는 성능을 개선할 뿐만 아니라 Docker 이미지를 더 가볍고, 더 안전하며, 관리하기 쉽게 만든다.

9. 컨테이너 실행

port publish

컨테이너에 고립된 앱을 외부 네트워크에 연결할 수 있도록 포트를 publish한다.

```
docker run -d -p HOST_PORT:CONTAINER_PORT nginx
```

HOST_PORT: 트래픽을 수신하려는 호스트 머신의 포트 번호
CONTAINER_PORT: 연결을 수신하는 컨테이너 내의 포트 번호

예를 들어 머신의 8080포트로 전송된 모든 트래픽이 컨테이너 내의 80포트로 전달되도록 하려면 다음과 같이 명령을 실행한다.

```
docker run -d -p 8080:80 nginx
```

컨테이너 내에서의 포트만 지정하려면 다음과 같이 실행한다.

```
docker run -p 80 nginx
```

ps 명령으로 실행중인 컨테이너 목록을 확인해보자

```
docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                     NAMES
a527355c9c53   nginx     "/docker-entrypoint.â¦"   4 seconds ago    Up 3 seconds    0.0.0.0:54772->80/tcp     romantic_williamson
```

위 결과를 보면 내부 포트는 지정한 80이지만 호스트의 포트는 지정하지 않았기 때문에 임의의 포트 값(54772)이 할당된 것을 볼 수 있다.

컨테이너 이미지를 생성할 때, 이 EXPOSE 명령어는 패키징된 애플리케이션의 포트를 지정한다. 하지만 이러한 포트는 기본적으로 publish되지 않는다. EXPOSE로 지정한 포트를 publish 하려면 실행 시 -P 옵션을 추가한다.

```
docker run -P nginx
```

컴포즈 시 포트를 지정하려면 다음과 같다.

<compose.yaml>

```
services:
  app:
    image: docker/welcome-to-docker
    ports:
```

```
      - 8080:80
```

이 파일에서 포트를 지정하며 [호스트 포트 : 컨테이너 내 포트] 를 의미한다

컨테이너 볼륨

데이터를 지속하기 위해 볼륨을 생성할 수도 있다.

```
docker volume create log-data
```

위 명령으로 컨테이너에 log-data라는 이름의 볼륨이 생성되었다. 이 볼륨이 포함된 컨테이너를 실행해 보자.

```
-v <디스크디렉토리명>:<도커볼륨명>
docker run -d -p 80:80 -v /logs:/log-data docker/welcome-to-docker
```

위 명령을 실행하면 docker/welcome-to-docker 이미지가 실행되고 이 컨테이너에 생성된 볼륨 log-data는 호스트 컴퓨터의 /logs 디렉토리에 마운트된다. 만약 볼륨을 생성하지 않고 이 명령을 실행한다면 자동으로 볼륨을 생성해 준다.

이 컨테이너에서 파일들을 저장한다면 모두 /logs에 저장되고 만약 이 컨테이너를 삭제하고 이 볼륨을 사용하는 새 컨테이너를 실행한다면 /logs에 있는 파일들은 그대로 남아있다.

볼륨 관련 명령어들

- docker volume ls
  볼륨 목록
- docker volume rm <volume-name-or-id>
  볼륨 제거(볼륨이 어떤 컨테이너에도 연결되지 않은 경우에만 작동)
- docker volume prune
  사용되지 않는(연결되지 않은) 모든 볼륨을 제거

다중 컨테이너 실행은 컴포즈로 해결

다중 컨테이너를 개별 실행하여 연결하면 다음과 같은 단점이 발생할 수 있다.

- 애플리케이션은 종종 서로 의존한다. 특정 순서로 컨테이너를 수동으로 시작하고 네트워크 연결을 관리하는 것은 스택이 확장됨에 따라 어려워진다.

- 각 애플리케이션은 docker run명령이 필요하므로 개별 서비스를 확장하기 어렵다. 전체 애플리케이션을 확장한다는 리소스를 낭비할 가능성이 있음을 의미한다.

- 각 애플리케이션에 대한 데이터를 유지하려면 각 docker run 명령 내에서 별도의 볼륨 마운트 구성이 필요하다. 이는 분산된 데이터를 관리하는 비용이 발생할 수 있다.

- 각 애플리케이션에 대한 환경 변수를 별도의 docker run명령 마다 설정하는 것은 지루하고 오류가 발생하기 쉽다.

바로 이러한 점들이 Docker Compose가 요구되는 부분이다.

Docker Compose는 compose.yml이라는 단일 YAML 파일에 전체 멀티 컨테이너 애플리케이션을 정의한다. 이 파일은 모든 컨테이너, 종속성, 환경 변수, 심지어 볼륨과 네트워크에 대한 구성을 지정한다.

Docker Compose 장점
- docker run 명령을 여러 번 실행할 필요가 없다. 단일 YAML 파일에 전체 멀티 컨테이너 애플리케이션을 정의하기만 하면 된다. 이렇게 하면 구성이 중앙 집중화되고 관리가 간소화된다.
- 특정 순서로 컨테이너를 실행하고 네트워크 연결을 쉽게 관리할 수 있다.
- 멀티 컨테이너 설정 내에서 개별 서비스를 간단히 확장하거나 축소할 수 있다. 이를 통해 실시간 요구에 따라 효율적인 할당이 가능하다.
- 손쉽게 영구 볼륨을 구현할 수 있다.
- Docker Compose 파일에서 환경 변수를 한 번만 설정하면 된다.

이처럼 Docker Compose를 사용하여 여러 컨테이너 설정을 실행하면 모듈성, 확장성, 일관성을 핵심으로 복잡한 애플리케이션을 구축할 수 있다.

실습

이미지 구축

```
docker build -t nginx .
docker build -t web .
```

컨테이너 실행

멀티 컨테이너 애플리케이션을 실행하기 전에 모든 애플리케이션이 통신할 수 있는 네트워크 생성

```
docker network create sample-app
```

필요한 컨테이너들 실행

```
docker run -d   --name redis --network sample-app --network-alias redis redis
docker run -d --name web1 -h web1 --network sample-app --network-alias web1 web
docker run -d --name web2 -h web2 --network sample-app --network-alias web2 web
docker run -d --name nginx --network sample-app   -p 80:80 nginx
```

실행중인 컨테이너들 확인

```
docker ps
```

실행결과

```
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS                  NAMES
2cf7c484c144   nginx     "/docker-entrypoint.…"   9 seconds ago        Up 8 seconds        0.0.0.0:80->80/tcp     nginx
7a070c9ffeaa   web       "docker-entrypoint.s…"   19 seconds ago       Up 18 seconds                              web2
6dc6d4e60aaf   web       "docker-entrypoint.s…"   34 seconds ago       Up 33 seconds                              web1
008e0ecf4f36   redis     "docker-entrypoint.s…"   About a minute ago   Up About a minute   6379/tcp               redis
```

위 실습은 여러 컨테이너들을 따로 실행하는 방법을 사용했다. 이제 컴포즈를 사용해 보자. 먼저 compose.yaml 파일을 만들자

```
services:
  redis:
    image: 'redislabs/redismod'
    ports:
      - '6379:6379'
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
  nginx:
    build: ./nginx
    ports:
    - '80:80'
    depends_on:
    - web1
```

```
    - web2
```

컴포즈 실행

```
docker compose up -d --build
```

실행결과

Running 5/5

✔□□ Network nginx-nodejs-redis_default Created 0.0s

✔□□ Container nginx-nodejs-redis-web1-1 Started 0.1s

✔□□ Container nginx-nodejs-redis-redis-1 Started 0.1s

✔ □□ Container nginx-nodejs-redis-web2-1 Started 0.1s

✔□□ Container nginx-nodejs-redis-nginx-1 Started0.1s