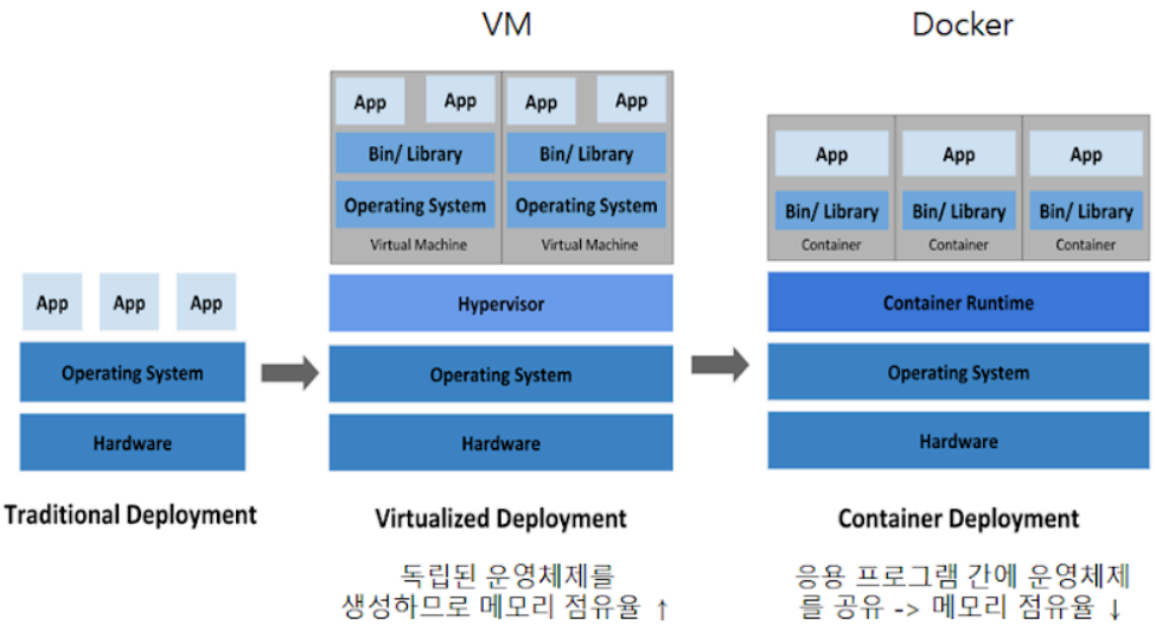
Docker

Docker란

- Container를 활용해 어플리케이션에 필요한 환경을 구축, 배포하게 도와주는 플랫폼
- Virtual Machine처럼 Host서버와 독립된 새로운 환경을 만들어 주는 도구



- 기존 가상환경은 VM에 또 OS를 설치하는 방식 => Docker는 Host OS를 활용
- ubuntu/postgres 같은 이름의 Docker image들이 있는데, 만약 host os가 redhat이면 ubuntu가 설치되는 것 아닌가?
  - 운영체제의 두가지 주요 영역
    - Kernel Space
      - 하드웨어 자원 관리, 시스템 호출 처리
      - 메모리 관리, 프로세스 스케줄링, 디스크및 파일 시스템 관리, 네트워크 처리, 하드웨어 · 소프트웨어 간 중재 등 수행
    - User Space
      - 애플리케이션, 프로세스 실행되는 공간
      - 커널을 통해서만 간접적으로 하드웨어 접근 가능
  - 커널은 host OS것 쓰고, 사용자 공간만 이미지에서 정의한 운영체제에서 동작하는 것 처럼 보임

Docker 사용 이유

- 가상머신보다 경량화 되어 있고, 커널을 공유하여 리소스 사용 최소화
- Docker-hub에 다양한 이미지 존재해서 활용하기 좋음(Harbor로 이 역할 했음)
- 호스트 OS와 무관하게 Docker 설치되어 있다면 빌드 된 이미지를 활용해 빠른 실행 가능
- 컨테이너를 활용한 캡슐화로 환경에 무관하게 서비스 제공가능
- 경량화, 이식성, 확장성의 장점을 가짐.

컨테이너

- Docker가 강점을 갖게 해주는 요인
- 경량화, 이식성, 확장성이라는 장점은 컨테이너 때문이라고 볼 수 있음.
- 격리된 환경을 제공하여 타 애플리케이션, host와의 충돌 방지
- 이미지를 기반으로 실행

이미지

- 컨테이너를 생성하는 기반으로 컨테이너 실행에 필요한 정보를 하나로 묶은 패키지
- Layer로 구성되어 변경사항 발생시 변경사항만 추가로 생성되어 저장 공간, 빌드 속도에 장점을 갖음

Docker 엔진

- Docker Daemon(dockerd)
  - Docker의 백그라운드 프로세스, 컨테이너와이미지 생성, 관리
  - 사용자 명령 수신 및 작업
  - Rest API를 통해 Docker CLI 및 외부 어플리케이션과 통신
- Docker CLI
  - 사용자와 Docker Daemon간의 인터페이스 제공
  - 명령어를 통해 이미지 빌드 및 컨테이너 관리

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

- Docker API
  - Docker Daemon이 다른 어플리케이션과 통신하기 위한 REST API
  - 도커 제어시 사용(Python, Go, Docker CLI 등으로)

Docker 작업 흐름

- 컨테이너생성 및 실행
  1. 사용자의 명령어(CLI)로 컨테이너실행 명령
  2. 로컬에 이미지 존재 확인 후 없을 경우 Docker Hub(or Harbor같은 사설 저장소)에서 이미지 pull
  3. 컨테이너 생성 및 서비스 실행
- 이미지 빌드
  1. Dockerfile 작성
  2. 파일 내 명령 실행

    - Docker Daemon이 Dockerfile 명령어 단계별로 실행해 레이어 생성
3. 이미지 저장
    - 빌드된 이미지는 저장되어 재사용 가능

Docker File

- Docker 이미지 생성을 위한 스크립트 파일로, 이미지 빌드 과정을 단계별로 정의한 파일
- cf) docker commit <컨테이너 ID> <이미지 이름>:<태그>
    - Base image로 컨테이너 실행 후, 하나하나 세팅한 뒤 docker commit으로 이미지 생성도 가능
    - 하지만 수작업으로 인한 오류 가능성, 불필요한 레이어 발생 가능성 등으로 인해 DockerFile을 이용하는 것을 더 추천

```
# Node.js LTS 버전 사용
FROM node:16

# 작업 디렉토리 설정
WORKDIR /usr/src/app

# package.json 파일 복사
COPY package*.json ./

# 종속성 설치
RUN npm install

# 애플리케이션 파일 복사
COPY . .

# 애플리케이션 실행
CMD ["node", "server.js"]
```

Volume

- Volume을 설정하지 않는다면 container 삭제시, 컨테이너 내부의 데이터가 전부 삭제됨
- 필요한 데이터는 남아 있을 수 있도록 Host 서버에 데이터를 저장
- 혹은 여러 컨테이너에서 같은 경로를 바라보면 데이터를 공유해야할 경우에도 활용 가능
- 기존의 경로를 Mount하여 사용할 수도있고, Docker가 관리하는 Volume을 생성해서 사용도 가능

Docker compose

- Docker 컨테이너를 정의하고, 여러 어플리케이션을 관리하기 위한 도구
- YAML파일에 컨테이너의 설정 정의를 통해 컨테이너 생성, 종료 가능
- ex) Web, WAS, DB로 구성될 서비스가 있는 경우 3개의 컨테이너를 하나의 YAML파일을 통해 관리

```
version: "3.9"  # Docker Compose 버전
```

```yaml
services:
  # FastAPI 애플리케이션 서비스
  fastapi:
    build:
      context: ./fastapi  # FastAPI Dockerfile 경로
    container_name: fastapi_app
    ports:
      - "8000:8000"  # 외부에서 FastAPI에 접근할 수 있도록 포트 연결
    networks:
      - app_network
    depends_on:
      - db  # FastAPI가 MySQL 서비스 실행 이후 시작
    environment:
      - DATABASE_URL=mysql+pymysql://root:example@db:3306/mydb

  # Nginx 프록시 서비스
  nginx:
    image: nginx:latest
    container_name: nginx_proxy
    ports:
      - "80:80"  # Nginx 기본 포트
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro  # Nginx 설정 파일 연결
    networks:
      - app_network
    depends_on:
      - fastapi  # FastAPI 서비스 실행 이후 Nginx 시작

  # MySQL 데이터베이스 서비스
  db:
    image: mysql:5.7
    container_name: mysql_db
    ports:
      - "3306:3306"  # MySQL 기본 포트
    environment:
      MYSQL_ROOT_PASSWORD: example  # MySQL 루트 비밀번호
      MYSQL_DATABASE: mydb  # 초기화할 데이터베이스 이름
      MYSQL_USER: user  # MySQL 사용자 이름
      MYSQL_PASSWORD: password  # MySQL 사용자 비밀번호
    volumes:
      - db_data:/var/lib/mysql  # MySQL 데이터 저장

networks:
  app_network:  # 모든 서비스가 공유하는 네트워크
    driver: bridge

volumes:
  db_data:  # MySQL 데이터 저장을 위한 볼륨
```

추천 실습 환경

- 윈도우니까 wsl2로 ubuntu 실행해서 docker 설치해서 해보는게 어떨까...
- 아니면 라즈베리파이 하나 밀고 제공?