# Docker 가상화

## 배포 환경의 가상화

[AI/6] 1반 김도엽

# 목차

- Docker가 없던 시절의 배포 방법, 그리고 문제점
- Hypervisor 방식의 배포 방법
- Docker 란?
- Docker container 방식의 배포의 장점
- 한번 배포해봅시다

# Docker 없는 배포

# Docker가 없던 시절의 배포

- 운영체제에 어플리케이션을 실행할 환경을 직접 구축해야합니다.
- 예시를 보여드리겠습니다

# 예전에 배포해봤던 경험....

환경 구축

```
# aws coreetto 다운로드
sudo curl -L https://corretto.aws/downloads/latest/amazon-corretto-11-x64-linux-jdk.rpm -o

# jdk11 설치
sudo yum localinstall jdk11.rpm -y

# jdk version 선택
sudo /usr/sbin/alternatives --config java

# java 버전 확인
java --version

# 다운받은 설치키트 제거
rm -rf jdk11.rpm
```

배포 스크립트

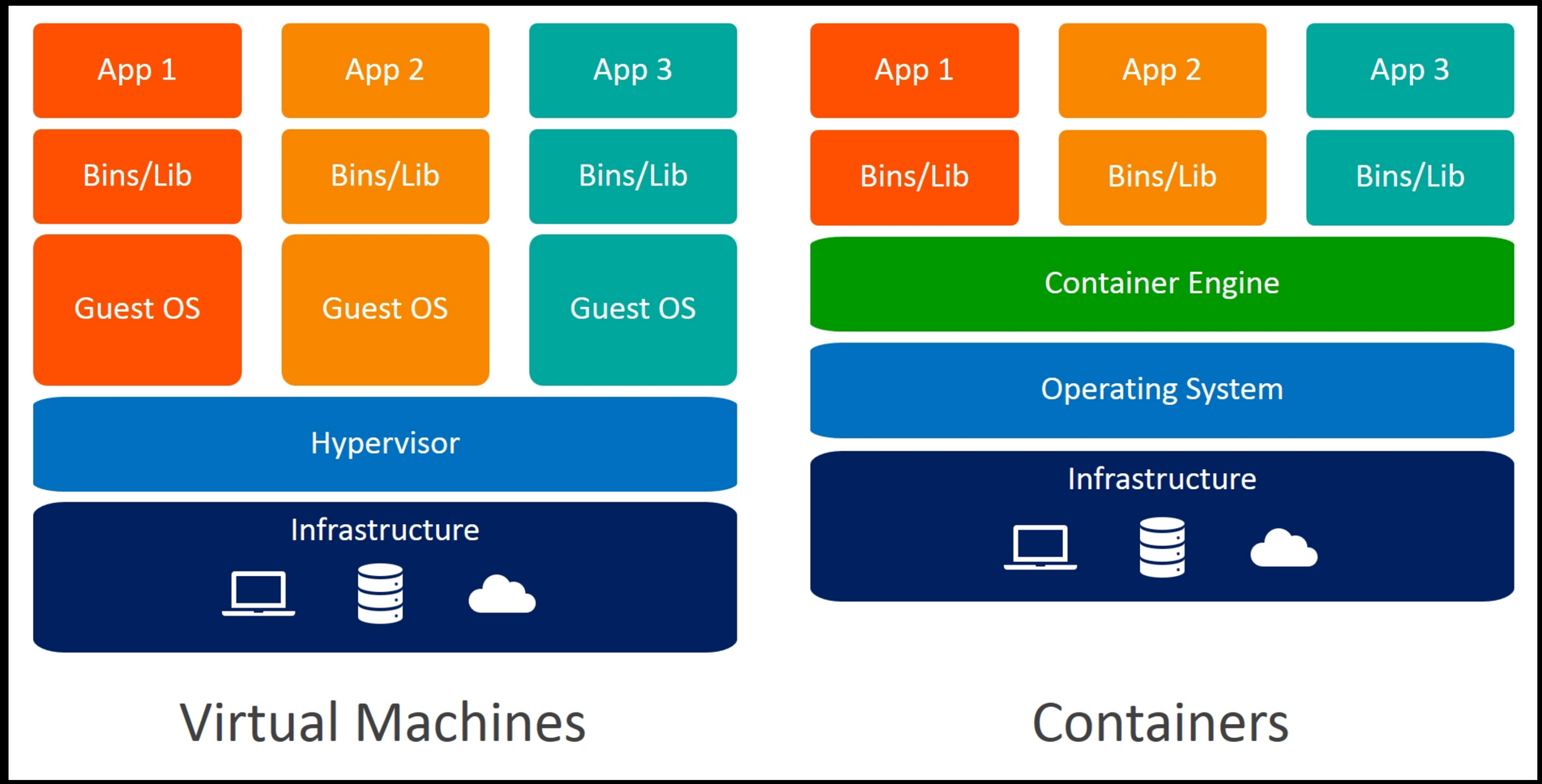
```
# 프로젝트 폴더가 위치해있는 디렉토리
REPOSITORY=/home/ec2-user/sources

# 프로젝트 디렉토리로 들어가기
cd $REPOSITORY/order-example

echo "> GIT PULL"

git pull

echo "> START TO BUILD PROJECT"

./gradlew build

echo "> COPY JAR FILES"

cp ./build/libs/*.jar $REPOSITORY/

echo "> 현재 구동중인 어플리케이션의 PID 확인"

CURRENT_PID=$(pgrep -f devops-test)

if [ -z $CURRENT_PID ]; then
        echo "현재 구동중인 APPLICATION이 없으므로 종료하지 않습니다."
else
        echo "kill -15 $CURRENT_PID"
        kill -15 $CURRENT_PID
        sleep 5
fi

echo "DEPLOY NEW APPLICATION"

JAR_NAME=$(ls $REPOSITORY |grep 'devops-test' | tail -n 1)

echo "JAR NAME : $JAR_NAME"

# nohup을 통한 무중단 배포
nohup java -jar $REPOSITORY/$JAR_NAME &
```

# 이 방식의 문제점

- 가상머신에 직접 환경을 설치하는 방식이기 때문에 다수가 배포를 관리해야하는 경우 관리가 힘듭니다 (문서화를 일일이 해줘야함)
- 필요한 경우 운영체제 자체 이미지를 관리해야하는데, 용량이 꽤 큽니다
- 가상머신 하나하나 단위로 관리를 해야하기 때문에 다수의 트래픽을 소화하는건 힘들수도 있습니다
- 어찌저찌 관리를 잘 한다고 해도 장애 발생에 대처가 어렵습니다. (가상머신이 뻗으면 회복하는 스크립트를 일일이 짜야하기 때문입니다)

# Hypervisor 방식

# Hypervisor 가상화



# Hypervisor 방식

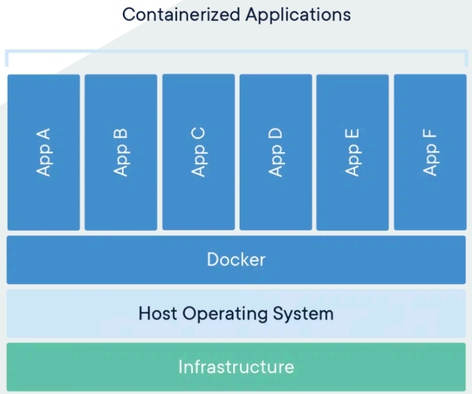
- Hypervisor 방식은 이전의 방식과 다르게, 하나의 OS 위에 여러개의 OS를 격리시키는 방식으로 배포하는 방법입니다.
- OS를 격리하여 배포하는 특징 때문에, 여러 OS에 대해서 환경 격리가 가능하기 때문에 배포의 이점을 가져올수 있습니다.

# Hypervisor 방식의 단점

- OS위에 OS를 올리는 방식이다보니 기능이 중복됩니다. (프로세스 스케쥴링 등등…)
- 상대적으로 무겁습니다. 일반적인 배포 방식에 비해서 오버헤드가 10~20% 정도가 더 발생한다고 알려져있습니다.
- 이전의 방식보다는 배포 방식이 더 어렵습니다. OS위에 OS를 한단계 더 가상화해야하기 때문입니다.
- 게다가 여전히 OS 위에서 환경을 정의하는 방식이다보니, 배포스크립트를 구성해야하는 건 여전합니다.

# Docker

# Docker란?




## Package Software into Standardized Units for Development, Shipment and Deployment

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers that run on Docker Engine:

- **Standard:** Docker created the industry standard for containers, so they could be portable anywhere
- **Lightweight:** Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- **Secure:** Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

# Docker의 특징

- Hypervisor와는 다르게, 하나의 Hosting OS 위에 Docker engine을 올리고, Docker engine 위에서 어플리케이션을 동작시키는 방식입니다.
- 가벼우며, 어플리케이션 간에 격리가 되어있으며 (어플리케이션 간에 간섭이 일어나지 않는다), 그리고 하나의 도커 컨테이너는 해당 어플리케이션의 모든 실행 환경을 포함합니다.
- 애플리케이션을 도커 환경에 올리기 위한 모든 환경, 코드 등을 포함한 것을 도커 이미지라고 부르며, 도커 위에 올라간 이미지를 container 라고 부릅니다.

# Docker의 장점

- 애플리케이션 실행 환경 자체를 Dockerfile 이라는 파일 하나로 관리하기 때문에 명시적입니다. 그리고 코드로 실행 환경이 정의되어있기 때문에 관리가 쉽습니다.
- 도커 이미지만 있다면 커맨드 몇 줄로 바로 배포하는게 가능합니다.
- 이미지로 모든 실행환경이 정의되어있기 때문에 필요할 때마다 해당 애플리케이션 실행 개수를 늘렸다, 줄였다 하기 편합니다 (흔히 수평적 확장이라 부릅니다)
- 뭣보다 컨테이너 환경의 배포 표준이기 때문에 어딜가든지 큰 차이점이 존재하지 않는 것도 장점이라고 할수있습니다

# Docker 명령어

# Docker 명령어

- docker build . -t [tag] : Docker build
- docker images : 도커 이미지들을 조회
- docker ps : 실행중인 도커 컨테이너를 조회
- docker ps -a : 모든 도커 컨테이너를 조회
- docker stop [container_name] : 실행중인 컨테이너 종료
- docker rm [container_name] : 컨테이너 삭제
- docker rmi [image] : 도커 이미지 삭제

# Docker 명령어

- docker run [options] [image] : 도커 이미지를 도커 엔진에 컨테이너로 적재
- docker run -d [image] : 도커 이미지를 백그라운드에서 실행
- docker exec -it [container] bash : 컨테이너에 접속하여 bash shell 실행
- docker logs -f [container] : 컨테이너의 로그를 실시간으로 추적

# Docker 이미지 정의 방법

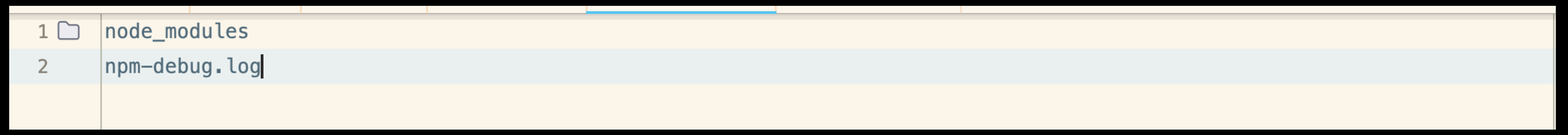
# Docker 이미지 정의 방법

- node.js 프로젝트의 루트 폴더에 Dockerfile 이름의 파일 하나를 생성합니다.
- 그리고 아래와 같이 작성해줍니다. (어떤 node.js 프로젝트든 저기서 큰 차이는 없을겁니다)

```
1  # 경량화된 node image를 끌어와서
2  FROM node:18-alpine
3  # docker 작업 공간을 /opt/app 으로 지정하여
4  WORKDIR /opt/app
5  # wildcard를 이용하여 package.json, package-lock.json 을 가져와서
6  COPY package*.json ./
7  # 모든 의존성을 설치한 후
8  RUN npm install
9  # 디렉토리를 /opt/app 으로 복사하여
10 COPY . .
11 # 3500번 포트를 노출한 다음에
12 EXPOSE 3500
13 # 엔트리포인트를 정의한다
14 CMD ["npm", "start"]
```

# 참고

- 중간에 node install 한 후에 COPY . . 을 하여 프로젝트를 모두 docker 내부에 복사하는 로직이 존재합니다. 이 때 node_modules가 덮어씌여지지 않도록 dockerignore 라는 파일을 생성하여 아래와 같이 작성합니다.



# 한번 실습해보겠습니다

# docker를 사용하기 위한 사전작업

1. AWS amazon-linux 가상머신을 사용하겠습니다

2. 가상머신에 접속한 후에, yum 패키지 매니저를 이용해서 최신 버전으로 업데이트 후, git과 docker를 설치하겠습니다.

```
sudo yum update -y
sudo yum install git -y
sudo yum install docker -y
sudo chmod 666 /var/run/docker.sock
```

3. git으로부터 node.js 프로젝트를 복사해옵니다

# 프로젝트를 실행해보겠습니다

1. 도커 이미지를 빌드합니다 -> docker build . -t node-example:latest
2. 도커 이미지가 잘 빌드되었는지 조회합니다 -> docker images
3. 도커 이미지를 컨테이너로 적재합니다 -> docker run -d -p 3500:3500 node-example:latest
4. 도커 컨테이너가 잘 적재됐는지 확인합니다 -> docker ps
5. 테스트합니다 -> curl [ip]:3500/health
6. 컨테이너를 삭제합니다 -> docker stop [container_id] && docker rm [container_id]