kakao tech
bootcamp

kakao × goorm

클라우드 환경의 이해

개발자가 되기 전에 알았으면 좋았을 것들

rex.kim

꼭 컨테이너를 써야할까요?

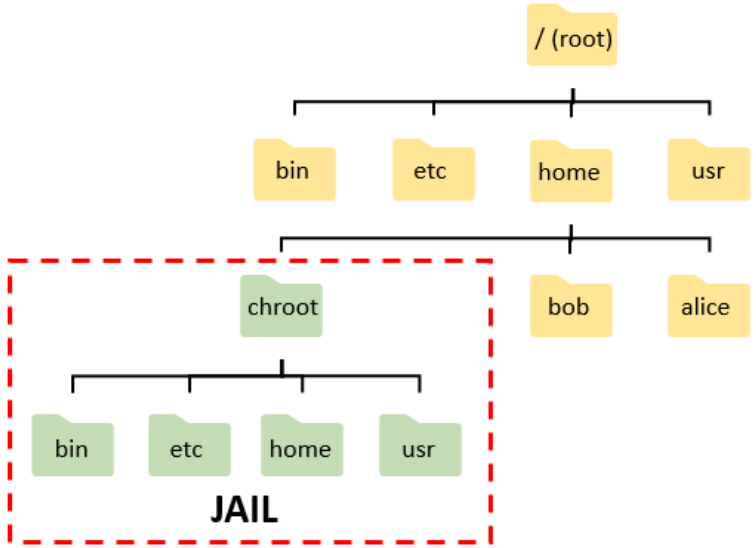
- **컨테이너를 쓰면 좋은 점**
  - 서비스를 운영하는데 필요한 환경이 컨테이너 구성에 포함되어 있어 어디서든 동일한 환경에서 실행된다.
  - 하나의 VM에서 다수의 컨테이너를 동작시켜 CPU, 메모리와 같은 자원들을 효율적으로 이용할 수 있다.
  - 컨테이너 이미지를 사용하여 보다 빠른 애플리케이션을 배포하고, 롤백할 수 있게 된다.
- **오히려 독이 되는경우?**
  - 컨테이너 환경을 구성하고, 서비스 운영을 위해 컨테이너 오케스트레이션 도구(ex. k8s)를 사용하는 것은 소규모 프로젝트에서는 오버 엔지니어링이 될 수 있다.
  - 성능이 매우 중요한 애플리케이션을 개발하는 경우, 오버헤드를 최소화하기 위해 베어메탈 서버를 이용하는 것이 나을 수 있다.
  - 애플리케이션의 구조가 단순한 경우, 컨테이너를 이용하는 것이 오히려 복잡도를 늘릴 수 있다.

컨테이너의 이해

- **컨테이너 기술의 3요소**
  - chroot
    - 프로세스에 대하여 새로운 root 디렉토리를 지정
    - 이렇게 새로운 root 디렉토리가 지정된 프로세스는 새롭게 할당된 root 디렉토리 하위로만 접근이 가능
    - `파일 시스템 격리`
  - namespace
    - 프로세스 별 리소스 사용을 분리
    - 서로 다른 namespace에 존재하는 프로세스들은 각각의 namespace에서 리소스 관리를 하여, 서로 영향을 줄 수 없음
    - `리소스 환경 격리`
  - cgroup
    - 프로세스에서 사용하는 시스템의 자원을 관리
    - `시스템 자원 격리 및 제한`

컨테이너의 이해 - chroot

- chroot(Change Root Directory)
  - 실제로 chroot를 `/home/chroot`에 적용하면 아래와 같은 모습으로 격리가 이뤄진다



컨테이너의 이해 - chroot

- chroot(Change Root Directory)

```
Usage: chroot [OPTION] NEWROOT [COMMAND [ARG]...]
or:  chroot OPTION

Run COMMAND with root directory set to NEWROOT.
  --groups=G_LIST       specify supplementary groups as g1,g2,..,gN
  --userspec=USER:GROUP  specify user and group (ID or name) to use
  --skip-chdir          do not change working directory to '/'
  --help      display this help and exit
  --version   output version information and exit
```

컨테이너의 이해 - chroot

- chroot(Change Root Directory)

```
# mkdir jail
# chroot jail /bin/bash
# chroot: failed to run command '/bin/bash': No such file or directory
```

  - 새로 생성하려고 하는 `root` 를 기준으로 `bash` 명령어를 찾을 수 없어 발생하는 에러

```
# which bash # 환경에 따라 위치가 다를 수 있으니 실습 진행 시 확인 필수!
/usr/bin/bash

# mkdir -p ./jail/bin
# cp /usr/bin/bash ./jail/bin/bash
# chroot jail /bin/bash
# chroot: failed to run command '/bin/bash': No such file or directory
```

  - `bash` 명령어가 참조하는 라이브러리가 없어 또! 에러 발생

컨테이너의 이해 - chroot

- chroot(Change Root Directory)

```
# ldd /usr/bin/bash # (which bash를 한 bash의 실제 위치를 기준으로 명령어 입력)
linux-vdso.so.1 (0x00007fffd6ff5000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x000079fb33aa0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000079fb33800000)
/lib64/ld-linux-x86-64.so.2 (0x000079fb33c48000)
```

  - 해당 라이브러리들을 `/lib` 와 `/lib64` 에 맞게 복사합니다.

```
# mkdir  -p ./jail/lib
# mkdir  -p ./jail/lib64
# cp /lib/x86_64-linux-gnu/libtinfo.so.6  ./jail/lib/
# cp /lib/x86_64-linux-gnu/libc.so.6 ./jail/lib/
# cp /lib64/ld-linux-x86-64.so.2 ./jail/lib64/
```

컨테이너의 이해 - chroot

- chroot(Change Root Directory)

```
# chroot jail /bin/bash
bash-5.2# pwd
/
bash-5.2# cd ..
bash-5.2# pwd
/
bash-5.2# echo "A" > a.txt
bash-5.2# exit
# cd /home/jail
# ls
a.txt  bin  lib  lib64
# cat a.txt
A
```

  - `/home/jail` 을 기준으로 root 디렉토리가 고정된 것을 확인할 수 있다.

컨테이너의 이해 - chroot 탈옥하기

- **chroot만으로 구현된 격리는 완벽하지 않다!**
  - 아래 코드는 널리 알려진 chroot 탈옥 코드

```
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
 mkdir(".out",0755);
 chroot(".out");
 chdir("../../../../../");
 chroot(".");

 return execl("/bin/sh","-i",NULL);
}
```

컨테이너의 이해 - chroot 탈옥하기

- **실제 탈옥 과정**

```
root@ip-10-6-0-163:/home# chrootjail /bin/bash
bash-5.2# ./escape
# ls
bin         lib             opt         snap
bin.usr-is-merged  lib.usr-is-merged  proc        srv
boot            lib64       root            sys
dev             lost+found      run         tmp
etc         media           sbin        usr
home            mnt             sbin.usr-is-merged  var
```

  - 원래는 접근 불가능한 host의 root에 접근한 모습

컨테이너의 이해 - 탈옥 방지하기

- **pivot_root를 활용하여 탈옥을 막을 수 있다**
  - chroot로 격리한 공간은 루트 파일 시스템이 동일하여 탈옥이 가능했으나,
    `pivot_root` 를 이용하면 루트 파일 시스템의 Mount Point를 변경하여 상위 권한 탈취를 방지할 수 있다

```
# unshare --mount /bin/bash #(Mount Namespace를 격리하여 shell 실행)

# mkdir pivot_jail          pivot_root여야함?

# mount -t tmpfs -o size=100M none pivot_jail #(루트 파일 시스템으로 사용하기 위해 마운트)
# cp -r /home/jail/* /home/pivot_jail/
# mkdir -p /home/pivot_jail/old_fs
# cd /home/pivot_jail
# pivot . old_fs #(현재 위치가 새로운 루트 파일 시스템이 되며, 기존 루트 파일 시스템은 old_fs에서 확인할 수 있음)
# ./escape #(탈옥에 실패하는 것을 확인)
```

컨테이너의 이해 - Namespace

- **Namespace의 종류**
  - MNT: 마운트 시, 파일 시스템을 격리한다.
  - PID: Process ID를 격리한다. Namespace가 서로 다른 프로세스끼리는 접근할 수 없게 된다.
  - NET: 네트워크 리소스를 격리한다. 가상의 네트워크 장치를 할당한다.
  - IPC: IPC(Inter-Process Communication)를 격리하여, 다른 프로세스의 접근을 방지한다.
  - UTS: 호스트 이름과 도메인 이름을 격리한다. (UNIX Timesharing System)
  - User: UID,GID 정보를 격리한다.

컨테이너의 이해 - Namespace

- Namespace의 분리

```
# unshare --help
Usage:
unshare [options] [<program> [<argument>...]]

Run a program with some namespaces unshared from the parent.
```

컨테이너의 이해 - Namespace

- **실제 분리된 Namespace 확인해보기**
  - 1번 프로세스를 기준으로 Namespace 정보를 확인

```
# /proc/[PID]/ns에서 해당 프로세스의 Namespace 정보를 확인할 수 있다
# ls -al /proc/1/ns
total 0
dr-x--x--x 2 root root 0 Jul 31 09:10 .
dr-xr-xr-x 9 root root 0 Jul 31 08:29 ..
lrwxrwxrwx 1 root root 0 Jul 31 15:53 cgroup → 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 ipc → 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 mnt → 'mnt:[4026531841]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 net → 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 Jul 31 09:10 pid → 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 pid_for_children → 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 time → 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 time_for_children → 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 user → 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Jul 31 15:53 uts → 'uts:[4026531838]'
```

컨테이너의 이해 - Namespace

- **실제 분리된 Namespace 확인해보기**
  - 다른 Namespace를 사용하는 환경을 구성하기 위해 Docker Container 실행

```
# docker pull nginx
# docker run -itd --name test nginx
# ps -ef | grep nginx
root      4875   4851  0 15:55 pts/0    00:00:00 nginx: master process nginx -g daemon off;
message+  4921   4875  0 15:55 pts/0    00:00:00 nginx: worker process
message+  4922   4875  0 15:55 pts/0    00:00:00 nginx: worker process
root      4954   4936  0 16:01 pts/1    00:00:00 grep --color=auto nginx
```

컨테이너의 이해 - Namespace

- **실제 분리된 Namespace 확인해보기**
  - 해당 컨테이서 실행되고 있는 nginx process(PID: 4875)를 기준으로 Namespace 정보 확인

```
# ls -al /proc/4875/ns
total 0
dr-x--x--x 2 root root 0 Jul 31 15:55 .
dr-xr-xr-x 9 root root 0 Jul 31 15:55 ..
lrwxrwxrwx 1 root root 0 Jul 31 15:55 cgroup → 'cgroup:[4026532292]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 ipc → 'ipc:[4026532228]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 mnt → 'mnt:[4026532226]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 net → 'net:[4026532230]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 pid → 'pid:[4026532229]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 pid_for_children → 'pid:[4026532229]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 time → 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 time_for_children → 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 user → 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Jul 31 15:55 uts → 'uts:[4026532227]'
```

컨테이너의 이해 - CGroup

- **CGroup (Control Group)을 활용해 제한 가능한 자원의 종류**
  - CPU: 스케쥴러를 사용하여 해당 CGroup에 속한 프로세스의 CPU 사용시간을 제어
  - memory: 프로세스의 메모리 사용량을 제어
  - blkio: Block I/O에 대한 제한 설정
  - cpuset: 개별 CPU 및 메모리 노드를 CGroup에 바인딩하기 위한 서브 시스템
  - devices: 장치에 대한 액세스를 허용하거나 거부
  - etc.

컨테이너의 이해 - CGroup

- **Docker Inspect를 확인해보자**

```
# docker inspect test
[{
 ...
 "HostConfig": {
    ...
    "CpuShares": 0,
    "Memory": 0,
    "CgroupParent": "",
    "BlkioWeight": 0,
    "CpuPeriod": 0,
    "CpuQuota": 0,
    "CpusetCpus": "",
    "CpusetMems": "",
    "Devices": [],
    ...
 }
}]
```

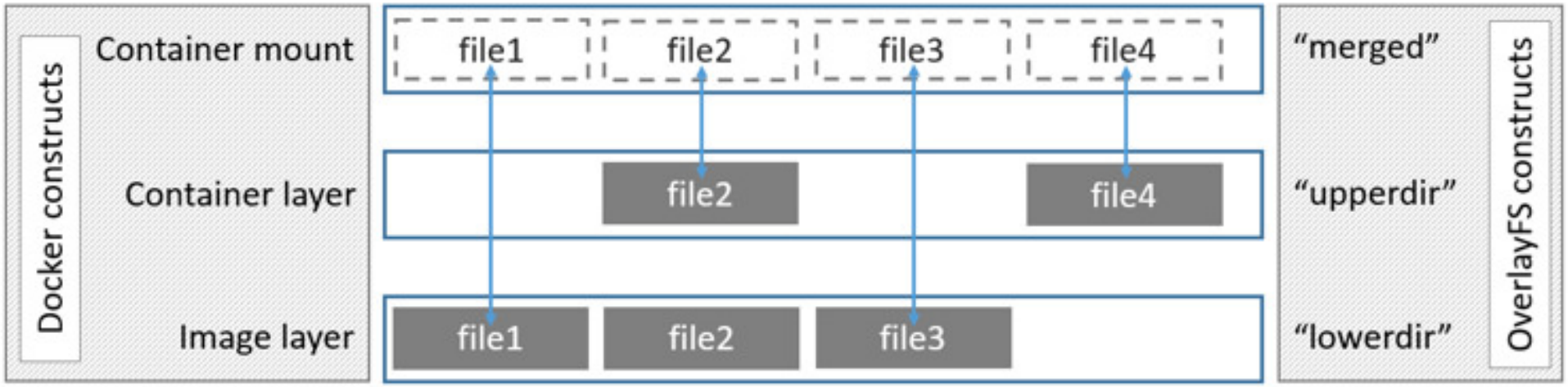
컨테이너의 이해 - CGroup

- **CGroup을 활용하여 리소스 제한하기**
  - 편리한 제한 환경을 구성하기 위해 Docker를 이용한다

```
docker run -itd --name [컨테이너 명]
--memory (컨테이너에서 사용 가능한 메모리 제한)
--cpu-shares (기본 값 1024를 기준으로하는 가중치)
--cpuset-cpus (특정 CPU만 사용하도록 설정)
--cpu-period, --cpu-quota (컨테이너가 할당받는 CPU 주기를 결정)
--cpus (보다 직관적으로 사용하는 CPU 개수 지정)
--device-write-bps, --device-read-bps (Bloci I/O 지정)
[이미지명]
```

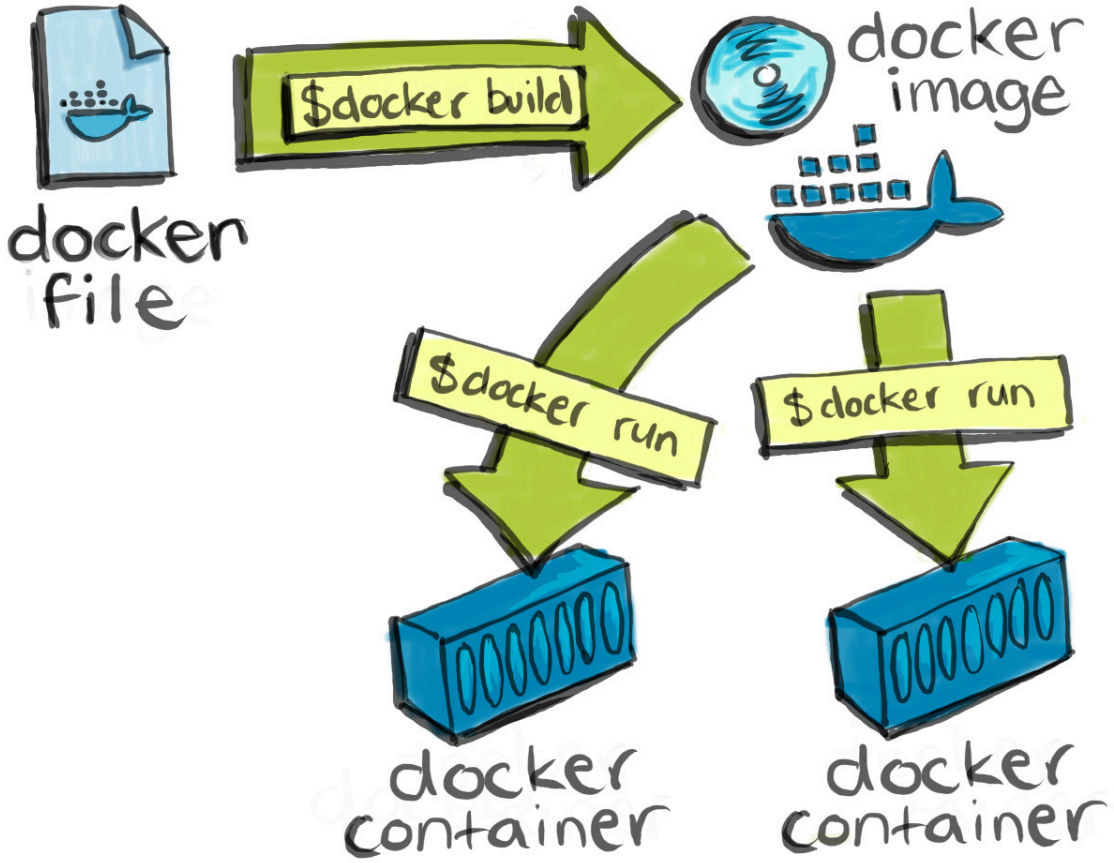
컨테이너의 이해 - Union File System

- **Union File System이란**
  - 여러 개의 파일 시스템(Layer)를 하나의 파일 시스템(Union)으로 취급하는 파일 시스템이다.
  - 이미지 영역과 사용자 영역이 분리되어 있으며 Copy on Write(COW) 전략을 사용하여 상위 레이어(사용자 영역)에서 하위 레이어(이미지 영역)에 영향을 주지 않는다.



Union File System 중 하나인 Overlay2의 구조

컨테이너 기반의 서비스 운영에 대하여



컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile이란?**
  - Docker 이미지를 생성하기 위한 스크립트 파일이다.
  - 애플리케이션이 실행되기 위한 환경을 정의하고 이를 실행하기 위한 명령어가 포함되어 있다.

```
# Node.js 애플리케이션 이미지를 생성하기 위한 Dockerfile 예시
FROM node:22

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 8080
CMD ["node", "app.js"]
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile Command**
  - FROM
    - 베이스 이미지를 지정한다.
      기본적으로는 `https://hub.docker.com`에 등록된 이미지를 사용
    - `[이미지명]:[태그]`의 형태이며 태그를 지정하지 않으면 `latest`가 지정된다.

```
FROM ubuntu:20.04
```

  - LABEL
    - 메타데이터를 추가한다.

```
LABEL version="1.0"
LABEL description="KAKAO TECH BOOTCAMP"
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile Command**
  - COPY
    - 호스트의 파일을 컨테이너 내의 파일 시스템으로 복사한다.

```
COPY . /app
```

  - ADD
    - COPY와 비슷하나 URL에서 파일을 다운로드하거나 압축 파일을 자동으로 풀어주는 기능이 있다. 다만 이런 경우가 아니라면 COPY를 쓰는 것을 권장한다.(Docker 권장사항)

```
ADD https://example.com/file.tar.gz /app/
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- Dockerfile Command
  - WORKDIR
    - 작업을 수행할 경로를 지정합니다. 이후 명령어는 해당 경로에서 실행됩니다.

```
WORKDIR /app
```

  - RUN
    - 이미지 빌드 과정에서 실행할 명령어를 지정하여, 실행한다.

```
RUN apt-get update && apt-get install -y nginx
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile Command**
  - CMD
    - 컨테이너가 시작될 때 실행할 명령어를 지정한다. (1번 프로세스가 된다.)
      해당 명령어가 종료되면 컨테이너도 종료된다.

```
CMD ["python3", "app.py"]
```

  - ENTRYPOINT
    - 컨테이너가 시작될 때 실행할 명령어를 지정한다. CMD와 결합하여 파라미터를 넘기는 형태로 사용될 수 있으며, `무조건` 최초에 실행된다.

```
ENTRYPOINT ["python3", "app.py"]
##
ENTRYPOINT ["python3", "app.py"]
CMD ["params"]
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile Command**
  - CMD와 ENTRYPOINT
    - 언뜻 보기에 동일한 명령어로 보이지만, `CMD` 의 경우 컨테이너를 실행할 때, 명령어를 덮어 씌울 수 있고 `ENTRYPOINT` 의 경우 덮어 씌우는 것이 불가능하다.

```
# CMD 버전
FROM ubuntu

CMD ["echo", "CMD test"]
```

```
#ENTRYPOINT 버전
FROM ubuntu

ENTRYPOINT ["echo", "ENTRYPOINT test"]
```

컨테이너 기반의 서비스 운영에 대하여 - Dockerfile

- **Dockerfile Command**
  - ENV
    - 환경변수를 설정한다.

```
ENV APP_ENV=production
```

- ARG
  - Dockerfile 내에서 사용할 변수를 정의한다. (빌드 과정에서만 유효)

```
ARG version=1.0
```

- EXPOSE
  - 컨테이너가 수신할 포트를 지정한다

```
EXPOSE 80
```

컨테이너 기반의 서비스 운영에 대하여 - Docker Image

- **Dockerfile로 이미지 만들기**

```
# cd WORKING_DIR
# docker build -t [이미지명:태그] .
```

  - -t 옵션을 줄 경우 이미지에 태그를 지정할 수 있다.

- **생성한 이미지로 컨테이너 실행하기**

```
# docker run [이미지명:태그]
Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

  - -i, --interactive: 표준입력을 활성화한다.
  - -t, --tty: TTY모드를 사용한다.
  - --name: 컨테이너 이름을 설정
  - -d, --detach: 컨테이너를 백그라운드로 실행한다.

컨테이너 기반의 서비스 운영에 대하여 - Docker Image

- **생성한 이미지로 컨테이너 실행하기**
  - -p, --public: 포트포워딩을 진행한다. [호스트 포트]:[컨테이너 포트]
  - --privileged: 컨테이너 안에서 호스트의 커널 기능을 모두 사용할 수 있도록 권한을 부여한다. (되도록 사용하지 않는 것이 좋다)
  - -v, --volume: 마운트로 호스트와 컨테이너의 디렉토리를 연결한다.
  - --gpus: 컨테이너에서 GPU 장치를 사용한다.

컨테이너 기반의 서비스 운영에 대하여 - Docker Image

- **Repo에 등록하기**

```
# docker login
# docker tag my-node-app your-dockerhub-username/my-node-app
docker push your-dockerhub-username/my-node-app
# docker push your-dockerhub-username/my-node-app
```

컨테이너 기반의 서비스 운영에 대하여 - Jenkins

- **Jenkins란?**
  - Jenkins는 오픈 소스 자동화 도구로, 소프트웨어 개발과 관련된 다양한 작업을 자동화하기 위해 사용된다.
    주로 지속적 통합(Continuous Integration, CI)과 지속적 배포(Continuous Deployment, CD) 파이프라인을 구축하는 데 사용한다.
  - 플러그인을 사용하여 Git, Docker, k8s, Slack과 연동이 가능하다.

컨테이너 기반의 서비스 운영에 대하여 - Jenkins

- Jenkins Pipeline

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building...'
        // 빌드 명령어 추가
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying...'
        // 배포 명령어 추가
      }
    }
  }
}
```