Docker in Windows

Docker

1.1 Docker 란?

Docker는 컨테이너 기반의 오픈소스 가상화 플랫폼입니다. 애플리케이션과 그 실행에 필요한 모든 의존성을 컨테이너라는 표준화된 유닛으로 패키징하여, 어떤 환경에서도 동일하게 실행될 수 있도록 보장합니다. 이는 개발, 테스트, 배포 과정을 더욱 효율적이고 일관성 있게 만들어줍니다.

Docker의 주요 장점 중 하나는 격리된 환경에서 애플리케이션을 실행할 수 있다는 것입니다. 각 컨테이너는 독립적으로 실행되며, 호스트 시스템이나 다른 컨테이너와의 충돌 없이 필요한 리소스를 사용할 수 있습니다. 또한 Docker는 마이크로서비스 아키텍처를 구현하는 데 이상적인 도구로 널리 사용되고 있습니다.

1.2 Docker 설치

Docker는 운영체제별로 설치 방법이 다릅니다. Windows와 macOS 사용자는 Docker Desktop을 설치하여 손쉽게 Docker 환경을 구축할 수 있습니다. Linux 사용자는 패키지 관리자를 통해 Docker Engine을 직접 설치합니다.

각 운영체제별 설치 방법은 다음과 같습니다:

- Windows: Docker 웹사이트에서 Docker Desktop for Windows 다운로드 및 설치
- macOS: Docker Desktop for Mac 다운로드 및 설치
- Linux: apt, yum 등의 패키지 관리자로 Docker Engine 설치

이제 Windows 11에서 Docker를 설치하고 운영하는 방법을 단계별로 설명하겠습니다. Windows 11에서는 WSL 2(Windows Subsystem for Linux 2)를 기반으로 Docker를 운영하는 것이 표준적인 방법입니다. 다음은 상세한 설치 및 운영 과정입니다.

1. 시스템 요구 사항 확인

- **운영체제**: Windows 11 Home, Pro, Enterprise 또는 Education 버전 (빌드 22000 이상)
- **하드웨어**: CPU의 가상화(VT-x/AMD-V) 지원 및 최소 4GB RAM 권장
- **WSL 2 지원**: WSL 2 활성화 필요

설치 전에 시스템이 최신 상태인지 확인하세요:

- **설정 > Windows 업데이트**에서 최신 업데이트를 설치합니다.

2. WSL 2 설치

Docker Desktop은 WSL 2를 백엔드로 사용하므로 먼저 이를 설정해야 합니다.

1. **관리자 권한으로 PowerShell 실행**:
   - 시작 메뉴에서 "PowerShell"을 검색하고, "관리자 권한으로 실행"을 선택합니다.
2. **WSL 설치**: `wsl --install`
   - 이 명령어로 WSL과 기본 Linux 배포판(Ubuntu)이 설치됩니다. 설치 후 재부팅이 필요할 수 있습니다.
3. **WSL 2로 기본 설정**: `wsl --set-default-version 2`
   - WSL 2가 기본으로 설정되었는지 확인합니다.
4. **설치된 배포판 확인**: `wsl --list`
   - Ubuntu가 없다면 Microsoft Store에서 "Ubuntu"를 검색하여 설치하세요.
   - `wsl --install -d Ubuntu-22.04`
5. wsl 상태 확인
   - `wsl -l -v`

```
(base) PS C:\Users\Administrator> wsl -l -v
  NAME                   STATE           VERSION
* Ubuntu-22.04           Running         2
```

6. 터미널 열기 - `wsl -d Ubuntu-22.04`
7. Vscode 실행 code - `code .`
8. 패키지 업데이트
    - sudo apt update
    - sudo apt upgrade -y
9. 도커 설치 및 systemd 활성 권한 부여

- sudo apt install -y docker.io
- sudo usermod -aG docker $USER
- newgrp docker
- sudo update-alternatives --config iptables → legacy 선택
- sudo update-alternatives --set iptables /usr/sbin/iptables-legacy
- sudo systemctl start docker
- sudo systemctl status docker
- sudo nano /etc/docker/daemon.json

```
{
  "iptables": false,
  "bridge": "none"
}
```

이 설정은 windows 에서 ROS2나 ROS1을 자체적으로 운용하게 할 수 있지만 DDS 를 사용 할때 같은 subnet 으로 만들 수 없다. WSL 자체가 내부 NAT로 설정이 되어 있기 때문이다. 이 설정은 Hyper-V 관리자를 통해서 이루어 지는데 bridge 로 바꾸려면 Hyper-V 옵션을 수정해야 한다.

3. 가상환경에서의 Docker

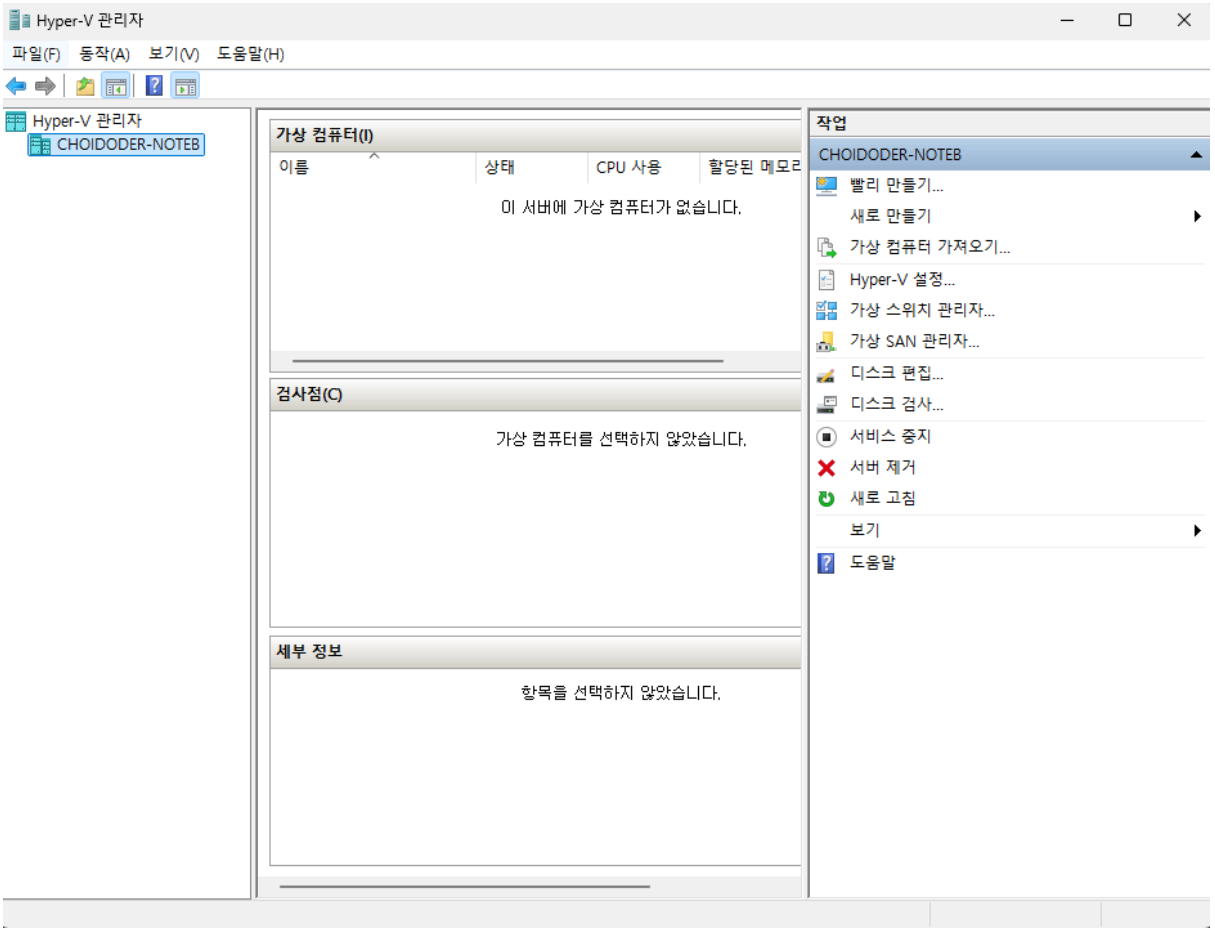
기본설정은 똑같이 한다.

디스플레이 관련해서 Wayland 로 설정이 되어 있으면 GUI 가 전달이 안되므로 로그인 환경에서 X11로 바꾼다.

그리고 외부의 local 접근을 허용하기 위해 아래 코드를 실행 한다.

```
xhost +local:
```

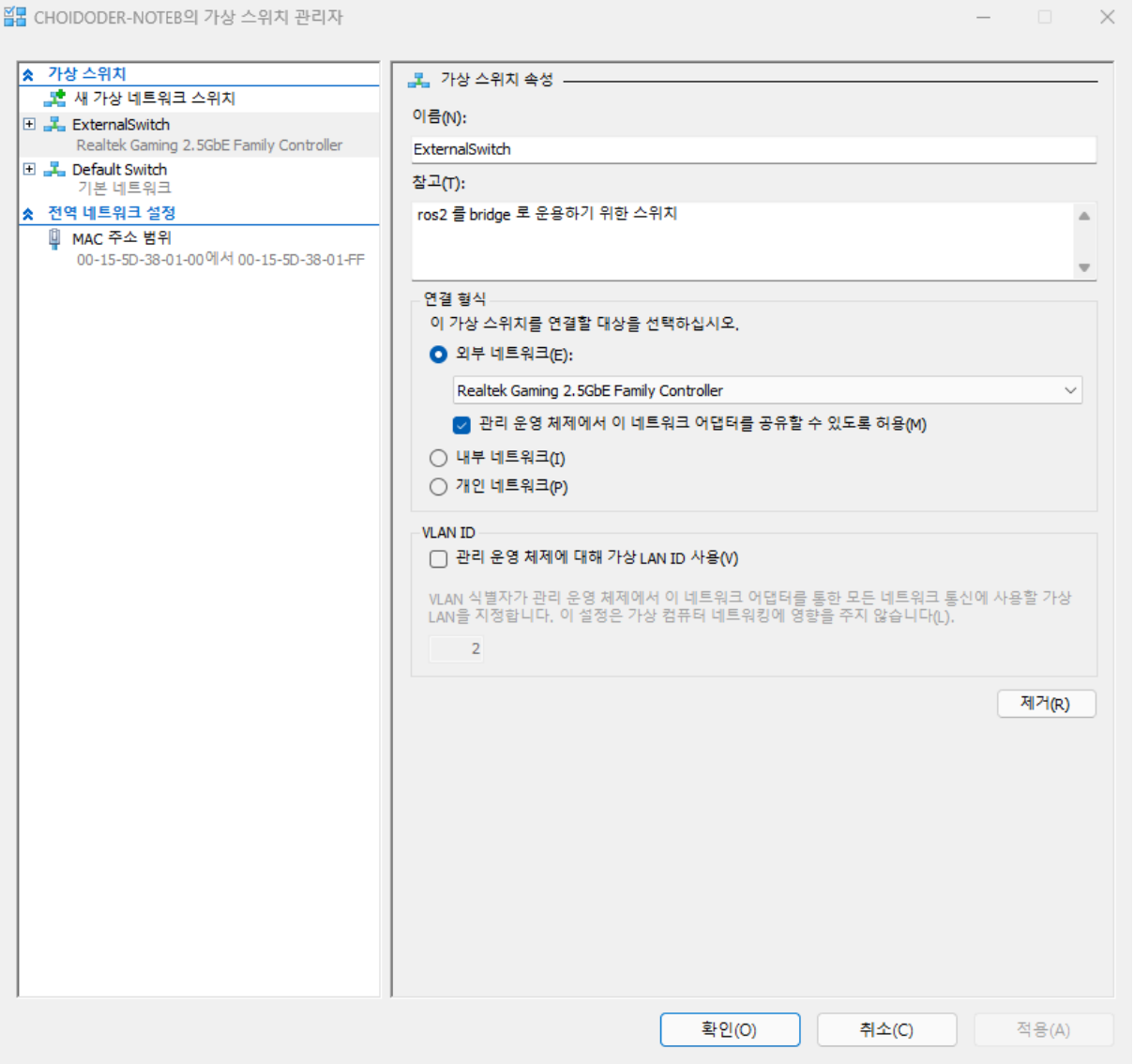
1.3 WSL2 를 브릿지로 설정 하는 방법



가상 스위치 관리자를 선택한다.

새 가상 네트워크 스위치를 만들고

ExternalSwitch 로 이름을 지정하고 외부 네트워크에 사용하는 랜카드를 지정한다.



- Hyper-V 활성화 확인
  - `Get-WindowsOptionalFeature -Online | Where-Object { $_.FeatureName -eq "Microsoft-Hyper-V" }`
- WSL 2에 외부 스위치 연결 - 이 설정은 윈도우 11이상에서만 지원 됨.
  - `notepad "$env:USERPROFILE\.wslconfig"`

```
[wsl2]
networkingMode=bridged
vmSwitch=ExternalSwitch
```

- networkingMode=mirrored
  - 네트 워크가 잡히기는 하는데 Xlaunch 와 ros2 topic list 가 정상적으로 실행 되지 않는다.

`docker run -it --net=host  --gpus all -e DISPLAY=192.168.0.7:0 osrf/ros:humble-desktop`

추가 설치

```
apt update
apt install -y libx11-xcb1 libxcb-xinerama0 libxcb-icccm4 libxcb-image0 libx
cb-keysyms1 libxcb-randr0 libxcb-render-util0 libxcb-shape0 libxcb-sync1 l
ibxcb-xfixes0 libxcb-xkb1
apt install -y qt5-default qtbase5-dev qtbase5-dev-tools
```

```
sudo apt update
sudo apt upgrade
sudo apt install ros-humble-gazebo-*
sudo apt install ros-humble-dynamixel-sdk
sudo apt install ros-humble-turtlebot3-msgs
sudo apt install ros-humble-turtlebot3

mkdir -p ~/turtlebot3_ws/src/
cd ~/turtlebot3_ws/src/
git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3_simulation
s.git
cd ~/turtlebot3_ws && colcon build --symlink-install
source ./install/setup.bash
```

~/.bashrc 추가

```
export TERM=xterm-256color
export PS1="\[\e[01;32m\]\u@\h:\[\e[01;34m\]\w\[\e[00m\]\$ "
alias ls='ls --color=auto'
alias killgazebo='killall -9 gazebo & killall -9 gzserver  & killall -9 gzclient'
source /opt/ros/humble/setup.bash
source ~/turtlebot3_ws/install/local_setup.bash
source /usr/share/gazebo/setup.bash
export ROS_DOMAIN_ID=12
export TURTLEBOT3_MODEL=burger
export DISPLAY=192.168.0.7:0
export XDG_RUNTIME_DIR=/run/user/$(id -u)
```

```
mkdir -p /run/user/$(id -u)
chmod 700 /run/user/$(id -u)
```

- dockerfile

```
FROM osrf/ros:humble-desktop

# 환경 변수 설정
ENV DEBIAN_FRONTEND=noninteractive
ENV TURTLEBOT3_MODEL=burger
ENV ROS_DOMAIN_ID=12
ENV TERM=xterm-256color
ENV DISPLAY=192.168.0.7:0
ENV XDG_RUNTIME_DIR=/run/user/1000

# 기본 패키지 업데이트 및 필요 패키지 설치
RUN apt-get update && apt-get upgrade -y && \
    apt-get install -y \
    ros-humble-gazebo-* \
    ros-humble-dynamixel-sdk \
    ros-humble-turtlebot3-msgs \
    ros-humble-turtlebot3 \
    git && \
    rm -rf /var/lib/apt/lists/*

# 사용자 작업 디렉토리 생성 및 TurtleBot3 시뮬레이션 소스 클론
WORKDIR /root
RUN mkdir -p /root/turtlebot3_ws/src/ && \
    cd /root/turtlebot3_ws/src/ && \
    git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3_simulati
ons.git && \
    cd /root/turtlebot3_ws && \
    /bin/bash -c "source /opt/ros/humble/setup.bash && colcon build --symli
nk-install"
```

```
# bashrc 설정 추가
RUN echo 'export TERM=xterm-256color' >> ~/.bashrc && \
    echo 'export PS1="\[\e[01;32m\]\u@\h:\[\e[01;34m\]\w\[\e[00m\]\$ "' >>
~/.bashrc && \
    echo 'alias ls="ls --color=auto"' >> ~/.bashrc && \
    echo 'alias killgazebo="killall -9 gazebo & killall -9 gzserver & killall -9 gz
client"' >> ~/.bashrc && \
    echo 'source /opt/ros/humble/setup.bash' >> ~/.bashrc && \
    echo 'source ~/turtlebot3_ws/install/local_setup.bash' >> ~/.bashrc && \
    echo 'source /usr/share/gazebo/setup.bash' >> ~/.bashrc && \
    echo 'export ROS_DOMAIN_ID=12' >> ~/.bashrc && \
    echo 'export TURTLEBOT3_MODEL=burger' >> ~/.bashrc && \
    echo 'export DISPLAY=192.168.0.7:0' >> ~/.bashrc && \
    echo 'export XDG_RUNTIME_DIR=/run/user/$(id -u)' >> ~/.bashrc && \
    echo 'mkdir -p /run/user/$(id -u)' >> ~/.bashrc && \
    echo 'chmod 700 /run/user/$(id -u)' >> ~/.bashrc

# XDG 런타임 디렉토리 설정
RUN mkdir -p ${XDG_RUNTIME_DIR} && chmod 700 ${XDG_RUNTIME_DI
R}

# 시작 시 bash 실행
CMD ["/bin/bash"]
```

1.4 Docker 의 기본 기능

1.4.1 컨테이너 개념과 가상화 비교 (VM vs Container)

컨테이너와 가상 머신(VM)은 모두 가상화 기술이지만, 그 접근 방식에서 중요한 차이가 있습니다. VM은 하드웨어 수준에서 가상화를 구현하여 각각의 VM이 자체 운영체제를 포함하는 반면, 컨테이너는 호스트 OS의 커널을 공유하면서 애플리케이션 실행에 필요한 최소한의 구성 요소만을 포함합니다. 이러한 차이로 인해 컨테이너는 VM에 비해 더 가볍고 빠르게 시작되며, 리소스 효율성이 높습니다.

호스트 OS 의 커널을 사용하기 때문에리눅스 컨테이너는 윈도우 운영체제에서 직접 실행할 수 없습니다. 이러한 이유로 Windows에서는 WSL2(Windows Subsystem for Linux)나

Hyper-V와 같은 가상화 계층이 필요합니다. 이는 Windows 환경에서 Linux 컨테이너를 실행하기 위한 필수적인 구성 요소입니다.

macOS에서는 Linux와 마찬가지로 Unix 계열 운영체제이기 때문에 컨테이너 실행을 위한 기본적인 호환성이 있습니다. 하지만 macOS도 네이티브 Linux 커널을 사용하지 않기 때문에, Docker Desktop for Mac은 경량 가상화 기술을 사용합니다. 특히 Apple Silicon(M1/M2) 칩을 사용하는 최신 Mac의 경우, 전용 가상화 프레임워크를 통해 효율적으로 컨테이너를 실행할 수 있습니다.

Docker Desktop for Mac은 내부적으로 경량 가상 머신을 사용하여 Linux 컨테이너를 실행하며, 이 과정이 사용자에게는 완전히 투명하게 처리됩니다. 이러한 구현 방식은 Windows의 WSL2와 유사하지만, macOS의 고유한 가상화 기술을 활용하여 더 나은 성능과 호환성을 제공합니다.

Docker는 **하나의 컨테이너에 하나의 주요 프로세스**를 실행하는 것을 권장합니다. 이러한 설계 원칙에 따라, Docker는 컨테이너 내부에서 `systemd`와 같은 초기화 시스템(init system)의 사용을 지양합니다. 그 이유는 다음과 같습니다:

1. **호스트 OS와의 충돌 가능성:** `systemd`는 파일 시스템 마운트, 커널 파라미터 제어 등 호스트 시스템 수준의 작업을 수행합니다. 이러한 기능은 Docker의 격리 모델과 충돌할 수 있으며, 컨테이너의 독립성을 저해할 수 있습니다. Stack Overflow
2. **보안 및 안정성 문제:** `systemd`를 컨테이너에서 실행하려면 추가적인 권한 부여가 필요하며, 이는 보안상의 취약점을 초래할 수 있습니다. 또한, `systemd`는 호스트 시스템의 리소스를 직접적으로 조작하기 때문에, 컨테이너의 안정성에도 부정적인 영향을 미칠 수 있습니다.

여러 프로세스의 관리 대안:

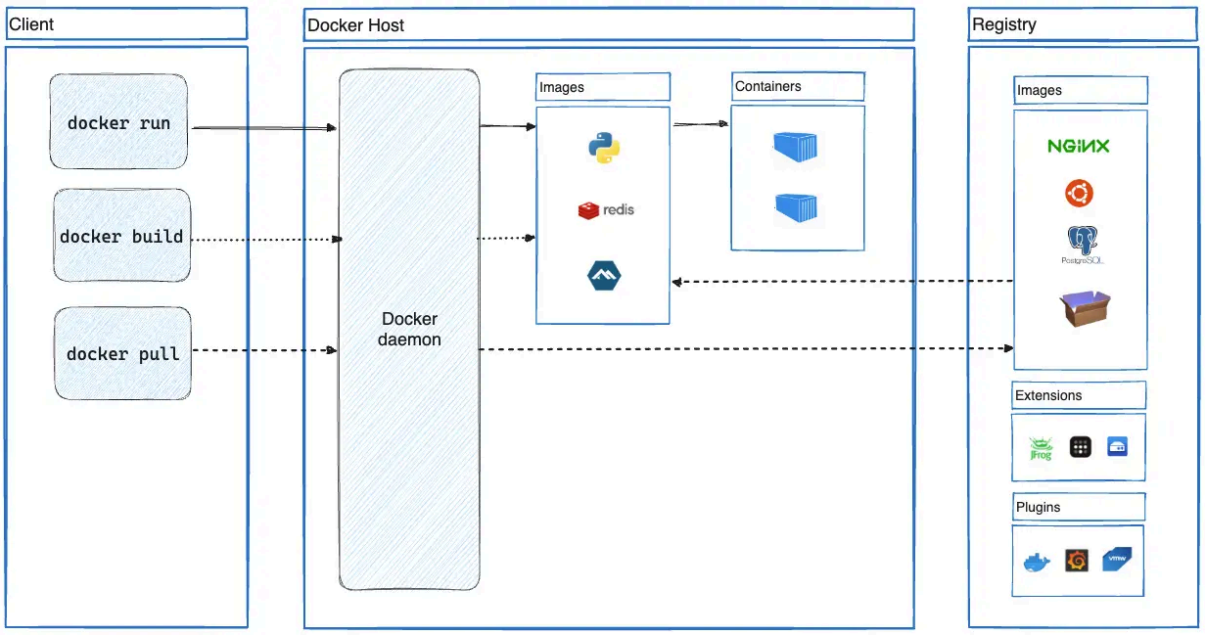
컨테이너 내부에서 여러 프로세스를 관리해야 하는 경우, `systemd` 대신 다음과 같은 경량 프로세스 관리 도구를 사용하는 것이 일반적입니다:

1. **Supervisor:** Python으로 작성된 프로세스 제어 시스템으로, 여러 프로세스의 시작, 종료, 재시작 등을 관리할 수 있습니다. 컨테이너 내부에서 `supervisord`를 실행하여 다중 프로세스를 효과적으로 제어할 수 있습니다.
2. **s6:** 경량화된 프로세스 관리 도구로, 컨테이너 환경에서 효율적으로 다중 프로세스를 관리할 수 있습니다. 특히, 초기화 시스템이 필요한 복잡한 컨테이너 설정에서 유용하게 사용됩니다.
3. **runit:** 간단하고 신뢰성 있는 프로세스 관리 도구로, 빠른 시작과 종료를 지원하며, 컨테이너 내부에서 여러 프로세스를 관리하는 데 적합합니다.

모범 사례:

가능한 경우, 각 프로세스를 별도의 컨테이너로 분리하여 실행하는 것이 Docker의 철학에 부합하며, 이는 시스템의 확장성과 유지보수성을 향상시킵니다. 그러나 여러 프로세스를 하나의 컨테이너에서 실행해야 하는 상황이라면, 위에서 언급한 경량 프로세스 관리 도구를 활용하여 효율적으로 관리하는 것이 바람직합니다.

1.4.2 Docker 아키텍처 및 주요 구성요소(Image, Container, Engine)



https://docs.docker.com/get-started/docker-overview/#what-can-i-use-docker-for

Docker 데몬

Docker 데몬(`dockerd`)은 Docker API 요청을 수신하고 이미지, 컨테이너, 네트워크, 볼륨과 같은 Docker 객체를 관리합니다. 또한 다른 데몬과 통신하여 Docker 서비스를 관리할 수 있습니다.

Docker 클라이언트

Docker 클라이언트(`docker`)는 사용자가 Docker와 상호 작용하는 주요 수단입니다. `docker run`과 같은 명령을 실행하면 클라이언트가 이를 `dockerd`로 전달하여 실행합니다. Docker 클라이언트는 Docker API를 사용하며, 여러 데몬과 동시에 통신할 수 있습니다.

도커 데스크톱

Docker Desktop은 Mac, Windows, Linux 환경에서 컨테이너화된 애플리케이션과 마이크로서비스를 쉽게 빌드하고 공유할 수 있는 애플리케이션입니다. Docker Desktop은 Docker 데몬(`dockerd`), Docker 클라이언트(`docker`), Docker Compose, Docker Content Trust, Kubernetes, Credential Helper를 포함합니다. 자세한 내용은 Docker Desktop을 참조하세요.

Docker 레지스트리

Docker 레지스트리는 Docker 이미지를 저장하는 저장소입니다. Docker Hub는 누구나 사용할 수 있는 공개 레지스트리이며, Docker는 기본적으로 Docker Hub에서 이미지를 검색합니다. 필요한 경우 사용자가 직접 개인 레지스트리를 운영할 수 있습니다.

`docker pull`이나 `docker run` 명령을 실행하면 Docker는 설정된 레지스트리에서 필요한 이미지를 가져옵니다. `docker push` 명령을 사용하면 이미지를 레지스트리에 업로드할 수 있습니다.

Docker 객체

Docker를 사용하면 이미지, 컨테이너, 네트워크, 볼륨, 플러그인 등 다양한 객체를 생성하고 사용할 수 있습니다. 이 섹션에서는 주요 객체들에 대해 간단히 살펴보겠습니다.

이미지

이미지는 Docker 컨테이너를 생성하기 위한 읽기 전용 템플릿입니다. 여기서 컨테이너 이미지가 등장합니다. 컨테이너 이미지는 컨테이너를 실행하는 데 필요한 모든 파일, 바이너리, 라이브러리 및 구성을 포함하는 표준화된 패키지입니다.

대부분의 이미지는 다른 이미지를 기반으로 하며 추가적인 사용자 정의를 포함합니다. 예를 들어, `ubuntu` 이미지를 기반으로 하여 Apache 웹 서버와 애플리케이션, 필요한 구성을 추가한 새로운 이미지를 만들 수 있습니다.

이미지는 직접 만들거나 다른 사람이 레지스트리에 공유한 것을 사용할 수 있습니다. 이미지를 직접 만들려면 Dockerfile을 작성해야 하는데, 이는 이미지 생성에 필요한 단계를 정의하는 간단한 문법을 사용합니다. Dockerfile의 각 명령은 이미지에 새로운 레이어를 생성하며, 변경 사항이 있을 때는 해당 레이어만 다시 빌드됩니다. 이러한 특성 덕분에 Docker 이미지는 다른 가상화 기술에 비해 더 가볍고 빠릅니다.

이미지에는 두 가지 중요한 원칙이 있습니다.

1. 이미지는 변경할 수 없습니다. 이미지를 만든 후에는 수정할 수 없습니다. 새 이미지를 만들거나 기존 이미지 위에 변경 사항을 추가하는 것만 가능합니다.

2. 컨테이너 이미지는 여러 레이어로 구성됩니다. 각 레이어는 파일을 추가, 제거 또는 수정하는 파일 시스템 변경 사항 집합을 나타냅니다.

컨테이너

컨테이너는 이미지의 실행 가능한 인스턴스입니다. Docker API나 CLI를 통해 컨테이너를 생성, 시작, 중지, 이동, 삭제할 수 있으며, 네트워크 연결, 스토리지 연결, 현재 상태를 기반으로 한 새로운 이미지 생성도 가능합니다.

> 컨테이너란: 앱의 각 구성 요소를 위한 격리된 프로세스
> 컨테이너는 실행에 필요한 모든 파일을 갖춘 고립된 프로세스입니다.

컨테이너는 기본적으로 다른 컨테이너와 호스트 시스템으로부터 격리되어 있습니다. 사용자는 컨테이너의 네트워크, 스토리지, 기타 하위 시스템의 격리 수준을 직접 제어할 수 있습니다. 컨테이너의 장점은 다음과 같습니다.

- 독립형. 각 컨테이너는 호스트 머신에 미리 설치된 종속성에 의존하지 않고도 작동하는 데 필요한 모든 것을 갖추고 있습니다.
- 격리됨. 컨테이너는 격리된 상태로 실행되므로 호스트와 다른 컨테이너에 미치는 영향을 최소화하여 애플리케이션의 보안을 강화합니다.
- 독립적입니다. 각 컨테이너는 독립적으로 관리됩니다. 컨테이너 하나를 삭제해도 다른 컨테이너에는 영향을 미치지 않습니다.
- 이동성이 뛰어납니다. 컨테이너는 어디에서나 실행할 수 있습니다! 개발 머신에서 실행되는 컨테이너는 데이터 센터나 클라우드 어디에서나 동일하게 작동합니다!

컨테이너는 기본 이미지와 생성 시 지정된 구성 옵션에 의해 정의됩니다. 컨테이너를 삭제하면 영구 저장소에 저장되지 않은 모든 상태 변경사항이 함께 제거됩니다.

1.4.3 Docker 설치 실습 (Ubuntu 기준)

Ubuntu에서 Docker를 설치하는 과정은 다음과 같습니다:

1. APT 패키지 인덱스를 업데이트하고 필요한 패키지를 설치합니다:

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
```

2. Docker의 공식 GPG 키를 추가합니다:

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --de
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

3. Docker 리포지토리를 설정합니다:

```
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/d
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

또는 다음과 같이 간편하게 설치할 수 있습니다.

1. apt 를 이용한 설치 - 간편

```
- sudo apt install -y docker.io
- sudo usermod -aG docker $USER
- newgrp docker
```

두 설치 방법의 주요 차이점:

- **첫 번째 방법 (GPG 키 사용):** Docker의 공식 리포지토리를 사용하며, 보안성이 더 높고 최신 버전을 받을 수 있습니다. GPG 키 검증을 통해 패키지의 신뢰성을 보장합니다.
- **두 번째 방법 (apt install):** Ubuntu의 기본 리포지토리를 사용하는 간편한 방법입니다. 설치는 더 쉽지만, 최신 버전을 받지 못할 수 있고 보안 검증 단계가 생략됩니다.

권장사항: 프로덕션 환경이나 보안이 중요한 경우 첫 번째 방법을 사용하고, 테스트나 학습 목적이라면 두 번째 방법도 충분합니다.

1.4.4 컨테이너 실행: run

1. 기본 컨테이너 실행

- **명령어**: docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

- **예제**:
  - docker run ubuntu:24.04 (최신 Ubuntu 이미지로 컨테이너 실행)
  - docker run -it ubuntu:24.04 sh (대화형 셸 실행)
- **설명**: 태그(:24.04)로 버전 지정, 다이제스트(@sha256:...)로 고유 이미지 실행 가능.

---

2. 태그 vs 다이제스트

- **태그 실행**:
  - docker run ubuntu:24.04
  - 간단하지만 태그는 변경될 수 있음 (개발용 추천).
- **다이제스트 실행**:
  - docker run ubuntu@sha256:9cacb71397b...
  - 고유 이미지 보장 (프로덕션용 추천).

---

3. 주요 옵션

- **이름 지정**: docker run --name my_container ubuntu
- **백그라운드 실행**: docker run -d nginx
- **포그라운드 상호작용**: docker run -it ubuntu sh
- **로그 확인**: docker logs my_container
- **재연결**: docker attach my_container

---

4. 네트워킹

- **사용자 지정 네트워크 생성**:
  - docker network create my-net
  - docker run -d --name web --network my-net nginx:alpine
  - docker run -it --network my-net busybox ping web
- **설명**: 동일 네트워크 내 컨테이너는 이름으로 통신 가능.

---

5. 파일 시스템 마운트

- **볼륨 마운트**:
  - docker run --mount source=my_volume,target=/app busybox echo "test" > /app/file.txt

    - docker run --mount source=my_volume,target=/app busybox cat /app/file.txt
- **바인드 마운트**:
    - docker run -it --mount type=bind,source=./host_dir,target=/app busybox
- **설명**: 볼륨은 영구 저장, 바인드는 호스트와 공유.

---

6. 리소스 제약

- **메모리 제한**: docker run -it -m 300M ubuntu:24.04
- **CPU 제한**: docker run -it --cpus=0.5 ubuntu:24.04
- **설명**: -m으로 메모리, --cpus로 CPU 사용량 조정.

---

7. 환경 변수 설정

- **명령어**: docker run -e MY_VAR=value ubuntu env
- **예제**:
    - export TODAY=Wednesday
    - docker run -e TODAY ubuntu env
- **설명**: -e로 컨테이너 내 변수 설정.

---

8. 기본값 재정의

- **진입점 재정의**: docker run -it --entrypoint /bin/sh ubuntu
- **작업 디렉토리**: docker run -w /my_dir ubuntu pwd
- **설명**: 이미지 기본 설정(ENTRYPOINT, WORKDIR)을 런타임에 변경.

---

9. 종료 코드 확인

- **예제**:
    - docker run busybox /etc; echo $? (126: 실행 불가)
    - docker run busybox foo; echo $? (127: 명령 없음)
- **설명**: 종료 코드로 실행 상태 확인.

---

10. 실습 예제

1. **Nginx 백그라운드 실행**:

- docker run -d --name web nginx
- docker ps (확인)
- docker logs web (로그 확인)

2. **Ubuntu 셸 실행**:
   - docker run -it --name ubuntu_test ubuntu:24.04 sh
   - 컨테이너 종료 후: docker rm ubuntu_test
3. **볼륨 데이터 유지**:
   - docker run --mount source=test_vol,target=/data busybox echo "persistent" > /data/test.txt
   - docker run --mount source=test_vol,target=/data busybox cat /data/test.txt

컨테이너 실행: run -원문

1.4.5. `docker run` , `ps` , `exec` , `rm` , `logs` 등 컨테이너 명령어 실습

- docker ps : docker ps는 현재 실행 중인 컨테이너 목록을 보여줍니다.

```
aa@choidoder-notebook:~$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED        STATUS         PORTS     NAMES
206df7ad5602   osrf/ros:humble-desktop  "/ros_entrypoint.sh …"   23 hours ago   Up 5 minutes             sad_bose
aa@choidoder-notebook:~$
```

- docker exec - 컨테이너 내부에서 명령 실행

```
docker exec -it <컨테이너_이름> /bin/bash

- i: 인터랙티브 모드 (입력 가능)
- t: 터미널 할당
- /bin/bash: 실행할 쉘 명령어
```

- docker rm - 컨테이너 삭제

```
docker stop <컨테이너_이름>
docker rm <컨테이너_이름>
docker rm -f <컨테이너_이름>
- 실행 중인 컨테이너를 강제로 삭제하려면 -f 옵션을 사용
```

컨테이너를 삭제하면 해당 컨테이너의 데이터도 사라지니, 중요한 데이터는 볼륨으로 백업하는 것이 좋습니다.

- docker logs - 컨테이너 로그 확인

```
docker logs <컨테이너_이름>
# 실시간으로 로그를 확인하려면 -f 옵션 사용
docker logs -f <컨테이너_이름>
```

1.4.6. Docker 이미지 구조 및 관리 (`pull` , `tag` , `rmi` , `prune`)

- docker pull - 이미지 다운로드
  - `docker pull alpine:latest`
- 컨테이너를 만들고 실행 환경을 조성
- `docker run -it --net=host alpine:latest /bin/sh`
- `apk add --no-cache gcc g++ cmake make`
- docker tag - 이미지에 별칭 추가

```
docker tag alpine:latest my-alpine:cmake # 새로운 태그 추가
docker images # 이미지 목록 확인
```

docker tag는 동일한 이미지에 새로운 이름을 붙여 관리 편의성을 높입니다. 예를 들어, CMake 환경임을 나타내는 태그를 사용할 수 있습니다. Image id 를 확인 하면 똑같은 ID 라는 것을 알 수 있다.

- docker rmi - 이미지 삭제

```
docker images
docker rmi my-alpine:cmake
docker stop <컨테이너_이름> # 사용중이라면 중지 해야 함.
```

- docker prune - 사용하지 않는 리소스 정리

```
docker system prune
docker image prune
docker image prune -a
```

prune은 실행 중이지 않은 리소스를 정리하며, -a 옵션은 태그 없는 이미지뿐 아니라 사용되지 않는 모든 이미지를 제거합니다.

- CMake 프로젝트 실습

```
docker run -it --net=host -v $(pwd)/app:/app -w /app alpine:latest /bin/sh
# -v: 로컬 디렉토리를 컨테이너에 마운트
# -w: 작업 디렉토리 지정
# apk add --no-cache gcc g++ cmake make
```

- 간단한 프로그램 만들기

```
#include <iostream>
int main() {
    std::cout << "Hello, Alpine Docker!" << std::endl;
    return 0;
}
```

```
cmake_minimum_required(VERSION 3.10)
project(HelloAlpine)
add_executable(hello hello.cpp)
```

- 추가 팁 : 이미지를 새로 만들어서 운영하면 편한다.

```
FROM alpine:latest
RUN apk add --no-cache gcc g++ cmake make git vi \
    && git clone https://github.com/catchorg/Catch2.git \
    && cd Catch2 \
    && cmake -Bbuild -S. -DBUILD_TESTING=OFF \
    && make -C build install
WORKDIR /app
CMD ["/bin/sh"]
```

-
    - build : `docker build --network host -t my-image .`
    - run: `docker run -it --net=host -v $(pwd)/app:/app -w /app my-image:latest /bin/sh`

- Catch 를 활용한 테스트 방법

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch_all.hpp>

int add(int a, int b)
{
    return a + b;
}

TEST_CASE("Addition works", "[add]")
{
    REQUIRE(add(2, 3) == 5);
    REQUIRE(add(-1, 1) == 0);
    REQUIRE(add(0, 0) == 0);
}
```

```
cmake_minimum_required(VERSION 3.10)
project(HelloAlpine)

# CTest 활성화화
```

```
include(CTest)
enable_testing()

# 메인 실행 파일
add_executable(hello hello.cpp)

# Catch2 라이브러리 찾기
find_package(Catch2 REQUIRED)

include(Catch)  # Catch2 테스트 통합 활성화
add_executable(tests test/test.cpp)  # 테스트 설정
target_link_libraries(tests PRIVATE Catch2::Catch2WithMain)
catch_discover_tests(tests)  # Catch2의 테스트 자동 검색 기능 사용
# 테스트 설정

# 테스트 등록
add_test(NAME AllTests COMMAND tests)
```

```
mkdir build && cd build
cmake ..
make
make test
./hello
```

1.4.9. Docker compose 란 무엇인가?

이제는 데이터베이스, 메시지 큐, 캐시 등 다양한 서비스를 실행하는 복잡한 작업을 수행하고 싶어질 것입니다. 이런 경우 모든 것을 단일 컨테이너에 설치할까요, 아니면 여러 컨테이너로 나눌까요? 여러 컨테이너를 사용한다면 이들을 어떻게 연결해야 할까요?

컨테이너의 핵심 모범 사례는 각 컨테이너가 한 가지 작업만 잘 수행해야 한다는 것입니다. 물론 예외는 있지만, 하나의 컨테이너에 여러 작업을 넣는 것은 피하는 것이 좋습니다.

여러 `docker run` 명령으로 다수의 컨테이너를 실행할 수 있습니다. 하지만 곧 네트워크 관리와 컨테이너 연결에 필요한 플래그 설정 등 많은 작업이 필요하다는 것을 알게 됩니다. 더구나 작업 완료 후의 정리도 복잡해집니다.

Docker Compose를 사용하면 모든 컨테이너와 설정을 하나의 YAML 파일로 정의할 수 있습니다. 이 파일을 코드 저장소에 포함하면, 저장소를 복제한 사용자는 단일 명령으로 Docker Compose를 실행할 수 있습니다.

Compose는 선언적 도구라는 점이 중요합니다. 설정을 정의하고 실행하기만 하면 됩니다. 매번 처음부터 다시 설정할 필요가 없습니다. 변경 사항이 있을 때 `docker compose up`을 다시 실행하면, Compose가 자동으로 변경 사항을 감지하고 적절히 적용합니다.

도커컴포즈의 설치 - v2

```
sudo apt install docker-compose-v2
```

plugin

도커에 제공하는 패키지 주소를 포함해야 한다.

```
sudo apt-get install -y apt-transport-https ca-certificates curl software-pro
perties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dea
rmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo   "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyri
ngs/docker-archive-keyring.gpg] https://download.docker.com/linux/ubunt
u \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /
dev/null
```

1. docker compose 를 용한 간단한 예제

flask 를 이용한 python 프로그램을 만든다.

```
import time
```

```
import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return f'Hello World! I have been seen {count} times.\n'
```

dockerfile 을 생성 해서 간단한 컨테이너를 설정한다.

```
# syntax=docker/dockerfile:1
FROM python:3.10-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run", "--debug"]
```

compose.yaml 을 만들어서 파이썬과 redis 를 이용한 서비스가 실행 되도록 만든다.

```
version: '3.8'

services:
  web:
    build:
      context: .
      # network: host
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

💡 apine 기능으로 apk add 를 할 때 인터넷이 설정 되도록 network: host 옵션을 넣는다.

localhost:8000 으로 접속하면 실행 화면이 보인다.

2. Compose Watch 의 기능

compose.yaml 에 watch 기능을 추가로 넣을 수 있다.

```
services:
  web:
    build:
      context: .
      network: host
    ports:
      - "8000:5000"
    develop:
      watch:
        - action: sync
```

```
        path: .
        target: /code
  redis:
    image: "redis:alpine"
```

watch 기능을 활성화 하려면 명령어를

`docker compose watch` 로 실행 해야 한다.

3. 서비스 분할

여러 개의 Compose 파일을 사용하면 다양한 환경이나 워크플로에 맞게 Compose 애플리케이션을 사용자 지정할 수 있습니다. 이는 수십 개의 컨테이너를 사용하고 여러 팀에 소유권이 분산된 대규모 애플리케이션에 유용합니다.

```
services:
  redis:
    image: "redis:alpine"
```

```
include:
   - infra.yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
    develop:
      watch:
        - action: sync
          path: .
          target: /code
```

hits 기능을 영속성 있게 하려면 volume 을 사용하면 된다.

```
services:
  redis:
    image: "redis:alpine"
```

```
    volumes:
      - redis-data:/data
    command: redis-server --appendonly yes --save 60 1

volumes:
  redis-data:
```

참고

https://docs.docker.com/get-started/resources/

https://docs.docker.com/engine/storage/