



docker 설치 및 사용

Status Draft

Ubuntu 24.04 LTS (Noble Numbat) 환경에서의 Docker 설치 및 활용 종합 매뉴얼

서론

Docker는 애플리케이션을 신속하게 구축, 테스트 및 배포할 수 있도록 하는 개방형 플랫폼이다. Docker는 소프트웨어를 컨테이너라는 표준화된 단위로 패키징하며, 이 컨테이너에는 라이브러리, 시스템 도구, 코드, 런타임 등 소프트웨어를 실행하는 데 필요한 모든 것이 포함되어 있다. 이를 통해 개발자는 환경에 구애받지 않고 애플리케이션을 동일하게 실행할 수 있으며, 인프라와 애플리케이션을 분리하여 운영 효율성을 극대화할 수 있다.

본 매뉴얼은 Ubuntu 24.04 LTS (Noble Numbat) 환경에서 Docker를 설치하고, 기본적인 명령어를 통해 컨테이너와 이미지를 관리하며, Python 애플리케이션을 컨테이너화하는 방법, 일반적인 문제 해결 방안, 그리고 Docker 보안 강화 전략에 이르기까지 포괄적인 내용을 다룬다. 각 단계별 상세한 설명과 예제를 통해 사용자가 Docker를 효과적으로 이해하고 활용할 수 있도록 지원하는 것을 목표로 한다.

1. Ubuntu 24.04 Docker 설치 준비

Docker Engine을 Ubuntu 24.04에 성공적으로 설치하기 위해서는 몇 가지 사전 준비 사항을 확인하고 충족해야 한다.

1.1. 시스템 요구 사항 및 사전 확인

- **운영체제:** Ubuntu Noble 24.04 (LTS)의 64비트 버전이 필요하다. Docker Engine은 x86_64 (amd64), armhf, arm64, s390x, ppc64le (ppc64el) 아키텍처를 지원한다. Linux Mint와 같은 Ubuntu 파생 배포판에서의 설치는 공식적으로 지원되지 않을 수 있다.
- **커널 버전:** Docker는 Linux 커널의 특정 기능을 활용하므로, 최신 상태의 커널을 사용하는 것이 권장된다. 일반적으로 Ubuntu 24.04 LTS의 기본 커널은 Docker 실행에 충분하다.
- **저장 공간:** Docker 이미지와 컨테이너는 디스크 공간을 차지하므로, 충분한 여유 공간을 확보해야 한다. `/var/lib/docker/` 디렉토리에 주로 데이터가 저장된다.
- **방화벽 제한 사항:** Docker를 사용하여 컨테이너 포트를 노출할 경우, `ufw` 또는 `firewalld`와 같은 방화벽 규칙을 우회할 수 있다. Docker는 `iptables-nft` 및 `iptables-legacy`와 호환되며, `nft`로 생성된 방화벽 규칙은 지원하지 않는다. 방화벽 규칙은 `iptables` 또는 `ip6tables`로 생성하고 `DOCKER-USER` 체인에 추가해야 한다.

1.2. 기존 Docker 버전 또는 충돌 가능 패키지 제거

새로운 버전의 Docker Engine을 설치하기 전에, 시스템에 이전 버전의 Docker나 비공식 Docker 패키지가 설치되어 있다면 이를 제거하는 것이 좋다. 충돌을 유발할 수 있는 패키지에는 `docker.io`, `docker-doc`, `docker-compose`, `docker-compose-v2`, `podman-docker` 등이 있다. 또한, Docker Engine이 자체적으로 `containerd.io`와 `runc`를 번들로 제공하므로, 이들이 별도로 설치되어 있다면 제거해야 한다.

다음 명령어를 사용하여 관련 패키지를 제거할 수 있다:

```
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc;
do sudo apt-get remove $pkg; done
```

이 명령어는 패키지를 제거하지만, /var/lib/docker/에 저장된 기존 이미지, 컨테이너, 볼륨, 네트워크는 자동으로 삭제되지 않는다. 완전히 새로 설치하고자 한다면, Docker 제거 후 해당 디렉토리를 수동으로 삭제해야 할 수 있다.

2. Ubuntu 24.04에 Docker Engine 설치 (APT 저장소 방식)

Ubuntu에 Docker Engine을 설치하는 가장 권장되는 방법은 Docker의 공식 APT 저장소를 사용하는 것이다. 이 방법을 사용하면 설치 및 업데이트 관리가 용이하다.

2.1. Docker 공식 GPG 키 추가

GPG 키는 다운로드하는 패키지의 진위성을 확인하여 소프트웨어 무결성을 보장하는 데 필수적이다. 이 과정은 신뢰할 수 있는 소스에서 Docker를 설치하고 있음을 시스템에 알리는 역할을 한다. 만약 GPG 키 검증 없이 패키지를 설치한다면, 중간자 공격(Man-in-the-Middle attack)이나 악의적으로 수정된 소프트웨어에 노출될 위험이 있다. 따라서 GPG 키 추가는 보안상 매우 중요한 단계이다.

1. 먼저, 시스템 패키지 목록을 업데이트하고 HTTPS를 통해 저장소를 사용하는 데 필요한 패키지를 설치한다:

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

2. GPG 키를 저장할 디렉토리를 생성하고 적절한 권한을 설정한다:

```
sudo install -m 0755 -d /etc/apt/keyrings
```

3. Docker의 공식 GPG 키를 다운로드하여 지정된 경로에 저장한다:

```
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
```

4. 다운로드한 GPG 키 파일에 대해 모든 사용자가 읽을 수 있도록 권한을 변경한다:

```
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

2.2. Docker APT 저장소 설정

GPG 키를 추가한 후, APT가 Docker 패키지를 찾을 수 있도록 Docker의 공식 저장소를 시스템의 소스 목록에 추가해야 한다. 공식 저장소를 사용하면 Docker 팀이 직접 제공하고 관리하는 최신 버전의 안정적이고 안전한 패키지를 받을 수 있다. 운영체제의 기본 저장소에 포함된 Docker 패키지는 오래되었거나 수정되었을 가능성이 있으므로, 공식 저장소 사용은 최신 기능과 보안 패치를 보장받는 최선의 방법이다.

1. 다음 명령어를 실행하여 Docker 저장소를 APT 소스 목록에 추가한다. 이 명령어는 시스템 아키텍처와 Ubuntu 코드를 자동으로 감지하여 적절한 저장소를 설정한다:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://dow
```

2. 저장소를 추가한 후, 다시 패키지 목록을 업데이트하여 APT가 새로운 Docker 저장소의 패키지 정보를 가져오도록 한다:

```
sudo apt-get update
```

2.3. Docker Engine 패키지 설치

저장소 설정이 완료되면 Docker Engine 및 관련 패키지를 설치할 수 있다.

1. 최신 버전의 Docker Engine, CLI, containerd, Docker Compose 플러그인, Buildx 플러그인을 설치하려면 다음 명령어를 실행한다:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

여기서 각 패키지의 역할은 다음과 같다:

- docker-ce: Docker Community Edition, 즉 Docker 엔진 자체이다.
- docker-ce-cli: Docker 명령줄 인터페이스(CLI)를 제공한다.
- containerd.io: 컨테이너 런타임으로, 컨테이너의 생명주기를 관리한다.
- docker-buildx-plugin: 다중 플랫폼 빌드 등 향상된 이미지 빌드 기능을 제공한다.
- docker-compose-plugin: YAML 파일을 사용하여 다중 컨테이너 Docker 애플리케이션을 관리할 수 있게 한다.

1. 특정 버전의 Docker Engine을 설치하려면, 먼저 사용 가능한 버전을 확인한다:

```
apt-cache madison docker-ce | awk '{ print $3 }'
```

출력된 목록에서 원하는 버전 문자열(예: 5:28.2.1-1~ubuntu.24.04~noble)을 선택한 후, 해당 버전으로 설치한다:

```
VERSION_STRING=5:28.2.1-1~ubuntu.24.04~noble
```

```
sudo apt-get install docker-ce=$VERSION_STRING docker-ce-cli=$VERSION_STRING containerd.io
docker-buildx-plugin docker-compose-plugin
```

2.4. 설치 확인

Docker Engine이 성공적으로 설치되었는지 확인하기 위해 hello-world 이미지를 실행한다. 이 이미지는 간단한 테스트용으로, 실행 시 Docker 설치가 정상적으로 완료되었음을 알리는 메시지를 출력한다.

```
sudo docker run hello-world
```

이 명령어를 실행하면 hello-world 이미지가 로컬에 없는 경우 Docker Hub에서 자동으로 다운로드되고, 컨테이너로 실행된 후 메시지를 출력하고 종료된다. "Hello from Docker!"와 유사한 메시지가 나타나면 설치가 성공적으로 완료된 것이다.

설치된 Docker 버전을 확인하려면 다음 명령어를 사용한다:

```
sudo docker --version
```

출력 예: Docker version 26.1.4, build 5650f9b

3. Docker 설치 후속 조치

Docker Engine 설치 후, 사용 편의성과 시스템 관리를 위해 몇 가지 추가 설정을 수행하는 것이 좋다.

3.1. sudo 없이 Docker 명령어 사용 설정 (비루트 사용자 접근 활성화)

기본적으로 Docker 명령어는 루트 권한을 요구하므로 sudo를 함께 사용해야 한다. 매번 sudo를 입력하는 번거로움을 줄이고자 한다면, 현재 사용자를 docker 그룹에 추가할 수 있다. 이 설정은 편의성을 제공하지만, 보안적 측면을 고려해야 한다. docker 그룹에 속한 사용자는 Docker 데몬과 직접 통신할 수 있게 되며, Docker 데몬은 루트 권한으로 실행되므로 이는 해당 사용자에게 사실상 루트와 동등한 권한을 부여하는 것과 같다. 특히 다중 사용자 환경에서는 이 설정이 보안 위험을 증가시킬 수 있으므로 신중하게 결정해야 한다. 단일 사용자 환경이나 개발 환경에서는 편의를 위해 설정하는 경우가 많다.

1. docker 그룹이 시스템에 존재하지 않을 경우 생성한다. -f 옵션은 그룹이 이미 존재하더라도 오류를 발생시키지 않는다:

```
sudo groupadd -f docker
```

2. 현재 로그인된 사용자를 docker 그룹에 추가한다. 다른 사용자를 추가하려면 \$USER 대신 해당 사용자명을 입력한다:

```
sudo usermod -aG docker $USER
```

또는 특정 사용자를 추가하려면:

```
sudo usermod -aG docker [username]
```

3. 그룹 변경 사항을 현재 세션에 적용한다. 이를 위해서는 `newgrp docker` 명령어를 실행하거나, 시스템에서 로그아웃 후 다시 로그인해야 한다. `newgrp` 명령어는 새로운 셸을 시작하여 변경된 그룹 멤버십을 즉시 적용한다.

```
newgrp docker
```

이후에는 `sudo` 없이 `docker ps`와 같은 Docker 명령어를 실행할 수 있다.

```
docker ps
```

실행 중인 컨테이너가 없다면 빈 목록이 출력될 것이다.

3.2. Docker 서비스 관리

Docker는 시스템 서비스로 실행되므로 `systemctl` 명령어를 사용하여 관리할 수 있다.

- **부팅 시 Docker 자동 시작 활성화:** 시스템이 시작될 때 Docker 서비스가 자동으로 실행되도록 설정한다.

```
sudo systemctl enable docker
```

- **Docker 서비스 상태 확인:** Docker 서비스의 현재 실행 상태를 확인한다.

```
sudo systemctl status docker
```

출력에서 `active (running)` 상태를 확인하면 정상적으로 실행 중인 것이다.

- **Docker 서비스 시작:** 중지된 Docker 서비스를 시작한다.

```
sudo systemctl start docker
```

- **Docker 서비스 중지:** 실행 중인 Docker 서비스를 중지한다.

```
sudo systemctl stop docker
```

- **Docker 서비스 재시작:** Docker 서비스를 재시작한다. 설정 변경 후 적용하거나 문제가 발생했을 때 사용한다.

```
sudo systemctl restart docker
```

4. Docker 기본 명령어 활용

Docker 설치 후에는 다양한 명령어를 사용하여 컨테이너와 이미지를 관리할 수 있다. 다음은 핵심적인 기본 명령어들이다.

4.1. docker ps: 실행 중인 컨테이너 목록 확인

`docker ps` 명령어는 현재 실행 중인 컨테이너의 목록을 보여준다. (User Query)

```
docker ps
```

모든 컨테이너(중지된 컨테이너 포함)를 보려면 `-a` 또는 `--all` 옵션을 추가한다:

```
docker ps -a
```

출력에는 컨테이너 ID, 사용된 이미지, 실행된 명령어, 생성 시간, 상태, 포트, 이름 등의 정보가 포함된다.

4.2. docker stop: 실행 중인 컨테이너 중지

`docker stop [container_name_or_id]` 명령어는 실행 중인 컨테이너를 중지시킨다. (User Query) 이 명령어는 컨테이너의 주 프로세스에 `SIGTERM` 신호를 보내 정상적으로 종료하도록 요청한다. 일정 시간(기본 10초) 내에 종료되지 않으면 `SIGKILL` 신호를 보내 강제 종료한다.

```
docker stop my_container
```

(User Query) 여러 컨테이너를 한 번에 중지시킬 수도 있다:

```
docker stop container1 container2
```

모든 실행 중인 컨테이너를 중지하려면 다음 명령어를 사용할 수 있다:

```
docker stop $(docker ps -q)
```

여기서 `docker ps -q`는 실행 중인 컨테이너의 ID만 나열한다.

4.3. docker start: 중지된 컨테이너 재시작

`docker start [container_name_or_id]` 명령어는 중지된 컨테이너를 다시 시작시킨다. (User Query)

```
docker start my_stopped_container
```

(User Query)

4.4. docker logs: 컨테이너 로그 확인

`docker logs [container_name_or_id]` 명령어는 컨테이너 내부에서 발생하는 로그(표준 출력 및 표준 에러)를 확인할 때 사용한다. (User Query)

```
docker logs my_running_container
```

실시간으로 로그를 계속 보려면 `-f` 또는 `--follow` 옵션을 추가한다:

```
docker logs -f my_running_container
```

최근 N개의 로그 라인만 보려면 `--tail N` 옵션을 사용한다:

```
docker logs --tail 50 my_running_container
```

4.5. docker rm: 컨테이너 삭제

`docker rm [container_name_or_id]` 명령어는 하나 이상의 컨테이너를 삭제한다. (User Query) 컨테이너는 중지된 상태여야 삭제할 수 있다.

```
docker rm my_stopped_container
```

실행 중인 컨테이너를 중지하지 않고 즉시 삭제하려면 `-f` 또는 `--force` 옵션을 추가한다. 이 옵션은 컨테이너에 SIGKILL 신호를 보내 강제 종료 후 삭제한다. (User Query)

```
docker rm -f my_running_container
```

모든 중지된 컨테이너를 삭제하려면 다음 명령어를 사용할 수 있다:

```
docker rm $(docker ps -a -f status=exited -q)
```

또는 `docker container prune` 명령어를 사용하면 중지된 모든 컨테이너를 한 번에 삭제할 수 있다 (사용자 확인 후 진행):

```
docker container prune
```

4.6. docker rmi: 이미지 삭제

`docker rmi [image_name_or_id]` 명령어는 로컬 시스템에서 하나 이상의 Docker 이미지를 삭제한다. (User Query)

```
docker rmi my_image:tag
```

이미지가 실행 중인 컨테이너나 다른 이미지의 부모 이미지로 사용되고 있다면 삭제되지 않는다. 강제로 삭제하려면 `-f` 또는 `--force` 옵션을 사용할 수 있지만, 주의해야 한다.

```
docker rmi -f my_image:tag
```

태그가 없는 이미지(댕글링 이미지, dangling images)만 삭제하려면 다음 명령어를 사용한다:

```
docker image prune
```

모든 이미지를 삭제하려면 (사용 중이지 않은 이미지 대상):

```
docker rmi $(docker images -a -q)
```

5. Docker 컨테이너 실행: docker run 상세 분석

docker run 명령어는 Docker에서 이미지를 기반으로 새로운 컨테이너를 생성하고 실행하는 핵심 명령어이다. 이 명령어는 다양한 옵션을 통해 컨테이너의 동작 방식을 세밀하게 제어할 수 있다. (User Query)

5.1. docker run 기본 구문

docker run 명령어의 일반적인 형태는 다음과 같다:

docker run IMAGE

- ``: 컨테이너의 동작을 설정하는 다양한 옵션들이다. (User Query)
- IMAGE: 컨테이너를 생성할 기반 Docker 이미지의 이름이다. 태그(:TAG)를 생략하면 기본적으로 :latest 태그가 사용된다. 이미지 다이제스트(@DIGEST)를 사용하여 특정 버전의 이미지를 정확히 지정할 수도 있다. (User Query)
- ``: 컨테이너가 시작될 때 내부에서 실행할 명령어이다. 이미지에 기본 실행 명령어가 정의되어 있다면 생략 가능하다. (User Query)
- : 에 전달할 인자들이다. (User Query)

예를 들어, busybox 이미지를 사용하여 "Hello, World!"를 출력하는 간단한 컨테이너를 실행하는 명령어는 다음과 같다:

```
docker run busybox echo "Hello, World!"
```

5.2. docker run 주요 옵션

docker run 명령어와 함께 자주 사용되는 주요 옵션들은 다음과 같다. (User Query)

- **d, --detach**: 컨테이너를 백그라운드에서 실행한다 (데몬 모드). (User Query) 터미널에 연결되지 않고 백그라운드에서 실행되므로 웹 서버나 데이터베이스와 같은 서비스를 실행할 때 유용하다.

```
docker run -d nginx
```

이 옵션은 장기 실행 서비스에 적합하며, 컨테이너 ID만 출력하고 터미널 제어권을 즉시 반환한다. 반면, 대화형 세션이 필요한 경우 (예: 셸 접속) 이 옵션은 적합하지 않다.

- **p, --publish HOST_PORT:CONTAINER_PORT**: 호스트와 컨테이너 간의 포트를 연결(매핑)한다. (User Query) 호스트의 HOST_PORT로 들어오는 트래픽을 컨테이너 내부의 CONTAINER_PORT로 전달한다. 예: -p 8080:80은 호스트의 8080번 포트를 컨테이너의 80번 포트로 연결한다. (User Query)

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

(User Query) 이 포트 매핑 기능은 컨테이너 내부에서 실행 중인 애플리케이션을 외부 네트워크에 노출시키는 핵심 방법이다. 호스트 포트는 호스트 머신에서 사용 가능한 포트여야 하며, 컨테이너 포트는 컨테이너 내 애플리케이션이 수신 대기하는 포트이다. 이를 통해 동일한 컨테이너 이미지를 여러 개 실행하면서 각각 다른 호스트 포트에 매핑하여 서비스 충돌 없이 여러 인스턴스를 운영할 수 있다.

- **v, --volume HOST_PATH:CONTAINER_PATH 또는 VOLUME_NAME:CONTAINER_PATH**: 호스트와 컨테이너 간의 디렉토리나 파일을 공유(마운트)하거나, 명명된 볼륨을 컨테이너에 연결한다. (User Query)
- **바인드 마운트 (Bind Mounts)**: 호스트의 특정 경로(HOST_PATH)를 컨테이너의 특정 경로(CONTAINER_PATH)에 직접 연결한다. 호스트 파일 시스템의 변경 사항이 컨테이너에 반영되고, 그 반대도 가능하다. 예: 호스트의 /opt/data 디렉토리를 컨테이너의 /app 디렉토리와 공유한다.

```
docker run -v /opt/data:/app myapp
```

(User Query)

- **명명된 볼륨 (Named Volumes)**: Docker가 관리하는 영구 저장 공간인 명명된 볼륨(VOLUME_NAME)을 컨테이너의 특정 경로(CONTAINER_PATH)에 연결한다. 데이터베이스 데이터나 애플리케이션 상태를 영구적으로 저장하는 데 권장된다. 예: my_volume이라는 명명된 볼륨을 컨테이너의 /data 경로에 마운트한다.

```
docker run --mount source=my_volume,target=/data myimage
```

(또는 -v my_volume:/data)

- **-name CONTAINER_NAME:** 컨테이너에 고유한 이름을 지정한다. (User Query) 이름을 지정하면 컨테이너 ID 대신 이름으로 컨테이너를 참조할 수 있어 관리가 용이하다.

```
docker run -d --name my_web_server nginx
```

- **-env KEY=VALUE 또는 -e KEY=VALUE:** 컨테이너 내부에 환경 변수를 설정한다. (User Query) 애플리케이션 설정, 데이터베이스 연결 정보 등을 전달하는 데 사용된다. 예: DATABASE_URL 환경 변수를 설정한다.

```
docker run --env DATABASE_URL=mysql://user:password@localhost:5432/db myapp
```

(User Query)

- **i, --interactive:** 컨테이너의 표준 입력(STDIN)을 열린 상태로 유지한다. 주로 -t 옵션과 함께 사용된다. (User Query)
- **t, --tty:** 가상 터미널(pseudo-TTY)을 할당한다. 주로 -i 옵션과 함께 사용되어 컨테이너와 상호작용하는 셸 환경을 제공한다. (User Query) 예: Ubuntu 이미지로 컨테이너를 실행하고 bash 셸에 접속한다.

```
docker run -it ubuntu /bin/bash
```

- **o** 옵션은 컨테이너 내부에서 명령어를 직접 입력하고 결과를 확인해야 하는 디버깅이나 개발 작업에 필수적이다. **-d** 옵션과는 상반되는 용도로, 포그라운드에서 대화형 세션을 유지한다.

5.3. docker run 추가 옵션

자주 사용되는 옵션 외에도 다음과 같은 유용한 옵션들이 있다. (User Query)

- **-rm:** 컨테이너가 종료될 때 자동으로 삭제한다. 일회성 작업이나 테스트용 컨테이너에 유용하다.

```
docker run --rm busybox echo "Task complete, removing container"
```

- **-link OTHER_CONTAINER_NAME:ALIAS:** 다른 컨테이너와 네트워크 연결을 설정한다 (레거시 기능, 사용자 정의 네트워크 사용 권장). (User Query)

- **e, --env-file FILE_PATH:** 환경 변수가 정의된 파일로부터 환경 변수를 읽어와 컨테이너에 설정한다. (User Query)

```
docker run --env-file ./env.list myapp
```

- **-restart RESTART_POLICY:** 컨테이너가 종료되었을 때 재시작 정책을 설정한다. (User Query) 예: --restart always (항상 재시작), --restart on-failure (오류로 종료 시 재시작).

```
docker run -d --restart always my_reliable_service
```

- **-network NETWORK_NAME:** 컨테이너가 속할 네트워크를 지정한다. (User Query) 기본 브리지 네트워크 대신 사용자 정의 네트워크에 연결할 때 사용한다.

```
docker run --network my_custom_network myapp
```

- **-add-host HOSTNAME:IP_ADDRESS:** 컨테이너 내부의 /etc/hosts 파일에 호스트 이름과 IP 주소 매핑을 추가한다.

- **w, --workdir PATH:** 컨테이너 내부의 작업 디렉토리를 설정한다. COMMAND가 이 디렉토리에서 실행된다. (docker exec에서도 사용)

docker run 명령어는 매우 강력하며, 위에 언급된 옵션 외에도 다양한 옵션들이 존재한다. docker run --help 명령어를 통해 모든 옵션과 설명을 확인할 수 있다.

6. Docker 컨테이너 관리 심화

컨테이너를 실행한 후에는 컨테이너 내부와 상호작용하거나, 컨테이너의 생명주기를 보다 세밀하게 제어해야 하는 경우가 발생한다.

6.1. 실행 중인 컨테이너에 명령어 전달: docker exec

docker exec 명령어는 이미 실행 중인 컨테이너 내부에서 새로운 명령어를 실행하는 데 사용된다. (User Query) 이는 컨테이너를 중지하거나 새로 시작하지 않고도 내부 상태를 확인하거나, 파일을 수정하거나, 추가적인 프로세스를 실행할 수 있게 해준다. docker run이 새로운 컨테이너를 생성하여 명령어를 실행하는 반면, docker exec는 기존 컨테이너의 네임스페이스 내에서 작동한다. 이 명령어는 컨테이너의 주 프로세스(PID 1)가 실행 중일 때만 작동하며, docker exec로 실행된 명령어는 컨테이너 재시작 시 자동으로 다시 실행되지 않는다.

기본 구문:

docker exec CONTAINER COMMAND

가장 일반적인 사용 사례는 실행 중인 컨테이너의 셸에 접근하는 것이다:

docker exec -it [container_name_or_id] /bin/bash

(User Query) 또는 Alpine Linux와 같이 /bin/bash가 없는 최소 이미지의 경우 /bin/sh를 사용한다:

docker exec -it [container_name_or_id] sh

- i, --interactive: 표준 입력을 열린 상태로 유지한다.
- t, --tty: 가상 터미널을 할당한다.
- d, --detach: 명령어를 백그라운드에서 실행한다.
- e, --env KEY=VALUE: exec 프로세스에 대한 환경 변수를 설정한다.
- w, --workdir PATH: exec 프로세스가 실행될 컨테이너 내부의 작업 디렉토리를 지정한다.

예를 들어, my_container라는 이름의 실행 중인 컨테이너 내부에 /tmp/execWorks라는 파일을 생성하는 명령어는 다음과 같다 (백그라운드 실행):

docker exec -d my_container touch /tmp/execWorks

6.2. 컨테이너에서 빠져나오기

- it 옵션으로 컨테이너의 셸에 접속한 경우, 다음 두 가지 방법으로 빠져나올 수 있다. (User Query)
- **컨테이너를 종료하지 않고 빠져나오기:** Ctrl+P를 누른 후 Ctrl+Q를 차례로 입력한다. (User Query) 이렇게 하면 컨테이너는 백그라운드에서 계속 실행된 상태로 현재 터미널 세션만 컨테이너에서 분리된다.
- **컨테이너를 종료하면서 빠져나오기:** 셸에서 exit 명령어를 입력하거나 Ctrl+D를 누른다. 또는 Ctrl+C를 사용하여 현재 실행 중인 포그라운드 프로세스(예: 셸)를 종료시킬 수 있으며, 이것이 컨테이너의 주 프로세스(PID 1)라면 컨테이너 자체가 종료될 수 있다. (User Query)

6.3. 컨테이너 강제 종료: docker kill

docker stop 명령어가 컨테이너에 SIGTERM 신호를 보내 정상 종료를 시도하는 반면, docker kill [container_name_or_id] 명령어는 즉시 SIGKILL 신호를 보내 컨테이너를 강제로 종료시킨다. (User Query)

docker kill my_unresponsive_container

(User Query) docker stop으로 컨테이너가 정상적으로 종료되지 않거나 즉각적인 종료가 필요할 때 사용된다.

6.4. 모든 컨테이너 중지 및 삭제

- **모든 컨테이너 중지:**

docker stop \$(docker ps -a -q)

(User Query) 이 명령어는 실행 중인 컨테이너와 중지된 컨테이너 모두를 대상으로 stop 명령을 시도한다 (docker ps -a -q는 모든 컨테이너 ID를 반환). 실제로는 실행 중인 컨테이너만 중지된다.

- **모든 컨테이너 삭제 (중지 후):** 먼저 모든 컨테이너를 중지한 후, 모든 컨테이너를 삭제한다.

docker stop \$(docker ps -a -q)

docker rm \$(docker ps -a -q)

이 두 명령어를 순차적으로 실행하면 시스템의 모든 컨테이너가 정리된다.

7. Docker 시스템 관리

Docker를 장기간 사용하다 보면 이미지, 컨테이너, 볼륨, 네트워크 등 사용하지 않는 리소스들이 누적되어 디스크 공간을 불필요하게 차지할 수 있다. Docker는 이러한 리소스를 효율적으로 관리하고 시스템 상태를 점검할 수 있는 명령어들을 제공한다.

7.1. Docker 디스크 사용량 확인: `docker system df`

`docker system df` 명령어는 Docker가 사용 중인 디스크 공간에 대한 요약 정보를 보여준다. (User Query)

`docker system df`

출력에는 이미지, 컨테이너, 로컬 볼륨, 빌드 캐시 각각에 대한 총 개수, 활성 개수, 사용 중인 크기, 그리고 회수 가능한 공간(unused and reclaimable space)이 표시된다.

더 상세한 정보를 보려면 `-v` 또는 `--verbose` 옵션을 사용한다:

`docker system df -v`

이 옵션은 각 이미지, 컨테이너, 볼륨별로 개별 크기와 회수 가능 여부 등의 자세한 정보를 제공하여 어떤 요소가 디스크 공간을 많이 차지하는지 파악하는 데 도움이 된다.

출력 형식을 JSON으로 지정하여 스크립트나 다른 도구에서 파싱하기 용이하게 만들 수도 있다:

`docker system df --format json`

7.2. 불필요한 Docker 리소스 정리: `docker system prune`

`docker system prune` 명령어는 사용하지 않는 Docker 리소스(중지된 컨테이너, 사용되지 않는 네트워크, 덩글링 이미지, 빌드 캐시)를 한 번에 정리하여 디스크 공간을 확보하는 데 매우 유용하다. (User Query)

`docker system prune`

이 명령어를 실행하면 삭제될 대상과 함께 경고 메시지가 표시되며, 계속 진행할지 여부를 묻는다.

`docker system prune` 명령어는 다음과 같은 주요 옵션들을 제공한다:

- **a, --all:** 기본적으로 `docker system prune`은 덩글링 이미지(태그가 없고 다른 이미지의 부모도 아닌 이미지)만 삭제한다. `-a` 옵션을 추가하면 사용 중이지 않은 모든 이미지(태그가 있는 이미지 포함)까지 삭제 대상으로 포함한다.

`docker system prune -a`

- **-volumes:** 기본적으로 명명된 볼륨은 데이터 손실을 방지하기 위해 삭제되지 않는다. 이 옵션을 사용하면 사용 중이지 않은 볼륨(어떤 컨테이너에도 연결되지 않은 볼륨)도 함께 삭제한다. **이 옵션은 중요한 데이터가 저장된 볼륨을 실수로 삭제할 수 있으므로 사용에 각별한 주의가 필요하다.**

`docker system prune --volumes`

- **f, --force:** 사용자 확인 프롬프트를 생략하고 즉시 정리 작업을 수행한다. 스크립트나 자동화된 작업에 유용하다.

`docker system prune -f`

- **-filter "until=timestamp":** 지정된 타임스탬프 이전에 생성된 리소스만 삭제 대상으로 필터링한다. Go duration 문자열(예: "24h", "10m")도 사용할 수 있다.

`docker system prune --filter "until=24h"`

가장 강력한 정리 명령어 중 하나는 다음과 같다:

`docker system prune --all --volumes -f`

(User Query) 이 명령어는 확인 절차 없이 중지된 모든 컨테이너, 사용되지 않는 모든 네트워크, 사용되지 않는 모든 이미지(덩글링 이미지 포함), 사용되지 않는 모든 볼륨, 그리고 빌드 캐시를 삭제한다. Docker 환경을 초기 상태에 가깝게 정리할 수 있지만, 데이터 손실 위험이 크므로 신중하게 사용해야 한다.

docker system prune 외에도 특정 리소스 타입별로 정리하는 명령어들이 존재한다:

- docker container prune: 중지된 모든 컨테이너 삭제.
- docker image prune: 댕글링 이미지 삭제. -a 옵션 사용 시 사용되지 않는 모든 이미지 삭제.
- docker volume prune: 사용되지 않는 로컬 볼륨 삭제 (기본적으로 익명 볼륨만, -a 옵션으로 명명된 볼륨 포함).
- docker network prune: 사용되지 않는 네트워크 삭제.

주기적으로 docker system df로 디스크 사용량을 확인하고, docker system prune 관련 명령어들을 적절히 사용하여 시스템을 깨끗하게 유지하는 것이 좋다.

8. Docker 컨테이너 내 Python 모듈 설치

Docker를 사용하여 Python 애플리케이션을 컨테이너화할 때, 필요한 Python 모듈(패키지)을 이미지에 포함시키거나 실행 중인 컨테이너에 설치하는 방법을 이해하는 것이 중요하다.

8.1. Dockerfile을 이용한 Python 모듈 설치 (이미지 빌드 시)

Python 애플리케이션용 Docker 이미지를 빌드할 때, Dockerfile 내에서 pip 명령어를 사용하여 의존성 모듈을 설치하는 것이 일반적이다.

- 기본 Python Dockerfile 구조 예시:

```
# Use an official Python runtime as a parent image
FROM python:3.11-slim-bullseye

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy the current directory contents into the container at /usr/src/app
# For optimization, copy requirements.txt first and install dependencies
COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code
COPY . .

# Make port 80 available to the world outside this container (if your app uses a port)
# EXPOSE 80

# Define environment variable (example)
# ENV NAME World

# Run app.py when the container launches
CMD ["python", "your_app.py"]

(Outline, based on common Python Dockerfile patterns)
```

- **requirements.txt 파일을 사용한 모듈 설치:** 가장 권장되는 방법은 requirements.txt 파일에 필요한 모든 모듈과 버전을 명시하고, 이를 Dockerfile에서 설치하는 것이다.

1. 프로젝트 루트에 requirements.txt 파일을 생성하고 의존성을 나열한다. 예시 requirements.txt:

```
SomePackage==1.0.4
AnotherPackage>=2.0
requests
```

(Outline, User Query)

2. Dockerfile에서 requirements.txt 파일을 먼저 복사하고 pip install을 실행한다.

COPY requirements.txt./

RUN pip install --no-cache-dir -r requirements.txt

(Outline) --no-cache-dir 옵션은 pip 캐시를 저장하지 않아 이미지 크기를 줄이는 데 도움이 된다. (Outline)

이러한 방식은 Docker의 빌드 캐시를 효율적으로 활용하는 데 매우 중요하다. Docker 이미지는 여러 계층(layer)으로 구성되며, Dockerfile의 각 명령어는 새로운 계층을 생성한다. Docker는 빌드 시 이전 빌드와 변경되지 않은 계층은 캐시에서 재사용한다. Python 애플리케이션의 의존성(requirements.txt)은 소스 코드 자체보다 변경 빈도가 낮다. 따라서 requirements.txt를 먼저 복사하고 의존성을 설치하는 계층을 만든 후, 애플리케이션 소스 코드를 복사하는 계층을 만들면, 소스 코드만 변경되었을 경우 의존성 설치 계층은 캐시에서 재사용되어 빌드 시간을 크게 단축할 수 있다. 만약 모든 파일을 한 번에 복사(COPY..)한 후 pip install을 실행하면, 코드의 작은 변경만으로도 의존성 설치 단계까지 캐시가 무효화되어 매번 새로 설치하게 된다. 이 원리는 Dockerfile 작성 시 빌드 최적화의 핵심 개념 중 하나이다.

- **Dockerfile에서 직접 pip install 실행 (단일 모듈 또는 특정 경우):** requirements.txt를 사용하지 않고 Dockerfile 내에서 직접 RUN 명령어로 pip를 실행하여 모듈을 설치할 수도 있다.
- 특정 버전 설치: RUN python -m pip install SomePackage==1.0.4 (User Query)
- 업그레이드: RUN python -m pip install --upgrade SomePackage (User Query)
- 최소 버전 지정: RUN python -m pip install "SomePackage>=1.0.4" (User Query)

8.2. 실행 중인 컨테이너에 Python 모듈 설치 (디버깅 및 테스트용)

이미 실행 중인 컨테이너에 임시로 Python 모듈을 설치해야 할 경우, docker exec 명령어를 사용할 수 있다.

1. 먼저, 대상 컨테이너의 셸에 접속한다:

docker exec -it <container_name_or_id> /bin/bash

(Outline)

2. 컨테이너 내부 셸에서 pip 명령어를 사용하여 모듈을 설치한다:

python -m pip install SomePackage

(Outline) 또는 특정 버전:

python -m pip install SomePackage==1.2.3

주의: 이 방법으로 설치된 모듈은 해당 컨테이너 인스턴스에만 적용되며, 컨테이너가 중지되고 삭제되면 변경 사항은 사라진다. 즉, 일시적인 변경이며 영구적이지 않다. 프로덕션 환경에서는 Dockerfile을 통해 이미지에 의존성을 포함시키는 것이 올바른 방법이다. 디버깅이나 빠른 테스트 목적으로만 사용하는 것이 좋다. 만약 이러한 변경 사항을 영구적으로 유지하고 싶다면 docker commit 명령어를 사용하여 새로운 이미지를 생성할 수 있지만, 이는 일반적으로 권장되지 않는 워크플로우이다.

9. Ubuntu에서 Docker 사용 시 일반적인 문제 해결

Docker를 사용하다 보면 몇 가지 일반적인 문제에 직면할 수 있다. 여기서는 자주 발생하는 포트 바인딩 오류와 볼륨 지정 오류에 대한 해결 방법을 설명한다.

9.1. 포트 바인딩 오류

컨테이너 실행 시 -p 또는 --publish 옵션으로 포트를 매핑할 때, 호스트의 해당 포트가 이미 사용 중이면 오류가 발생한다.

- **오류 메시지:**
- Error starting userland proxy: listen tcp 0.0.0.0:<port_number>: bind: address already in use

- Bind for 0.0.0.0:<port_number> failed: port is already allocated
- 원인 및 해결:
 1. **"address already in use"**: 이 오류는 Docker 컨테이너가 아닌 다른 프로세스가 호스트 머신의 해당 포트를 이미 사용하고 있을 때 발생한다.
 - **진단**: `sudo lsof -i:<port_number>` (예: `sudo lsof -i:8080`) 명령어를 사용하여 어떤 프로세스가 해당 포트를 점유하고 있는지 확인한다. 또는 `sudo netstat -tulnp | grep <port_number>` 또는 `sudo ss -tulnp | grep <port_number>` 명령어를 사용할 수도 있다.
 - **해결 방법 1: 충돌 프로세스 종료**: 확인된 프로세스 ID(PID)를 `sudo kill <PID>` 명령어로 종료시킨다. 이후 Docker 컨테이너를 다시 실행한다.
 - **해결 방법 2: 다른 호스트 포트 사용**: Docker 컨테이너를 실행할 때, 호스트의 다른 사용 가능한 포트로 매핑한다. 예를 들어, 8080 포트가 사용 중이라면 8081 포트로 변경한다: `docker run -p 8081:80....`
 1. **"port is already allocated"**: 이 오류는 다른 Docker 컨테이너가 이미 해당 호스트 포트를 사용하고 있을 때 발생한다.
 - **진단**: `docker ps` 명령어로 실행 중인 컨테이너 목록과 매핑된 포트를 확인하여 어떤 컨테이너가 해당 포트를 사용 중인지 찾는다.
 - **해결 방법**: 충돌하는 기존 컨테이너를 중지(`docker stop <container_id>`)하거나, 새로 실행할 컨테이너에 다른 호스트 포트를 할당한다.

오류로 인해 컨테이너 시작에 실패한 경우, 해당 컨테이너는 중지된 상태로 남아있을 수 있다. `docker rm <container_name_or_id>` 명령어로 불필요한 컨테이너를 정리하는 것이 좋다.

9.2. 볼륨 지정 오류

`docker run` 명령어에서 `-v` 또는 `--mount` 옵션으로 볼륨을 마운트할 때 경로 지정 문제로 오류가 발생할 수 있다.

- **오류 메시지**: Error response from daemon: invalid volume specification: "<volume_string>"
- **주요 원인**: 컨테이너 내부 마운트 경로를 절대 경로로 지정하지 않은 경우이다. Docker에서 호스트 OS와 컨테이너 간 디렉토리를 마운트할 때, 컨테이너 측 경로는 반드시 절대 경로여야 한다. 호스트 측 경로는 상대 경로일 수 있으며, 이 경우 Docker CLI가 실행되는 현재 작업 디렉토리를 기준으로 해석되거나 Docker 데몬의 특정 기본 경로를 기준으로 할 수 있다. 그러나 컨테이너 파일 시스템은 자체적으로 격리된 루트(/)를 가지므로, 컨테이너 내의 경로는 이 루트를 기준으로 명확하게 지정되어야 한다. 상대 경로를 사용하면 Docker가 컨테이너 내에서 정확한 마운트 지점을 결정할 수 없어 오류가 발생한다.
- **잘못된 예시**:


```
docker container run -v /abc/webap:/nginx/html... # 컨테이너 경로가 상대 경로
docker container run -v /abc/webap:./nginx/html... # 컨테이너 경로가 상대 경로
```
- **올바른 예시**:


```
docker container run -v /abc/webap:/nginx/html... # 컨테이너 경로가 절대 경로
```

또한, 호스트 경로가 실제로 존재하는지, 명명된 볼륨을 사용하는 경우 볼륨 이름의 구문이 올바른지 확인하는 것도 중요하다. 이러한 경로 해석의 미묘한 차이를 이해하는 것은 볼륨 마운트 관련 문제를 예방하고 신속하게 해결하는데 도움이 된다.

10. Ubuntu Docker 보안 강화

Docker를 프로덕션 환경에서 사용하거나 민감한 데이터를 다룰 때는 보안을 강화하는 것이 매우 중요하다. 다음은 Ubuntu에서 Docker 보안을 향상시키기 위한 주요 원칙과 방법들이다.

10.1. Docker 보안 기본 원칙

- **최소 권한의 원칙 (Principle of Least Privilege):** 컨테이너와 사용자에게 필요한 최소한의 권한만 부여한다. 예를 들어, 컨테이너 내부에서 애플리케이션을 루트 사용자로 실행하지 않도록 한다.
- **심층 방어 (Defense in Depth):** 단일 보안 계층에 의존하지 않고, 호스트, Docker 데몬, 이미지, 컨테이너, 네트워크 등 여러 계층에 걸쳐 보안 조치를 적용한다.
- **정기적인 업데이트:** Docker Engine, 호스트 운영체제(Ubuntu), 사용 중인 모든 Docker 이미지를 최신 상태로 유지하여 알려진 취약점을 패치한다.

10.2. 민감 데이터의 안전한 관리

애플리케이션 비밀번호, API 키, TLS 인증서와 같은 민감한 정보는 Docker 이미지에 직접 포함하거나 환경 변수를 통해 평문으로 전달하는 것을 피해야 한다.

- **환경 변수의 한계:** 환경 변수(-e 또는 --env 옵션)는 간단하게 데이터를 전달할 수 있지만, 다음과 같은 보안적 약점이 있다 :
 - docker inspect <container_id> 명령어로 노출될 수 있다.
 - 컨테이너 내부에서 실행되는 모든 프로세스가 접근할 수 있다.
 - 로그 파일에 실수로 기록될 수 있다.
 - 매우 민감한 데이터에는 적합하지 않다.
- **Docker Secrets (주로 Swarm 모드용):** Docker Secrets는 Docker Swarm 모드에서 민감한 데이터를 안전하게 관리하기 위해 설계된 기능이다. (User Query)
- **개요:** 비밀번호, API 키, TLS 인증서 등의 민감 데이터를 중앙에서 관리하고, 필요한 서비스(컨테이너)에만 안전하게 전달한다. 데이터는 전송 중 및 Swarm Raft 로그에 저장될 때 암호화된다.
- **작동 방식:** Secrets는 컨테이너 내부에 인메모리 파일 시스템으로 마운트된다 (Linux의 경우 기본적으로 /run/secrets/<secret_name>). 애플리케이션은 이 파일을 읽어 민감한 데이터를 사용한다.
- **중요 사항:** Docker Secrets는 공식적으로 Swarm 서비스에서만 사용 가능하다. 독립 실행형 컨테이너에서 사용하려면 해당 컨테이너를 단일 노드 Swarm의 서비스로 실행해야 하는 번거로움이 있다.
- **Secrets 생성:**
 - 표준 입력(STDIN)으로부터: `printf "my_password" | docker secret create my_db_password -`
 - 파일로부터: `docker secret create my_site_cert /path/to/cert.pem`
 - 목록 확인: `docker secret ls`
 - 상세 정보 확인: `docker secret inspect <secret_name>`
 - 삭제: `docker secret rm <secret_name>` (실행 중인 서비스에서 사용 중이면 삭제 불가)
- **서비스에서 Secrets 사용:**
 - docker service create 사용 시: `docker service create --name myservice --secret my_secret_data myimage`
 - docker-compose.yml 파일 (docker stack deploy 용):


```
version: '3.8' # 또는 secrets를 지원하는 최신 버전

services:
  myapp:
    image: myapp_image
    secrets:
      - db_password # 짧은 구문: secret 이름 'db_password', 컨테이너 내 경로 /run/secrets/db_password
```

```
# - source: app_config_secret # 긴 구문 예시
# target: /etc/app/config
# uid: '1000'
# gid: '1000'
# mode: 0400
environment:
# 애플리케이션이 이 파일에서 비밀번호를 읽도록 설정
DB_PASSWORD_FILE: /run/secrets/db_password
secrets:
db_password:
file:./db_password.txt # 호스트의 로컬 파일에서 secret 값 로드
# app_config_secret:
# external: true # Swarm에 이미 secret이 존재하는 경우

애플리케이션 코드는 환경 변수(예: DB_PASSWORD_FILE)가 가리키는 파일 경로에서 실제 secret 값을 읽도록
수정되어야 한다. 예를 들어, 공식 MySQL 이미지는 MYSQL_ROOT_PASSWORD_FILE 환경 변수를 통해 파일에
서 비밀번호를 읽는 기능을 지원한다.
```

Docker Secrets는 Swarm 환경에서 편리하고 통합된 민감 데이터 관리 방법을 제공하지만, 모든 시나리오에 적합한 것
은 아니다. 독립 실행형 컨테이너나 Kubernetes와 같은 다른 오케스트레이션 환경에서는 HashiCorp Vault , AWS
Secrets Manager, Azure Key Vault와 같은 외부 전용 시크릿 관리 도구를 고려하는 것이 좋다. 이는 Docker가 제공
하는 기본 기능 외에, 조직의 요구사항이나 기존 인프라에 따라 더 강력하고 중앙화된 솔루션을 선택할 수 있음을 시사한
다. 즉, 시크릿 관리 전략은 배포 모델과 조직의 보안 정책에 따라 계층적으로 접근해야 한다.

- **외부 시크릿 관리 도구:** HashiCorp Vault와 같은 도구는 보다 정교한 시크릿 관리, 접근 제어, 감사 로깅 기능을 제
공하며, 다양한 환경과 통합될 수 있다.

10.3. 컨테이너화된 웹 애플리케이션 SSL/TLS 적용

웹 애플리케이션을 Docker 컨테이너로 배포할 때, 클라이언트와 서버 간의 통신을 암호화하기 위해 SSL/TLS를 적용하
는 것은 필수적이다. (User Query)

- **일반적인 패턴: Nginx 리버스 프록시와 SSL 종료 (SSL Termination)** Nginx 컨테이너가 HTTPS (443 포트) 요
청을 수신하여 SSL/TLS 핸드셰이크를 처리하고, 암호화된 트래픽을 복호화한 후 내부 네트워크를 통해 HTTP로 애
플리케이션 컨테이너(예: Node.js, Python 앱)에 전달하는 방식이 널리 사용된다.

- **SSL/TLS 인증서 생성:**

- **자체 서명 인증서 (개발/테스트용):** OpenSSL 등을 사용하여 생성할 수 있다. 브라우저에서 기본적으로 신뢰하지 않
으므로 경고가 표시된다.

```
# 1. OpenSSL 설정 파일 준비 (예: my-site.conf [span_184](start_span)[span_184](end_span))
```

```
# 2. 개인 키 및 인증서 생성
```

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout my-site.key -out my-site.crt -config
my-site.conf
```

- **Let's Encrypt (프로덕션용):** 무료이며 자동화된 방식으로 신뢰할 수 있는 인증서를 발급받을 수 있다. Certbot과
같은 클라이언트를 사용하며, 도메인 소유권 검증이 필요하다.

- **Nginx SSL 설정:** Nginx 설정 파일 (nginx.conf)에 SSL 관련 지시어를 추가한다. 예시 nginx.conf 일부:

```
server {
```

```

listen 80;

server_name your_domain.com;

location / {
    return 301 https://$host$request_uri; # HTTP를 HTTPS로 리디렉션
}

server {
    listen 443 ssl;
    server_name your_domain.com;

    ssl_certificate /etc/ssl/certs/my-site.crt;    # Nginx 컨테이너 내부의 인증서 경로
    ssl_certificate_key /etc/ssl/private/my-site.key; # Nginx 컨테이너 내부의 개인 키 경로
    # 추가적인 SSL 설정 (프로토콜, 암호화 스위트 등)
    # ssl_protocols TLSv1.2 TLSv1.3;
    # ssl_ciphers 'ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:...';

    location / {
        proxy_pass http://app_container_name:app_port; # 업스트림 애플리케이션 컨테이너
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}

```

Nginx용 Dockerfile에서는 이 설정 파일과 인증서/키 파일을 이미지에 복사하거나, 볼륨 또는 Docker Secrets(Swarm 사용 시)를 통해 마운트할 수 있다.

FROM nginx:latest

COPY nginx.conf /etc/nginx/nginx.conf

COPY my-site.crt /etc/ssl/certs/my-site.crt

COPY my-site.key /etc/ssl/private/my-site.key

Docker Compose를 사용하면 Nginx와 애플리케이션 컨테이너를 함께 정의하고 관리하기 용이하다.

10.4. 주요 Docker 보안 모범 사례

다음은 Docker 환경의 보안을 강화하기 위한 핵심적인 모범 사례들이다. 이러한 관행들은 개발 초기 단계부터 보안을 고려하는 "Shift Left" 접근 방식을 반영하며, 이는 단순히 런타임 설정뿐만 아니라 이미지 생성 및 관리 전반에 걸친 지속적인 노력을 포함한다.

- **컨테이너를 비루트(non-root) 사용자로 실행:** 기본적으로 컨테이너는 내부에서 루트 사용자로 실행될 수 있으며, 이는 컨테이너가 손상될 경우 심각한 보안 위협이 될 수 있다.
- Dockerfile에서 USER <non_root_user>[:<group>] 지시어를 사용하여 컨테이너 실행 사용자를 지정한다. 필요한 경우 먼저 사용자와 그룹을 생성해야 한다 (예: RUN groupadd -r appgroup && useradd -r -g appgroup appuser).
- docker run 명령어 실행 시 -u <username_or_UID> 옵션을 사용할 수 있다.

- **최소한의 기본 이미지 사용:** 이미지 크기를 줄이고 불필요한 구성 요소 및 종속성을 제거하여 공격 표면을 최소화한다.
- alpine, busybox와 같은 경량 이미지나 Google의 distroless 이미지(예: gcr.io/distroless/static-debian11)를 사용한다.
- **이미지 정기적 취약점 스캔:** 이미지에 포함된 소프트웨어 패키지의 알려진 취약점을 탐지하고 수정한다.
- docker scan <image_name> (Snyk 또는 Docker Scout 기반) 명령어를 사용하거나, Trivy, Clair와 같은 오픈소스 스캐너를 활용한다.
- CI/CD 파이프라인에 이미지 스캔 단계를 통합하여 자동화한다.
- 신뢰할 수 있는 레지스트리(예: Docker Hub 유료 플랜, Harbor)의 내장 스캔 기능을 활용한다.
- **컨테이너 리소스 사용량 제한:** 단일 컨테이너가 과도한 시스템 리소스(CPU, 메모리)를 사용하여 서비스 거부(DoS) 공격을 유발하거나 다른 컨테이너의 성능에 영향을 미치는 것을 방지한다.
- docker run 명령어의 --memory <limit>, --cpus <limit>, --pids-limit 등의 옵션을 사용하여 리소스를 제한한다. 이는 Linux의 cgroups 기능을 통해 구현된다.
- **보안 컨테이너 레지스트리 사용 및 관리:**
 - Docker Hub의 공식 이미지나 검증된 게시자의 이미지를 우선적으로 사용한다.
 - 자체 레지스트리를 운영하는 경우, TLS를 사용하여 통신을 암호화하고, 역할 기반 접근 제어(RBAC)를 구현하여 이미지 접근 권한을 관리한다.
- **네트워크 분리 및 API 보안:**
 - 사용자 정의 네트워크를 생성하여 컨테이너 간 통신을 격리하고, 필요한 경우에만 서로 통신하도록 허용한다.
 - Docker 데몬 API 접근을 엄격히 제어하고, 가능하면 네트워크에 노출하지 않는다. 원격 접근이 필요하다면 TLS로 암호화하고 인증을 적용한다.
- **Docker 데몬 및 호스트 OS 최신 상태 유지:**
 - Docker Engine 자체와 호스트 운영체제의 커널 및 관련 패키지에 대한 보안 업데이트를 정기적으로 적용한다.
- **Dockerfile 및 애플리케이션 코드 정적 분석:**
 - Hadolint, SonarQube와 같은 도구를 사용하여 Dockerfile의 잠재적인 보안 문제나 비효율적인 구성을 검사하고, 애플리케이션 코드의 보안 취약점을 분석한다.
- **Docker 데몬 루트리스(Rootless) 모드 사용 (고급):**
 - Docker 데몬과 컨테이너를 일반 사용자로 실행하여 보안을 크게 향상시킨다. 호스트 시스템에 대한 잠재적 위험을 줄일 수 있다.
 - dockerd-rootless-setup.sh install 스크립트를 사용하여 설정할 수 있다. 일부 기능 제한이 있을 수 있으나, 점차 개선되고 있다.

이러한 보안 관행들을 체계적으로 적용하면 Docker 환경의 전반적인 보안 수준을 크게 높일 수 있다. 다음 표는 주요 보안 모범 사례를 요약한 것이다.

표 10.1: Docker 보안 모범 사례 체크리스트

카테고리	모범 사례	근거/이점	구현 방법 (간략히)
이미지 보안	최소 기본 이미지 사용	공격 표면 감소, 이미지 크기 최적화	alpine, distroless 등 사용
	이미지 취약점 스캔	알려진 취약점 사전 탐지 및 완화	docker scan, Trivy, Clair 등 CI/CD 통합
	신뢰할 수 있는 이미지 소스 사용	악성 코드나 변조된 이미지 위험 감소	Docker Hub 공식 이미지, 검증된 게시자, 자체 보안 레지스트리 사용

	멀티스테이지 빌드 활용	최종 이미지에서 빌드 도구 및 불필요한 아티팩트 제거, 이미지 크기 최적화	Dockerfile에서 여러 FROM 문 사용
컨테이너 런타임 보안	비루트 사용자로 컨테이너 실행	컨테이너 탈출 시 호스트 시스템 피해 최소화 (최소 권한 원칙)	Dockerfile에 USER 지시어 사용, docker run -u 옵션
	리소스 사용량 제한	DoS 공격 방지, 시스템 안정성 확보	docker run 시 --memory, --cpus 등 옵션 사용
	읽기 전용 루트 파일 시스템 사용	컨테이너 내부 파일 시스템 변경 방지, 불변성 강화	docker run --read-only 옵션 사용 (쓰기 가능한 tmpfs 마운트 필요할 수 있음)
	AppArmor/Seccomp 프로파일 적용	컨테이너의 시스템 콜 제한, 커널 수준 보안 강화	Docker 기본 프로파일 사용 또는 사용자 정의 프로파일 적용
호스트 및 데몬 보안	호스트 OS 및 Docker Engine 최신 상태 유지	알려진 취약점 패치	정기적인 시스템 업데이트 및 Docker 버전 업그레이드
	Docker 데몬 API 접근 제어	무단 접근 및 데몬 제어 방지	Unix 소켓 사용 권장, TCP 소켓 사용 시 TLS 인증 적용
	루트리스 모드 사용 (가능한 경우)	데몬 및 컨테이너를 비루트 사용자로 실행하여 호스트 보안 강화	dockerd-rootless-setuptool.sh 사용
데이터 보안	민감 데이터 Docker Secrets 사용 (Swarm)	비밀번호, API 키 등을 안전하게 관리 및 컨테이너에 전달	docker secret create, 서비스 정의 시 secrets 블록 사용
	외부 시크릿 관리 도구 활용	Swarm 외부 환경 또는 고급 시크릿 관리 요구사항 충족	HashiCorp Vault, AWS Secrets Manager 등 연동
	볼륨 데이터 암호화 (필요시)	저장된 데이터 보호	호스트 수준 디스크 암호화 (예: LUKS) 또는 애플리케이션 수준 암호화
네트워크 보안	사용자 정의 네트워크로 컨테이너 분리	불필요한 컨테이너 간 통신 차단, 네트워크 세분화	docker network create, 컨테이너 실행 시 --network 옵션 사용
	웹 애플리케이션에 SSL/TLS 적용	클라이언트-서버 간 통신 암호화	리버스 프록시(Nginx 등)에서 SSL 종료, Let's Encrypt 등 사용
	불필요한 포트 노출 최소화	공격 표면 감소	Dockerfile의 EXPOSE는 문서화 목적, docker run -p로 실제 필요한 포트만 매핑

11. 결론 및 추가 자료

본 매뉴얼은 Ubuntu 24.04 LTS 환경에서 Docker를 설치하고 기본적인 사용법부터 컨테이너 관리, 시스템 유지보수, Python 애플리케이션 컨테이너화, 일반적인 문제 해결, 그리고 보안 강화 방안에 이르기까지 포괄적인 내용을 다루었다. Docker는 개발 환경의 일관성을 제공하고, 배포 프로세스를 단순화하며, 마이크로서비스 아키텍처 구현을 용이하게 하는 등 현대 애플리케이션 개발 및 운영에 있어 강력하고 유연한 도구임이 확인되었다.

특히, Dockerfile 최적화, 리소스 관리, 보안 모범 사례 적용은 안정적인 Docker 활용을 위해 매우 중요하다. 예를 들어, requirements.txt를 활용한 의존성 관리와 Docker 빌드 캐시의 이해는 이미지 빌드 시간을 단축시키며, docker system prune과 같은 명령어를 통한 주기적인 시스템 정리는 디스크 공간을 효율적으로 관리하는 데 기여한다. 또한, Docker Secrets나 SSL/TLS 적용과 같은 보안 조치는 민감한 데이터를 보호하고 안전한 서비스를 운영하는 데 필수적이다.

Docker의 세계는 방대하며, 본 매뉴얼에서 다룬 내용은 그 시작점에 해당한다. 사용자의 숙련도와 요구사항에 따라 더 많은 고급 기능과 개념을 학습할 필요가 있다.

추가 학습을 위한 자료:

- **Docker 공식 문서 (Docker Docs):** <https://docs.docker.com/>

- 가장 정확하고 최신의 정보를 제공하는 기본 자료이다. 설치, 명령어 레퍼런스, 네트워킹, 스토리지, 보안 등 모든 주제를 망라한다.
- **Docker Hub:** <https://hub.docker.com/>
- 수많은 공식 및 커뮤니티 제공 Docker 이미지를 검색하고 공유할 수 있는 레지스트리이다.
- **Docker 커뮤니티 포럼 (Docker Community Forums):** <https://forums.docker.com/>
- 다른 Docker 사용자와 질문을 주고받거나 문제 해결 방법을 논의할 수 있는 공간이다.
- **관련 GitHub 저장소:**
- Docker Engine: <https://github.com/moby/moby>
- Docker Compose: <https://github.com/docker/compose>

향후 탐색해 볼 고급 주제:

- **Docker Compose:** YAML 파일을 사용하여 다중 컨테이너 애플리케이션을 정의하고 실행하는 도구이다.
- **Docker Swarm 및 Kubernetes:** 컨테이너 오케스트레이션 도구로, 여러 호스트에 걸쳐 컨테이너화된 애플리케이션을 배포, 확장, 관리하는 기능을 제공한다.
- **고급 네트워킹:** 오버레이 네트워크, 맥브랜(macvlan) 네트워크 등 다양한 네트워크 드라이버와 구성 방법을 이해한다.
- **최적화된 멀티스테이지 Dockerfile 빌드:** 빌드 컨텍스트를 줄이고 최종 이미지 크기를 최소화하기 위한 고급 Dockerfile 작성 기법이다.
- **루트리스 Docker (Rootless Docker):** 보안을 강화하기 위해 Docker 데몬과 컨테이너를 일반 사용자로 실행하는 방법이다.

꾸준한 학습과 실습을 통해 Docker의 강력한 기능을 최대한 활용하여 개발 및 운영 효율성을 높일 수 있기를 바란다.

참고 자료

1. How to Install Docker on Ubuntu 22.04 and 24.04 - phoenixNAP, <https://phoenixnap.com/kb/install-docker-ubuntu>
2. Ubuntu | Docker Docs, <https://docs.docker.com/engine/install/ubuntu/>
3. How to Install Docker on Ubuntu 24.04 | Vultr Docs, <https://docs.vultr.com/how-to-install-docker-on-ubuntu-24-04>
4. How To Remove Docker Images, Containers, and Volumes ..., <https://www.digitalocean.com/community/tutorials/how-to-remove-docker-images-containers-and-volumes>
5. Simplifying Containerization With Docker Run Command - BuildPiper, <https://www.buildpiper.io/blogs/simplifying-containerization-with-docker-run-command/>
6. Running containers | Docker Docs, <https://docs.docker.com/engine/containers/run/>
7. docker container exec, <https://docs.docker.com/reference/cli/docker/container/exec/>
8. Docker Prune: A Complete Guide with Hands-On Examples - DataCamp, <https://www.datacamp.com/tutorial/docker-prune>
9. Tidy up with docker system prune - Depot, <https://depot.dev/blog/docker-system-prune>
10. How to use docker system df command to check disk usage - LabEx, <https://labex.io/tutorials/docker-how-to-use-docker-system-df-command-to-check-disk-usage-555247>
11. docker system - Docker Docs, <https://docs.docker.com/reference/cli/docker/system/>
12. Docker Error Bind: Address Already in Use | Baeldung on Linux, <https://www.baeldung.com/linux/docker-address-already-in-use>
13. [Docker CE] 도커 디렉토리 마운트 설정 에러 (Error response from ..., <https://nirsa.tistory.com/54>
14. Docker Security: 5 Risks and 5 Best Practices for Securing Your Containers - Tigera.io, <https://www.tigera.io/learn/guides/container-security-best-practices/docker-security/>
15. Docker Container Security Best Practices for Modern Applications - Wiz, <https://www.wiz.io/academy/docker-container-security-best-practices>
16. Are there any comprehensive beginner-friendly guides to secure Docker or should I just switch to Podman? - Reddit, https://www.reddit.com/r/selfhosted/comments/1iwwk6/are_there_any_comprehensive_beginnerfriendly/
17. Securing Passwords in Docker | Baeldung on Ops, <https://www.baeldung.com/ops/docker-securing->

passwords 18. Manage sensitive data with Docker secrets | Docker Docs, <https://docs.docker.com/engine/swarm/secrets/> 19. How to use secrets in Docker Compose, <https://docs.docker.com/compose/how-to-use-secrets/> 20. docker secret create - Docker Docs, <https://docs.docker.com/reference/cli/docker/secret/create/> 21. How to secure a Docker registry with SSL/TLS encryption - LabEx, <https://labex.io/tutorials/docker-how-to-secure-a-docker-registry-with-ssl-tls-encryption-411601> 22. Securing Docker Containers with SSL/TLS Certificates - GlobalSign, <https://www.globalsign.com/en/blog/securing-docker-containers-ssl-tls-certificates> 23. SSL with Docker images using nginx as reverse proxy - GitHub Gist, <https://gist.github.com/dahlsailrunner/679e6dec5fd769f30bce90447ae80081> 24. How To Secure a Containerized Node.js Application with Nginx, Let's Encrypt, and Docker Compose | DigitalOcean, <https://www.digitalocean.com/community/tutorials/how-to-secure-a-containerized-node-js-application-with-nginx-let-s-encrypt-and-docker-compose>