

docker - Dockerfile ENTRYPOINT 사용



도커 빌드할때 entrypoint 에 대해서 알려줘

도커에서 **ENTRYPOINT**는 컨테이너가 실행될 때 반드시 실행되는 기본 명령을 지정하는 부분입니다.

조금 풀어서 설명드리면:

- **ENTRYPOINT vs CMD**

- **CMD** : 컨테이너가 실행될 때 기본으로 실행되는 **명령어/인자**. 하지만 **docker run** 뒤에 명령을 주면 CMD는 덮어씌워집니다.
- **ENTRYPOINT** : 컨테이너 실행 시 **항상 실행되어야 하는 실행 파일**을 지정. **docker run** 뒤에 붙는 명령은 **ENTRYPOINT의 인자**로 전달됩니다.

- **형식**

1. **exec form** (권장)

```
dockerfile

ENTRYPOINT ["executable", "param1", "param2"]
```

→ 셸을 거치지 않고 바로 실행. (예: ["/bin/bash", "-c", "echo hello"])

2. **shell form**

```
dockerfile

ENTRYPOINT command param1 param2
```

→ /bin/sh -c로 실행되므로 signal 전달이나 PID 1 문제에서 불리할 수 있음.

- **예시**

1. 고정 실행 파일을 지정하고, 실행할 인자를 바꾸고 싶을 때:

```
dockerfile

FROM ubuntu:24.04
ENTRYPOINT ["echo"]
CMD ["hello"]
```

```
nginx

docker run myimage          # => echo hello
docker run myimage world    # => echo world
```

2. 스크립트를 항상 실행해야 할 때:

```
dockerfile
```

```
COPY start.sh /usr/local/bin/
ENTRYPOINT ["/usr/local/bin/start.sh"]
```

→ 컨테이너가 실행되면 무조건 **start.sh** 실행.

👉 정리하자면:

- **CMD**는 "기본 옵션" 같은 개념. 쉽게 덮어씌워짐.
- **ENTRYPOINT**는 "이 컨테이너의 본질적인 실행파일"을 지정. 덮어씌워지지 않고, run 시 인자를 붙여 전달받음.

혹시 원하시는 게 이론 설명일까요, 아니면 실제 Dockerfile에서 자주 쓰이는 패턴들을 예시로 몇 가지 더 보여드릴까요?



Dockerfile 에 직접 ndtrypoint 를 기술할 수 있어?

네, 맞습니다. **Dockerfile 안에서 직접 ENTRYPOINT를 기술할 수 있습니다.** 실제로 많이 쓰이는 방식이에요.

예를 들어:

```
dockerfile
```

```
FROM ubuntu:24.04
```

```
# 컨테이너 안에 실행할 스크립트 복사
COPY start.sh /usr/local/bin/
```

```
# 실행 권한 부여
RUN chmod +x /usr/local/bin/start.sh
```

```
# 컨테이너 실행 시 무조건 실행될 프로그램 지정
ENTRYPOINT ["/usr/local/bin/start.sh"]
```

이렇게 해두면 **docker run**을 할 때마다 항상 **start.sh**가 실행됩니다.

몇 가지 주의할 점

1. Dockerfile에는 ENTRYPOINT를 한 번만 쓸 수 있음

여러 번 쓰면 마지막에 쓴 게 최종적으로 적용됩니다.

2. 형식

- 권장: JSON 배열 형태 (exec form)

```
dockerfile
```

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

이렇게 하면 신호 전달(signal handling)과 PID 1 문제에 안전합니다.

- 셸 형태 (shell form)

```
dockerfile

ENTRYPOINT python3 app.py
```

내부적으로 `/bin/sh -c "python3 app.py"`로 실행되기 때문에 SIGTERM 같은 게 바로 전달되지 않는 문제가 있을 수 있습니다.

3. CMD와 같이 쓰기

ENTRYPOINT는 "메인 실행파일", CMD는 "디폴트 인자"를 설정하는 패턴이 많습니다.

```
dockerfile

FROM ubuntu:24.04
ENTRYPOINT ["ping"]
CMD ["localhost"]
```

```
arduino

docker run myimage # ping localhost
docker run myimage 8.8.8.8 # ping 8.8.8.8
```

👉 결론: Dockerfile 안에서 **ENTRYPOINT**를 직접 기술하는 건 당연히 가능하고, 오히려 공식 이미지들도 그렇게 하고 있습니다.

궁금한 게, 지금은 ****ENTRYPOINT**를 써야 하는 상황(고정 실행 스크립트, 실행파일 지정)**을 알고 싶으신 건가요, 아니면 단순히 "문법적으로 쓸 수 있는지"만 확인하신 걸까요?



Dockerfile 에서 CMD 로 PATH를 설정하고 다음 CMD 또는 다음 RUN에서 그 PATH를 사용할 수 있어?

좋은 질문이에요 😊

여기서 중요한 포인트는 **CMD는 실행 시점(runtime)** 설정이고, **RUN은 빌드 시점(build-time)** 명령어라는 점이에요.

1. CMD는 실행 시점에만 동작

```
dockerfile

CMD ["echo", "hello"]
```

- 이건 **이미지를 실행할 때** 컨테이너 안에서 실행되는 기본 명령어예요.
- 따라서 **RUN** 단계에서는 **CMD**의 내용이 전혀 반영되지 않습니다.

즉, **CMD로 PATH를 설정해도 그 값은 빌드 도중의 RUN 명령어에서 사용할 수 없습니다.**

2. 빌드 도중 PATH를 설정하려면 → ENV

빌드 과정(**RUN**)에서 PATH를 반영하려면 ****ENV****를 써야 합니다.

```
dockerfile

FROM ubuntu:24.04

ENV PATH="/usr/local/bin:${PATH}"

RUN echo $PATH    # 여기서도 적용됨
```

3. 실행 시 PATH를 바꾸고 싶으면 → ENTRYPOINT 스크립트 활용

```
dockerfile

FROM ubuntu:24.04

COPY entrypoint.sh /usr/local/bin/
RUN chmod +x /usr/local/bin/entrypoint.sh

ENTRYPOINT ["/usr/local/bin/entrypoint.sh"]
CMD ["python3", "app.py"]
```

entrypoint.sh 안에서 PATH를 동적으로 수정할 수 있어요:

```
bash

#!/bin/bash
export PATH="/my/custom/bin:$PATH"
exec "$@"
```

이러면 컨테이너 실행할 때 PATH가 적용된 상태에서 CMD 명령(**python3 app.py**)이 실행됩니다.

✓ 정리

- **CMD**는 빌드 단계에서 **RUN**에 영향 못 줌.
- 빌드 단계에서 PATH 설정하려면 → **ENV**.
- 실행 단계에서 PATH 바꾸고 싶으면 → **ENTRYPOINT 스크립트** 활용.

혹시 원하시는 게 **이미지 빌드 시 PATH 확장**이세요, 아니면 **컨테이너 실행 시 PATH를 바꿔 쓰는 것**을 하고 싶으신 건가요?



도커를 빌드하면서 명령(cmd, run)에서 실행한 명령의 결과를 호스트pc에 출력하는 방법은?

좋은 질문이에요. “호스트 PC에 출력”이 무엇을 의미하느냐에 따라 나뉩니다.

1) 빌드 중 (RUN) 출력

- **RUN**에서 나오는 **stdout/stderr**는 그대로 호스트 터미널에 스트리밍됩니다.
즉, 기본적으로 **docker build** 화면에 다 찍혀요.

- 더 자세히 보려면:

```
bash

DOCKER_BUILDKIT=1 docker buildx build --progress=plain .
```

BuildKit의 예쁜 요약 뷰 대신 **평문 로그**로 모든 출력이 보입니다.

- 로그를 파일로도 저장하려면:

```
bash

DOCKER_BUILDKIT=1 docker buildx build --progress=plain . 2>&1 | tee build.log
```

- 커맨드 에코/에러 추적까지 보고 싶다면 Dockerfile에:

```
dockerfile

SHELL ["/bin/bash", "-eo", "pipefail", "-c"]
RUN set -x; your commands...
```

⚠ 빌드 중 컨테이너에서 호스트 파일시스템에 직접 쓰는 건 불가능합니다. 보안 모델상 빌드 단계는 호스트를 건드릴 수 없고, 출력은 “로그”로만 나옵니다.

빌드 산출물을 호스트로 “꺼내고” 싶다면

- BuildKit의 **output** 기능 사용:

```
bash

docker buildx build \
  --output type=local,dest=./out \
  .
```

빌드 최종 스테이지의 파일들이 호스트 **./out**으로 떨어집니다.
(로그 출력과는 별개로, 파일 결과물을 호스트에 받는 정석 방법)

2) CMD 출력

- **CMD는 빌드 시점엔 실행되지 않습니다.**
따라서 **CMD**의 결과를 “빌드하면서” 볼 순 없어요.
- **CMD** 결과를 보려면 이미지 빌드 후 **컨테이너를 실행**하세요:

```
bash

docker run --rm -it your-image
```

→ 이때 컨테이너 표준출력이 그대로 호스트 터미널에 표시됩니다.
(백그라운드로 띄웠다면 **docker logs <컨테이너>**)

3) 실행 중 로그를 호스트 파일로 남기고 싶다면

- 실행 시 **바인드 마운트**로 로그 디렉터리 연결:

```
bash

docker run --rm -v "$PWD/logs:/app/logs" your-image
```

컨테이너가 `/app/logs/app.log`에 쓰면 호스트 `./logs/app.log`에 파일이 생깁니다.

- 또는 단순히 **호스트 셸 리다이렉션**:

```
bash

docker run --rm your-image 2>&1 | tee run.log
```

짧은 정리

- RUN 출력 → 빌드 시 이미 **호스트 콘솔에 표시됨** (`--progress=plain` 추천).
- CMD는 빌드 시점에 안 돌음 → 이미지를 실행해서 보세요.
- 빌드 중 호스트에 파일 쓰기 → 불가. `**--output**`으로 산출물만 복사 가능.
- 실행 중 로그를 호스트에 남기기 → **바인드 마운트**나 호스트 리다이렉션 사용.

필요하시면, 지금 쓰고 계신 Dockerfile에 맞춰서 **로그 더 잘 보이게** 손보는 스니펫도 바로 잡아드릴게요.



apache-airflow-2.9.2-python3.11을 base로 해서 이미지를 만들었거든. 만들때 도커파일에서 requirements.txt를 이용해서 파이썬 모듈을 설치하게 했어. 이때 모듈중에 apache-airflow===2.9.2 를 추가했거든. 문제없겠지?

네, 말씀하신 상황을 정리하면:

- **베이스 이미지:** `apache/airflow:2.9.2-python3.11`
- **Dockerfile**에서 `requirements.txt`를 설치하면서 거기에 `apache-airflow===2.9.2`를 넣음.

어떻게 동작할까?

- 베이스 이미지에는 이미 **Airflow 2.9.2**가 설치되어 있습니다.
- `pip install -r requirements.txt` 과정에서 `apache-airflow===2.9.2`가 있으면, pip은
 1. 현재 설치된 버전 확인 → 이미 2.9.2
 2. 동일 버전이면 보통 `**"Requirement already satisfied"**`라고 하고 넘어갑니다.
 3. 환경에 따라 재설치를 시도할 수도 있는데, 같은 버전이므로 결과적으로 다시 2.9.2로 덮어씌우는 수준입니다.

잠재적인 문제점

1. 빌드 시간 증가

- 이미 설치된 패키지를 다시 확인하거나, 경우에 따라 재다운로드할 수 있어 불필요하게 시간이 늘어납니다.

2. 의존성 충돌 위험

- 공식 이미지의 `apache-airflow`는 `constraints` 파일로 버전 고정된 수많은 의존성을 맞춰둡니다.
- `pip install apache-airflow==2.9.2`가 다시 실행되면 `constraints` 없이 풀려서 설치될 수도 있고, 그러면 하위 라이브러리 버전이 달라져 미묘한 충돌이 생길 수 있습니다.

3. Best practice 위배

- Airflow 공식 문서에서도, base image를 쓸 때는 `apache-airflow`를 다시 설치하지 말고, 추가 모듈만 설치하는 걸 권장합니다.
- Airflow를 업데이트하려면 base 이미지를 바꾸는 게 더 안전합니다.

정리

- 큰 문제는 없지만 비효율적이고, 경우에 따라 의존성 꼬임이 생길 수 있습니다.
- 이미지를 안정적으로 쓰고 싶으시면 `requirements.txt`에서는 `apache-airflow` 항목을 빼고, 필요한 플러그인/추가 모듈만 두는 게 좋아요.

혹시 제가 확인차 여쭙보고 싶은 건,

`requirements.txt`에 `apache-airflow==2.9.2`를 넣으신 이유가 “명시적으로 버전을 고정해두려는 목적”이신가요, 아니면 단순히 “다른 모듈 설치하면서 같이 넣은 것”일까요?



왜 이렇게 했냐면 `airflow-jdbc` 모듈을 설치하는데, 컨테이너 내부에서 `airflow` 커맨드를 못찾는다고 에러가 나더라고

맥락 이해했어요. `apache/airflow:2.9.2-python3.11`를 베이스로 쓰는데, `airflow-jdbc` 모듈(= JDBC 프로바이더)을 설치하다가 컨테이너 안에서 `airflow` 커맨드를 못 찾는 문제가 생긴 거죠.

이 이슈는 보통 아래 셋 중 하나입니다.

1. 설치한 사용자와 PATH 미스매치

- 베이스 이미지는 기본 `USER airflow`이고, `pip install --user`로 깔면 `/home/airflow/.local/bin`에 바이너리가 생깁니다.
- 반대로 `USER root` 상태에서 `--user`로 깔면 `/root/.local/bin`에 생기고, 그 경로는 `PATH`에 없어서 `airflow`가 “없다”고 뜹니다.
- 해결:
 - 아예 `root`로 시스템 전역 설치(권장) → `pip install --no-cache-dir ...` (※ `--user` 쓰지 않기)
 - 또는 `USER airflow`로 전환한 뒤 설치 + `ENV PATH="/home/airflow/.local/bin:$PATH"` 보장

2. ENTRYPOINT/CMD를 커스텀하며 기본 엔트리포인트를 덮어씀

- 공식 이미지의 `/entrypoint`가 `PATH` 등 런타임 환경을 잘 세팅해줍니다.

- Dockerfile에서 **ENTRYPOINT**를 바꾸면 이 초기화가 빠져서 **airflow**가 PATH에 안 잡힐 수 있어요.
- 해결: ENTRYPOINT를 건드리지 않거나, 반드시 **원래 엔트리포인트를 체이닝**하세요.
 - (공식 기본값을 그대로 두는 게 가장 안전)

3. 패키지 이름/설치 방식 문제

- JDBC는 **apache-airflow-providers-jdbc** 가 정식 이름입니다.
- Airflow 버전과 맞는 **constraints**로 설치해야 의존성 꼬임이 없습니다.

안전한 Dockerfile 예시 (권장)

```
dockerfile

FROM apache/airflow:2.9.2-python3.11

# 필요 시 OS 패키지
USER root
# RUN apt-get update && apt-get install -y --no-install-recommends <deps> && rm -rf
/var/lib/apt/lists/*

# Airflow 2.9.2 + Python 3.11 에 맞춘 constraints로 JDBC 프로바이더 설치
ARG AIRFLOW_VERSION=2.9.2
ARG PYTHON_VERSION=3.11
ARG
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/
constraints-${PYTHON_VERSION}.txt"
RUN pip install --no-cache-dir "apache-airflow-providers-jdbc" --constraint "${CONSTRAINT_URL}"

# root로 전역 설치했으면 PATH는 이미 OK. (ENTRYPOINT는 공식 기본 그대로 유지!)
USER airflow
```

포인트

- **apache-airflow**를 다시 설치하지 않습니다. 베이스 이미지가 이미 2.9.2니까 **프로바이더만** 추가하세요.
- **constraints**를 꼭 씁니다(버전 호환).
- **ENTRYPOINT**는 건드리지 않는 게 안전합니다.

빠른 진단 체크리스트

컨테이너에서 아래로 확인해보세요.

```
bash

which airflow
echo "$PATH"
python -c "import airflow, airflow.providers.jdbc; print('OK!')"
```

- **which airflow**가 **/home/airflow/.local/bin/airflow** 또는 **/usr/local/bin/airflow**를 가리키면 정상입니다.
- 안 나오면 위의 사용자/ENTRYPOINT/설치 방식 중 하나가 문제예요.

요약

- `apache-airflow===2.9.2`를 `requirements.txt`에 넣어 재설치하는 건 비추(빌드 시간 증가 + 의존성 위험).
- 정식 이름 `apache-airflow-providers-jdbc` 를 **constraints와 함께** 설치하세요.
- **ENTRYPOINT**는 그대로, 설치하는 **root 전역**으로 두는 편이 PATH 문제를 줄입니다.

원하시면 지금 사용 중인 Dockerfile 일부만 보여주셔도 돼요. 어디서 PATH가 끊기는지 바로 짚어드릴게요.



airflow provider만 설치하라는게 정확히 뭐야? 그리고 entrypoint를 그대로 두라는게 뭐야? 자세히 설명해줘

좋은 포인트예요. 두 가지를 나눠서 딱 정리드릴게요.

1) "Provider만 설치"가 정확히 뭔가요?

- **Airflow Core:** `apache-airflow` 패키지(웹서버/스케줄러/CLI 등 핵심).
- **Provider:** `apache-airflow-providers-<이름>` 패키지(특정 시스템용 Hook/Operator/Sensor/Connection).
예) JDBC → `apache-airflow-providers-jdbc`, Postgres → `apache-airflow-providers-postgres`

****"Provider만 설치"는 이미 베이스 이미지에 들어있는 Core(= `apache-airflow`)는 그대로 두고, 필요한 Provider 패키지들만 `pip install` 하는 걸 말합니다.**

이때 **Airflow 버전과 맞는 constraints**를 꼭 적용해 의존성 충돌을 막습니다.

예시 (권장 Dockerfile 스니펫)

```
dockerfile

FROM apache/airflow:2.9.2-python3.11

USER root
ARG AIRFLOW_VERSION=2.9.2
ARG PYTHON_VERSION=3.11
ARG CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt"

# 필요한 프로바이더만 추가 (JDBC 예시)
RUN pip install --no-cache-dir "apache-airflow-providers-jdbc" --constraint "${CONSTRAINT_URL}"

# 다른 프로바이더가 필요하면 같은 줄에 추가
# RUN pip install --no-cache-dir \
#   "apache-airflow-providers-postgres" \
#   "apache-airflow-providers-amazon" \
#   --constraint "${CONSTRAINT_URL}"

USER airflow
```

포인트

- **apache-airflow(Core)**는 다시 설치하지 않음.
- 프로바이더만 설치하되 **constraints**로 버전 호환을 보장.
- 설치하는 **root 전역 설치**가 PATH/권한 이슈를 줄입니다.

2) “ENTRYPOINT를 그대로 두라”는 게 뭔가요?

Airflow 공식 베이스 이미지는 이미 공식 **ENTRYPOINT 스크립트**(그리고 **dumb-init**)를 포함하고 있어요. 이 엔트리포인트는 런타임에 다음을 해줍니다(요지):

- 환경 변수/AIRFLOW_HOME/권한 세팅
- PATH 정리
- 전달된 인자에 따라 **webserver, scheduler, worker** 등 기동
- PID 1 신호 처리(정상 종료/재시작에 중요)

여기서 **Dockerfile에서 ENTRYPOINT를 새로 지정해 덮어쓰면** 위 초기화가 통채로 빠질 수 있고, 그 결과:

- **airflow** 바이너리를 PATH에서 못 찾는대거나
- 권한/유저 전환이 안 된대거나
- 종료 신호가 제대로 안 전파되는 문제가 쉽게 납니다.

그래서 ****가장 안전한 방법은 “ENTRYPOINT를 건드리지 않는 것”****입니다.

즉, **Dockerfile에 ENTRYPOINT를 쓰지 않으면** 베이스 이미지의 엔트리포인트가 **그대로 유지**됩니다.

그래도 커스텀 초기화가 꼭 필요할 때(체이닝)

- 기존 엔트리포인트를 **대체하지 말고 호출**하세요(체이닝).
- 작은 래퍼 스크립트를 만들고 **마지막에 원래 엔트리포인트로 exec** 하도록 합니다.

```
dockerfile

FROM apache/airflow:2.9.2-python3.11
USER root

# 1) 나만의 초기화 스크립트
COPY my-entrypoint.sh /usr/local/bin/
RUN chmod +x /usr/local/bin/my-entrypoint.sh

# 2) 원 엔트리포인트를 체이닝하도록 설정
# - 실제 경로/값은 이미지 버전에 따라 다를 수 있어 래퍼에서 원본을 호출하는 방식이 안전합니다.
ENTRYPOINT ["/usr/local/bin/my-entrypoint.sh"]

USER airflow
```

my-entrypoint.sh 예시:

```
bash

#!/usr/bin/env bash
set -euo pipefail
```

```
# (필요 시) PATH/ENV 조정
# export PATH="/home/airflow/.local/bin:$PATH"

# ✔️ 중요: 마지막에 원래 엔트리포인트로 완전 이양
exec /entrypoint "$@"
```

핵심은 `exec /entrypoint "$@"` 로 원래 동작을 그대로 이어받게 하는 겁니다.

(공식 이미지에선 보통 `dumb-init + /entrypoint` 조합을 쓰므로, `dumb-init` 유지가 필요하면 래퍼에서 그것까지 포함해 호출하세요. 가장 안전한 건 아예 ENTRYPOINT를 건드리지 않는 것입니다.)

보너스: airflow 커맨드 못 찾는 문제 빠른 점검

컨테이너 안에서:

```
bash

which airflow
echo "$PATH"
id -u -n
```

- 전역 설치면 보통 `/usr/local/bin/airflow`
- 유저 설치면 `/home/airflow/.local/bin/airflow`
- **USER**와 설치 위치가 엇갈리면 PATH에서 못 잡힙니다.
→ 전역 설치(권장) 또는 `ENV PATH="/home/airflow/.local/bin:$PATH"` 보강

요약

- **Provider만 설치** = Core는 그대로 두고 `apache-airflow-providers-*`만 `constraints`와 함께 설치.
- **ENTRYPOINT는 그대로** = 베이스 이미지의 초기화/신호 처리/경로 세팅을 믿고 덮어쓰지 않기.
꼭 필요하면 래퍼 스크립트로 체이닝해서 원 엔트리포인트로 `exec`하세요.

지금 쓰신 Dockerfile을 살짝만 보여주시면, 어디서 PATH/ENTRYPOINT가 꼬였는지 정확히 짚어서 한 번에 정리해드릴게요.



생성한 airflow 도커 이미지를 실행하는 방법

만드신 이미지를 실행하는 방법은 크게 두 가지 단계로 보시면 됩니다.

1. 이미지 확인

빌드한 이미지가 로컬에 있는지 먼저 확인하세요:

```
bash

docker images
```

예를 들어 결과가

```
perl

REPOSITORY          TAG          IMAGE ID       CREATED        SIZE
my-airflow          2.9.2       abc123def456   5 minutes ago 1.1GB
```

이런 식으로 나오면 **my-airflow:2.9.2**라는 이름으로 빌드된 겁니다.

2. 실행하기

Airflow는 여러 컴포넌트(webserver, scheduler, worker 등)가 있어서 **어떤 프로세스를 띄울지** 지정해야 합니다. 공식 베이스 이미지에는 이미 엔트리포인트(/**entrypoint**)가 설정돼 있어서, 뒤에 오는 명령을 그대로 실행해 줍니다.

기본 실행 (예: 웹서버)

```
bash

docker run --rm -it -p 8080:8080 my-airflow:2.9.2 webserver
```

- **--rm** : 컨테이너 종료 시 자동 삭제
- **-it** : 터미널 붙이기
- **-p 8080:8080** : 호스트 8080 → 컨테이너 8080 포트 매핑
- **webserver** : 컨테이너에서 실행할 Airflow 서버커맨드

접속: <http://localhost:8080> ↗

스케줄러 실행

```
bash

docker run --rm -it my-airflow:2.9.2 scheduler
```

CLI 실행

Airflow 커맨드 직접 확인하고 싶다면:

```
bash

docker run --rm -it my-airflow:2.9.2 airflow version
```

환경 변수 지정

Airflow는 DB나 기타 설정을 환경 변수로 조정합니다. 예를 들어 SQLite를 쓰는 기본 테스트 환경:

```
bash

docker run --rm -it \
  -p 8080:8080 \
  -e AIRFLOW__CORE__EXECUTOR=SequentialExecutor \
  my-airflow:2.9.2 webserver
```

여러 컴포넌트 함께 띄우기

실전에서는 보통 docker-compose를 씁니다.

예시(docker-compose.yml):

```
yaml

version: '3'
services:
  airflow-webserver:
    image: my-airflow:2.9.2
    command: webserver
    ports:
      - "8080:8080"
    environment:
      - AIRFLOW__CORE__EXECUTOR=LocalExecutor

  airflow-scheduler:
    image: my-airflow:2.9.2
    command: scheduler
    depends_on:
      - airflow-webserver
```

👉 정리

- 단일 프로세스: `docker run my-airflow:2.9.2 webserver` (혹은 scheduler/worker 등).
- 여러 개를 동시에 돌리고 싶으면 docker-compose로 webserver/scheduler/worker를 분리 실행.

혹시 원하시는 게 **개발 테스트용으로 컨테이너 하나만 띄워서 DAG 확인**이신가요, 아니면 **운영환경처럼 웹서버+스케줄러+DB까지 묶어 올리는 전체 구성**을 원하시나요?



디버깅용으로 컨테이너 내부 접속해볼려고.

아, 단순히 컨테이너 안에 들어가서 환경을 보고 싶으신 거군요 👍

이미지 이름이 `my-airflow:2.9.2`라고 치면 이렇게 하시면 됩니다:

1) bash로 바로 들어가기

```
bash

docker run --rm -it my-airflow:2.9.2 bash
```

- 새 컨테이너를 만들고 곧장 bash 셸로 접속합니다.
- `which airflow`, `echo $PATH`, `pip list` 같은 걸 직접 확인 가능해요.

2) 백그라운드로 컨테이너 띄운 뒤 exec

```
bash

# 컨테이너 실행 (예: 웹서버)
docker run -d --name airflow-test my-airflow:2.9.2 webserver

# 실행 중인 컨테이너에 접속
docker exec -it airflow-test bash
```


- `docker exec` 방식은 지금 돌아가는 프로세스를 멈추지 않고 안에 들어갈 수 있습니다.
- `airflow scheduler`나 `webserver`가 돌아가고 있는 상태 그대로 내부 확인 가능해요.

3) 셸이 없는 경우 (최소 이미지일 때)

혹시 `bash`가 안 깔려 있으면:

```
bash

docker run --rm -it my-airflow:2.9.2 sh
```

 들어가서 확인하면 좋은 것들:

```
bash

which airflow
airflow version
echo $PATH
pip list | grep airflow
```

→ PATH 문제, 설치된 패키지, 버전 충돌 여부 등을 바로 볼 수 있습니다.

컨테이너 안에 들어가면 주로 **PATH/USER 차이 때문에 airflow 명령을 못 찾는 문제**가 확인될 가능성이 큰데요, 제가 기본 점검 순서를 알려드릴까요?