



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 -c`로 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)** 명령이라는 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 충돌**이 생길 수 있습니다.

3. Best practice 위배

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

정리

- 큰 문제는 없지만 **비효율적**이고, 경우에 따라 **의존성 꼬임**이 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.txt에 **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 **PATH**에 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.
- constraints를 꼭 씁니다(버전 호환).
- ENTRYPOINT는 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 설명해 줘

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 필요하다면 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 필요하다면 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?



airflow 를 못찾는다고 오류가 나는 상황을 정리해줄께.

```
docker run --rm -it airflow:2.9.2-0.3-python3.11-spark3.5.3 bash 이렇게 실행하면
ERROR! Maxium number of retries (20) reached. Last check result: $ airflow db check
```

/entrypoint: line 20: airflow: command not found 라고 오류가 뜨는데,

내가 설치한 Dockerfile의 베이스 이미지는 apache/airflow:2.9.2-python3.11 이고, pip install 로 requirements.txt 을 설치하고 있는데, 목록은 apache-airflow-providers-jdbc 이걸 설치하면 오류가 나네.

상황 딱 보입니다. **bash**로 들어가려 했는데도 **베이스 이미지의 엔트리포인트**가 먼저 돌면서 **airflow db check**를 치고, 여기서 **airflow: command not found**가 터진 거예요.

즉, **엔트리포인트가 실행되는 시점에 airflow 바이너리가 PATH에 없습니다.**

아래 순서로 바로 잡아봅시다.

0) 먼저 엔트리포인트 우회해서 들어가기

엔트리포인트가 망가진 상태에선 평소처럼 **docker run ... bash**가 안 됩니다. **엔트리포인트를 교체**해서 들어가세요.

```
bash

docker run --rm -it --entrypoint /bin/bash \
  airflow:2.9.2-0.3-python3.11-spark3.5.3
```

들어가면 바로 점검:

```
bash

which airflow
echo "$PATH"
pip show apache-airflow
pip list | grep -E 'airflow|jdbc'
ls -l /usr/local/bin/airflow
id -u -n
```

- 정상이라면 **which airflow**가 보통 **/usr/local/bin/airflow**를 가리킵니다.
- 안 보이면 **PATH**가 덮였거나, 설치가 꼬였거나, 유저/설치 위치가 엇갈린 겁니다.

1) 흔한 원인 3가지

1. PATH를 잘못 덮어씀

- Dockerfile에서 **ENV PATH=/something**처럼 **기존 \$PATH를 포함하지 않고** 새로 세팅하면 **/usr/local/bin**이 날아 갑니다.
 - 증상: 엔트리포인트에서 **airflow** 못 찾음.
 - 해결: 항상 **append** 형태로.

```
dockerfile

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

(가장 안전하게 표준 경로를 먼저 적고 **\${PATH}**를 뒤에 덧붙이세요.)

2. 프로바이더 설치를 constraints 없이 실행 → 의존성/스크립트 꼬임

- **apache-airflow-providers-jdbc**를 **constraints 없이** 깔면, 하위 의존성 충돌로 **console_scripts(= airflow CLI 엔트리포인트)** 재생성이 어긋날 수 있습니다.

- 해결: **항상 constraints로 설치.**

```
dockerfile

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}"
```

3. 설치 유저/위치 문제 (--user 사용, USER 전환 순서)

- **USER root**로 **pip install --user**를 쓰면 **/root/.local/bin**에 스크립트가 생기고 **PATH**에 안 잡힙니다.
- 반대로 **USER airflow**에서 **--user**면 **/home/airflow/.local/bin**.
- 해결:
 - **전역 설치(권장): USER root**에서 **pip install --no-cache-dir ...** (절대 **--user** 쓰지 않기)
 - 또는 **USER airflow**로 설치하되 **PATH** 보강:

```
dockerfile

ENV PATH="/home/airflow/.local/bin:${PATH}"
```

2) 바로 고치는 Dockerfile 예시 (안전판)

```
dockerfile

FROM apache/airflow:2.9.2-python3.11

USER root

# 혹시 이전에 PATH를 건드렸었다면, 표준 PATH를 확실히 복구
ENV PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"

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"

# JDBC 프로바이더만 설치 (Core 재설치 금지)
RUN pip install --no-cache-dir apache-airflow-providers-jdbc --constraint "${CONSTRAINT_URL}"

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

USER airflow
```


포인트

- **apache-airflow**는 다시 깔지 않습니다(베이스 이미지를 신뢰).
- **constraints**로 의존성 맞춤.
- **PATH**를 덮어쓰지 않거나, 덮어썼다면 표준 경로 + **\${PATH}**로 복구.
- **--user** 사용 금지(전역 설치로 PATH 문제 차단).

3) 이미지 재빌드 후 빠른 건강검진

```
bash

# 엔트리포인트 우회로 먼저 체크
docker run --rm -it --entrypoint /bin/bash my-fixed-airflow:2.9.2
which airflow && airflow version

# 이제 정상 실행 (엔트리포인트 사용)
docker run --rm -it -p 8080:8080 my-fixed-airflow:2.9.2 webserver
```

4) 당장 우회해서 확인만 하고 싶다면

엔트리포인트의 사전 체크를 끄고 **airflow**만 확인:

```
bash

docker run --rm -it --entrypoint /bin/bash \
  airflow:2.9.2-0.3-python3.11-spark3.5.3 -lc 'which airflow && airflow version || true'
```

한 줄 정리

- 지금 오류는 **엔트리포인트** 시점 **PATH**에 **airflow**가 없어서 생깁니다.
- **PATH** 덮어쓰기, **constraints** 미사용, **--user** 설치가 주범.
- 위 Dockerfile 예시대로 **전역 설치 + constraints + PATH** 복구로 끝납니다.

원하시면 현재 Dockerfile의 **ENV PATH, USER, pip install** 부분만 붙여주셔도 돼요. 어디서 비틀렸는지 딱 잡아드릴게요.



나는 base image 즉 Dockerfile에서 FROM apache/airflow:2.9.2-python3.11 을 했는데, 컨테이너에 들어와서 pip show apache-airflow 하니깐 Version 3.0.6 이 나오네, 그리고 어디에도 airflow 실행파일이 존재하지 않아.

말씀하신 증상 두 가지가 핵심이네요.

1. 베이스는 2.9.2인데 컨테이너에서 pip show apache-airflow → 3.0.6

2. airflow 실행파일이 PATH 어디에도 없음

이건 빌드 과정에서 베이스 이미지에 들어있던 Airflow 2.9.2가 다른 버전(3.0.6)으로 교체되면서, 동시에 CLI 엔트리포인트(= /usr/local/bin/airflow)가 PATH 밖으로 밀리거나 생성이 꼬인 상황으로 보는 게 맞습니다. 보통 아래 시나리오 중 하나로 재현됩니다.

왜 이런 일이 생기나? (가능성 높은 원인)

A. requirements.txt 또는 추가 설치에서 Core를 다시 설치/업그레이드

- apache-airflow가 명시돼 있거나, 제약(Constraints) 없이 추가로 뭔가를 설치하며 의존성 해석 과정에서 Core가 3.0.x로 올라감
- 결과: 베이스의 2.9.2가 3.0.6으로 바뀜 → 이미지가 가정하던 바이너리/스크립트 상태가 달라짐

B. Constraints 미사용

- Airflow는 버전별로 **constraints** 파일을 강하게 권장합니다.
- constraints 없이 apache-airflow-providers-jdbc 등을 설치하면 하위 의존성들이 엇갈려서 console_scripts(= airflow CLI) 생성이 꼬이는 사례가 납니다.

C. PATH를 덮어씀

- Dockerfile에서 ENV PATH=...를 하면서 기존 \$PATH를 포함하지 않으면 /usr/local/bin이 날아가 airflow 바이너리를 못 찾습니다.
- 실행파일은 있는데 which airflow가 못 찾는 전형적인 패턴.

D. 설치 위치/유저 불일치

- --user로 설치하거나 PYTHONUSERBASE/PIP_TARGET을 건드리면 스크립트가 ~/.local/bin 등 다른 곳에 생기고 PATH에 안 잡힙니다.

바로 확인해볼 체크리스트 (컨테이너 안)

```
bash

# 엔트리포인트 우회해서 진입
docker run --rm -it --entrypoint /bin/bash <YOUR-IMAGE>

# 1) 실제 설치/경로 상태
python -c "import airflow, sys; print('airflow:', airflow.__version__, 'py:', sys.version)"
pip show apache-airflow | egrep 'Version|Location'
python -c "import site, sys; print('paths=', site.getsitepackages(), '\nuserbase=', site.USER_BASE)"

# 2) 실행파일 유무와 PATH
echo "$PATH"
ls -l /usr/local/bin/airflow || true
ls -l /home/airflow/.local/bin/airflow || true
command -v airflow || which airflow || true
```

```
# 3) PATH를 덮어쓴 흔적?
env | grep -E 'PIP_TARGET|PYTHONUSERBASE' || true
```

- 정상이라면 `/usr/local/bin/airflow` 또는 `/home/airflow/.local/bin/airflow` 중 하나가 보여야 합니다.
- `PATH`에 `/usr/local/bin`이 없다면 `Dockerfile`의 `ENV PATH=...`가 범인일 확률 큼니다.

안전하게 고치는 Dockerfile 패턴 (권장)

핵심 원칙 세 가지:

- **Core 재설치 금지:** `apache-airflow`는 베이스 걸 그대로 둡니다.
- **항상 constraints 사용:** 프로바이더 설치 시 필수.
- **PATH 덮어쓰지 않기** (덮어썼다면 복구).

```
dockerfile

FROM apache/airflow:2.9.2-python3.11

USER root

# 혹시 이전에 PATH를 만졌다면 표준 경로 + 기존 PATH로 복구
ENV PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"

# Airflow 2.9.2 / Py3.11용 constraints 고정
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"

# ✅ Core는 건드리지 않고, 필요한 'provider'만 constraints와 함께 설치
RUN pip install --no-cache-dir \
    apache-airflow-providers-jdbc \
    --constraint "${CONSTRAINT_URL}"

# (중요) --user, PIP_TARGET, PYTHONUSERBASE 등 쓰지 마세요.

USER airflow
```

포인트

- `requirements.txt`에서 `apache-airflow` 항목은 제거하세요.
- 다른 프로바이더도 같은 줄에 추가해도 되지만, **항상 같은 constraints**를 사용하세요.
- `ENTRYPOINT`는 **베이스 그대로 유지**(따로 지정 X). 그게 `PATH`/권한/신호처리를 잘 해줍니다.

이미 꼬여버린 이미지에서의 응급 복구 (선택)

“당장” 확인만 하려면, `PATH`를 복구하고 `CLI`를 다시 만들어볼 수 있습니다. (임시 처방)

```
bash

docker run --rm -it --entrypoint /bin/bash <BROKEN-IMAGE> -lc '
set -e
```

```
export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"
which airflow || true
# 그래도 없으면 재생성 시도 (권장: 재빌드가 더 낫습니다)
pip install --no-cache-dir --force-reinstall "apache-airflow==2.9.2" \
  --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-
2.9.2/constraints-3.11.txt"
which airflow && airflow version
```

하지만 가장 안전한 건 **Dockerfile**을 고치고 처음부터 다시 빌드하는 겁니다.

한줄 결론

- 베이스가 2.9.2인데 컨테이너에서 3.0.6이 보인다는 건, **빌드 중에 Core가 업그레이드**됐다는 뜻입니다.
- 그 과정에서 **constraints 미사용 / PATH 덮어쓰기 / 설치 위치 불일치**가 겹치면 **airflow CLI**가 “사라진 것처럼” 보입니다.
- **Core는 건드리지 말고(providers만 설치), constraints를 꼭 쓰고, PATH를 덮어쓰지 않는 Dockerfile**로 재빌드하면 깔끔히 해결됩니다.

원하시면 지금 쓰신 Dockerfile에서 **ENV PATH, pip install**(특히 **apache-airflow** 항목 유무), **USER** 전환되는 부분만 붙여주세요. 정확히 어디서 꼬였는지 한 줄 단위로 정리해드릴게요.



참고로 내가 requirements.txt 파일내용 알려줄께.

```
setuptools===60.0.0
pyspark===3.5.3
apache-airflow-providers-apache-spark===5.0.1
apache-airflow-providers-jdbc===5.0.1
pyarrow
```

이렇게 적었거든.

공유해주신 **requirements.txt** 구성, 핵심만 보면 이렇게예요:

```
diff

setuptools===60.0.0    ← 강하게 고정
pyspark===3.5.3
apache-airflow-providers-apache-spark===5.0.1
apache-airflow-providers-jdbc===5.0.1
pyarrow               ← 버전 미고정
```

여기서 문제가 생길 지점이 몇 가지 있습니다.

왜 문제가 터졌을까

1. constraints 없이 프로바이더를 설치함

- Airflow는 버전별로 “constraints 파일”로 의존성을 촘촘히 고정합니다.
- constraints 없이 `apache-airflow-providers-*`를 설치하면, 하위 의존성 해석 과정에서 **Core(apache-airflow)**가 **최신(예: 3.x)**으로 끌어올라갈 수 있습니다.
→ 그래서 컨테이너에서 `pip show apache-airflow`가 **3.0.6**으로 보였던 겁니다.

2. setuptools==60.0.0 핀 고정

- Python 3.11 환경에서 너무 낮은 setuptools를 강제로 고정하면,
 - `console_scripts(= airflow CLI)` 엔트리포인트 생성/업데이트가 꼬일 수 있고
 - 다른 패키지 설치/빌드 휠에서 충돌이 생길 수 있습니다.
→ 결과적으로 `/usr/local/bin/airflow` 같은 실행 파일이 **안 생기거나 PATH 밖에 놓이는** 증상으로 이어질 수 있어요.

3. pyarrow 미고정 + Spark

- Spark와 Arrow는 짝이 있습니다. `pyarrow`를 풀어두면, 설치 시점에 올라간 버전이 Spark가 기대하는 버전과 **미세하게 어긋나** 런타임 경고/오류의 씨앗이 됩니다. (지금 증상과 직접 원인은 아니지만, 같이 잡아두면 좋습니다.)

권장 수정안

A) requirements.txt를 이렇게 정리

가장 안전한 형태는 **버전 표기 최소화 + constraints에 맡기기**입니다.

옵션 1 (심플 & 안전):

```
makefile

pyspark==3.5.3
apache-airflow-providers-apache-spark
apache-airflow-providers-jdbc
pyarrow==14.0.2
```

- `apache-airflow` 절대 넣지 않기
- `setuptools` 삭제
- `pyarrow`는 Spark 3.5.x와 잘 맞는 14.x로 고정(예: 14.0.2).
(다른 버전을 이미 쓰고 계시면 그 버전으로 고정해도 됩니다. 핵심은 “고정”)

옵션 2 (버전까지 명시하고 싶다면):

지금 쓰신 버전을 유지하되 반드시 **constraints와 함께 설치**하세요.

```
ini

pyspark==3.5.3
apache-airflow-providers-apache-spark==5.0.1
apache-airflow-providers-jdbc==5.0.1
pyarrow==14.0.2
```

그래도 `setuptools`는 빼는 걸 강력 권장드립니다.

B) Dockerfile에서 “constraints”로 설치

```

dockerfile

FROM apache/airflow:2.9.2-python3.11

USER root

# (중요) Airflow 2.9.2 / Py3.11에 맞는 constraints 지정
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"

# requirements 반영
COPY requirements.txt /requirements.txt

# 전역 설치( --user 금지 ), constraints 적용
RUN pip install --no-cache-dir -r /requirements.txt --constraint "${CONSTRAINT_URL}"

# PATH는 건드리지 않는 것이 가장 안전합니다
# (예전에 PATH를 덮어썼다면 표준 경로+${PATH}로 복구하세요)
# ENV PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"

USER airflow

```

포인트

- **Core 재설치 금지:** apache-airflow는 requirements에 넣지 않습니다.
- **constraints 필수:** 프로바이더/부가 패키지 설치 시 반드시 적용.
- **전역 설치:** --user 사용하지 않습니다.
- **ENTRYPOINT는 베이스 그대로:** 따로 바꾸지 마세요.

재빌드 후 빠른 체크

```

bash

# 엔트리포인트 우회해서 먼저 확인
docker run --rm -it --entrypoint /bin/bash my-airflow:2.9.2 -lc '
  which airflow && airflow version && \
  pip show apache-airflow | egrep "Version|Location" && \
  echo "$PATH"
'

# 정상 보이면 웹서버 기동
docker run --rm -it -p 8080:8080 my-airflow:2.9.2 webserver

```

- airflow version이 2.9.2로 출력되어야 하고, pip show apache-airflow도 2.9.2여야 정상입니다.
- which airflow가 /usr/local/bin/airflow를 가리키면 깔끔합니다.

한 줄 정리

- 지금 증상은 **constraints 없이 프로바이더를 설치 + 낮은 setuptools 핀 고정**이 겹치면서 **Airflow Core가 3.x**로 올라가고 **CLI 생성이 꼬인** 케이스입니다.
- **apache-airflow**는 제거, **setuptools** 핀 해제, **constraints와 함께 설치**로 재빌드하시면 **airflow CLI와 버전이** 정상으로 돌아옵니다.

원하시면, 지금 쓰신 Dockerfile 일부(특히 **ENV PATH, USER, pip install** 줄)만 붙여주시면 줄 단위로 확인해서 최적화까지 같이 잡아드릴게요.



root의 비밀번호는 뭐지

도커 컨테이너 안에서 말씀하시는 **root 계정 비밀번호**라면, 보통은 **설정되어 있지 않습니다**.

- 대부분의 공식 Docker 이미지(**apache/airflow** 포함)는 **비밀번호 로그인용 root 계정**이 **비활성화**되어 있어요.
- 컨테이너 안에서 root 권한이 필요하다면 **docker run** 할 때 **--user root**를 주거나, Dockerfile에서 **USER root**를 선언하는 방식으로 들어갑니다.

예시:

```
bash

# root 권한으로 셸 진입
docker run -it --rm --user root apache/airflow:2.9.2-python3.11 bash
```

컨테이너 안에서 root로 들어갔을 때는 이미 비밀번호 확인 없이 root 권한을 바로 쓸 수 있어요. 즉, "root 비밀번호"라는 개념 자체가 따로 없고, 기본값도 없습니다.

👉 정리하면:

- 공식 Airflow 이미지는 root 계정에 **비밀번호가 없음**.
- 필요한 경우 **--user root**로 실행해서 root 셸을 바로 여시면 됩니다.

혹시 지금 root로 로그인하려는 상황이 **컨테이너 내부에서 su/sudo로 전환하려는 상황**인가요, 아니면 **호스트 서버 root 패스워드**를 묻는 건가요?



아까처럼 도커를 빌드하고 컨테이너로 들어가면 airflow 실행파일이 안보여. **pip show apache-airflow** 하면 **Version:3.0.6** 이라고 나오고, 녀——ㅁ교 :**Programmatically author, schedule and monitor data pipelines** 이렇게 나오고..뭔가 base image 가 설치가 안된 느낌이야.

지금 보이는 증상은 세 줄로 요약됩니다.

1. 베이스는 **apache/airflow:2.9.2-python3.11**인데 컨테이너 안 **apache-airflow**가 **3.0.6**으로 바뀌었다.
2. **airflow** 실행파일(콘솔 스크립트)이 **생성되지 않았다**(PATH 문제이거나 애초에 스크립트가 안 만들어짐).
3. 이 둘은 보통 **constraints 없이 프로바이더 설치 + 낮은 setuptools 고정**이 겹칠 때 재현됩니다.

왜 이런 일이 생기나요?

- `apache-airflow-providers-*`를 **constraints 없이** 설치하면 pip가 의존성을 재해석하면서 **Core(=apache-airflow)**를 최신으로 끌어올릴 수 있습니다. 그래서 2.9.2 → 3.0.6이 된 겁니다.
- `setuptools==60.0.0` 고정은 **Python 3.11**에서 너무 낮아 콘솔 스크립트 생성(= `/usr/local/bin/airflow`)이 깨질 수 있습니다. 결과적으로 `airflow` 바이너리가 안 생기거나, 생겨도 인식이 꼬입니다.
- 추가로 Dockerfile 어딘가에서 `ENV PATH=...`로 기존 `PATH`를 덮어썼다면 `/usr/local/bin`이 빠져 `airflow`를 못 찾을 수도 있습니다.

고치는 방법 (가장 안전한 루트)

1) requirements.txt 정리

`apache-airflow`와 `setuptools` 핀을 제거하고, 필요한 것만 둡니다. (constraints가 버전 호환을 보장)

추천 예시

```
ini

pyspark==3.5.3
apache-airflow-providers-apache-spark==5.0.1
apache-airflow-providers-jdbc==5.0.1
pyarrow==14.0.2
```

핵심: **apache-airflow**는 절대 넣지 않기, **setuptools** 고정 지우기

`pyarrow`는 Spark 3.5.x와 궁합 좋은 14.x로 고정(예: 14.0.2). 이미 다른 버전을 쓰시면 그걸로 “고정”만 해주세요.

2) Dockerfile 수정

```
dockerfile

FROM apache/airflow:2.9.2-python3.11

USER root

# (선택) 혹시 과거에 PATH를 덮어쓴 적이 있다면 표준 PATH로 복구
ENV PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"

# Airflow 2.9.2 / Python 3.11에 맞는 constraints
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"

COPY requirements.txt /requirements.txt

# 전역 설치( --user 금지 ), 반드시 constraints 사용
RUN pip install --no-cache-dir -r /requirements.txt --constraint "${CONSTRAINT_URL}"

# 빌드 타임에 바로 검증(옵션이지만 강력 추천)
RUN which airflow && airflow version && \
    python -c "import airflow; print('airflow==', airflow.__version__)"
```


USER airflow

중요 포인트

- Core(=apache-airflow)는 **베이스 그대로** 유지합니다.
- 프로바이더·부가 패키지는 **항상 constraints로 설치**합니다.
- **--user**는 쓰지 말고 **전역 설치**로 PATH/권한 꼬임을 차단합니다.
- **ENTRYPOINT**는 **베이스 이미지 그대로** 두세요. (바꾸면 초기화/신호 처리/경로 세팅이 깨질 수 있음)

지금 이미지에서 바로 확인할 체크리스트

엔트리포인트를 우회해서 들어가 진단해보세요:

```
bash

docker run --rm -it --entrypoint /bin/bash <YOUR-IMAGE>
```

들어가서:

```
bash

echo "$PATH"
pip show apache-airflow | egrep 'Version|Location'
python -c "import airflow, sys; print('airflow:', airflow.__version__, 'py:', sys.version)"
ls -l /usr/local/bin/airflow || true
ls -l /home/airflow/.local/bin/airflow || true
command -v airflow || which airflow || true
```

- **Version이 3.0.6**으로 뜨면 이미 빌드 과정에서 Core가 **업그레이드**된 상태입니다.
- **실행파일 경로가 비어 있거나 PATH에 /usr/local/bin이 없다면** PATH/설치 문제가 맞습니다.

당장 임시 복구(응급)

지금 이미지 안에서만 급히 정상화 확인을 원하시면:

```
bash

docker run --rm -it --entrypoint /bin/bash <BROKEN-IMAGE> -lc '
set -e
export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"
python -c "import sys; print(sys.version)"
pip install --no-cache-dir --force-reinstall "apache-airflow==2.9.2" \
--constraint "https://raw.githubusercontent.com/apache/airflow/constraints-
2.9.2/constraints-3.11.txt"
which airflow && airflow version
'
```

그러나 가장 안전한 해법은 위 **Dockerfile로 처음부터 다시 빌드**하는 것입니다.

덧: 진짜 베이스가 맞는지 확인

혹시 멀티스테이지에서 뒤에 다른 **FROM**으로 덮었거나 실수로 다른 이미지를 태깅했을 수도 있어요. 아래로 베이스를 점검해보세요.

```
bash

docker history <YOUR-IMAGE>
docker inspect <YOUR-IMAGE> | grep -E '"Image"|Entrypoint|Cmd|WorkingDir|Env' -n
```

한 줄 정리

지금 상태는 **프로바이더를 constraints 없이 설치 + 낮은 setuptools 핀** 때문에 Core가 3.0.6으로 올라가고 콘솔 스크립트 생성이 꼬여 **airflow 실행파일이 사라진 것처럼 보이는** 케이스입니다.

apache-airflow와 **setuptools** 핀을 제거하고 **constraints로 재설치**하면, **airflow CLI**와 버전이 **2.9.2**로 정상 복구됩니다.



Dockerfile 알려줄게.

```
FROM nexus.....:5000/apache/airflow:2.9.2-python3.11
USER root
ARG AIRFLOW_VERSION=2.9.2
ARG PYTHON_VERSION=3.11
..... 이런식으로
```

공유 감사합니다. 핵심만 보면 지금 Dockerfile/빌드 흐름에서 몇 군데가 꼬였을 가능성이 높아요. 바로 잡을 수 있게 “원인 후보 → 확실한 수정안 → 빠른 검증” 순서로 드릴게요.

왜 2.9.2 베이스에서 3.0.6이 되었나

1. 프로바이더를 constraints 없이 설치

- **apache-airflow-providers-***를 constraints 없이 설치하면 **pip**가 의존성 재해석하면서 **Core(apache-airflow)**를 **3.x**로 끌어올릴 수 있습니다.

2. setuptools==60.0.0 고정(이전 메시지 기준)

- Py3.11에서 너무 낮아서 **console_scripts(= airflow 실행파일)** 생성이 **꼬일** 수 있습니다.

3. PATH/ENTRYPOINT 손댐

- **ENV PATH=...**로 기존 PATH를 덮었거나(→ **/usr/local/bin** 빠짐), **ENTRYPOINT**를 커스텀해 초기화가 깨지면 **airflow**를 못 찾습니다.

4. 레지스트리 미리 이미지

- **nexus.../apache/airflow:2.9.2-python3.11** 태그가 공식 이미지와 다른 내역일 수도 있습니다(내부에서 재빌드된 경우).

5. 오타로 ARG가 안 먹힘

- ARG PYTHON_VERSION=3.11 ← 오타(PYTHON_VERSION)면, 이 값을 쓰는 구문(예: constraints URL)이 있다면 쓸 값이 비어 잘못된 설치가 이루어질 수 있어요.

바로 적용 가능한 “안전 Dockerfile” (권장 템플릿)

아래 그대로 쓰시면 99% 정리됩니다.

```
dockerfile

# 1) 베이스: 공식과 동일한 내용이 들어있는지 신뢰 가능한 이미지 사용
FROM nexus.....:5000/apache/airflow:2.9.2-python3.11

# 2) 전역 설치를 위해 root로 전환
USER root

# 3) (선택) 예전에 PATH를 덮어썼다면 표준 경로로 복구
ENV PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}"

# 4) Airflow 버전/파이썬 버전 ARG (오타 주의!)
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"

# 5) requirements 반영 (Core는 절대 넣지 않기)
# 예: pyspark==3.5.3, providers-jdbc/Apache Spark, pyarrow==14.0.2 등만
COPY requirements.txt /requirements.txt

# 6) constraints를 반드시 적용해서 설치 (전역 설치, --user 금지)
RUN pip install --no-cache-dir -r /requirements.txt --constraint "${CONSTRAINT_URL}"

# 7) 빌드 타임에 바로 건강검진(강력 추천)
RUN set -e; \
    which airflow; \
    airflow version; \
    python -c "import airflow; print('airflow==', airflow.__version__)"; \
    pip show apache-airflow | grep 'Version|Location'

# 8) 런타임 유저 복귀
USER airflow
```

requirements.txt (예시)

```
ini

pyspark==3.5.3
apache-airflow-providers-apache-spark==5.0.1
apache-airflow-providers-jdbc==5.0.1
pyarrow==14.0.2
```

- 절대 apache-airflow 넣지 마세요.
- setuptools 핀 고정 제거 권장.

빠른 진단/검증 루틴

이미지를 재빌드한 뒤 아래 두 단계로 확인하세요.

A) 엔트리포인트 우회 진입(환경 확인)

```
bash

docker run --rm -it --entrypoint /bin/bash <이미지태그> -lc '
  echo "$PATH";
  which airflow || true;
  airflow version || true;
  pip show apache-airflow | egrep "Version|Location";
  python -c "import airflow, sys; print(airflow.__version__, sys.version)"
'
```

- 기대값: `airflow version`이 2.9.2, `pip show apache-airflow`도 2.9.2.

B) 정상 실행

```
bash

docker run --rm -it -p 8080:8080 <이미지태그> webserver
```

추가 체크(문제 계속 시)

- 베이스 진짜 확인

```
bash

docker history <이미지태그>
docker inspect <이미지태그> | egrep -n 'Entrypoint|Cmd|Env'
```

→ 내부에서 재빌드된 “비공식 2.9.2”일 수도 있습니다.

- **PATH 덮어쓰기 흔적**
Dockerfile에 `ENV PATH=...`가 있으면 반드시 `${PATH}`를 포함하는 형태로 수정하세요.
- **ENTRYPOINT 건드렸다면**
일단 지우고 베이스 기본 엔트리포인트를 그대로 쓰세요. 정말 필요하다면 래퍼에서 `exec /entrypoint "$@"`로 체이닝.

한 줄 정리

- **오tak(PYTHON_VERSION) 수정 + constraints 적용 + Core 재설치 금지** 이 3가지로 끝납니다.
- 재빌드 후 `which airflow` && `airflow version`을 빌드 단계에서 바로 체크하면 다음 번엔 컨테이너 들어가서 당황할 일이 없습니다.

지금 쓰신 Dockerfile의 `ENV PATH`, `ENTRYPOINT`, `pip install` 줄을 그대로 보내주시면, 거기서 딱 필요한 부분만 최소 수정해 드릴게요.



지금 쓰고 있는 도커파일에서 ENV없고 ENTRYPOINT 없고
requirements.txt에서
apache-airflow-providers-jdbc 만 추가하면 저런 현상이 나네

요약부터 할게요.

- **원인:** `apache-airflow-providers-jdbc`를 *constraints* 없이 설치하면, `pip`가 의존성 해석 과정에서 **Core(`apache-airflow`)를 최신(3.x)**으로 올려버릴 수 있습니다. 그러면 베이스(2.9.2)가 덮여서 `airflow` 콘솔 스크립트 생성/경로가 함께 꼬이고, 결과적으로 “`airflow` 실행파일이 없다”는 증상까지 이어집니다.
→ ENV/ENTRYPOINT를 안 건드려도 **그 한 줄만으로** 재현될 수 있어요.

해결책 (재현 방지용 베스트 프랙티스)

1) constraints를 반드시 써서 설치

Airflow 2.9.2 + Py3.11용 constraints URL을 고정하세요.

```
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"

# JDBC provider만 설치 (중요: --upgrade 사용하지 말기)
RUN pip install --no-cache-dir \
    "apache-airflow-providers-jdbc==5.0.1" \
    --constraint "${CONSTRAINT_URL}"

# 빌드 타임에 바로 검증(강추)
RUN which airflow && airflow version && \
    pip show apache-airflow | grep 'Version|Location'

USER airflow
```

포인트

- `requirements.txt`를 쓰신다면 동일하게 `--constraint "${CONSTRAINT_URL}"`를 붙여 설치하세요.
- `pip install -U(= --upgrade)` 붙어 있으면 뺍니다.
- `apache-airflow(Core)`는 `requirements`에 넣지 않습니다.

내부망(깃허브 접근 불가)일 때

constraints 파일을 미리 받아 빌드 컨텍스트에 넣고 사용하세요.

dockerfile

```
COPY constraints-3.11-2.9.2.txt /tmp/constraints.txt
RUN pip install --no-cache-dir \
    "apache-airflow-providers-jdbc==5.0.1" \
    --constraint /tmp/constraints.txt
```

2) 그래도 업그레이드가 일어난다면 “핀 고정 + 같은 커맨드”로 강제

정상에선 필요 없지만, 환경상 **pip**가 고집을 부리면 다음처럼 같은 한 줄에서 **Core** 버전도 명시해 해결할 수 있습니다. (재설치를 유발하긴 해도 버전은 안전합니다.)

dockerfile

```
RUN pip install --no-cache-dir \
    "apache-airflow==2.9.2" \
    "apache-airflow-providers-jdbc==5.0.1" \
    --constraint "${CONSTRAINT_URL}"
```

3) 설치 뒤 즉시 헬스체크 (필수에 가깝게 습관화)

bash

```
docker run --rm -it --entrypoint /bin/bash <이미지> -lc '
    which airflow && airflow version && \
    pip show apache-airflow | grep "Version|Location" && \
    echo "$PATH"
'
```

- 기대: **airflow version** → **2.9.2**, **pip show apache-airflow** → **2.9.2** / 위치는 보통 **/usr/local/lib/python3.11/site-packages**
- **which airflow** → **/usr/local/bin/airflow** 가 깔끔

왜 “jdbc만 추가”로 이런 일이?

정상이라면 **apache-airflow-providers-jdbc**의 요구조건이 **2.9.x**를 만족하므로 Core를 올릴 필요가 없습니다. 하지만 **constraints** 미사용 + (혹시) **-u/업그레이드 옵션**이 섞이면, pip가 “최신이 더 안전”하다고 판단해 **Core를 3.x**로 끌어올리는 경우가 있습니다. → 그래서 **항상 constraints**가 필요합니다. (Airflow 팀이 강력 권장)

딱 필요한 체크리스트

- Dockerfile의 설치 줄에 **--constraint** 붙였는지
- **pip install**에 **-U/--upgrade** 없는지
- 빌드 끝에 **which airflow && airflow version**을 한 번 실행해보는지

원하시면 현재 Dockerfile의 `pip install` 한 줄만 그대로 붙여주세요. 그 줄을 안전한 형태로 바꿔서 드릴게요.



requirements.txt 목록통해서 모듈을 설치하는데,
constraints.txt는 어떻게 사용하지?

좋은 질문이에요 👍

지금 쓰시는 방식이 `requirements.txt`에 패키지 목록을 적어놓고 `pip install -r requirements.txt`로 설치하는 거잖아요. 이때 **constraints** 파일은 “패키지 버전 상한/하한을 강제로 묶어주는 추가 가이드라인”으로 쓰입니다.

1. requirements.txt vs constraints.txt 차이

- **requirements.txt**

→ 내가 꼭 설치하고 싶은 패키지들 명시

예:

```
ini

pyspark==3.5.3
apache-airflow-providers-jdbc==5.0.1
```

- **constraints.txt**

→ 직접 설치하지 않지만, 의존성들이 설치될 때 버전을 강제로 이 버전으로 맞춰라 라는 규칙집

예: (Airflow 팀이 관리하는 공식 constraints)

```
makefile

alembic==1.12.0
flask==2.3.2
sqlalchemy==1.4.46
...
```

즉, `requirements.txt`는 “내가 설치할 모듈 목록”,

`constraints.txt`는 “그 모듈과 함께 설치될 다른 모듈 버전을 이 범위로 고정”입니다.

2. Dockerfile에서 함께 쓰는 방법

```
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"

COPY requirements.txt /requirements.txt
```

```
RUN pip install --no-cache-dir -r /requirements.txt --constraint "${CONSTRAINT_URL}"
USER airflow
```

포인트

- **--constraint "\${CONSTRAINT_URL}"** 옵션을 추가하면, pip이 requirements.txt에 적힌 패키지들을 설치하면서 **constraints.txt**에서 정의된 버전을 따라가게 됩니다.
- 만약 requirements.txt와 constraints.txt가 충돌하면, **requirements.txt** 우선이에요. (내가 직접 지정한 게 이깁니다)

3. 오프라인 환경 (내부망)일 때

인터넷 차단 환경이라면 constraints 파일을 미리 받아서 Docker 컨텍스트에 넣어둡니다.

```
bash

curl -o constraints-3.11-2.9.2.txt \
  https://raw.githubusercontent.com/apache/airflow/constraints-2.9.2/constraints-3.11.txt
```

그리고 Dockerfile에서:

```
dockerfile

COPY requirements.txt /requirements.txt
COPY constraints-3.11-2.9.2.txt /constraints.txt

RUN pip install --no-cache-dir -r /requirements.txt --constraint /constraints.txt
```

✅ 정리

- **requirements.txt** = 내가 설치할 패키지 목록
- **constraints.txt** = 의존성 버전 고정 규칙
- **pip install -r requirements.txt --constraint constraints.txt** 형태로 같이 사용

혹시 지금 환경이 인터넷 연결 가능한 곳인가요, 아니면 사내 Nexus 같은 프록시 저장소만 사용 가능한 환경인가요? 그에 따라 constraints.txt를 URL 그대로 쓸지, 파일로 가져다 둘지가 달라집니다.