**تفاوت Single-Stage vs Multi-Stage در ساخت ایمیج داکر**

**تعریف کوتاه**

* **Single-stage**: یک Dockerfile با یک FROM؛ همه‌ی ابزارهای Build و Runtime در همان ایمیج نهایی می‌مانند.
* **Multi-stage**: چند FROM پشت‌سرهم؛ مرحله‌ی Build از مرحله‌ی Runtime جداست و فقط خروجی‌های لازم (artifacts) به ایمیج نهایی کپی می‌شوند.

**پیامدهای کلیدی**

* **حجم ایمیج**
  + Single-stage: بزرگ‌تر چون کامپایلرها، هِدرها، cacheها و… هم در ایمیج نهایی می‌مانند.
  + Multi-stage: کوچک‌تر؛ فقط باینری/کد و وابستگی‌های Run لازم منتقل می‌شود. به‌عنوان Best Practice توصیه شده است.
* **امنیت**
  + Single-stage: سطح حمله (attack surface) بزرگ‌تر چون ابزارهای Build و Dev داخل ایمیج باقی می‌مانند.
  + Multi-stage: سطح حمله کوچک‌تر با حذف ابزارهای غیرضروری در Runtime.
* **سرعت Build و Cache**
  + Single-stage: ساده‌تر اما با تغییرات کوچک، cache لایه‌ها ممکن است بی‌اثر شود و Build طولانی شود.
  + Multi-stage: امکان موازی‌سازی منطقی مراحل و استفاده‌ی بهتر از cacheها (با ساختاردهی مناسب به مراحل).
* **قابلیت نگه‌داری**
  + Single-stage: ساده‌تر برای شروع، ولی با رشد پروژه مدیریت آن سخت و پرریسک‌تر می‌شود.
  + Multi-stage: تفکیک نقش‌ها (build vs run) نگه‌داری و ارتقا را تمیزتر می‌کند.

**چه زمانی کدام را انتخاب کنیم؟**

* **Single-stage** مناسب: پروژه‌های خیلی کوچک، نمونه‌سازی سریع، یا سناریوهای آموزشی کوتاه.
* **Multi-stage** مناسب پیشنهادی برای Production: وقتی حجم ایمیج، امنیت، سرعت انتشار و قابلیت کش مهم است (اکثر سرویس‌های واقعی).

**الگوی پایه برای پروژه‌های Python/FastAPI**

**Single-stage ساده اما حجیم**

# ---------- Single-Stage (DEV / ساده) ----------

FROM python:3.12-slim

# تنظیمات مفید پایتون برای کانتینر

ENV PYTHONDONTWRITEBYTECODE=1 \

PYTHONUNBUFFERED=1

WORKDIR /usr/src/core

# فقط فایل نیازمندی‌ها را اول کپی کن تا cache بهتر عمل کند

COPY ./requirements.txt .

# نصب وابستگی‌ها

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

# کد اپلیکیشن

COPY ./core .

# در محیط dev میشه از fastapi CLI استفاده کرد

EXPOSE 8000

CMD ["fastapi", "run", "--host", "0.0.0.0", "--port", "8000"]

* مشکل: ابزارهای build و cacheها در ایمیج نهایی می‌مانند ⇒ حجیم/کم‌امنیت.

**Multi-stage پیشنهادی**

# ---------- Stage 1: Builder ----------

FROM python:3.12-slim AS builder

# در صورت نیاز به build deps (فقط در استیج بیلدر نصب می‌مانند)

# اگر پکیج‌ها pure-python هستند، می‌توانید این خط را حذف کنید

RUN apt-get update && apt-get install -y --no-install-recommends build-essential \

&& rm -rf /var/lib/apt/lists/\*

WORKDIR /usr/src/core

# فقط requirements را کپی کن تا لایه‌ی کش پایدار بماند

COPY ./requirements.txt .

# ساخت wheelها (آفلاین نصب می‌کنیم)

RUN pip wheel --no-cache-dir --wheel-dir /wheels -r ./requirements.txt

# ---------- Stage 2: Runtime ----------

FROM python:3.12-slim AS runtime

ENV PYTHONDONTWRITEBYTECODE=1 \

PYTHONUNBUFFERED=1

WORKDIR /usr/src/core

# یک کاربر غیر روت برای امنیت بهتر

RUN useradd --create-home --shell /bin/bash appuser

# فقط wheelهای آماده و فایل requirements را از بیلدر کپی کن

COPY --from=builder /wheels /wheels

COPY ./requirements.txt .

# نصب صرفاً از wheelها (بدون دسترسی اینترنت و بدون build)

RUN pip install --no-cache-dir --no-index --find-links=/wheels -r ./requirements.txt

# کد اپلیکیشن

COPY ./core .

# مالکیت را به کاربر نهایی بده

RUN chown -R appuser:appuser /usr/src/core

USER appuser

# در Production معمولاً gunicorn/uvicorn به‌صورت workerدار اجرا می‌شود

EXPOSE 8000

# نمونه با uvicorn خالص:

# CMD ["uvicorn", "main:app", "--host=0.0.0.0", "--port=8000"]

# اگر ساختار پروژه‌ات فایل main را در core دارد:

# core/

# main.py -> app = FastAPI()

# آنگاه ماژول باید اینطور رفرنس شود: "core.main:app"

CMD ["uvicorn", "core.main:app", "--host=0.0.0.0", "--port", "8000"]

* مزیت: ابزارهای build وارد ایمیج نهایی نمی‌شوند؛ فقط wheelها و کد ⇒ ایمیج کوچک‌تر و امن‌تر.

الگوی تفکیک build و run که داکر توصیه کرده است.

نکته: اگر در Stage اول از Poetry/UV/… استفاده می‌کنیم، در Stage نهایی آن‌ها را منتقل نکنیم؛ تنها خروجی نصب (wheels/site-packages) را کپی کن تا حجم کم شود.

**نکات تکمیلی برای بهینه‌سازی**

* **.dockerignore** را تنظیم کنیم تا فایل‌های غیرلازم (venv، .git، tests، docs، media) وارد کانتکست نشود.
* **ترتیب لایه‌ها** را پایدار بچینیم تا cache بهتر عمل کند ابتدا dependency files، سپس کد.
* **Base image مناسب**: برای Python عموماً python:\*-slim عملی‌تر از Alpine است .
* **Rebuild منظم**: به‌صورت دوره‌ای build بدون cache برای دریافت به‌روزرسانی‌های امنیتی لایه‌های پایه.
* **Multi-platform** (در صورت نیاز): با docker buildx build --platform linux/amd64,linux/arm64 … اما این با multi-stage فرق دارد.

**در آخر:**

* **چرا Multi-stage؟** برای کاهش حجم، افزایش امنیت و جداسازی روشن Build/Run.
* **چه تغییری می‌دهد؟** چند FROM، کپی انتخابی artifacts به Stage نهایی.
* **نتیجه‌ی عملی**: ایمیج سبک‌تر (pull سریع‌تر، هزینه‌ی ذخیره‌سازی کمتر، سطح حمله کم‌تر، CI/CD سریع‌تر.