Docker

By Eng: Abdelrahman Ayman Lotfy

DevOps track

Table of contents

How to think effectively before writing docker file?

Step 1: Define the Goal of Your Dockerfile

Before writing a Dockerfile, you need to decide what this container is for. That decision will drive all the other choices (base image, size, layers, tools, optimization, etc.).

1. Runtime Environment

You only want the container to run your application.

- Use lightweight images → small, fast, production-ready.
- Avoid extra tools (no compilers, debuggers).

2. Development Environment

You want developers to work inside the container.

- Add dev tools: git, curl, vim, compilers, linters, etc.
- Mount code as a volume so edits apply live.
- Image will be bigger (not production-friendly).

3. Build Environment

You only use this container to build artifacts (not run them).
Use multi-stage builds to keep the final image clean.

4. Testing Environment

You need an image for CI/CD pipelines to run tests.

- Include testing frameworks.
- Doesn't need to be small.

- Not usually shipped to production.

4. Testing Environment

You need an image for **CI/CD pipelines** to run tests.

- Include testing frameworks.
- Doesn't need to be small.
- Not usually shipped to production.

Questions to Ask Yourself

1. Where will I run this container?
    - ➢ Local dev, staging, production, CI/CD?
2. What's my top priority?
    - ➢ Small image? Debuggability? Speed? Convenience?
3. Who will use it?
    - ➢ Developers, CI/CD pipeline, or end users?
4. Do I need build tools inside the final image?
    - ➢ If no ➔ use multi-stage build.
5. Do I need easy configuration?
    - ➢ If yes ➔ use environment variables instead of hardcoding.

Step 2: choose your base image

After defining the Goal of Your Dockerfile (Run time | Development | Build environment | Testing)

Now choose your base image

- ❖ If the application is java so the image could be openjdk:17-jdk-alpine
- ❖ If the application is Node.js so the image could be node:20-alpine

- ❖ If the application is python so the image could be python:3.12-slim

If your environment is production so choose the most lightweight base image like alpine and slim images don't use a latest version to ensure that the image is stable and not changing by time.

There is Official language images (e.g., node, python, openjdk, golang) use the best tag of it like alpine and slim

- Vendor/OS images (mcr.microsoft.com, gcr.io, eclipse-temurin, amazoncorretto) suitable when you build .net or JVM apps
- Distroless / Minimal Runtime (gcr.io/distroless) like gcr.io/distroless/java17 ➔ doesn't contain shell or package manager suitable for production as low size and high security ,Also suitable for multi stage when you only need to use the artifact (.jar or binary) from the build stage to the run stage.
- If your environment is testing you can use big images like ubuntu:22.04

Step 3: Handling your Dockerfile

- Make sure to copy the file like package.json or requirements.txt first before copying code as these files don't change too much so make sure to copy it first
- Layers optimization don't forget that each line in your docker file represents a layer so many lines cause many layers so big size and time arise to avoid this you can use many commands in one line like:

```
RUN apt-get update && apt-get install -y \
curl git \
&& rm -rf /var/lib/apt/lists/*
```

  Here I put all in one layer using &&
- Also you can use .dockerignore to avoid put files like (.git , node_modules)

- Take care about security don't use root user if it isn't necessary

---

Step 4: Review the difference between Docker instructions

❖ **CMD vs ENTRYPOINT**

The following example shows that CMD the first thing that will be executed also you can override it in the RUN time

```
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ docker run cmd:v1
Hello from CMD
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ docker run cmd:v1 echo "Override CMD"
Override CMD
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ cat Dockerfile
FROM ubuntu:rolling
CMD ["echo","Hello from CMD"]
```

The following shows the ENTRYPOINT you can't modify it in run but you can add an argument to it

```
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ docker run cmd:v2
Message:
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ docker run cmd:v2 attach argument
Message:  attach argument
abdu@DESKTOP-7UHR1JU:/mnt/d/depi_tasks/test$ cat Dockerfile
FROM ubuntu:rolling
ENTRYPOINT ["echo","Message: "]
```

❖ **ADD vs COPY**

```
# Copy app source code
COPY . /app
```

The previous copy the source code only

```
ADD app.tar.gz /app
ADD https://example.com/file.txt /tmp/file.txt
```

ADD like COPY but using ADD it will extract the files from .tar.gz also you can download a URL but it isn't preferred as low security.

- **ARG vs ENV**

  ARG in the build time only you can use it to store in it image name as it will disappear after build but ENV appear in run time

---

Step 5: Build & RUN image

docker build name:tag .

docker run name:tag

check that image exists using docker images

you can also check docker history name:tag to see layers

Conclusion

Writing a good Dockerfile is about clear goals, smart layering, minimal and secure images, and correct runtime configuration.
If you think in this order: Goal ➔ Base ➔ Layers ➔ Security ➔ ENTRYPOINT/CMD ➔ Variables ➔ Inspect then you'll always end up with a professional, efficient Dockerfile.