Dockerfile : Best practices for building an image

Maxime Lafond Follow
Jun 17 · 5 min read ★

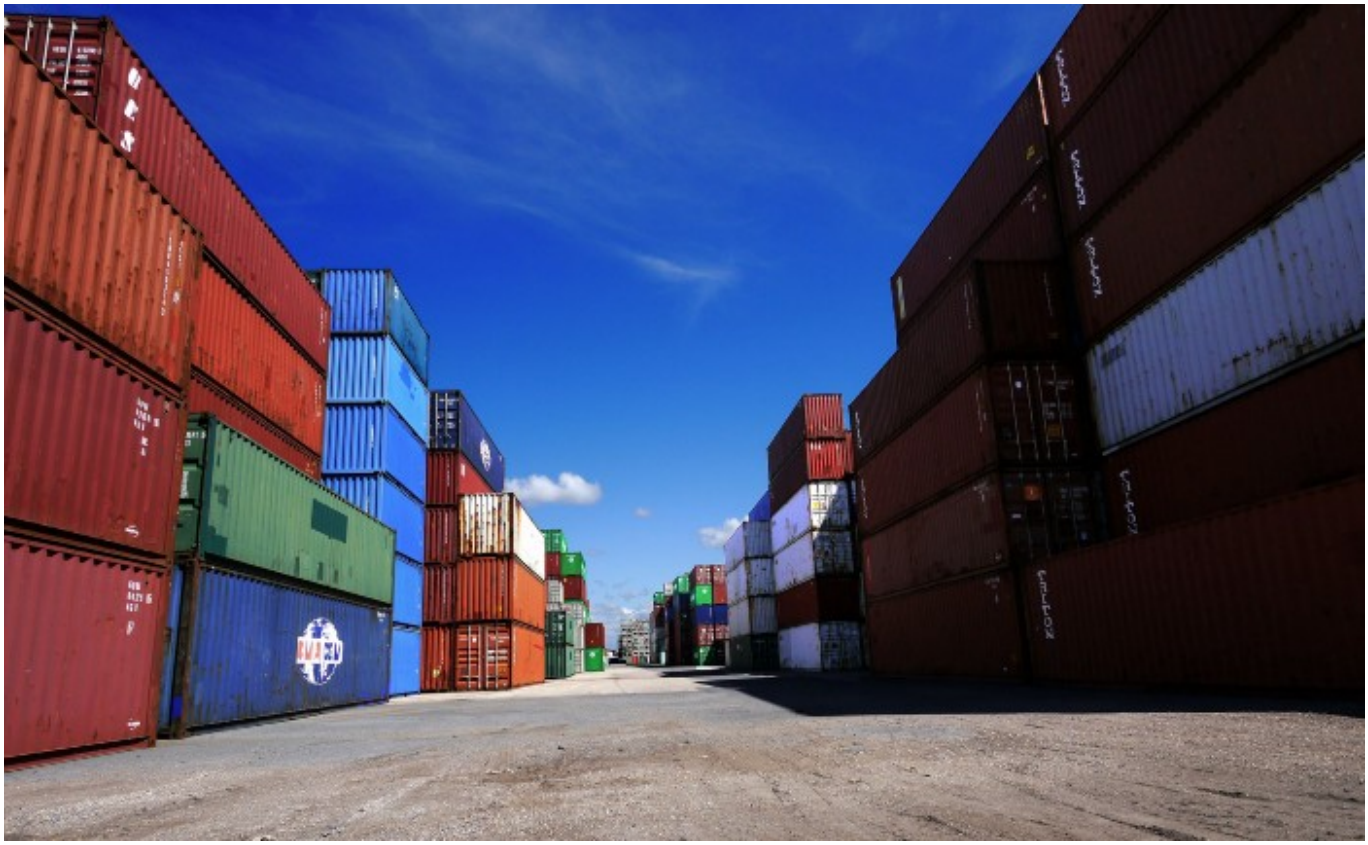

Photo : trackmetal from Pixabay

The adoption of the containers by the community and within enterprises in the last few years is incredible. How can we not like a technology who bring us more stability, more control on the solution and take generally less time to deploy versus the same application on a server ? It's surely some of the reasons why the containers are so popular when DevOps engineers are designing a CI/CD system.

The flexibility of creating an image is a good asset but need to be oriented by some guidance and good practices. Why ? Mostly for fighting the size of the container, who can impact the performance and the security vulnerabilities, but also for improving the re usability, the readability and the maintainability.

The list of good practices below is mostly from my personal experience working as a DevOps Specialist for different companies. It's not in a particular order, they are all for me important to be understood and to be applied.

1. **OS base image** : The OS base image can have a big impact on the size of an image and the security vulnerabilities. It's always a good practice to go with a lightweight base image like alpine (~5 MB) versus a standard base image like ubuntu (~188 MB). The tools can be manually added for your needs.

2. **Base images** : The base images between the OS base image and the final image need to be designed properly for having a good re usability and maintainability. Each one need to have a specific goal and to be as generic as possible for being reused by other images. For a microservice, the image architecture could be as follow :
   (1) OS base image
   (2) Middleware image
   (3) Tooling image
   (4) Application image
   We can in this case reused the first 3 base images for the microservices with the same needs.

3. **Define users with minimal privileges** : The users we create in a Dockerfile need to follow the same good practices as when creating Linux users. They should only have the privileges needed and no more. The ROOT user should not be used for starting the container for obvious reasons and it's a good idea to not change user to often in the Dockerfile because of layer created each time.

4. **Leverage .dockerignore** : The .dockerignore file is really useful to exclude from the context when building an image all the files who are not necessary in the build. This optimization is really appropriate when building several times a day multiple application in a CI/CD system.

5. **Be specific with ADD or COPY** : Selecting only the folder and /or the files can take a little more time but it is an easy way to optimize the size of an image and reduce the security vulnerabilities.

6. **Specific tag instead of fixed tag** : A tag is an alias for a specific image version. One of the biggest confusion around the tagging is the “latest” tag. It’s in theory used for the latest image version available but because it’s manually managed, we have no guarantee it’s really the case. Maybe the maintainer manage the latest tag in his own way or just forgot to update the tag. Also, using a fixed tag like “latest” mean that the image version can change and so rebuilding an image can bring some side effects. It’s recommended to use a specific tag like the image version. This version is upgraded when you are ready with a proper life cycle management.

7. **Image life cycle management** : Security breach can happen if you are on a deprecated version of an image. A proper life cycle management need to be define in your team to validate and to update if necessary the image version. It can be summarize to : Who is in charge ? What’s the frequency ?

8. **Sensitive information** : Environment variables are not a good solution for managing secrets because (1) no encryption is done and (2) how easy it is to retrieve the data. Solutions exists on the market for managing secrets and few of them are designed for Kubernetes. An example is Hashicorp vault and their architecture for yes managing secrets but also handling the vault connection. The Kubernetes secrets need to be avoid because of the data who is only encoded with base64. No encryption method is used.

9. **Multistage builds** : The multistage builds feature introduced in Docker V17.05 can be very useful when having some processing in the construction of the image who imply tools / files but only for a short period of time. These tools or files are not needed for the actual process running in the ENTRYPOINT. By example, a java microservice using maven could add the steps of building the artifact directly in the Dockerfile. It’s great but no need to keep the tools like maven or the temporary files, only the artifact produced is needed. The multistage build who is basically multiple FROM in the Dockerfile create this possibility to only import in the latest FROM the file(s) needed. See Example 1

10. **Combine commands** : Each command in a Dockerfile add one layer to the image. We reduce the size of the image when reducing the number of layers and one way to do it is by combining commands. The easiest example is the RUN command. Instead of having multiple RUN command, each instructions can be combined and separated by “&&”. See Example 2

Example 1 : Multistage builds

```
FROM 3.6.3-jdk-11
COPY src ~/app/src
COPY pom.xml ~/app
RUN mvn clean install -f ~/app/pom.xml

FROM openjdk:11-jre-slim
COPY — from=0 ~/app/target/ARTIFACT-1.0.0-SNAPSHOT.jar
/SOME/FOLDER/ARTIFACT-1.0.0-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT [“java”,”-jar”,”/SOME/FOLDER/ARTIFACT-1.0.0-SNAPSHOT.jar”]
```

Example 2 : Combining RUN commands

```
RUN apt-get update
RUN apt-get install -y python3-pip
RUN cd ~

# Can be converted to

RUN apt-get update && apt-get install -y python3-pip && cd ~
```

Of course a lot more can be discussed around the proper way to build an image. The image architecture, multistage builds, secrets management and image sizing can all have a dedicated story. There is some links to the official documentations if you are interested to go deeper

- Multistage builds : https://docs.docker.com/develop/develop-images/multistage-build/
- Best practices : https://docs.docker.com/develop/dev-best-practices/

Docker Dockerfiles Docker Image