10 Tips on Writing a Proper Dockerfile

Writing a proper Dockerfile is not too difficult

Meysam Azad May 7, 2020 · 6 min read

Image via IMGBIN.com

There are some tips and tricks to write a proper Dockerfile. And for the most part, writing a “proper” Dockerfile is simple, though not easy.

Without further ado, let’s dive right in to the topic.

1. The most lightweight base image possible

Right from the very beginning, you’d wanna start building from the most lightweight base image possible. Something usually related to the alpine docker image.

For example for running a python web application inside a docker container, I’d start my Dockerfile with something like the following:

```
FROM python:3.8.2-alpine
```

This makes your final image much smaller and desirable.

2. Place your most static commands at the top

Any command that is susceptible to the least changes possible in the future, should be placed at the top of your Dockerfile. That way you can take advantage of caching layers of docker, which is the act of using the result of

previous builds in the current one.

For example, I would put the following lines just below my `FROM` line.

```
RUN wget https://dumb-init-url -O /usr/local/bin/dumb-init && \
    chmod +x /usr/local/bin/dumb-init && \
    apk update && \
    apk add curl

LABEL maintainer="Meysam Azad"
LABEL company="My awesome company"

ARG service_workdir=/app
ARG service_port=8000
ARG username=webapp
ARG data_path=/data
ARG wheelhouse_directory=/wheelhouse
```

No matter what happens in the future, I am sure that these lines will most probably never change. These are my “static” lines and are placed at the top of my Dockerfile.

The rest of the commands would normally be placed below these, and should always be the commands that change more often in each build.

3. Never ever run your app with superuser

Just because your application is running inside a docker container, which is a isolated entity with it’s own file system and specifications, doesn’t mean you can ignore years of Linux security best practices.

That being said, the following lines would normally come after my “static” lines.

```
RUN adduser -D ${username} && \
        mkdir -p ${data_path} && \
        chown ${username} ${data_path}

...
# somewhere along the way
USER ${USERNAME}  # initialized below
```

3 points worth mentioning here:

1. I have taken advantage of docker build arguments `username` from the above. These arguments are configurable upon every build.
2. These commands are valid inside an alpine image, so you’d better find your own commands instead of memorizing these lines.
3. You can see that these lines are also “static” lines and would never change except for the times that a build argument have changed. You can call these lines “semi-static”.

4. Use ENV and ARG instead of hard-coded values

As you saw previously, we made use of `ARG` command. And next, it’s cousin is `ENV`. You would totally use these combination to take your docker image to the next level by making it much more configurable for every use case.

For example these would be my `ENV` lines.

```
ENV WHEELHOUSE_DIR=${WHEELHOUSE_DIR:-$wheelhouse_directory}
ENV SERVICE_WORKDIR=${SERVICE_WORKDIR:-$service_workdir}
ENV SERVICE_PORT=${SERVICE_PORT:-$service_port}
ENV USERNAME=${USERNAME:-$username}
ENV DATA_PATH=${DATA_PATH:-$data_path}
```

I have used `:-` syntax in above lines which means: “if the argument on the left side is empty, use the value provided on the right side”. And in the above cases, all the right side arguments are variables holding a value from previous `ARG` lines.

5. Make use of docker multi-stage feature

This feature is a practical one. You can write commands on different stages, and run them on different occasions: `dev`, `test` or `prod`. You can also use this feature if you want to build on different versions. Like on A/B testing for example.

For example I would start my first line of docker like this:

```
FROM python:3.8.2-alpine AS base
```

And then, down the line when I’m ready to start the application, I would write something similar to this:

```
FROM base AS test
# some commands to run application on testing environment
...

FROM base AS dev
# other commands to run application on development environment
...

FROM base AS prod
# yet another set of commands
...
```

Now every time you want to build your application, you could either use:

```
docker build --target dev -t my-cool-image .
```

or use the following in a docker-compose file:

```
build:
  context: .
  target: dev
```

6. Copy contents within different stages if needed

Suppose you're trying to run a web app inside a container. And let's say your application needs to be compiled, or something similar.

You don't need to have both of *compile* and *run* phase in the same stage, and rather you'd fetch dependencies and compile on one stage, and then run the binary executable output on another.

For example when I want to run a python app and there are some external dependencies involved, I would download those dependencies in one stage:

```
FROM base as download_wheels

RUN apk add gcc musl-dev linux-headers

COPY --chown=${USERNAME} setup.py ./

RUN pip wheel -w ${WHEELHOUSE_DIR} .
```

And then I would copy these dependencies to my final stage to be able to run the app:

```
COPY --from=download_wheels --chown=${USERNAME} \
    ${WHEELHOUSE_DIRECTORY} ${WHEELHOUSE_DIRECTORY}

...

RUN pip install --no-index -f ${WHEELHOUSE_DIRECTORY} .
```

You can see that I have also changed the owner of the copied file to my own desirable user with `--chown`.

7. Always put a health check

You wouldn't know if your application is running in full-power just by simply using `docker ps`. Because this command is only capable of telling if a container is running or not, and not able to figure if your application has been stuck somewhere, or whether it is fully operational and ready to service.

I would normally add a line like this for my web application:

```
HEALTHCHECK \
    --interval=10s \
    --timeout=5s \
    --start-period=10s \
    --retries=5 \
    CMD curl localhost:${SERVICE_PORT}/v1/ \
    || exit 1
```

Everything is self explanatory I hope, but let's not be presumptuous.

We try to run the command `curl localhost:${SERVICE_PORT}/v1/` every 10 seconds, waiting 5 second for it to respond, starting 10 seconds after the container is started, and retrying 5 times if it fails. Otherwise just return a

non-zero status code, informing the docker service that we're not feeling good.

The endpoint `/v1/` for me, is usually an endpoint which checks if all the other dependencies are also up and running. Like RabbitMQ, Redis or any other service that my app is interacting with.

8. Expose your ports if it's meant to be

In a typical web application, you would run your app in some port, and so my advice to you is to expose that. This is desirable because when someone else is working with your image and inspects it using `docker inspect` they would easily find out which port to communicate.

It's pretty obvious but just for reference:

```
EXPOSE ${SERVICE_PORT}
```

9. Also expose your working directory

Has it ever occurred to you that you have pulled a docker image from the registry, and without any knowledge about the image, you had to go through `docker inspect` to figure out it's port and it's data directory. Like `postgres` for example which I almost always forget where it stores its data, right before I check it using `docker inspect postgres`.

That's why exposing your working directory is important and desirable. Just like exposing your ports, other people, or even yourself when coming back to it after sometime, would need to see the working directory (or data directory) of an image Therefore try to do this in your image where possible:

```
WORKDIR ${DATA_PATH}
```

10. Have different ENTRYPOINT and CMD

This is mainly because, on a regular day job, you would do the following whenever you feel the need:

```
docker exec -it my-container bash
```

This command will be executed after the `ENTRYPOINT` and so you would place something powerful in there.

For me I would normally use something like this:

```
ENTRYPOINT ["dumb-init", "--"]
CMD ["sh", "entrypoint.sh"]
```

Notice how I used brackets `[` and `]`. This is most desirable. Because if you use this notion instead:

```
CMD sh entrypoint.sh
```

docker would run `["sh", "-c", "sh", "entrypoint.sh"]` which is very unnecessary if you ask me.

Dockerfile

If you want to get your hands on the complete Dockerfile, this is for you. Enjoy!

```dockerfile
# ================================ Base Stage ================================
FROM python:3.8.2-alpine as base

RUN wget https://github.com/Yelp/dumb-init/releases/download/v1.2.2/dumb-init_1.2.2_amd6
        -O /usr/local/bin/dumb-init && \
        chmod +x /usr/local/bin/dumb-init && \
        apk add curl  # used for healthcheck

LABEL maintainer="Meysam Azad <MeysamAzad81@yahoo.com>"
LABEL company="My Awesome Company"
LABEL name="WebApp"

ARG wheelhouse_directory=/wheelhouse
ARG service_workdir=/app/
ARG service_version=latest
ARG service_port=8000
ARG username=webapp
ARG data_path=/data

RUN adduser -D ${username} && \
        mkdir -p ${data_path} && \
        chown ${username} ${data_path}

ENV WHEELHOUSE_DIRECTORY=${WHEELHOUSE_DIRECTORY:-$wheelhouse_directory}
ENV SERVICE_WORKDIR=${SERVICE_WORKDIR:-$service_workdir}
ENV SERVICE_VERSION=${SERVICE_VERSION:-$service_version}
ENV SERVICE_PORT=${SERVICE_PORT:-$service_port}
ENV USERNAME=${USERNAME:-$username}
ENV DATA_PATH=${DATA_PATH:-$data_path}

WORKDIR ${SERVICE_WORKDIR}

VOLUME [ "${DATA_PATH}" ]

RUN pip install -U pip

# ============================== Download Wheels ==============================
FROM base as download_wheels

RUN apk add gcc musl-dev linux-headers

COPY --chown=${USERNAME} setup.py ./

RUN pip wheel -w ${WHEELHOUSE_DIRECTORY} .

# ================================ Build Service ================================
FROM base as builder

EXPOSE ${SERVICE_PORT}

HEALTHCHECK --interval=10s --timeout=5s --start-period=10s --retries=5 \
        CMD curl localhost:${SERVICE_PORT}/v1/ || exit 1

COPY --chown=${USERNAME} ./src .
COPY --chown=${USERNAME} entrypoint.sh .
COPY --from=download_wheels --chown=${USERNAME} ${WHEELHOUSE_DIRECTORY} ${WHEELHOUSE_DIR
COPY --chown=${USERNAME} setup.py ./

RUN pip install --no-index -f ${WHEELHOUSE_DIRECTORY} .
Your email
USER ${USERNAME}

ENTRYPOINT [ "dumb-init", "--" ]
CMD [ "sh", "entrypoint.sh" ]
```

Dockerfile hosted with ❤ by **GitHub** **view raw**

Conclusion

That’s it guys. In this article I have shared with you one of the coolest Dockerfile that I’ve come up with. I hope you enjoyed and got something out of it.
I’m also hoping to see your cool Dockerfile sometimes.

Acknowledgement

If you liked the above content, follow me as I plan to write regularly. Which I would say feels pretty good t feel free to take a look at my other works as well.

Tmux: A Beginner's Guide

Turn your single terminal into many.

medium.com

Clean Architecture Simplified

Explaining how clean architecture works in a friendly manner

medium.com

How To Write Tests For Your Python Web App

A practical guide for adding tests and coverage to your project

medium.com
