Dockerfiles

Florent Glück - florent.gluck@hesge.ch

May 04, 2025

ISC - HEPIA

Dockerfile

- A Dockerfile is a text file containing instructions on how to build an image
- Dockerfile = an image' **source code** made up of **instructions**
- As simple as running **`docker buildx build`** on a Dockerfile to build an image!
- Requires the **`docker-buildx`** plugin on the client
  - on Ubuntu/Debian distributions, install it with:

```
sudo apt-get install docker-buildx
```

Why a Dockerfile?

- Most images are generic and won't fullfill our exact needs
- **Solution**: to customize an existing image!

Image creation and usage workflow (example)

1. Describe the **image** contents by writing instructions in a **Dockerfile**:

```
vim Dockerfile
```

Image creation and usage workflow (example)

1. Describe the **image** contents by writing instructions in a **Dockerfile**:

```
vim Dockerfile
```

2. **Build** the **image** from the Dockerfile:

```
docker buildx build -f Dockerfile -t myimage:v1 .
```

Image creation and usage workflow (example)

1. Describe the **image** contents by writing instructions in a **Dockerfile**:

```
vim Dockerfile
```

2. **Build** the **image** from the Dockerfile:

```
docker buildx build -f Dockerfile -t myimage:v1 .
```

3. **Instantiate a container** from the newly built image:

```
docker run -it --rm myimage:v1
```

Main instructions

`FROM`	base image to use
`ENV`	define an environment variable (exists in Dockerfile as well as at container execution)
`RUN`	execute commands
`COPY/ADD`	add files to the image
`CMD/ENTRYPOINT`	command to execute when the image is instantiated
`WORKDIR`	set the working directory (in Dockerfile and container execution), `/` by default; creates the directory if nonexistent
`USER`	set user name/UID and user group/GID to use for subsequent commands (including at container execution)

Dockerfile basic contents workflow (example)

1. Define the base image with `FROM`:

```
FROM alpine:3.21
```

Dockerfile basic contents workflow (example)

1. Define the base image with `FROM`:

```
FROM alpine:3.21
```

2. Add instructions, e.g. packages to install, actions to perform, etc. with `RUN`:

```
RUN apk update        # equivalent to apt-get update in Ubuntu/Debian
RUN apk add lighttpd  # the package creates a lighttpd user and group
```

Dockerfile basic contents workflow (example)

1. Define the base image with `FROM`:

```
FROM alpine:3.21
```

2. Add instructions, e.g. packages to install, actions to perform, etc. with `RUN`:

```
RUN apk update        # equivalent to apt-get update in Ubuntu/Debian
RUN apk add lighttpd  # the package creates a lighttpd user and group
```

3. Copy files from local filesystem into the image with `COPY` or `ADD`:

```
COPY index.html /var/www/localhost/htdocs
```

Dockerfile basic contents workflow (example)

1. Define the base image with `FROM`:

```
FROM alpine:3.21
```

2. Add instructions, e.g. packages to install, actions to perform, etc. with `RUN`:

```
RUN apk update        # equivalent to apt-get update in Ubuntu/Debian
RUN apk add lighttpd  # the package creates a lighttpd user and group
```

3. Copy files from local filesystem into the image with `COPY` or `ADD`:

```
COPY index.html /var/www/localhost/htdocs
```

4. Define default command/entry point with `CMD` or `ENTRYPOINT`:

```
CMD ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
```

Example: image description, creation and container instantiation

1. **Write** a file named `Dockerfile.lighttpd` with:

```
FROM alpine:3.21
RUN apk update
RUN apk add lighttpd
COPY index.html /var/www/localhost/htdocs
CMD ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
```

2. **Build** an image named `mylighttpd` and tagged `v1`:

```
docker buildx build -f Dockerfile.lighttpd -t mylighttpd:v1 .
```

Example: image description, creation and container instantiation

1. **Write** a file named `Dockerfile.lighttpd` with:

```
FROM alpine:3.21
RUN apk update
RUN apk add lighttpd
COPY index.html /var/www/localhost/htdocs
CMD ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
```

2. **Build** an image named `mylighttpd` and tagged `v1`:

```
docker buildx build -f Dockerfile.lighttpd -t mylighttpd:v1 .
```

3. **Instantiate a container** named `websrv` from the newly built image:

```
docker run -it --rm --name websrv mylighttpd:v1
```

Environment variables (ENV)

- Environment variables can be declared in a Dockerfile using `ENV`, e.g.:

```
ENV EDITOR=vi
```

- These variables **still exist** during the life-cycle of the container!
- The `DEBIAN_FRONTEND` variable is particularly useful when set to `noninteractive`[1]

[1] See `man 7 debconf` as to why

Injecting environment variables at runtime

- Environment variables (here `PIPO` and `blah`) can be injected when running a container (runtime):

```
docker run -e PIPO=1234 -e blah=xyz alpine
```

- Environment variables can also be defined in an *env* file:

```
docker run --env-file pipo alpine
```

  where the `pipo` *env* file contains:

```
PIPO=1234
blah=xyz
```

COPY vs ADD

- `COPY/ADD` both copy files from a "source" into a Docker image
- `COPY` can only copy a local file or directory from the host into the image
- `ADD` is similar to `COPY` but more powerful as it supports:
  - wildcards:
    - `ADD hom* /mydir/`
    - `ADD hom?.txt /mydir/`
  - decompressing local archives (`.tar.gz`, etc.) into the image
  - retrieving remote files (url)
  - cloning a git repository

COPY vs ADD: example

Copy `index.html` in the client's current dir to `/var/www` in the image:

```
COPY index.html /var/www
```

Copy `data.tar.gz` in the client's current dir to `/var/www` in the image:

```
COPY data.tar.gz /var/www
```

Decompress `data.tar.gz` to `/var/www` in the image:

```
ADD data.tar.gz /var/www
```

Clone the git repository into `/virtu` in the image:

```
ADD https://gitedu.hesge.ch/flg_courses/virtualization/virtualization_pub_spring25.git /virtu
```

CMD

- `CMD` specifies which command to execute when the image is instantiated:

```
CMD ["command","arg1","arg2",...]
CMD command arg1 arg2
```

  Avoid the last variant as it executes `command` with `/bin/sh -c`

- `CMD` can be overriden or set when running a container, as shown below with `COMMAND ARG...`:

```
docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]
```

- Only the **last** `CMD` of a Dockerfile is executed!

ENTRYPOINT

- `ENTRYPOINT` also specifies which command to execute when the image is instantiated
- Possible to **override** it when `--entrypoint` is passed to `run`
- What's the purpose of `ENTRYPOINT` given `CMD` already exists?
  - `CMD` is **appended** to `ENTRYPOINT`!

```
ENTRYPOINT ["git"]      # At container execution, will execute:
CMD ["--help"]          # git --help
```

- `ENTRYPOINT` + `CMD` allows to specify **default** arguments which can be **overriden**!

ENTRYPOINT vs CMD

- `ENTRYPOINT` should be defined when using the container as an executable
- `CMD` should be used as a way of defining default arguments for an `ENTRYPOINT` command or for executing commands in a container
- `CMD` is overridden when running the container with alternative arguments
- If `CMD` is defined in a parent image, setting `ENTRYPOINT` will **reset** `CMD` to an **empty value**
  - in this case, `CMD` must be defined in the current image to have a value

ENTRYPOINT vs CMD: examples (1/3)

```
FROM alpine
CMD ["/bin/echo"]
```

- `docker run image` ⟶ (prints an empty line)
- `docker run image blah` ⟶ `error: executable "blah" not found!`
- `docker run image /bin/echo blah` ⟶ `blah`

```
FROM alpine
ENTRYPOINT ["/bin/echo"]
```

- `docker run image` ⟶ (prints an empty line)
- `docker run image blah` ⟶ `blah`
- `docker run image pipo molo` ⟶ `pipo molo`

ENTRYPOINT vs CMD: examples (2/3)

```
FROM alpine
ENTRYPOINT ["/bin/echo"]
CMD ["blah"]
```

- `docker run image` ⟶ `blah`
- `docker run image pipo molo` ⟶ `pipo molo`

```
FROM alpine
CMD ["ls"]
CMD ["/bin/echo"]
```

- `docker run image` ⟶ (prints an empty line)
- `docker run image blah` ⟶ `error: executable "blah" not found!`
- `docker run image pipo molo` ⟶ `error: executable "pipo" not found!`

ENTRYPOINT vs CMD: examples (3/3)

```
FROM alpine
```

- `docker run image` ⟶ executes `/bin/sh` and terminates (since non-interactive)
- `docker run image /bin/echo blah` ⟶ `blah`

Identify an image's CMD or/and ENTRYPOINT

Let `myecho` be an image with the following contents:

```
FROM alpine
ENTRYPOINT ["/bin/echo"]
CMD ["pipo"]
```

Use `docker history` to display the `CMD` or `ENTRYPOINT` of an image:

```
$ docker history myecho
IMAGE          CREATED         CREATED BY                                      SIZE
f5044de936b8 2 months ago  CMD ["pipo"]                                    0B
<missing>      2 months ago  ENTRYPOINT ["/bin/echo"]                        0B
<missing>      2 months ago  CMD ["/bin/sh"]                                 0B
<missing>      2 months ago  ADD alpine-minirootfs-3.21.3-x86_64.tar.gz /…  7.83MB
```

Building an image

- Use `docker buildx build` to build an image from a Dockerfile and a *build context*
- The `build` command looks for a file named `Dockerfile`
  - to specify a different file, use `-f xxx`
- The build is run by the **`dockerd`** daemon ⟶ on the server
- The newly built image is stored in the **server's local repository**
- To build the `myimage:mytag` image whose contents is defined in `tagada`:

```
docker buildx build . -t myimage:mytag -f tagada
```

Build context (1/2)

- The **build context** is a file hierarchy, defined by a directory, and available to the image builder
- At build time, the client **sends** the **entire context** **recursively** to the daemon!
- Examples:
  - specifies the current directory (`.`) as build context:

```
docker buildx build . -t myimage:mytag
```

  - specifies `/tmp/pipo/` as build context:

```
docker buildx build /tmp/pipo -t myimage:mytag
```

Build context (2/2)

- Use `.dockerignore` to specify which files not to be sent to the daemon (similar to `.gitignore`)
- Files copied into the image with `ADD` or `COPY` **must be present** in the context
- Paths specified to `ADD` or `COPY` are **relative** to the context source!
- The location of the Dockerfile has no impact on `ADD` or `COPY`
- For **performance** and **security** reason, make sure the context is minimal!

Image size: why layers matter (1/4)

- Every `RUN`, `COPY` and `ADD` line creates a new layer
- Below, lines 1, 4, 5, 6, 7, 8, 9, and 10 each create a new layer:

```
1  FROM alpine:3.21
2  ENV EDITOR=vi
3  WORKDIR /tmp
4  RUN apk update
5  RUN apk add wget
6  RUN wget https://kernel.org/pub/linux/kernel/v3.0/linux-3.0.tar.gz
7  RUN tar fxz linux-3.0.tar.gz
8  RUN mv linux-3.0/Documentation /home/
9  RUN rm -rf linux-3.0
10 RUN rm linux-3.0.tar.gz
11 WORKDIR /home/Documentation
12 CMD ["ls -F"]
```

- What's the **issue** with this Dockerfile?

Image size: why layers matter (2/4)

- Output of `docker history` on the previous image:

```
$ docker history linuxdoc:bad

IMAGE          CREATED BY                                      SIZE
8e86bc5be9e1   /bin/sh -c #(nop)  CMD ["ls -F"]                0B
81f9e7660397   /bin/sh -c #(nop) WORKDIR /home/Documentation   0B
3997c801e030   /bin/sh -c rm linux-3.0.tar.gz                  0B
b3cea0d6da79   /bin/sh -c rm -rf linux-3.0                     0B
71a0aea74ee5   /bin/sh -c mv linux-3.0/Documentation /home/     14.3MB
2fbc5e9e0770   /bin/sh -c tar fxz linux-3.0.tar.gz              421MB
1e23ef77e48e   /bin/sh -c wget https://kernel.org/pub/linux...  96.7MB
d4a7ec1db569   /bin/sh -c apk add wget                          2.33MB
4c0591a76730   /bin/sh -c apk update                           2.47MB
970403b690cf   /bin/sh -c #(nop) WORKDIR /tmp                  0B
ff07dd56bc06   /bin/sh -c #(nop)  ENV EDITOR=vi                0B
8471affe5de5   /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      /bin/sh -c #(nop) ADD file:970e6b2578ef73457...  5.55MB
```

- Image size: 543MB!

Image size: why layers matter (3/4)

- Remember that:
  - each layer represents a **delta of the changes** from the previous layer
  - a layer's contents **is never** removed!
- Previous Dockerfile rewritten to avoid wasting space:

```
1  FROM alpine:3.21
2  ENV EDITOR=vi
3  WORKDIR /tmp
4  RUN apk update && apk add wget && \
5      wget https://kernel.org/pub/linux/kernel/v3.0/linux-3.0.tar.gz && \
6      tar fxz linux-3.0.tar.gz && \
7      mv linux-3.0/Documentation /home/ && \
8      rm -rf linux-3.0 && \
9      rm linux-3.0.tar.gz && \
10     apk del wget
11 WORKDIR /home/Documentation
12 CMD ["ls -F"]
```

Image size: why layers matter (4/4)

- Output of `docker history` on the previous image:

```
$ docker history linuxdoc:good

IMAGE          CREATED BY                                      SIZE
25572b8c3b4f   /bin/sh -c #(nop) CMD ["ls -F"]                 0B
8c2c839402cd   /bin/sh -c #(nop) WORKDIR /home/Documentation   0B
b36242c99985   /bin/sh -c apk update && apk add wget && wget https://kern... 16.77MB
994e124747d2   /bin/sh -c #(nop) WORKDIR /tmp                  0B
098cd05113f9   /bin/sh -c #(nop) ENV EDITOR=vi                 0B
8471affe5de5   /bin/sh -c #(nop) CMD ["/bin/sh"]               0B
<missing>      /bin/sh -c #(nop) ADD file:970e6b2578ef73457...  5.55MB
```

- Image size: 22.4MB

Exporting an image or container's filesystem

- `docker export` ⟶ exports a **container**'s filesystem into an image as a tar archive (to `stdout`)
  - archive contains the container's **root filesystem** only
- `docker save` ⟶ saves an **image**'s filesystem into an image as a tar archive (to `stdout`)
  - archive contains the image's various **layers' contents** and **meta-data** (e.g. entry-point, etc.)
- Note that `docker import` and `docker load` perform the opposite operations

Docker export: archive's contents

```
$ docker run --name myc openjdk:latest
$ docker export myc > rootfs.tar && tar tf rootfs.tar
.dockerenv
bin
boot/
dev/
dev/console
dev/full
dev/initctl
dev/null
dev/ptmx
dev/pts/
dev/random
dev/shm/
dev/tty
dev/tty0
dev/urandom
dev/zero
etc/
etc/aliases
etc/alternatives/
etc/alternatives/jar
etc/alternatives/jarsigner
...
```

Docker save: archive's contents

```
$ docker save openjdk:latest > rootfs.tar && tar tf rootfs.tar
34aba91dbd1358ac48d86995dad4620c73ead6466f94f8dfce622a59892fcb5f.json
8e3b009939a813b63c7c2bae06327fa868cdacb2f33edf524d436a1be3036b9a/
8e3b009939a813b63c7c2bae06327fa868cdacb2f33edf524d436a1be3036b9a/VERSION
8e3b009939a813b63c7c2bae06327fa868cdacb2f33edf524d436a1be3036b9a/json
8e3b009939a813b63c7c2bae06327fa868cdacb2f33edf524d436a1be3036b9a/layer.tar
b04eff89da618fd519087acde0f769f144c30e8b3b6c21cf2310248d24c52015/
b04eff89da618fd519087acde0f769f144c30e8b3b6c21cf2310248d24c52015/VERSION
b04eff89da618fd519087acde0f769f144c30e8b3b6c21cf2310248d24c52015/json
b04eff89da618fd519087acde0f769f144c30e8b3b6c21cf2310248d24c52015/layer.tar
eefb4b5e0bcf55f75dfdc77f8c0c5e4cfaf98e8ff48d350c9ac75768cd019631/
eefb4b5e0bcf55f75dfdc77f8c0c5e4cfaf98e8ff48d350c9ac75768cd019631/VERSION
eefb4b5e0bcf55f75dfdc77f8c0c5e4cfaf98e8ff48d350c9ac75768cd019631/json
eefb4b5e0bcf55f75dfdc77f8c0c5e4cfaf98e8ff48d350c9ac75768cd019631/layer.tar
manifest.json
repositories
```

Local archive as base image

- Although images can be customized, using existing images might not always be desirable
- Base image change over time: updates might slightly change behavior and break things
- We might want **full control** over the contents of the image
- We might want a **minimal image** containing **only** what the app. needs

Solution: instead of using a traditional base image from a repository, use a local archive as root filesystem

Scratch image

- A local root filesystem archive can be used as base image
- Require the use of a special empty image, named **scratch image**
- How to obtain a local root filesystem?
  - use the root filesystem from an existing container
  - use Debian's Debootstrap tool
  - manually create a root filesystem (huge task to do from scratch[1])

[1] Famous project for doing exactly this: https://www.linuxfromscratch.org/

Creating an image from a local root filesystem: example

1. Create container `alp` from alpine:3.21 image:

```
docker run --name alp alpine:3.21
```

2. Export `alp` container's filesystem into a `tar` archive:

```
docker export alp > alpine_3.21.tar
```

3. Write a Dockerfile that uses the archive as the root filesystem:

```
FROM scratch
ADD alpine_3.21.tar /
CMD ["sh"]
```

4. Build the image from the Dockerfile

Multi-stage builds

- What if you want to build an image containing a specific program that must be **generated from source**?
  - it requires the compiler, all dependencies, etc.
- The resulting image would be very large since the **full build environment is required**!

Multi-stage builds

- What if you want to build an image containing a specific program that must be **generated from source**?
  - it requires the compiler, all dependencies, etc.
- The resulting image would be very large since the **full build environment is required**!
- How to make the resulting image as small as possible?

Multi-stage builds

- What if you want to build an image containing a specific program that must be **generated from source**?
  - it requires the compiler, all dependencies, etc.
- The resulting image would be very large since the **full build environment is required**!
- How to make the resulting image as small as possible?
  - unfortunately, no easy, clean and generic way of doing so...
- Solution?
  - multi-stage builds!

Multi-stage builds: why?

1. When images require building binaries or artifacts
2. To reduce the size of the final image
   - create a cleaner **separation** between the building of the image and the final output

Multi-stage builds: how?

```
# Builder stage which builds ninvaders" from source
FROM ubuntu:24.04 AS builder
RUN apt-get update && apt-get install -y build-essential libncurses-dev
WORKDIR /src
RUN wget https://downloads.sourceforge.net/project/ninvaders/ninvaders/0.1.1/
ninvaders-0.1.1.tar.gz
RUN tar fxz ninvaders-0.1.1.tar.gz
WORKDIR ninvaders-0.1.1
RUN make
# make install installs ninvaders into /usr/local/bin
RUN make install

# Final stage which copies the "ninvaders" file from the builder stage
FROM alpine:latest
RUN apk update && apk add ncurses
COPY --from=builder /usr/local/bin/ninvaders /usr/bin
CMD ["/usr/bin/ninvaders"]
```

Passing build-time values to the builder

1. Use the `--build-arg` when building the image, e.g.:

```
docker buildx build . -t myimage --build-arg base_dir=pipo
```

2. In Dockerfile, declare the same variable with `ARG`, e.g.:

```
ARG base_dir
...
COPY ${base_dir} .
```

- Avoid using build arguments for passing secrets such as user credentials, etc. as they are visible in the image history!
  - use `RUN --mount=type=secret` instead

Best practices (1/2)

- Use multi-stage builds (when possible)
- Choose the right and smallest base images to avoid unecessary bloat
- Avoid installing unnecessary packages/files
- Minimize the image size by removing unnecessary files
  - e.g. with Ubuntu/Debian-like distributions:

```
apt autoremove -y && apt-get clean && rm -rf /var/lib/apt/lists /var/cache/apt
```

- Use `.dockerignore` to avoid sending all context files to the daemon

Best practices (2/2)

- Pin base image versions to avoid breaking changes
  - e.g. `FROM alpine:x.y` instead of `FROM alpine`
- Each container should only solve one problem
  - decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers
- Create Dockerfiles that define stateless images
  - any state should be kept outside of the container (typically in volumes)

Security considerations

- Do not blindly trust others' images
- Often rebuild your images
  - keep image up-to-date with updated dependencies
  - to avoid cache hits, consider building with `--no-cache` option

Vulnerability Analysis of 2500 Docker Hub Images Wist K., Helsem M., Gligoroski D. (2021)

Abstract — *The use of container technology has skyrocketed during the last few years, with Docker as the leading container platform. Docker's online repository for publicly available container images, called Docker Hub, hosts over 3.5 million images at the time of writing, making it the world's largest community of container images. We perform an extensive vulnerability analysis of 2500 Docker images.*

...

Our main findings reveal that (1) the number of newly introduced vulnerabilities on Docker Hub is rapidly increasing; (2) certified images are the most vulnerable; (3) official images are the least vulnerable; (4) there is no correlation between the number of vulnerabilities and image features (i.e., number of pulls, number of stars, and days since the last update); (5) the most severe vulnerabilities originate from two of the most popular scripting languages, JavaScript and Python; and (6) Python 2.x packages and jackson-databind packages contain the highest number of severe vulnerabilities. We perceive our study as the most extensive vulnerability analysis published in the open literature in the last couple of years.

Resources

- How the build cache works (official)
  https://docs.docker.com/build/cache/
- Dockerfile reference (official)
  https://docs.docker.com/reference/dockerfile/
- Best practices for writing Dockerfiles (official)
  https://docs.docker.com/build/building/best-practices/
- Create base images (official)
  https://docs.docker.com/build/building/base-images/
- Create multi-stage builds (official)
  https://docs.docker.com/build/building/multi-stage/
- Explaining Docker Image IDs
  https://windsock.io/explaining-docker-image-ids/