Dockerfiles

Florent Gluck - Florent.Gluck@hesge.ch

April 13, 2022

Dockerfile

- Docker can build images automatically by reading instructions from a Dockerfile
- **Dockerfile = "source code" or recipe on how to build an image**
- Does not install an image $\rightarrow$ instructions on how to build an image

Why a Dockerfile?

- When existing images don't satisfy our needs
- When we can't find the exact image we need
- Most images are generic and won't cover our exact scenario
- To customize an existing image to our needs

Image creation and usage workflow

1. **Write** a **Dockerfile** with instructions on how to build the image

Image creation and usage workflow

1. **Write** a **Dockerfile** with instructions on how to build the image
2. **Build** the **image** from the Dockerfile

Image creation and usage workflow

1. **Write** a **Dockerfile** with instructions on how to build the image
2. **Build** the **image** from the Dockerfile
3. **Create** the **container** from the newly built image

Dockerfile contents workflow

1. Define base image (`FROM`)

Dockerfile contents workflow

1. Define base image (`FROM`)
2. Set environment variables (`ENV`)

Dockerfile contents workflow

1. Define base image (`FROM`)
2. Set environment variables (`ENV`)
3. Add instructions, e.g. packages to install, etc. (`RUN`)

Dockerfile contents workflow

1. Define base image (`FROM`)
2. Set environment variables (`ENV`)
3. Add instructions, e.g. packages to install, etc. (`RUN`)
4. Add files (`COPY` or `ADD`)

Dockerfile contents workflow

1. Define base image (`FROM`)
2. Set environment variables (`ENV`)
3. Add instructions, e.g. packages to install, etc. (`RUN`)
4. Add files (`COPY` or `ADD`)
5. Define default command (`CMD` or `ENTRYPOINT`)

Example of Dockerfile

```
FROM alpine:3.12
ENV EDITOR=vi
RUN apk update
RUN apk add git
COPY hosts /etc/
WORKDIR /home
CMD git
```

Main commands

- `FROM`: base image to use
- `ENV`: define environment variables (in Dockerfile and container exec.)
- `RUN`: execute commands
- `COPY` / `ADD`: add files to the image
- `CMD` / `ENTRYPOINT`: command to execute when instantiating the image
- `WORKDIR`: working directory (in Dockerfile and container exec.), `/` by default
- `EXPOSE`: indicates ports the container listens to for connections

COPY vs ADD

- `COPY` / `ADD` both copy files from a specific location into a Docker image
- `COPY` can only copy a local file or directory from the host into the image
- `ADD` similar to `COPY` but more powerful:
  - can extract a `tar` archive into the image
  - can specify an URL instead of a local file or directory

CMD

- `CMD` specify which command to execute when the image is instantiated Preferred form:

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

  Shell form, which executes `command` with `/bin/sh -c`:

```
CMD command param1 param2
```

- `CMD` **can be overriden** when starting the container:

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

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

ENTRYPOINT

- `ENTRYPOINT` also specify which command to execute when the image is instantiated
- By opposition to `CMD`, `ENTRYPOINT` **cannot be overriden** when starting the container!
- What is the purpose of `ENTRYPOINT` since `CMD` already exists?
  - `CMD` contents is added to `ENTRYPOINT` as argument
  - `ENTRYPOINT` + `CMD` = a way of specifying **default**, but **overridable** arguments

```
ENTRYPOINT ["git"]
CMD ["--help"]
```

Examples (1/2)

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

- `docker run myecho` → (nothing printed)
- `docker run myecho blah` → `blah`
- `docker run myecho blah blah` → `blah blah`

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

- `docker run myecho` → (nothing printed)
- `docker run myecho blah` → `error: executable "blah" not found!`
- `docker run myecho /bin/echo blah` → `blah`

Examples (2/2)

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

- `docker run myecho` → `blah`
- `docker run myecho pipo molo` → `pipo molo`
- `docker run myecho blah` → `blah`

```
FROM alpine
```

- `docker run myecho /bin/echo` → (nothing printed)
- `docker run myecho /bin/echo blah` → `blah`

Building an image

- The `docker build` command builds an image from a Dockerfile and a build context
- **The build is run by the Docker daemon, not the client!**
- Example:

```
docker build . -t myimage:beta
```

Build context

- Build context = set of files at a specified PATH (local directory) or URL (git repository)
- A context is processed recursively:
  - PATH includes any subdirectories
  - URL includes a repository and its submodules
- Example: specify `/tmp/pipo/` as the build context:

```
docker build /tmp/pipo
```

- IMPORTANT: during build, the client **sends** the **entire context recursively** to the Docker daemon!
- Use `.dockerignore` to specify which files must not be sent to the daemon (similarly to `.gitignore`)

Building an image from a local root filesystem

- Using the `FROM` directive to specify a base image might not always be desirable
- The image will likely **change** over time
  - a specific tag doesn't prevent the image from being updated
  - bugs might be introduced
  - behavior might slightly change, enough to break things
- Alternative: instead of using a base image from a repository, use an immutable local archive as root filesystem

Exporting an image or container's filesystem

- `docker export` → exports a **container**'s filesystem into a tar archive (to `stdout`)
  - archive contains the filesystem only
- `docker save` → saves an **image**'s filesystem into a tar archive (to `stdout`)
  - archive contains the various layers' contents (filesystem) and image 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
etc/alternatives/java
etc/alternatives/javac
...
```

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
```

Creating an image from a local root filesystem

- A local root filesystem archive can be used as base image
- Requires the use of a special empty image named **scratch**
- Steps to create an image from a local root filesystem:

1) Export a container's filesystem into a `tar` archive

```
docker export alpine:3.12 > alpine_3.12.tar
```

2) Create a Dockerfile that uses the archive as the root filesystem

```
FROM scratch
ADD alpine_3.12.tar /
CMD sh
```

3) Build the image from the Dockerfile

Multi-stage builds

- What if you want to build a container with a specific program that must be generated from source?
- Resulting image would be very large $\rightarrow$ requires the full build environment!
- 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?

Purpose of multi-stage builds:

- When images require building binaries or artifacts
- Help keep image size minimal

Multi-stage builds: how?

```
# Builder stage
FROM golang AS builder
ADD . /app
WORKDIR /app
RUN go build

# Final stage which uses the builder stage
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/app .
CMD ["./app"]
```

Best practices

- One container should only solve one problem!
- Minimize number of layers (= minimize number of steps)
- Use `.dockerignore` to avoid sending all context files/dirs to the Docker daemon
- Create Dockerfiles that define stateless images
  - Any state should be kept outside of the container
- Order layers from less frequently changed to more frequently changed (ensure build cache is reusable)

Resources

- Dockerfile reference (official)
  https://docs.docker.com/engine/reference/builder/
- Dockerfile tutorial by example - basics and best practices
  https://takacsmark.com/dockerfile-tutorial-by-example-dockerfile-best-practices-2018/
- Best practices for writing Dockerfiles (official)
  https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- Create a base image and multi-stage builds (official)
  https://docs.docker.com/develop/develop-images/baseimages/
  https://docs.docker.com/develop/develop-images/multistage-build/
- Explaining Docker Image IDs
  https://windsock.io/explaining-docker-image-ids/