Deployment & Virtualization

Joseph Chazalon, Clément Demoulins {`firstname.lastname@lrde.epita.fr`}

October'19

EPITA Research & Development Laboratory (LRDE)

Dockerfile

What is it?

Dockerfile = recipe to build a container image

- Base image
- Metadata
- Build steps
- Run some commands
- Copy some files
- Etc.

FROM

The first instruction must be *FROM* (there is one exception). It define the parent image on which we will construct the new image.

```
FROM <image>[:<tag>] [AS <name>]
FROM <image>[@<digest>] [AS <name>]
```

Examples:

```
FROM python:slim
FROM debian
```

ARG

You can pass variables at build time using the *ARG* instruction and the *–build-arg* option.

If you define an *ARG* before a *FROM*, it will be available only for the *FROM* :

```
ARG version=stable
FROM debian:$version
```

MAINTAINER / LABEL

You can add metadata to an image with the *LABEL* instruction. A *LABEL* is a key-value pair.

Example :

```
LABEL version="1.0"
LABEL description="purpose of the image for example"
LABEL label1="value1" \
      label2="value2"
```

The *MAINTAINER* instruction set the Author field but is officially deprecated. The recommended way is to set a *LABEL* “maintainer”.

RUN

The *RUN* instruction is one of the 3 instructions that create new layers.

```
RUN <command>
RUN ["executable", "arg1", "arg2"]
```

As it creates a new layer each time, it is recommended to group multiple commands in one *RUN*, and sort the package names for installation commands (build cache optimization).

```
RUN <command> \
 && <command> \
 && <command>
```

Example :

```
RUN pip install -r requirements.txt
```

COPY / ADD

The 2 instructions *COPY* and *ADD* are very similar and create also new layers.

```
ADD [--chown=<user>:<group>] <src>… <dest>
COPY [--chown=<user>:<group>] <src>… <dest>
```

src path accept go file matching like shell expansion (*, ?) and must be in the build context. If src is a local tar archive, it will be automatically extracted. In the case of *ADD*, if src is an url, it will be fetched but be careful with the layer cache.

About the build cache (aka layer cache)

Docker builder keeps a cache of image layers which were generated during previous builds.

The image is indexed by the hash of the line which generated it (and the parent image).

If you change the line, then the image will not be reused.

But if you have the same sequence of lines in two Dockerfiles, then the cache be come into action.

If you do not want to use the cache at all, you can use the **`--no-cache=true`** option on the **`docker build`** command.

For more details see the official documentation.

USER

The *USER* instruction sets the user name (or UID). The following instruction will use that user and the default user in the final image will be changed.

```
USER <user>[:<group>]
USER <UID>[:<GID>]
```

The *USER* instruction doesn't create the user so you have to create it first :

```
RUN useradd -d /data -m -r web
USER web
```

WORKDIR

The *WORKDIR* instruction sets the working directory for the following instructions. The directory will be created if it doesn’t exist.

Example :

```
WORKDIR /data
# Create empty file in /data
RUN touch index.html
```

ENV

The *ENV* instruction sets the environment variable to the value .

```
ENV <key> <value>
ENV <key>=<value
ENV <key>=<value>, \
    <key>=<value
```

One common use case is to set locales variables :

```
ENV LANG=en_US.UTF-8   \
    LANGUAGE=en_US:en  \
    LC_ALL=en_US.UTF-8
```

EXPOSE

The *EXPOSE* instruction informs Docker that the image listens on the specified ports.

```
EXPOSE <port>[/<protocol>]
```

Examples :

```
# default is tcp
EXPOSE 80
EXPOSE 80/udp
```

It doesn't automatically export the exposed ports of a running containers. You can use the option "–publish-all" or "-P" to do that but the host port will be random. A more commonly used option is "–publish" or "-p" which **requires that you specify host and container ports.**

VOLUME

The *VOLUME* instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

```
VOLUME ["PATH1", "PATH2", …]
VOLUME PATH1 PATH2 …
```

Example :

```
FROM ubuntu
# files before the volume instruction will be copied on the volume
# when creating the container
RUN mkdir /database \
 && initialize_database.sh /database
VOLUME /database
# after, they will be ignored
COPY other_file.db /database/
```

ONBUILD

The *ONBUILD* instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the *FROM* instruction in the downstream Dockerfile.

Example from golang onbuild image :

```
FROM golang:1.6

RUN mkdir -p /go/src/app
WORKDIR /go/src/app

# this will ideally be built by the ONBUILD below ;)
CMD ["go-wrapper", "run"]

ONBUILD COPY . /go/src/app
ONBUILD RUN go-wrapper download
ONBUILD RUN go-wrapper install
```

CMD

CMD provides a default program to run when executing a container,
or parameters to a previously defined *ENTRYPOINT* if not executable.

There can only be one *CMD* instruction in a Dockerfile.
If you list more than one *CMD* then only the last *CMD* will take effect.

The CMD instruction has three forms:

1. **`CMD ["executable","param1","param2"]`** (*exec* form, this is the preferred form)
2. **`CMD ["param1","param2"]`** (as default parameters to *ENTRYPOINT*)
3. **`CMD command param1 param2`** (*shell* form)

In doubt, use the first case and no *ENTRYPOINT*.

ENTRYPOINT

An *ENTRYPOINT* allows you to configure a container that will run as an executable.

There can only be one *ENTRYPOINT* instruction in a Dockerfile.
If you list more than one *ENTRYPOINT* then only the last *ENTRYPOINT* will take effect.

Actual cases were using *ENTRYPOINT* makes sense:

- Use a custom *init* program for the container, forcing everything to be run by this program which will have container's PID 1 and handle all the signals.
- Use a weird custom script to handle signals, but, really, avoid it.

Interactions between CMD and ENTRYPOINT

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT [“exec_entry”, “p1_entry”]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

For more details see the official documentation

Build process

The single command line

There is only one command:

```
docker image build \
    --tag user/imagename:tag \
    [-f path/to/dockerfile] \
    BUILD_CONTEXT
```

usually looks like

```
docker image build -t myimage .
```

because:

- the current directory is the build context we want to send to the builder,
- and there is a file named **`Dockerfile`** in this directory.

Build context

What is it and why the hell a Dockerfile is not sufficient?

Build context

What is it and why the hell a Dockerfile is not sufficient?

The build is run by the Docker daemon, not by the CLI (client)! They can be on separate machines.

The build context can be a path (like `.`), an URL or even the standard input (`-`).

The first thing a build process does is send the entire context (recursively) to the daemon. (Think of it as a distant build.)

In most cases, it's best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.

`.dockerignore` files

Regardless of where the Dockerfile actually lives, all recursive contents of files and directories of the context directory are sent to the Docker daemon as the build context.

This may slow the build process, cause extra files to be added to the image, etc.

You can filter the files from the build context to transmit to the builder using a **`.dockerignore`**.

This file supports exclusion patterns similar to **`.gitignore`** files.

A closer look at `build` command options

Image/layer management

```
--build-arg list          Set build-time variables
--cache-from strings      Images to consider as cache sources
--compress                Compress the build context using gzip
--disable-content-trust   Skip image verification (default true)
-f, --file string        Name of the Dockerfile (Default is 'PATH/Dockerfile')
--force-rm                Always remove intermediate containers
--label list              Set metadata for an image
--no-cache                Do not use cache when building the image
--pull                    Always attempt to pull a newer version of the image
--rm                      Remove intermediate containers after a successful
                          build (default true)
-t, --tag list            Name and optionally a tag in the 'name:tag' format
--target string           Set the target build stage to build.
```

Build container management

```
--add-host list           Add a custom host-to-IP mapping (host:ip)
--cgroup-parent string    Optional parent cgroup for the container
--cpu-period int          Limit the CPU CFS (Completely Fair Scheduler) period
--cpu-quota int           Limit the CPU CFS (Completely Fair Scheduler) quota
-c, --cpu-shares int      CPU shares (relative weight)
--cpuset-cpus string      CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems string      MEMs in which to allow execution (0-3, 0,1)
--iidfile string          Write the image ID to the file
--isolation string        Container isolation technology
-m, --memory bytes        Memory limit
--memory-swap bytes       Swap limit equal to memory plus swap: '-1' to
                          enable unlimited swap
--network string          Set the networking mode for the RUN instructions
                          during build (default "default")
--security-opt strings    Security options
--shm-size bytes          Size of /dev/shm
--ulimit ulimit           Ulimit options (default [])
```

How images are built

1. The client sends the build context to the builder
2. The engine checks the syntax of the Dockerfile
3. It creates a new container (customisable isolation!) based on the image you chose
4. For each of your commands / changes in the Dockerfile:
    - If the cache is active (default), it checks for a cached image to use
    - It **applies** the changes, writing content to the container thin storage layer
    - It **commits** the changes, adding another layer to the resulting image
    - It sends progress to the client
5. It cleans up the context and return the final image id to the client

Remember:

- Each **RUN**, **ADD**, **COPY** instruction creates another layer, hence those ugly one-line commands.
- The others just update the container configuration which will be used at run-time.
- Docker leaves the unfinished image of failed build lying around.

How to debug a failed build?

Have the unfinished image is actually useful: we can perform an autopsy on it.

`docker image history` can help locate the failing line

You can start a container from the latest working layer to investigate: 1. Find the image id using **`docker image`** or **`docker container`** 2. Run a shell in a container based on this image (last working layer)

You can also check the content of the unfinished layer

- by showing changes:

  ```
  docker container diff CONTAINER
  ```

- or by inspecting the container, find the storage path and inspect it from the host.

Best practices

Stateless

Separation between process and data allow to scale horizontally easily.
Your complex web process can be put behind a load balancer and a cluster of docker container.

It allow also help in the process of releasing, testing and upgrading.
The exact same code can be tested on a copy of your production database

Minimal

In terms of size

because pulling a 1GB image is a waste of electricity

In terms of layers

because it tends to make the filesystem slower, and there are limits anyway

In terms of complexity

*because **you** may have to maintain it*

In terms of attack surface

because “fragiledatabase” does not need “bazookadebugger” to be installed with it

Some tips

Group changes
Group related commands in **RUN** instructions, or even use separate script to avoid multiplying layers

Smallest possible image
If you add files from a distribution bootstrap, or use static binaries, you may use the **`scratch`** image as base. It is a special image with no layer.

Separate build and runtime images

No pain, no gain: by using two images you will ensure that the runtime image contains the bare minimum. Lighter, smaller attack surface.

You can even use the multi-stage build (see the practice session).

Tagging images

Use semantic versioning.

You can use multiple tags.

```
$ docker build -t me/myapp:1.0.2 -t me/myapp:latest .
```

Use private images / registries

You can pull images from private / custom registries.

They are pretty simple to setup: the registry application can be run in a Docker container!

Usage:

1. (opt.) Use **`docker login`** to login to a registry
2. Pull images using **`docker image pull registry/user/image:tag`** or simply **`docker run`**
3. Build new images
4. Push them using **`docker image push registry/user/image:tag`**

Some examples

Some examples

Live discussion