Container files

Docker can build images automatically by reading the instructions from a `Dockerfile.` A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build,` users can create an automated build that executes several command-line instructions in succession.

- `Dockerfile` is the default name but can have any other name.
- Must exist in the root context of the project we are creating.

> It is always preferred and recommended to use a `Dockerfile` to create images rather than direct CLI for the `build` command. The Dockerfile uses `DSL Domain-Specific Language` to build the image. It is a more repeatable, transparent, and potent mechanism for creating images.

First Dockerfile

Let us build an image with a simple webserver.

Start by creating a folder for your project that will hold all our required files and be called `root context.`

Docker will upload this context and any files or directories to our container when executed.

```
$ mkdir static_web
$ cd static_web/
$ vim Dockerfile
```

Now let's prepare our Dockerfile by adding the following lines.

```
FROM ubuntu:16.04
RUN apt-get -yqq update
RUN apt-get install -yqq nginx
RUN echo "Hi, I am your Docker container!!!"> /var/www/html/index.html
EXPOSE 80
```

- Instructions in the file are processed TOP-DOWN, so be careful with overwriting an instruction.
- Each new line of instruction adds a new layer to the image and commits the image until all the instructions are completed.
- This means that if any instruction in the process fails, you will be left with an image that can be used and will help debug but will not be complete.

Let us now run our `build` to create the image.

```
$ docker build -t jmedinar/static_web .
```

The `-t` is to set a TAG to make it easier for us to reference this image later.

The trailing point `.` tells Docker to look in the current directory, but you can also specify a file. `docker build -f mydockerfile` or you can use a Git repository too `docker build github.com/username/static_web` .

You will get an output something like this (truncated for easy reading):

```
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
61e03ba1d414: Pull complete
4afb39f216bd: Pull complete
e489abdc9f90: Pull complete
999fff7bcc24: Pull complete
Digest:
sha256:6aab78d1825b4c15c159fecc62b8eef4fdf0c693a15aace3a605ad44e5e2df0c
Status: Downloaded newer image for ubuntu:16.04
 ---> 065cf14a189c

Step 2/5 : RUN apt-get -yqq update
 ---> Running in 40ca74eb79b3
Removing intermediate container 40ca74eb79b3
 ---> 2f7f283e50be

Step 3/5 : RUN apt-get install -yqq nginx
 ---> Running in 33e3397eeb0f
Selecting previously unselected package libxau6:amd64.
(Reading database ... 4785 files and directories currently installed.)
Preparing to unpack .../libxau6_1%3a1.0.8-1_amd64.deb ...
...
Removing intermediate container 33e3397eeb0f
 ---> 968cc9e4ee0f

Step 4/5 : RUN echo "Hi, I am your Docker container!!!" >
/var/www/html/index.html
 ---> Running in 20ad4c0c3b89
Removing intermediate container 20ad4c0c3b89
 ---> a12640aee4b7

Step 5/5 : EXPOSE 80
 ---> Running in de6a45abf235
Removing intermediate container de6a45abf235
 ---> 0636bd6423b9

Successfully built 0636bd6423b9
```

Notice:

1. Our `Dockerfile` had 5 lines so our build has 5 steps to complete and will end up with 5 Layers!.

```
$  docker history jmedinar/static_web
IMAGE          CREATED          CREATED BY
               SIZE      COMMENT
0636bd6423b9   14 minutes ago   /bin/sh -c #(nop)  EXPOSE 80
               0B
a12640aee4b7   14 minutes ago   /bin/sh -c echo "Hi, I am your
Docker contai…   34B
968cc9e4ee0f   14 minutes ago   /bin/sh -c apt-get install -yqq
nginx           56.6MB
2f7f283e50be   15 minutes ago   /bin/sh -c apt-get -yqq update
               30.9MB
065cf14a189c   2 weeks ago      /bin/sh -c #(nop)  CMD
["/bin/bash"]            0B
<missing>      2 weeks ago      /bin/sh -c mkdir -p /run/systemd
&& echo 'do…   7B
<missing>      2 weeks ago      /bin/sh -c rm -rf
/var/lib/apt/lists/*          0B
<missing>      2 weeks ago      /bin/sh -c set -xe   && echo
'#!/bin/sh' > /…   745B
<missing>      2 weeks ago      /bin/sh -c #(nop) ADD
file:4dd75f16753c9c921…   135MB
```

Notice how the `history` command also shows how the base layer was created and the space each action is taking.

2. Every action will create an intermediate layer to run the task and delete it, so only the layer with the final result remains.

Our image is ready!. Docker has committed after completing every step in the process.

```
$ docker images jmedinar/static_web
REPOSITORY            TAG       IMAGE ID       CREATED          SIZE
jmedinar/static_web   latest    0636bd6423b9   22 minutes ago   222MB
```

The build cache

All the layers created at build are known as `The build cache.`

If in our example [jmedinar/static_web], we do not change steps 1 to 3, but we add different steps 4 and 5.

```
FROM ubuntu:16.04
RUN apt-get -yqq update
RUN apt-get install -yqq nginx
RUN echo "This is a different container!!!"> /var/www/html/index.html
EXPOSE 8080
```

We build it

```
$ docker build -t jmedinar/static_web2 -f Dockerfile2 .

Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM ubuntu:16.04
 ---> 065cf14a189c

Step 2/5 : RUN apt-get -yqq update
 ---> Using cache
```

```
 ---> 2f7f283e50be

Step 3/5 : RUN apt-get install -yqq nginx
 ---> Using cache
 ---> 968cc9e4ee0f

Step 4/5 : RUN echo "This is a different container!!!" >
/var/www/html/index.html
 ---> Running in 63472653cc4f
Removing intermediate container 63472653cc4f
 ---> 865b026a1adc

Step 5/5 : EXPOSE 8080
 ---> Running in 5c2d55c394d4
Removing intermediate container 5c2d55c394d4
 ---> ebd61d3fe495

Successfully built ebd61d3fe495
Successfully tagged jmedinar/static_web2:latest
```

Notice how steps 1 to 3 are not doing anything, only `USING` the already existing layers!.

This is clever and efficient, but sometimes you want to ignore the cache and rebuild from scratch, like upgrading a system.

This can be done with the `--no-cache` option of the `docker build` command.

```
$ docker build --no-cache -t jmedinar/static_web2 -f Dockerfile2 .
```

.dockerignore

This is non-required unless we want to exclude some files.

To prevent large or unnecessary files from being considered.

It is placed in the context root directory and uses pattern matching.

.dockerignore syntax

Syntax	Description
#	Comment
*	One or more character wildcard
?	One character wildcard
[]	Range of characters
!	Exception (not)

example:

```
# My .dockerignorefile
*.md
!README*.md
README-secret.md
```

File format

The `Dockerfile` supports the following formats:

1. Comments starting with `#` symbol

```
# This is a comment
```

2. Instructions in the form: `INSTRUCTION arguments.`

```
RUN echo 'I Love Docker!'
```

The instruction is not case-sensitive. However, the convention is to be UPPERCASE to distinguish it from arguments more easily.

3. Line continuation with the `\` symbol

```
RUN echo \
     I \
     Love \
     Docker!
```

4. Multiple arguments

```
RUN echo 'I Love Docker!' \
     && echo 'I also love Linux!' \
     && echo 'and I love Collin College!'
```

5. Escape special characters with [] backslash or the [`] back-tick symbol.

The backslash is default, but since it is also used in Windows paths, we might have to scape it:

```
FROM microsoft/nanoserver
COPY testfile.txt c:\\
RUN dir c:\
```

When you try to run the above results in:

```
PS C:\project> docker build -t cmd .
Sending build context to Docker daemon 3.00 kB
Step 1/2: FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/2: COPY testfile.txt c:\\RUN dir c
GetFileAttributeEx c:RUN: The system cannot find the file
specified.
```

Notice how in line 5, when it should do the copy of the txt file, it fails to recognize the path `c:\\` because one backslash cancels the second…

So we have to change the scape symbol as follows.

```
# escape=`
FROM microsoft/nanoserver
COPY testfile.txt c:\
RUN dir c:\
```

Instructions

FROM

The base image from which this new image will be created

The `FROM` instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid `Dockerfile` must start with a `FROM` instruction.

```
FROM <image name>:tag/version [AS <name>]
FROM python:3.7
```

`FROM` can also be set as a name later referenced in a `multi-staged` build.

```
FROM golang:1.16 AS builder
```

`FROM` instructions support variables that are declared by any `ARG` instructions that occur before the first `FROM` .

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
ARG VERSION=3.7
FROM python:${VERSION}
```

RUN

`RUN` will execute a command and can be used in two forms:

- ```
  RUN <command>
  ```
- ```
  RUN [command, parameter1, parameter2, ...]
  ```

Example:

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

or

```
RUN ["/bin/bash", "-c", "source $HOME/.bashrc", ";", "echo $HOME"]
```

CMD

Similar to sending a command when executing `run.`

The primary purpose of a `CMD` is to provide defaults for an executing container.

These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

The instruction `CMD` can be called in 3 different ways:

```
# Exec form in Json array
CMD ["command", "parameter 1", "parameter 2"]
# As simple parameters for an ENTRYPOINT in Json Array
CMD ["parameter 1", "parameter 2"]
# Shell form
CMD command param1 param2
```

IMPORTANT:

- There can only be one `CMD` instruction in a `Dockerfile.`
- If you list more than one `CMD,` only the last `CMD` will take effect.
- If `CMD` is used to provide arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.
- The `docker run` command overwrites `CMD` in the `Dockerfile.`
- Do not confuse

  ```
  RUN
  ```

  with

  ```
  CMD
  ```

  .
  - `RUN` actually runs a command and commits the result
  - `CMD` does not execute anything at build time but specifies the intended command for the image.

ENTRYPOINT

An `ENTRYPOINT` allows you to set a container that will run as a program. So you only use it to set what should be executed once the container is started!.

You can do it in two forms:

```
# exec form
ENTRYPOINT ["executable", "param1", "param2"]
# shell form
ENTRYPOINT command param1 param2
```

IMPORTANT:

- Command line arguments to `docker run` command will be appended after all elements in an *exec* form `ENTRYPOINT,` and will override all elements specified using `CMD.` This allows arguments to be passed to the entry point, `docker run <image> -d argument` .
- You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.
- The *shell* form prevents any `CMD` or `run` command-line arguments from being used but has the disadvantage that your `ENTRYPOINT` will be started as a subcommand of `/bin/sh -c` . Which does not pass signals. This means that the executable will not be the container's

`PID 1` - and will *not* receive Unix signals - so your executable will not receive a `SIGTERM` from `docker stop <container>`.
- Only the last `ENTRYPOINT` instruction in the `Dockerfile` will have an effect.

exec form

```
FROM debian
ENTRYPOINT ["uname"]
CMD ["-a"]
```

This is the same as running a container as follows.

```
$ docker run -ti --rm --name get_uname_info debian uname -a
Linux ab0373fc27de 3.10.0-1160.31.1.el7.x86_64 #1 SMP Wed May 26 20:18:08 UTC 2021 x86_64 GNU/Linux
```

It is essential to know that ENTRYPOINTS in exec form do not use a `shell,` so it will not work if you use something like an environment variable.

```
ENTRYPOINT ["echo", "$PATH"]
```

You will have to force the shell to run with the `sh -c` command.

```
ENTRYPOINT ["sh", "-c", "echo $PATH"]
```

shell form

Alternatively, use the shell form instead that will by default add the `/bin/sh -c` to the command but ignore any `CMD` or `docker run` commands.

```
ENTRYPOINT echo $PATH
```

In the shell form, it is also essential to add the `exec` command to ensure that if we issue a `docker stop` command, it will correctly signal the process.

```
ENTRYPOINT exec top -b
```

LABEL

Add metadata to an image as `key:value` pairs; they are documentation and extra information for our image.

```
LABEL version="1.0"
LABEL location="New York" type="Data Center" role="Web Server"
LABEL description="Our labels \
can span multiple lines."
```

You can inspect the labels using the `docker inspect`

```
docker inspect jmedinar/apache
```

EXPOSE

Specifies (but does not open) the network port that our container will open, either TCP or UDP.

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

You have to use the `-p` flag on `docker run` to open the port to publish.

```
EXPOSE 80/tcp
EXPOSE 80/udp
 $ docker run -p 80:80/tcp -p 80:80/udp ...
```

ENV

Sets environment variables in our container.

```
ENV PROGRAM="/bin/myprogram"
ENV USERNAME="runuser"
ENV VERSION="1.0"
```

Since the environment variable will exist in the container at execution, it can also be used as a parameter to RUN something.

```
RUN $PROGRAM -u $USERNAME
```

will be executed as:

```
/bin/myprogram -u runuser
```

If an environment variable is only needed during build and not in the final image, consider setting a value for a single command instead:

```
RUN DEBIAN_FRONTEND=noninteractive apt-get update && apt-get install -y
...
```

Alternatively, using `ARG` , which is not persisted in the final image:

```
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y ...
```

You can also use them in other instructions.

```
ENV TARGET_DIR="/opt/app"
WORKDIR $TARGET_DIR
```

The Environment variables will persist in all the containers created with your image.

You can also pass Environment Variables at `run` with `-e` flag.

```
docker run -ti -e "WEB_PORT=8080" ubuntu env
```

WORKDIR

Sets a working directory for the container, same as executing a `cd - change directory command.

```
WORKDIR /opt/webapp/db
RUN bundle install
WORKDIR /opt/webapp
ENTRYPOINT [ "rackup" ]
```

You can also overwrite the `WORKDIR` at `run` with `-w`

```
docker run -ti -w /var/log ubuntu pwd /var/log
```

USER

Specify the user under which the image or process should be executed

```
USER nginx
```

Multiple combinations can be used:

```
USER username
USER username:group
USER uid
USER uid:gid
USER username:gid
USER uid:group
```

This can also be overwritten at `run` with -u flag, and if not specified, the default user is `root.`

```
WORKDIR /opt/webapp/db
USER bundleuser
RUN bundle install
WORKDIR /opt/webapp
USER 0:0
ENTRYPOINT [ "rackup" ]
```

ADD

Adds files and directories from our build environment into the image.

```
ADD software.lic /opt/application/software.lic
```

This will copy the `software.lic` file into the `/opt/application/software.lic` in the image.

The source can be:

- URL
- Filename
- Dicrectories

> You cannot `ADD` files outside the build context.

If the destination ends with "/" it will be considered a directory else a file.

```
1 ADD http://wordpress.org/latest.zip /root/wordpress.zip
```

`ADD` has some unique management for file packages like `tar, tar.gz, zip, gzip, bzip2, xz` and will unpack it for you.

```
1 ADD latest.tar.gz /var/www/wordpress
```

If the folder exists, it will NOT be overwritten.

If the destination does not exist, it will be created with mode 0755 and 0:0 uid.

`ADD` invalidates the cache. If there is a file already in the cache and in ADD the add will be placed!.

If the file PATH contains white spaces, the following format should be used:

```
1 ADD ["software.lic", "/opt/application licenses/software.lic"]
```

You can also use regular expressions.

```
1 ADD *.txt /my-text-files/
```

You can also modify the ownership of files while adding.

```
1 ADD --chown=user1:group1 files* /myfiles/
```

COPY

Copies a file into the image. Unlike `ADD,` it can copy from multiple locations and does not have extraction capabilities.

```
1 COPY conf.d/ /etc/apache2/
```

The source should be in the build context; nothing can be copied. The reason is that only this directory is loaded in the daemon; everything else is not visible. The destination should be an absolute path inside the container.

VOLUME

Creates mount points in the image in which volumes will be mounted.

A volume is a special directory that bypasses the union filesystem and can be used in multiple containers providing multiple valuable features:

- Volumes can be shared and reused among containers
- A container does not have to be running to share its content
- Changes to a volume are made directly
- Changes to a volume are not included when changes are made to an image
- Volume persist even if no containers are using them

This can be used in databases.

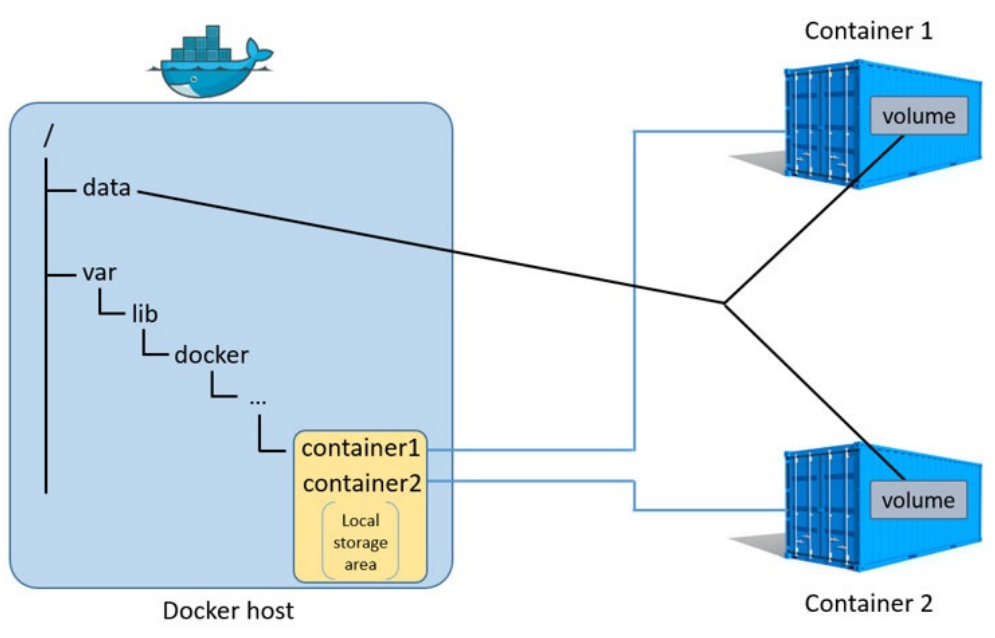

Fig. 1 - Overview of a mounted shared-volume inside two containers

We can specify one or multiple volumes.

```
VOLUME ["/opt/project", "/data"]
# or
VOLUME /opt/project
VOLUME /data
```

Keep the following things in mind about volumes in the `Dockerfile.`

- Volumes on Windows-based containers must be one of:
    - a non-existing or empty directory
    - a drive other than `C:`
- If any build steps change the data within the volume after it has been declared, those changes will be discarded.
- When using the JSON format, you must enclose words with double quotes (`"`).
- The mount point is host-dependent. This is to preserve image portability since a given host directory cannot be guaranteed to be available on all hosts. For this reason, you cannot mount a host directory from within the Dockerfile. You must specify the mount point when you create or run the container.

ARG

Defines variables that users can be pass at build time. You can only specify arguments that have been defined in the Dockerfile and can set one or more in a single Dockerfile.

```
# Only defining the variable
ARG build
ARG username
# Or also setting a default value
ARG webapp_user=user
ARG webapp_port=80
```

It can also be done with the `-build-arg` flag during the build.

```
docker build --build-arg build=1.0 -t jmedinar/webapp .
```

Important Do not pass important information using this method as passwords because they will be exposed during the build and will remain in the history of the image.

Scope: `ARG` variables are valid only after they are declared.

```
FROM busybox
USER ${user:-some_user}
ARG user
USER $user
```

- Line2: will set as user `some_user` because the `ARG` does not exist at that moment in code.
- Line4: will use the `ARG` user variable.

An `ARG` instruction goes out of Scope at the end of the build stage where it was defined. Each stage must include its own `ARG` instruction to use an' ARG' in multiple stages.

```
FROM busybox
ARG SETTINGS
RUN ./run/setup $SETTINGS

FROM busybox
ARG SETTINGS
RUN ./run/other $SETTINGS
```

HEALTHCHECK

Tells Docker how to test a container to check that it is still working correctly. Allows you to check things like the webserver is up and running, an API responds with the correct data.

When a container has a health check specified, it also has a status and its normal status.

```
HEALTHCHECK --interval=10s --timeout=1m --retries=5 CMD curl
http://localhost || exit 1
```

- `--interval` Defaults to 30 seconds. The time between checks.
- `--timeout` Defaults to 30 seconds. If a check takes longer, it will be considered failed.
- `--retries` Defaults to 3. A number of failed checks before the container is marked as unhealthy.

CMD can either be a shell command or an exec array. It should return a `0` if healthy or anything else if unhealthy.

We can see the result of the check with `docker inspect`

```
docker --inspect --format '{{.State.Health.Status}}' static_web
healthy
```

The history of checks can also be checked.

```
docker --inspect --format '{{range .State.Health.Status}} {{ExitCode}}
{{.Output}} {{end}}' static_web
0 Hi, I am in your container
```

There can only be one `HEALTCHECK` instruction in a `Dockerfile.` If there is more than one, only the last one will be considered.

You can also disable any health checks.

```
HEALTHCHECK NONE
```

Multi-Stage Builds

While building images, you should always keep them as small as possible!. Each line/instruction in a dockerfile adds a new layer to the final image, and sometimes we have artifacts created that are only required in that step in the process and nowhere else.

We can use shell tricks to make our images smaller… like running multiple steps on a single line:

```
FROM python:3.7
RUN set -ex \
    && curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg
| apt-key add - \
    && apt-get update \
    && apt-get upgrade -y \
    && apt-get -y install apt-transport-https ca-certificates curl
gnupg2 \
    && apt-get -y install software-properties-common \
    && apt-get purge -y mysql-common \
    && rm -rf /var/cache/apt/*
COPY file.txt /
COPY script.py /
```

The above will generate four layers:

FROM RUN COPY COPY

Even when the second layer is running multiple commands, the problem with this, besides the fact that it will be hard to maintain, is that if one of the steps fails, the whole build will fail, and we will have to start over.

```
REPOSITORY                                          TAG           IMAGE
ID       CREATED          SIZE
non-multi-staged                                    latest
3cba09ac9520   38 seconds ago   952MB
```

With Multi-stage builds, managing multiple states of a Dockerfile or multiple Dockerfiles is a lot simpler:

- You can use multiple `FROM` statements, and each can use a different base image.
- You can copy artifacts from one state to the next.
- You can remove anything you do not need from the final image.

```
FROM python:3.7 as updated-image
```

```
RUN set -ex \
    && curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - \
    && apt-get update \
    && apt-get upgrade -y

FROM updated-image as purged-image
RUN set -ex \
    && apt-get -y install apt-transport-https ca-certificates curl gnupg2 \
    && apt-get -y install software-properties-common \
    && apt-get purge -y mysql-common \
    && rm -rf /var/cache/apt/*

FROM purged-image
COPY file.txt /
COPY script.py /
```

In this simple example, we are not generating any artifact. We are not removing anything from the final image; we just run the same code in multiple stages and reduce the `Dockerfile` complexity.

```
REPOSITORY                                          TAG           IMAGE ID       CREATED          SIZE
multi-staged-build                                  latest
c2ae28085c9a   29 seconds ago   955MB
non-multi-staged                                    latest
3cba09ac9520   4 minutes ago    952MB
```

So we end up with a slightly larger image because of the extra information.

Notice we could also name our stages with the instruction `as <name>.` This is very useful to avoid confusion.

We can also `STOP` at any stage for debugging purposes. Let us pretend we want to stop once the `purged-image` is created we can build as follows:

```
$ Docker build --target purged-image -t debugging-purged-image .
```

You can also copy artifacts and stages from another image, not only the ones you are creating in the Dockerfile:

```
COPY --from=myotherimage:latest /etc/nginx/nginx.conf /nginx.conf
```

Of course, you can build multiple stages from one.

```
FROM python:3.7 as updated-image
RUN set -ex \
    && curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - \
    && apt-get update \
    && apt-get upgrade -y

FROM updated-image as purged-image
RUN set -ex \
    && apt-get -y install apt-transport-https ca-certificates curl gnupg2 \
    && apt-get -y install software-properties-common \
```

```
    && apt-get purge -y mysql-common \
    && rm -rf /var/cache/apt/*

FROM purged-image as image-with-file
COPY file.txt /

FROM purged-image as image-with-script
COPY script.py /
```