Docker fundamentals

Basic docker commands (IMAGES)

docker images -> list existing images
docker search ubuntu -> find images in public registry
docker image pull ubuntu:latest -> download image from registry to local

docker info -> get info on the engine
sudo ls /var/lib/docker (C:\ProgramData\DockerDesktop in windows) -> browse docker engine files
docker ps -> list instances

Dockerfile syntax

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

```
FROM ubuntu                 <-  reuse an already existing image
RUN echo “hello ynov”       <-  run an instruction
```

```
docker build .              <-  build an image from a Dockerfile
```

Create a file run.py with the following content:

```
print("Hello Docker - PY")
```

```
FROM python                 ←  reuse an already existing image
ADD . .                     ←  add src code
CMD python run.py           ←  execute runtime
```

```
docker build . -t test-python       <-  build an image from a Dockerfile and tag it
docker run test-python              <-  execute an image
docker inspect test-python
```

DOCKERFILE INSTRUCTIONS - RUN

RUN has 2 forms:

- RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- RUN ["executable", "param1", "param2"] (exec form)

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results.

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

← Shell form

```
RUN ["/bin/bash", "-c", "echo hello"]
```

← Exec form

DOCKERFILE INSTRUCTIONS - COPY or ADD

The COPY instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>

```
COPY hom* /mydir/
```

COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support)

ADD does that same but in addition, it also supports 2 other sources.

- A URL instead of a local file/directory.
- Extract tar from the source directory into the destination.

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

DOCKERFILE INSTRUCTIONS - ARGS or ENV

The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well. The environment variables set using ENV will persist when a container is run from the resulting image.

```
ENV MY_NAME="John Doe"
```

The ARG instruction defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.

```
FROM ubuntu
ARG CONT_IMG_VER
RUN echo $CONT_IMG_VER
```

```
docker build --build-arg CONT_IMG_VER=v2.0.1 .
```

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

DOCKERFILE INSTRUCTIONS

The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use as the default user and group for the remainder of the current stage. The specified user is used for RUN instructions and at runtime, runs the relevant ENTRYPOINT and CMD commands.

The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.

The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.

DOCKERFILE INSTRUCTIONS - ENTRYPOINT / CMD

An ENTRYPOINT allows you to configure a container that will run as an executable. This instructuon is placed at the end of dockerfile. You can override the ENTRYPOINT instruction using the docker run --entrypoint flag.

```
ENTRYPOINT ["executable", "param1", "param2"]
```

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

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 main 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.

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

Basic docker commands (others)

docker run -it -v $(pwd)/datadir:/usr/local/datadir ubuntu:latest -> run a container with datadir available
docker run -it -u 1000 ubuntu:latest -> run a container as user 1000
docker run -n myubuntu -d ubuntu:latest sleep 5000 -> run a container demonized
docker exec -it myubuntu -> exec in a container
docker stop myubuntu -> stop a container
docker rm myubuntu -> rm a container

Dockerfile best practices

- Separation of concern: Ensure each Dockerfile is, as much as possible, focused on one goal. This will make it so much easier to reuse in multiple applications.
- Avoid unnecessary installations: This will reduce complexity and make the image and container compact enough.
- Reuse already built images: There are several built and versioned images on Docker Hub; thus, instead of implementing an already existing image, it's highly advisable to reuse by importing.
- Have a limited number of layers: A minimal number of layers will allow one to have a compact or smaller build. Memory is a key factor to consider when building images and containers, because this also affects the consumers of the image, or the clients.

Dockerfile tuning

1. always use versioning on images:

es. use FROM python:slim-bullseye instead of python

2. reduce size of image

docker image ls python:latest → 921MB

docker image ls python:slim-bullseye → 126MB

docker image ls python:3.10-alpine → 48.7MB

Dockerfile tuning

3. run executables as unprivileged users:

define a USER in the Dockerfile with limited privileges

4. use 2 stages build to separate between build stage and execution stage: this results in lightweight containers