Productivity Tools for IC Design (EE2510). Oct-Nov 2025. Exercise Set 4.

Section 1

1. Install docker and basic session.

Installation (on Ubuntu):

```
$ sudo apt install -y docker
```

Get a shell within the container:

```
$ docker pull ubuntu:latest     # takes some time and displays some information
$ docker run -it ubuntu:latest /bin/bash
```

- see what is there on your system:

```
$ docker ps -a
```

If not clear read help;

```
$ docker help ps
```

Get information about images installed on your system:

```
$ docker image ls
```

```
REPOSITORY                          TAG          IMAGE ID       CREATED        SIZE
openroad/flow-ubuntu22.04-builder   13b2c4       63417459cb0c   4 weeks ago    4.58GB
openroad/flow-ubuntu22.04-dev       13b2c4       4b04b77b6b35   4 weeks ago    3.26GB
openroad/orfs                       latest       e643ab316a70   4 weeks ago    4.6GB
```

The image ID can be used to select an image for any other operation, e.g. see the output of

```
$ docker image history 63417459cb0c   #subsitute the image IDs you have
```

Explore the commands available for images and containers using

```
$ docker help image
```

```
$ docker help image history
```

2. Using a Dockerfile to make your own image.

Save the following into a file named Dockerfile in any folder (say ~/work/tmp here)

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y python3
COPY hello.py /usr/src/app/hello.py
WORKDIR /usr/src/app
CMD ["python3", "hello.py"]
```

Try to understand what the following commands do:

```
$ cd ~/work/tmp
```

```
$ docker build .       # or docker build -t mycont -f Dockerfile
$ docker image ls
$ docker run <image id>
```

What happens if you replace “CMD [“python3”, “hello.py”] “ with “CMD [“/bin/bash]”. Why do you not get a container shell at the prompt in this case?

Suppose you want to share the data between the host and container:

```
$ docker run -it -v ~/work/tmp:/data <image-id>  /bin/bash
```

This will map the host’s ~/work/tmp to /data in the container. The data written to that folder from either host or container is available to the other.

For one off copy in a running container you can also use:

```
$ docker cp constraints.sdc <container_id>:/OpenROAD/constraints/
```

To run a container you can also do:

```
$ docker exec -it <container_id> bash
```

What is the difference between run and exec?

You can share (map) network port from one to another:

```
$ docker run -d -p 8080:80 <image-id>
```

Which will pass data on hosts’ port 8080 to container’s port 80.

ENV and ARG variables can be used to make a parameterised Dockerfile. For example

```
FROM ubuntu:latest
ARG setwdir=/app
ARG appname=app.py
RUN apt-get update && apt-get install -y python3
COPY $appname /usr/src/app/$appname
ENV workdir=$setwdir
WORKDIR $workdir
CMD ["python3", "$appname"]
```

Then the following command can be used to get same effect as the previous build:

```
$ docker build --build-arg setwdir=/usr/src/app --build-arg appname=hello.py .
```

They are set and used similar to shell variables. ENV variables can be accessed by the programs running in containers, just like normal environmental variables. ARG variables only exist during the building of the image file.

3. Layers and modifications and saving of the images.

Each instruction in a Dockerfile creates a layer when a docker image is built. Each layer represents a filesystem change, is immutable, is cached and reused when possible (say in building other images).

So, if you use the following Dockerfile after the previous one:

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y python3
COPY hello.py /usr/src/app/hello.py
WORKDIR /usr/src/app
RUN echo “hi” > change.txt
CMD ["python3", "hello.py"]
```

The first four layers are reused from previous build. Confirm this using

```
$ docker history <imageid>
```

Can you explain why using (1) instead of (2) may be better?

```
(1) RUN apt-get update && apt-get install -y python3 && apt-get clean
```

```
(2) RUN apt-get update
RUN apt-get install -y python3
RUN apt-get clean
```

Also explain why it is better to place the less frequently changing commands first in the dockerfile.

Till now we have been modifying existing Ubuntu images by adding further layers to it. Another option is to modify the containers interactively and then save (“commit”ting) them. For example you can have the following session:

```
$ docker run -it ubuntu /bin/bash
Container > apt-get update && apt-get install -y curl
Container > echo "hi" > /changes.txt
Container > exit
$ docker commit <container_id> ubuntu:curl-added
$ docker run -it ubuntu:curl-added /bin/bash
Container > cat /changes.txt
Container > exit
```

This way of modification doesn’t record the modification, like the way you do in a Dockerfile. This new image captures all the data of the running container, and can be converted to a packed file (.tar files in Unix).

```
$ docker save -o ubuntu-curl-added.tar ubuntu:curl-added
```

This file can be copied to another computer and run there. This is how the cloud providers can move (“migrate”) running instances of servers to different computers.

A final point related to the layers is called multi-stage builds, please read about them here.