Docker review

- Review these docker commands:
- **run:**
  - Creates and runs a container from an image
  - It creates a writable layer above the image's layers. The layer persists when the container stops and remains intact when it is started later.
  - One can specify environment variables for the container
    - `docker run --env FOO=BAR Ubuntu:latest`
  - The container created from an image can have custom names
  - The container can be removed upon exit
    - `docker run --rm -it ubuntu:latest`
  - `The -t` dedicates a pseudo-tty connected to the container's stdin and `-i` binds the host terminal to that tty. Without `–i` the `bash` shell is inaccessible and without –t bash cannot properly interpret the inputs.
  - A container can connect to a specific network
    - `docker run --network somenetwork ubuntu:latest`
  - An important aspect of run is data storage through bind mounts or volumes
  - Bind mounts are a type of storage where a file or directory on the host machine is mounted into a container. A disadvantage of bind mount is its dependency on the host machine's file system.
    - `docker run --volume "$(pwd)"/target:/app image`
    - `docker run --mount type=bind,source="$(pwd)"/target,target=/app image`
  - Volumes are a data persistence layer managed by docker itself. Volumes can be defined by both `--volume` and `--mount` options.
    - `docker run --volume myvolname:/app image`
    - `docker run --mount source=myvolname,target=/app image`
  - `Command: docker run --name test --rm --env FOO=BAR –it Ubuntu:latest`
- **Create**:
  - Creates a container from a source image
  - It accepts all the options mentioned in `run` command
- **start:**
  - starts a container
  - A container can be started in an interactive manner
  - `docker start –i containername`
- **Attach**:
  - It binds the current terminal emulator to the running container such that we have access to stdin, stdout, and stderr
  - `docker attach containername`

- **Commit**:

    - As stated, `run` command will create a writable layer on top of the image layers. Commit will persist changes made on a running container in the form of a new image
    - Commit won't affect data on mounted volumes
    - `docker commit containername imagename`
- **Cp**:
    - Copies files from a source path to a destination path
    - Source and destination could be either on the host machine or container's file system
    - Container paths are relative to the root directory i.e. `/` (forward slash).
    - Providing forward slash is optional
    - Host machine paths could be absolute or relative to the current directory where command is run
    - Copies are performed recursively
    - The source and destination could be file or directory names. Respective rules for each scenario are provided in Docker's official documentation.
    - `docker cp ./filename.txt containername:/tmp/data`
- **Exec**:
    - Runs a command in a running container
    - It runs on the default directory of the container unless a different `--workdir` is provided
    - Chained command does not work
    - Example:
        - `docker run --name mycontainer -d -i -t alpine /bin/sh`
        - `docker exec -d mycontainer touch /tmp/execWorks`
        - `docker exec -it mycontainer sh`
- **Images**:
    - `Docker images`
    - Shows all images excluding intermediate images
    - To show all images
        - `docker images --all`
    - One can filter results based on certain criteria
        - `docker images –filter "label=lbl"`
- **Inspect:**
    - Returns information about a docker object
- **Kill**:
    - Kills a running container
        - `docker kill containername`
    - Instead of sending SIGTERM, as `docker stop` does, it sends SIGKILL signal
- **Login**:
    - Logins to a registry
    - **`docker login`**
- **Logs:**
    - Prints logs the output of a container

        - `Docker logs containername`
    - Logs could be audited in real-time
        - `Docker logs --follow containername`
    - Other options to filter our logs:
        - `--tail (number of lines shown from the end of logs)`
        - `--since`
        - `--until`
        - `-t (show --timestamps)`
- **Ps**:
    - `docker ps`
    - Lists running containers
    - To list all containers
        - `docker ps --all`
    - `docker ps --all --last 10 --filter “name=somename”`
- **Pull:**
    - Pulls an image from a docker registry
    - One can set a proxy to connect to the registry server by setting `HTTP_PROXY` and `HTTPS_PROXY` environment variables
    - If no tag is provided docker will use default `:latest tag`
    - A specific version of an image can be pulled using its SHA256 digest
        - `docker pull ubuntu@sha256:26c68657ccce2cb0a31b3...`
    - It’s possible to pull an image from another registry other than DockerHub by providing the registry’s path (protocol `http://` excluded)
        - `docker pull myregistry.local:5000/testing/test-image`
- **push:**
    - pushes an image to a registry
    - `docker push myregistry.local:5000/testing/test-image`
- **rm:**
    - removes a container
    - One can remove associated volumes
        - `docker rm --volumes ubuntu:latest`
    - removes a container
- **rmi:**
    - removes an image
    - Images can be pointed at using their tags, ID’s, or digests.
    - Removing a tag won’t remove the image unless there are no other tags bound to the image
    - `docker rmi Ubuntu:latest`
- **stop:**
    - stops a container
    - `docker stop containername`
- **tag**:
    - tag a source image

        - `docker tag source_name target_name`
        - `docker tag ubuntu:latest os/linux/ubuntu:latest`
    - Image names in Docker are slash-separated and optionally prefixed by registry hostname. (protocol excluded)

- What are layers in docker image and what are their benefits?
    - An image is basically a set of changes made on top of another image (excluding `scratch`)
    - In essence, each layer in a docker **tagged image**, the one we can pull from a registry by its tag name, is itself a docker image.
    - Each layer represents the changes made to the image it's based on. Each tagged image comprises one or more layers.
    - Each instruction in a Dockerfile result in a new layer on top of the previous one. Some of these layers are removed later and some of them persist in the image, namely the ones created using RUN, COPY, and ADD.
    - Layers provide a sense of modularity, which results in network and computational load mitigation. Redundant layers present in different images could be downloaded once, cached in the host and later used to create other images, which reduces network load. Moreover, if the upper layers of an image get modified, e.g. through image's Dockerfile, underlying layers remain intact and are reused to create the new image, hence less computation load.
    - **Remark**: If you add something in one layer, and delete in the next, the add operation is still there in the first layer and can be looked up if you share the image.

- What are differences between docker run and docker start?
    - `docker run` adds a writable layer on top of the base image, creates a container from it and starts that container. `docker start` only start a container that is previously created.

- What is the base image that docker image redis version 6.0.17-bullseye created from?
    - First approach
        - `docker pull redis:6.0.17-bullseye`
        - `docker history redis:6.0.17-bullseye` (IMG_ID missing)
        - `dive redis:6.0.17-bullseye` (dive is an external tool for inspecting layers)
            - Shows content inside `debconf` directory, which belongs to debian
            - The first layer's digest is:
                - `sha256:4695cdfb426a05673a100e69d2fe9810d9ab2b3dd88ead97c6a3627246d83815`
        - I expected pull debian@digest to return the exact variant but it returned manifest unknown error
        - `Docker pull debian@sha256:4695cdfb426a05673a100e69d2fe9810d9ab2b3dd88ead97c6a3627246d83815`

    - Second approach
        - Visit the official image’s Github page and inspect its Dockerfile
        - Base image is `debian:bullseye-slim`

- Run redis 6.0 container and check what is the value of REDIS_VERSION and GOSU_VERSION env var inside the container
    - First approach
        - `Docker pull redis:6.0`
        - `Docker run -it -d --name rds redis:6.0`
        - `Docker exec -it rds sh`
        - Inside the container’s shell
            - `echo $REDIS_VERION`
                - 6.0.17
            - `echo $GOSU_VERSION`
                - 1.16
    - Second approach
        - `docker history redis:6.0`
            - `ENV REDIS_VERSION=6.0.17`
            - `ENV GOSU_VERSION=1.16`

- Create a container from mongo image and a container from mongo-express image in the same network and connect them together and open the mongo express dashboard. (use docker run command)
    - `Docker network create mongo-network`
    - `Docker run --name mongo --network mongo-network -d mongo`
    - `Docker run --name express --network mongo-network -e ME_CONFIG_MONGODB_SERVER=mongo -p 8081:8081 -d mongo-express`
- What is the benefit of the docker compose
    - Docker compose allows for defining a set of services working collectively in an easy way. Defining shared networks and running each docker image individually while passing respective configurations quickly becomes hard when the application components grow. Documenting these steps is also inconvenient. Docker compose lets you define required networks for your components, pass arguments as environment variables, and run all components with a single command. The whole setup is documented in a single YAML file, which facilitates later use.

- Create a container from mongo image and a container from mongo-express image in the same network and connect them together and open the mongo express dashboard. (use docker compose)

```
version: "3.9"
services:
  mongo:
    image: mongo:latest
```

```
express:
  image: mongo-express:latest
  depends_on:
    - mongo
  ports:
    - 8081:8081
  environment:
    ME_CONFIG_MONGODB_SERVER: mongo
```

- What is difference between ENTRYPOINT and CMD?
  - The ENTRYPOINT is used to set executing instruction for a container when it is initiated. It cannot be overwritten by docker run arguments. The CMD is chiefly used to provide defaults for an executing container. If both the ENTRYPOINT and CMD are present inside a Dockerfile, parameters of CMD will be appended to ENTRYPOINT (only in exec form). If user provides arguments when running a container, those arguments will replace CMD.
  - ENTRYPOINT and CMD can be used toghether to provide executable and its default arguments. Nevertheless, one can use each of them individually to excecute commands.
- Create a simple api server and create a dockerfile for that. create an account in docker hub and push it to your docker registry. put a bash script containing all your commands you used+dockerfile+apiserver in a github repository and add mojtabaimani as a contributor. create a PR and ask me to review it. create a db container and then create a database table for your api server and import a csv file containing a list of sample data to your database table that your api server can read them. include all your commands and sample data in your repo so I can also run it. Your data inside the container should be persistent. So If I stop and delete the container and create it again, data should be there again.

  - `https://github.com/rapour/dockerized_api`