Docker

24 June 2020 17:36

© RAJAT KUMAR
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers.

Why Docker?

The use of Linux containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is. Containerization is increasingly popular because containers are:
Flexible: Even the most complex applications can be containerized :-

Lightweight: Containers leverage and share the host kernel.
Interchangeable: You can deploy updates and upgrades on-the-fly.
Portable: You can build locally, deploy to the cloud, and run anywhere.
Scalable: You can increase and automatically distribute container replicas.
Stackable: You can stack services vertically and on-the-fly.

The process can be summarized as follows :-
1. Set up your Docker environment
2. Build an image and run it as one container
3. Scale your app to run multiple containers
4. Distribute your app across a cluster
5. Stack services by adding a back-end database
6. Deploy your app to production

Getting Started with containers :-

Run a Container : docker run hello-world
docker run hello-world - does exactly what it sounds like. It runs an image named “hello-world.”
docker ps -a - lists the containers on our system:
Let’s run this image again with docker run hello-world. The output is almost the same…
Run docker ps -a

We see two stopped instances of hello-world, with two different names. Docker created an additional container. It didn’t reuse the first. When we told Docker to run an image named hello-world, it did exactly that; it ran a new instance of the image. If we want to reuse a container, we refer to it by name.

Reuse a container :-

Let’s try starting one of the stopped containers:

docker start attach <container name>
The attach tells Docker to connect to the container output so we can see the results.
We stop containers with docker stop <container name> and remove them with docker rm <container name>.

With a single Docker command, docker run -it ubuntu bash, we downloaded an Ubuntu Linux image and started a login shell as root inside it. The -it flags allow us to interact with the shell.

Stop and remove a container:-

We can stop it with docker stop…

```
$ docker stop compassionate_ritchie
```
…and remove the container with docker rm.

```
$ docker rm compassionate_ritchie
```

Share an image :-

We can upload our own images to Docker Hub for distribution, too.

The first step is to create an account on Docker Cloud. If you don’t already have an account, go and create one.

Next, we’ll log in to the Docker registry:

```
$ docker login
Username: rajatkumar
Password:
Login Succeeded
```
We’ll upload ubuntu18 to Docker Hub.

Before we do that, we should tag it. The format for Docker tags is username/repository:tag. Tags and repository names are effectively freeform.

```
$ docker tag ubuntu18 rajatkumar/customimages
```
Note that our image tag and ubuntu18 have the same image ID and size. Tags don’t create new copies of images. They’re pointers.

Now we can push the image to Docker Hub:
docker push rajatkumar/customimages

Create copy of a container:-

```
docker commit <hash tag of running container> new_image
docker run -it new_image bash
Docker start <hash tag of new container>
Docker exec -it  <hash tag of new container> bash
```

Copy data to aws:-

Docker Notes (PPT)

Docker
Notes

Docker Containers

Create an interactive terminal container with a name, an image, and a default command:
- Usage: docker create -it --name=<name> <image> <command>
- Example: docker create -it --name=foo ubuntu bash

List all running containers:
- docker container ls
- (list all containers, running or not): docker container ls -a

Start a docker container:
- Usage: docker start <container name/id>
- Example: docker start foo

Attach to a docker container:
- Usage: docker attach <container name/id>
- Example: docker attach foo

Remove a container:
- Usage: docker rm <container name/id>
- Example: docker rm foo
- Force remove: docker rm foo -f

Run a new container:
- Usage: docker run <image> <command>
- Example with options: docker run --name=bar -it ubuntu bash

Remove all containers:
- docker container ls -aq | xargs docker container rm

Execute a command in a running container:
- Usage: docker exec <container name/id> <command>
- Example (interactive, with tty): docker exec -it express bash

Docker Images

Remove a docker image:
- Usage: docker image rmi <image id>
- Example (only uses first 3 characters of image id): docker rmi 70b

Remove all images:
- docker image ls -aq | xargs docker rmi -f

Search for a docker image on dockerhub:
- Usage: docker search <image>
- Example: docker search ubuntu

List docker images:
- docker image ls

Build a Docker image:
- Usage: docker build <path>
- Example (also tags and names the build): docker build . -t org/serve:0.0.0

Dockerfiles

Specify a base image:
- Usage: FROM <base image>
- Example: FROM node:latest

Set a working directory for the container:
- Usage: WORKDIR <dir>
- Example: WORKDIR /usr/src/app

Run a command for the container image:
- Usage: RUN command
- Command: RUN npm install -g serve

Copy files into the container:
- Usage: COPY <local files/directories> <container files/directories>
- Example: COPY ./display ./display

Inform that a port should be exposed
- Usage: EXPOSE <port>
- Example: EXPOSE 80

Specify a default command for the container:
- Usage (shell format): CMD <default command>
- Example: CMD serve ./display
- Usage (exec format, *recommended*): CMD [“default command”,

Docker start <hash tag of new container>
Docker exec -it <hash tag of new container> bash

Copy data to aws:-
scp -r -i C:\Users\rajatkumar\Downloads\ubuntu18_.pem ./bacnet-stack-0.8.6 ubuntu@
3.17.133.23:/home/ubuntu

Building and running a docker file :-
(should be in sudo su -, root directory)
nano Dockerfile (Write Code)
docker build -t filename ./
docker run --rm -it filename

We passed two arguments to build:

-t filename gave Docker a tag for the image. Since we only supplied a name, we can see that Docker tagged this build as the latest in the last line of the build output.
The final argument, dot (or "."), told Docker to look for the Dockerfile in the current working directory.

Few Docker commands :-

```
install apt docker (In docker machine)
docker image rm -f 1c35c4412082
docker run hello-world
docker run -it ubuntu bash
docker exec -it f2c4a7d7249e bash                    // Run a container with its ID

docker ps -a            # Lists containers (and tells you which images they are spun from)
docker images           # Lists images
docker rm <container_id>   # Removes a container

docker rmi <image_id>      # Removes an image
                    # Will fail if there is a running instance of that image i.e. container

docker rmi -f <image_id>   # Forces removal of image even if it is referenced in multiple repositories,
                    # i.e. same image id given multiple names/tags
                    # Will still fail if there is a docker container referencing image

docker container ls -a
docker image ls
docker container rm <container_id>
docker image rm <image_id>

docker rename pensive_mirzakhani ubuntu
```

Run a Container as host

```
docker run -it --network host --name container_name image_name
```

Linux Container vs Windows Container :

- Linux containers and Windows containers differ in terms of the operating system that they run on.
- Linux containers run on a host operating system that is based on the Linux kernel, whereas Windows containers run on a host operating system that is based on the Windows kernel.
- As a result, Windows containers can only run Windows-based applications, while Linux containers can run a wider range of applications, including those built for Linux and Windows.
- Additionally, the management tools and ecosystem surrounding Linux containers are typically more mature and robust compared to those for Windows containers.

Docker File Code :-

```
#!/bin/bash
FROM ubuntu
#You can start with any base Docker Image that works for you

# Update aptitude with new repo RUN apt update && apt-get install -y git
#RUN pwd
WORKDIR /root/bacnet-stack-0.8.6/
RUN pwd

#copy all files in /root/bacnet-stack-0.8.6/ folder to ./ COPY ./* ./
[illegible]
```

Specify a default command for the container:

- Usage (shell format): CMD <default command>
- Example: CMD serve ./display
- Usage (exec format, *recommended*): CMD ["default command", "arguments"]
- Example: CMD ["node", "server.js"]

Cross-Container Storage

Volumes

Create a volume

- Usage: docker volume create <volume name>
- Example: docker volume create shared-vol

Inspect a volume

- Usage: docker volume inspect <volume name>
- Example: docker volume inspect shared-vol

Mount a container with a volume using docker run

- Usage: --mount source=<volume name>, target=<container dir>
- Example:

  docker run -it --name=foo --mount source=shared-vol,target=/src/shared ubuntu bash

Bind Mounts

Mount a container with a bind mount using docker run

- Usage: --mount type=bind source=<host dir>, target=<container dir>
- Example:

  docker run -it --name=foo --mount type=bind source=/Users/foo/bindmountdir, \

  target=/src/mountdir ubuntu bash

Tmpfs mounts

Mount a container with a tmpfs mount using docker run

- Usage: --mount type=tmpfs, destination=<container dir>
- Example:
  docker run -it --name=baz --mount type=tmpfs, destination=/tmpdir ubuntu bash

Docker Networking

List docker networks

- docker network ls

Inspect a docker network

- Usage: docker network inspect <network name>
- Example: docker network inspect bridge

Create a docker network

- Usage: docker network create <network name>
- Example: docker network create privatenw

Run a container with a custom docker network:

- Usage: --network=<network name>
- Example: docker run --network=privatenw -it --name=goo busybox

Docker Compose

Start a compose application

- At the root (where docker-compose.yml is located): docker-compose up

Start a compose application and rebuild images:

- Docker-compose up --build

docker-compose.yml

Version

- Current version is 3. So at the top of the file, specify: version: '3'

Services with builds

- Have a services key in the file. List out services one indent at a time.

Dependencies

- Use the depends_on key and specify dependencies with a list. Each container dependency is marked by a dash, such as: -backend

Volumes

- Have a volume key per service.
- Connect a Docker host directory to a container directory, by joining them with a colon.
- Example: ./dockerhostdir:/containerdir

Networks

- Declare networks at the bottom of the file.
- Specify each service's network(s) with the networks option for each service.

Docker Swarm

Initialize a swarm in a node

- Usage: docker swarm init --advertise-addr=<node ip>
- Example: docker swarm init --advertise-addr=172.31.17.31

After initializing the swarm, you will find a join command for worker/other manager nodes

- Example: docker swarm join --token

```
WORKDIR /root/bacnet-stack-0.8.6/
RUN pwd

#copy all files in /root/bacnet-stack-0.8.6/ folder to ./ COPY ./* ./
#RUN make clean . RUN make . RUN cp -r /root/bacnet-stack-0.8.6/bin/ ./
COPY ./bin/ ./
EXPOSE 47808
#WORKDIR ./bin/
ENV PATH="/root/bacnet-stack-0.8.6/bin:${PATH}"
#RUN $PATH RUN pwd RUN ls RUN ["bacserv"]
CMD ["./bacwi" , "-1"]
CMD ["./bacrp", "1234", "1", "1", "85"]
CMD ["./bacwp", "1234", "1", "1", "85", "16", "-1", "4", "10"]
```

- Example: docker swarm init --advertise-addr=172.31.17.31

After initializing the swarm, you will find a join command for worker/other manager nodes

- Example: docker swarm join --token SWMTKN-1-592fo0c31guwi9cw58jpaw89fafzyw5fzbk9dwiw8bm4xxpad-94vn587o9o3r73h3e5esujxm9 172.31.17.31:2377

List docker nodes from a manger:

- docker node ls

Create a service for the swarm:

- Usage: docker service create --name=<service name> --publish=<host port:service port> <service image>
- Example: docker service create --name=site --publish=80:80 nginx

List services:

- docker service ls

List the running tasks for a service:

- Usage: docker service ps <service name>
- Example: docker service ps site

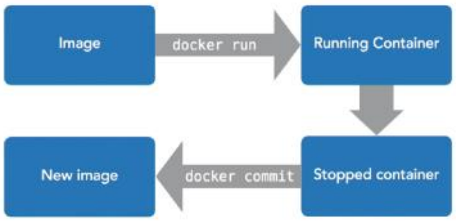
Update a service

- Usage: docker service update [options] <service name>
- Example: docker service update --replicas=6 site

Docker Commands with description :-

```
## List Docker CLI commands
docker
docker container --help


## Display Docker version and info
docker --version
docker version
docker info


## Execute Docker image
docker run hello-world


## List Docker images
docker image ls


## List Docker containers
(running, all, all in quiet mode)
docker container ls
docker container ls --all
docker container ls -aq


# Create image using this directory's Dockerfile
docker build -t friendlyhello .

# Run "friendlyname" mapping port 4000 to 80
docker run -p 4000:80 friendlyhello

# Same thing, but in detached mode
docker run -d -p 4000:80 friendlyhello

# List all running containers
docker container ls

# List all containers, even those not running
docker container ls -a

# Gracefully stop the specified container
docker container stop <hash>

# Force shutdown of the specified container
docker container kill <hash>

# Remove specified container from this machine
docker container rm <hash>

# Remove all containers
docker container rm $(docker container ls -a -q)

# List all images on this machine
docker image ls -a

# Remove specified image from this machine
docker image rm <image id>

# Remove all images from this machine
docker image rm $(docker image ls -a -q)

# Log in this CLI session using your Docker credentials
docker login

# Tag <image> for upload to registry
docker tag <image> username/repository:tag

# Upload tagged image to registry
docker push username/repository:tag
```

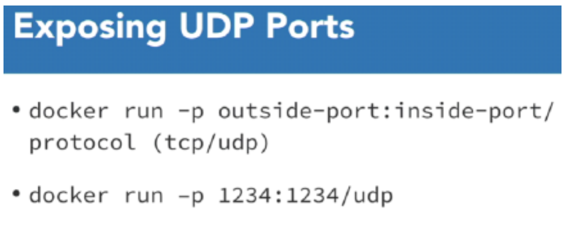
The Docker Flow :



Exposing Ports :

- docker run --rm -it -p 45678:45678 -p 45679:45679 --name echo-server ubuntu:14.04 bash
  (-p is for publish, we are publishing 45678 and connect it to port 45678, so we are just exposing the same port on the inside of the container as on the outside)
  (outside port : inside port)

  # if we don't provide outside port
- docker run --rm -it -p 45678 -p 45679--name echo-server ubuntu:14.04 bash
  # we can check using :
- docker port echo-server   (echo-server is container name)



Exposing UDP Ports

- docker run -p outside-port:inside-port/protocol (tcp/udp)
- docker run -p 1234:1234/udp

- docker run --rm -it -p 45678/udp --name echo-server ubuntu:14.04 bash

- docker network ls   (existing docker networks)
```
NETWORK ID    NAME     DRIVER   SCOPE
c7376020aba9  bridge   bridge   local
75919c8d5779  host     host     local
53f3bd0aeab7  none     null     local
```
  - Bridge is the network used by containers that don't specify a preference to be put into any other network.
  - Host is when you want a container to not have any network isolation at all. This does have some security concerns.
  - None is for when a container should have no networking.

- docker network create learning   (create new network)
- docker run --rm -it --net learning --name catserver ubuntu:14.04 bash   (using that created network using -net)
- Docker network create catsonly
- Docker network connect catsonly catserver

# Run image from a registry
docker run username/repository:tag

# List stacks or apps
docker stack ls

# Run the specified Compose file
docker stack deploy -c <composefile> <appname>

# List running services associated with an app
docker service ls

# List tasks associated with an app
docker service ps <service>

# Inspect task or container
docker inspect <task or container>

# List container IDs
docker container ls -q

# Tear down an application
docker stack rm <appname>

# Take down a single node swarm from the manager
docker swarm leave --force

Docker Commands :

- docker exec -it <container-id> /bin/bash (run a container using id/hash)
- Docker images (to list all docker images)
- Docker run -it <image-name> bash (running a container using an image)
- -it stands for interactive terminal
- docker run -it ubuntu:latest bash (running ubuntu container using image)
- docker run -it ubuntu bash (same as above, by default it take as latest)
- Docker ps (list running containers)
- docker container ls (same as above, list running containers)
- Docker ps -a (list all containers, stopped once also)
- docker container ls -a (same as above, list all containers, stopped once also)
- docker ps -l (print last container that we exit)
- docker container start <container-id/name> (starting a stopped container)
- docker container stop <container-id> (stopping a running container)
- docker container rm <container-id> (deleting a container)
- docker container kill <container-id> (force shutdown a container, it goes to stop stage)
- docker kill <container-id/name> (same as above)
- docker container rm $(docker container ls -a -q) (remove all containers)
- Docker commit <container-id> (make image using a container)
- Docker tag <image-id> <my-image-name> (giving a name to a image using it's id)
- docker tag windowswithnginx imrajat/windows-nginx:v2023.0 (creating new image with new name with tag)
- Docker commit <container-name> <my-image-name> (single command of commit + tag, default tag as latest)
- Docker commit <container-name> <my-image-name>:v1.01 (single command of commit + tag, with tag value as v1.01)
- Docker run --rm -it ubuntu bash (--rm is for delete container after it exit)
- Docker run -d -it ubuntu bash (-d means detach, the container will keep on running in background)
- Docker attach <container-name> (run container)
- Ctrl + p then ctrl + q (detach a container and keep it running while in a container)
- Docker image rm <image-name> (delete an image)
- Docker logs <container-name/id> (to check logs of a container)
- Docker run -it --name example ubuntu bash (--name to give name to a container)
- docker build -t hello . (build docker image with name as hello)
- docker run -it hello (to run the above)
- docker rmi <image-id> (deleting image)
- docker rmi <image-name:tag> (deleting image with tag)
- docker run -it -v D:\tempdocker:/shared-folder ubuntu bash (-v for volume to share data, the tempdocker folder data will be share in docker in shared-folder)
- docker run -it -v /shared-folder ubuntu bash (create a folder name shared-folder, it can be shared among other container)
- docker run -it --volumes-from <container-name> ubuntu bash (create a container with parameter --volumes-from having container name that we created with shared folder)
- Docker login (to login to docker)

  # To publish your image on docker
  - Docker tag ubuntu:latest imrajat/test-image-5:v1.1
  - Docker push imrajat/test-image-5:v1.1 (where imrajat is my docker username)

- docker rename <old-container-name> <new-container-name> (rename to change container name)
- docker volume create db (Create a volume by using the docker volume create command.)
- docker run -dp 3000:3000 --mount type=volume,src=db,target=/etc/todos getting-started (to save data in host, and use it in every container)
  - Mount example (using --mount) : type=volume,src=my-volume,target=/usr/local/data
- docker volume inspect db (to check where Docker is storing data when we use a volume)
- docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash

Resource Restriction :

Memory limits :
- Docker run --memory maximum-allowed-memory image-name command

CPU limits:
- Docker run --cpu-shares relative to other containers
- Docker run --cpu-quota to limit it in general

Legacy Linking :

- Links all ports, though only one way
- Secret environment variables are shared only one way
- Depends on startup order
- Restarts only sometimes break the links

- docker run --rm -e SECRET=theinternetlovescats -it --name catserver ubuntu:14.04 bash
  (-e is for setting environment variables)
- docker run --rm -it --link catserver --name dogserver ubuntu:14.04 bash
  (--link is to link the dogserver with catserver)

  nc -lp 1234 (run this in catserver)
  nc catserver 1234 (run this in dogserver)

  Not both are connecting, can pass message to each other.
  But if we run nc -lp on dogserver and nc dogserver on catserver it will not work.
  As we link dogserver with catserver not vice versa

- env (to check environment variables)

Docker Compose :

- Docker Compose is a tool that was developed to help define and share multi-container applications.
- With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

  # We create a Compose file by translating the commands we were using into the appropriate compose format.

- docker compose version (to check docker compose version)
- docker compose up -d (to run docker-compose.yml, -d flag to run everything in the background)
- docker compose down (the containers will stop, delete and the network will be removed)
  # By default, named volumes in your compose file are NOT removed when running docker compose down. If you want to remove the volumes, you will need to add the --volumes flag.

Dockerfile :

- This is a small "program" to create an image
- We run this program with
  - docker build -t <result-image-name> . (Note the . Here)
  - When it finishes, the result will be in your local docker registry

- Each line takes the image from the previous line and makes another image.
- Each line of a Docker file makes a new, independent image based on the previous line's image.
- The previous image is unchanged.
- It does not edit the state from the previous line.
- You don't want large files to span lines or your image will be huge

- Dockerfile look like shell scripts, but are not shell scripts

- MAINTAINER Firstname Lastname <email@example.com> (Defines the author of this Dockerfile)

  # Runs the command, waits for it to finish, and saves the result :
- RUN unzip install.zip /opt/install/
- RUN echo hello docker
- Use && to combine and run multiple commands

  # Adds locals files
- ADD project.tar.gz /install/ (This command notices that it is a compressed archive, and it uncompresses all the files to that directory.)
- ADD https://project.example.com/download/1.0/project.rpm/ /project/ (download project.rpm in /project directly)

  # Sets environment variables
- ENV DB_PORT=1234

  # The ENTRYPOINT and CMD statements
- ENTRYPOINT specifies the start of the command to run
- CMD specifies the whole command to run.
- If your container acts like a command-line program, you can use ENTRYPOINT

- Mount example (using --mount) : type=volume,src=my-volume,target=/usr/local/data
- docker volume inspect db   (to check where Docker is storing data when we use a volume)
- docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash
- A **bind mount** is another type of mount, which lets us share a directory from the host's filesystem into the container
  - docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash
    # The --mount option tells Docker to create a bind mount, where src is the current working directory on own host machine, and target is where that directory should appear inside the container (/src).
- docker logs -f <container-id>   (to check log)
- There are two ways to put a container on a network
  1. Assign it at start
  2. Connect an existing container
  - docker network create <network-name>
- docker compose version   (to check docker compose version)
- docker compose up -d   (to run docker-compose.yml, -d flag to run everything in the background)
- docker compose down   (the containers will stop, delete and the network will be removed)
  # By default, named volumes in your compose file are NOT removed when running docker compose down. If you want to remove the volumes, you will need to add the --volumes flag.
- Docker scan <image-name>   (Scan it for security vulnerabilities)
- docker build -t <result-image-name> .   (to build docker image)

# Saving and Loading Images :   (help in migrating between storage types)
- docker save -o my-images. tar.gz ldebian:sid busybox ubuntu: 14.04   (-o is for output name i.t. my-image, this command saves images in local)
- Docker load -i my-images.tar.gz   (to load images from local backup)

Different arguments used for Docker :

- -it
  - (-i stands for interactive, this allows us to interact with the container's terminal in the same way we interact with a terminal on our local machine)
  - (-t stands for "tty" and is used to allocate a pseudo-tty to the container. This provides a more interactive terminal experience, including proper terminal emulation and escape sequences)
  - docker run -it ubuntu:18.04 bash
- --restart-always   (if this container dies, restart it immediately)
- -d or --detach   (to run a Docker container in the background)
  - docker run -d <my-image>
- -p or --publish   (to publish the exposed ports to the host)
  - docker run -p 8080:80 <my-image>
- --name <my-container>   (to name the container)
  - docker run --name <my-container> -d ubuntu:18.04
-

Reference :
https://www.linkedin.com/learning/learning-docker-2018
https://docs.docker.com/get-started/

Problem:
Not able to access internet inside docker windows container

Solution :
Resolve it by setting the DNS.

If we are making the call during build time you can set it in your docker daemon. In Docker Desktop go to "Settings" > "Docker Engine" and add the following to the JSON config:

"dns": ["1.1.1.1"]

If we are experiencing the problem during run time we can add the dns argument

--dns 1.1.1.1

https://stackoverflow.com/questions/59766135/not-able-to-access-internet-inside-docker-windows-container

# The ENTRYPOINT and CMD statements
- ENTRYPOINT specifies the start of the command to run
- CMD specifies the whole command to run.
- If your container acts like a command-line program, you can use ENTRYPOINT
- If you are unsure, you can use CMD
- The 'CMD' statement specifies which command to run within the container.
- The 'ENTRYPOINT' statement is for making your containers look like normal programs.

# Shell Form vs Exec Form:
- ENTRYPOINT RUN and CMD can use either form
- Shell form looks like this:
  - nano notes.txt
- Exec form looks like this:
  - ["/bin/nano", "notes.txt"]

# The EXPOSE Statement:
- Maps a port into the container
  - EXPOSE 8080

# The VOLUME Statement:
- Defines shared or ephemeral volumes
  - VOLUME ["/host/path/" "/containe r/path/"]
  - VOLUME ["/shared -data"]
- Avoid defining shared folders in Dockerfiles because it has a dependency on host system

# The WORKDIR Statement:
- Sets the directory the container starts in   (kind of cd)
  - WORKDIR /install/

# The USER Statement:
- Sets which user the container will run as
  - USER rajat
  - USER 1000

```
FROM ubuntu:16.04 as builder
```
(# assign image name as builder, so we can use it later)
```
RUN apt-get update
RUN apt-get -y install curl
RUN curl https://google.com | wc -c > google-size

FROM alpine
COPY --from=builder /google-size /google-size
```
(# copy data from one image to another, it saves space because, ubuntu image is big as compared to alpine. Alpine is small base image)
```
ENTRYPOINT echo google is this big; cat google-size
```

Q) Where in your Dockerfile should you place the parts of your code that you have changed the most?
A) Include these at the end of your Dockerfile, so the parts before them do not need to be redone every time you change that part.

Name structure of Images :

- registry.example.com:port/organization/image-name: version-tag
- Organization/image-name is often enough

Volumes :
- Volume are sort of like shared folders.
- Virtual "discs" to store and share data, share them between the containers and between containers and host, or both.
- Two main varieties
  - **Persistent** Volume : We can put data there, and it will be available on the host, and when the containers goes away, the data will still be there.
  - **Ephemeral** Volume : These exist as long as the container is using them, but when no container is using them, they evaporate. They stick around as long as they are used, but they are not permanent.
- Not part of images.
- They are for our local data, local to this host.

Share Data with the host :
- "Shared folders" with the host
- Sharing a "single file" into a container
- Note that a file must exist before you start the container, or it will be assumed to be a directov

- docker run -it -v D:\tempdocker:/shared-folder ubuntu bash
  (-v for volume to share data, the tempdocker folder data will be share in docker in shared-folder)

  A **bind mount** is another type of mount, which lets us share a directory from the host's filesystem into the container
  Host location : We decide

  Mount example (using --mount) :
  type=bind,src=/path/to/data,target=/usr/local/data

- docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash

# The --mount option tells Docker to create a bind mount, where src is the current working directory on own host machine, and target is where that directory should appear inside the container (/src).

Sharing Data between Containers :

- volumes-from
- Shared "discs " that exist only as long as they are being used
- Can be shared between containers

- docker run -it -v /shared-folder ubuntu bash
  (create a folder name shared-folder, it can be shared among other container)
- docker run -it --volumes-from <container-name> ubuntu bash   (create a container with parameter --volumes-from having container name that we created with shared folder)

A **volume mount** is a great choice when you need somewhere persistent to store your application data.
Host location : Docker chooses

Mount example (using --mount) :
type=volume,src=my-volume,target=/usr/local/data

Docker Registries :

- Registries manage and distribute images
- Docker (the company) offers these for free
- We can run our own, as well

Finding Images :

- Docker search command
  - Example : Docker search ubuntu
- Search on https://hub.docker.com/