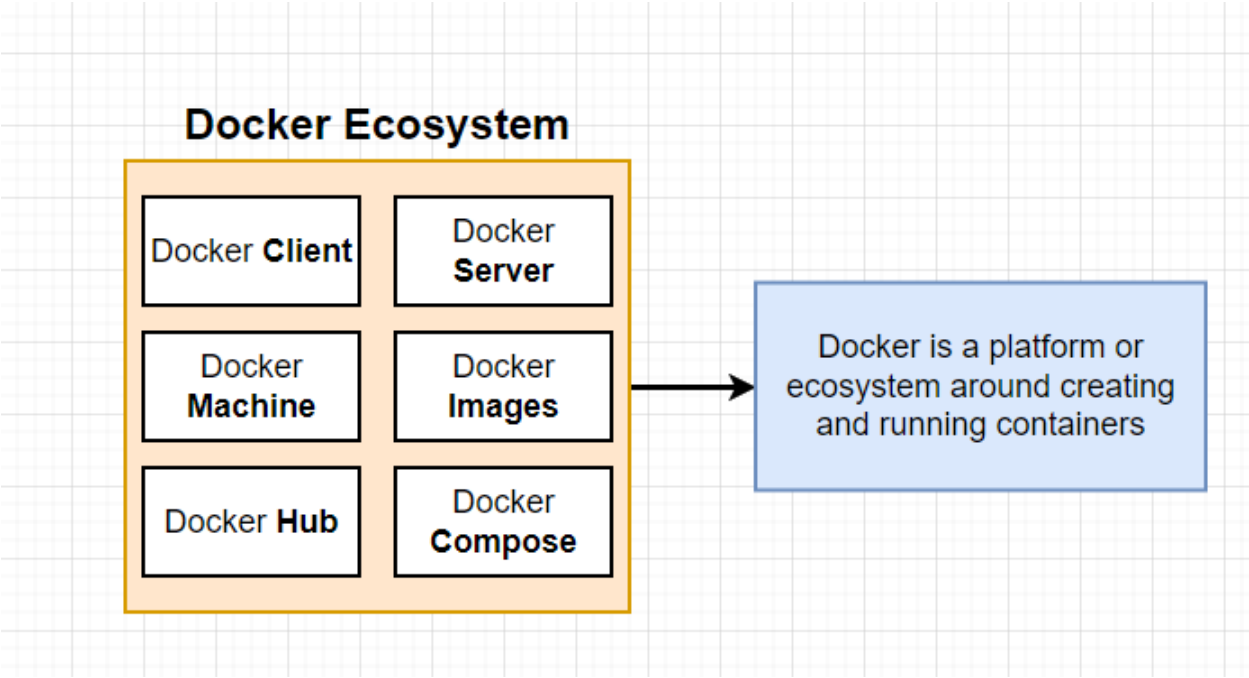
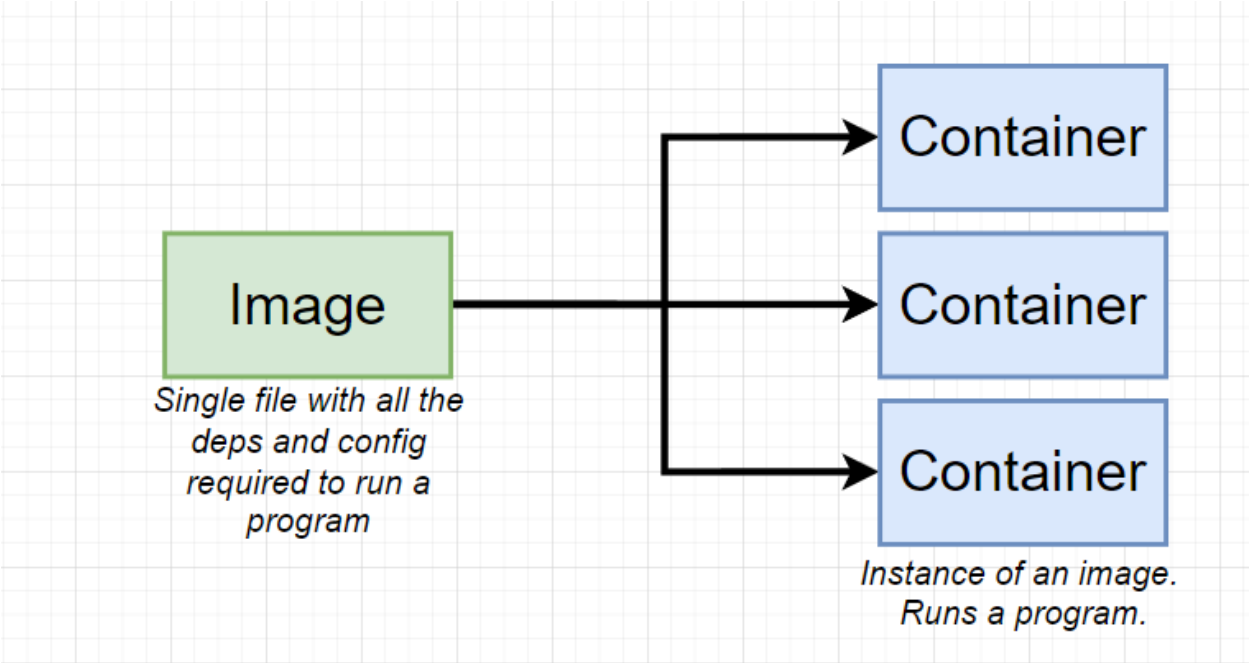
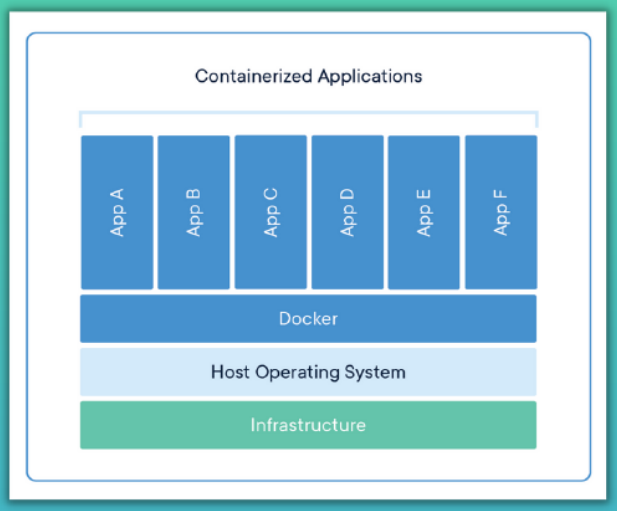
DockerFile Reference: https://docs.docker.com/engine/reference/builder/





Containers vs V

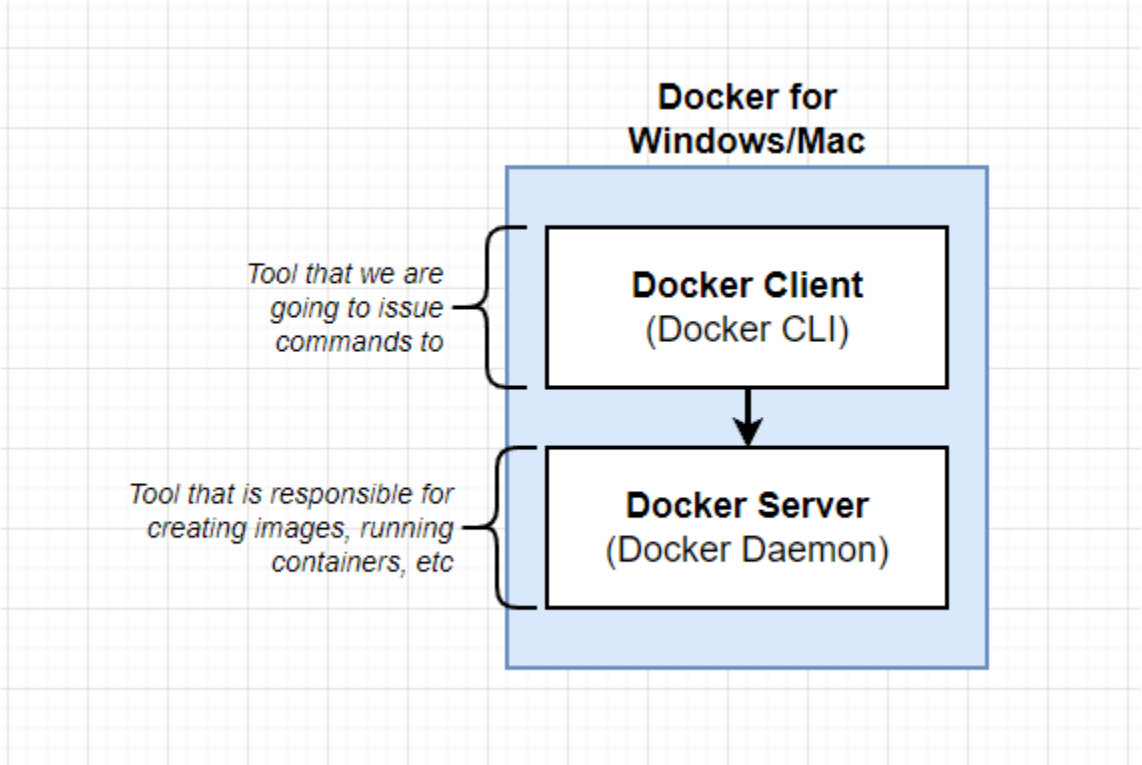


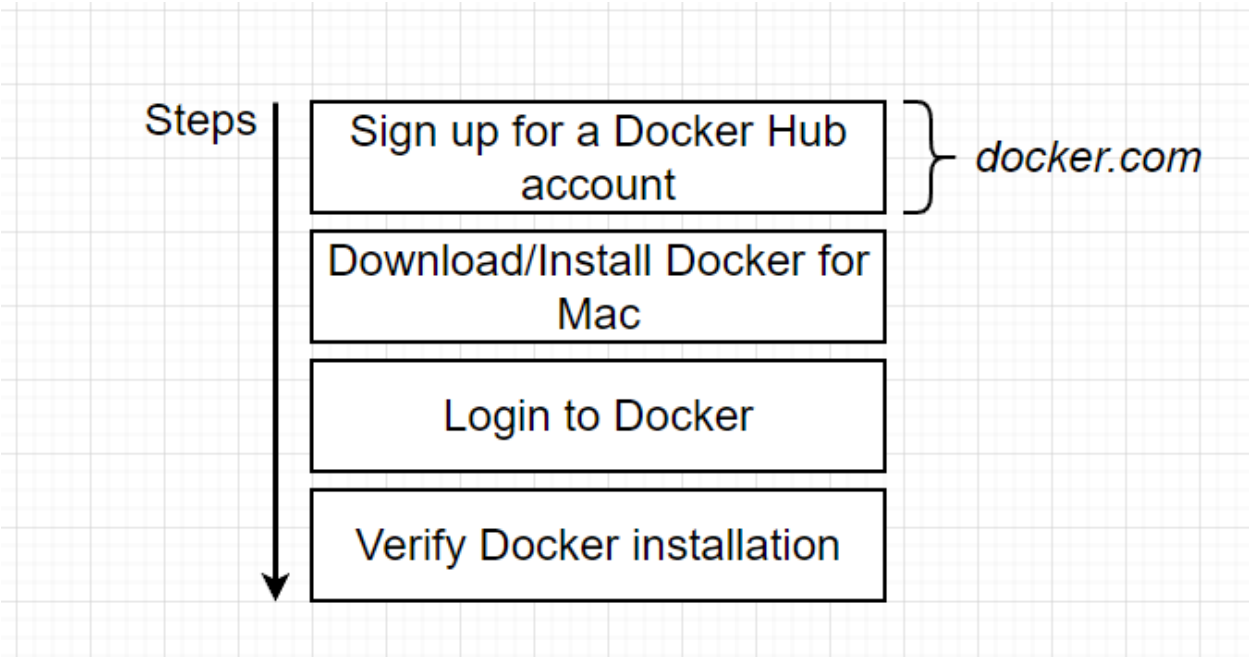
Size: Docker Image much smaller

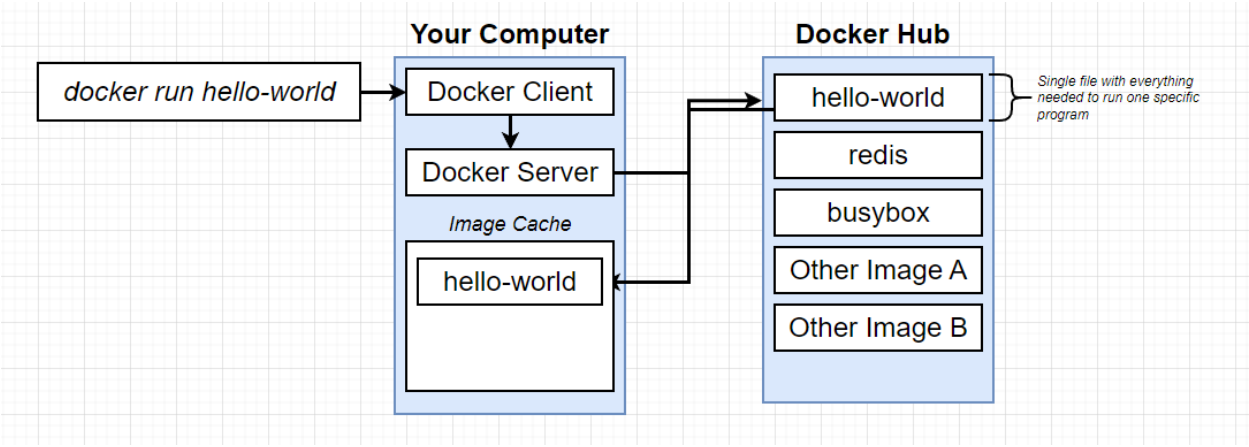
Speed: Docker containers start and run much faster

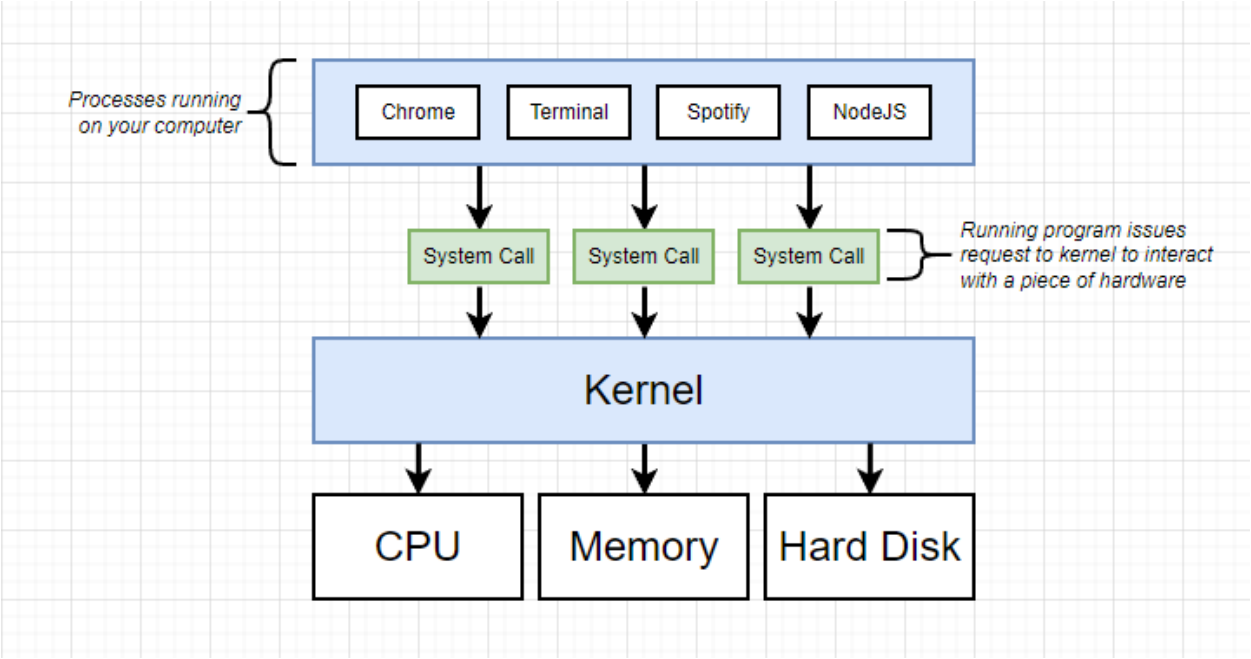
Compatibility: VM of any OS can run on any OS host

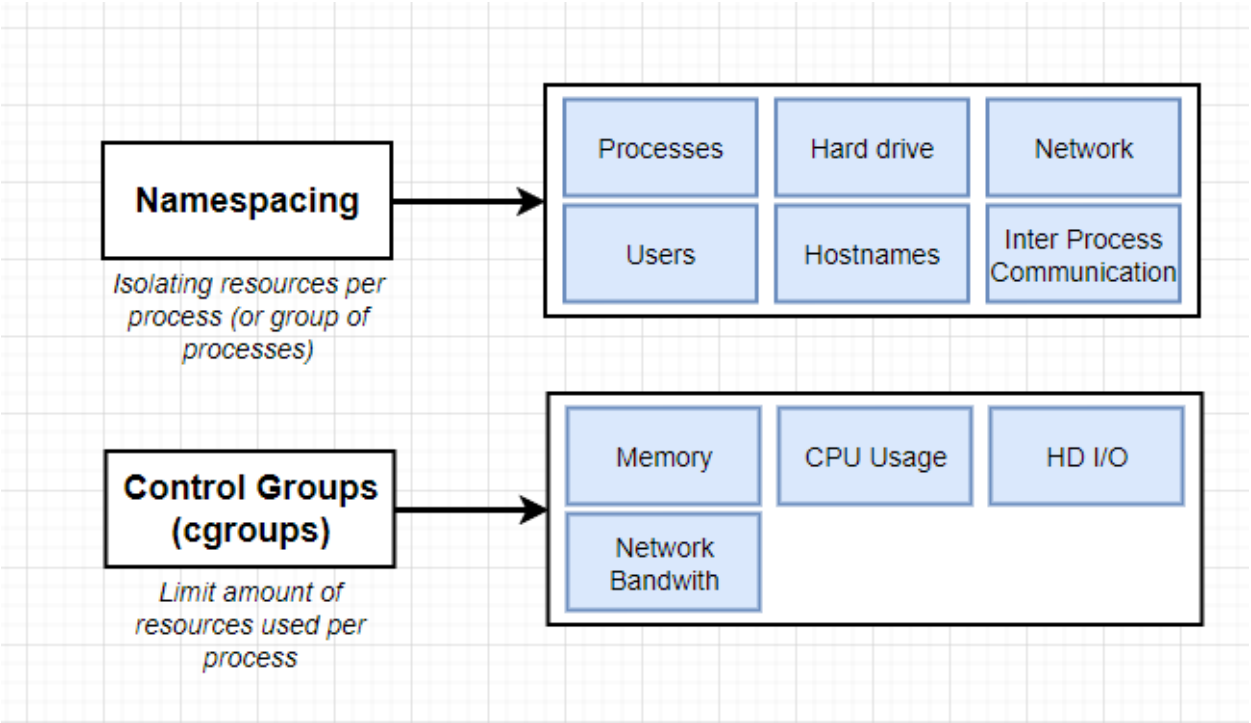
- Abstraction at the app layer
- Multiple containers share the OS kernel

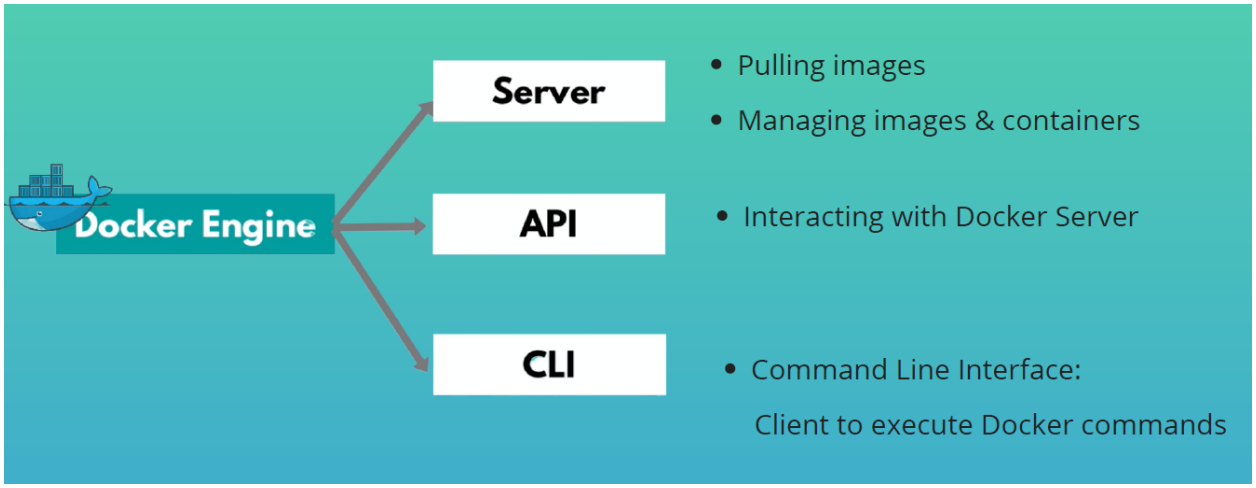


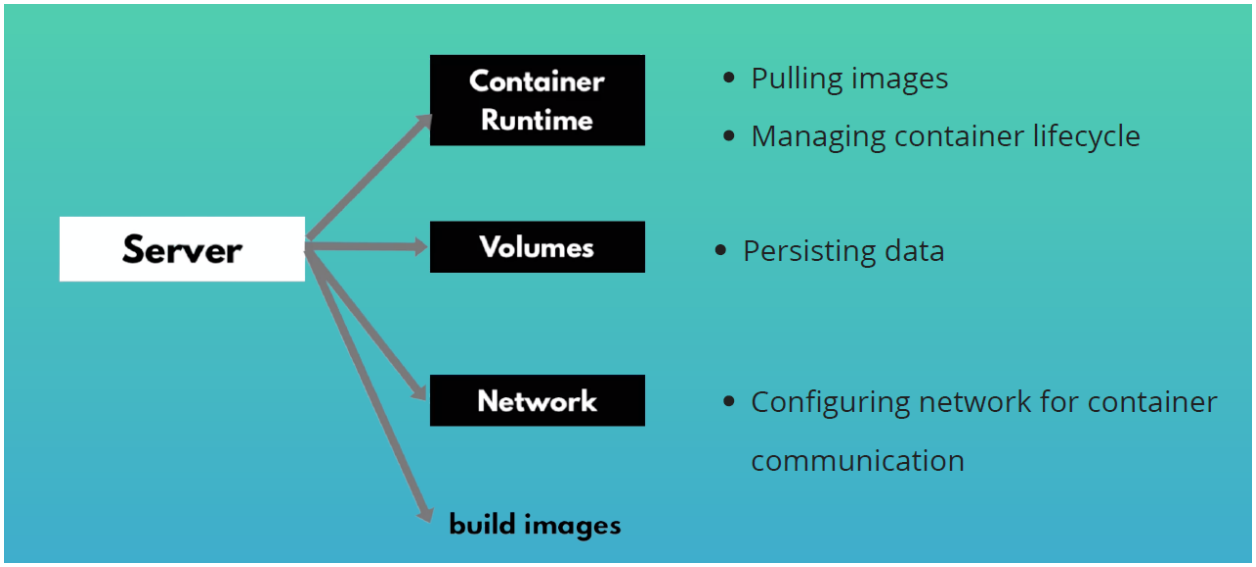

Steps
Sign up for a Docker Hub account
docker.com
Download/Install Docker for Mac
Login to Docker
Verify Docker installation


Your Computer
Docker Hub
docker run hello-world
Docker Client
Docker Server
Image Cache
hello-world
hello-world
redis
busybox
Other Image A
Other Image B
Single file with everything needed to run one specific program


Processes running on your computer
Chrome
Terminal
Spotify
NodeJS
System Call
System Call
System Call
Running program issues request to kernel to interact with a piece of hardware
Kernel
CPU
Memory
Hard Disk


Namespacing
Isolating resources per process (or group of processes)
Processes
Hard drive
Network
Users
Hostnames
Inter Process Communication
Control Groups (cgroups)
Limit amount of resources used per process
Memory
CPU Usage
HD I/O
Network Bandwith

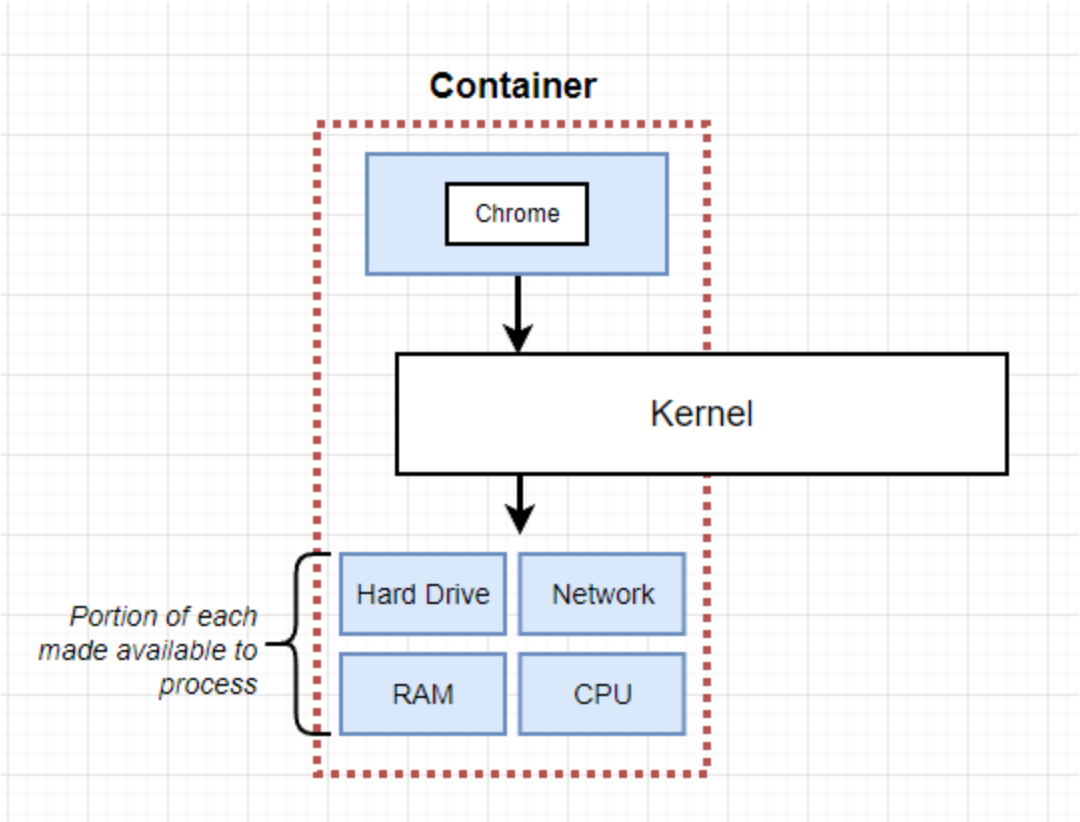

Docker Engine
Server
API
CLI
Pulling images
Managing images & containers
Interacting with Docker Server
Command Line Interface:
Client to execute Docker commands


Server
Container Runtime
Volumes
Network
build images
Pulling images
Managing container lifecycle
Persisting data
Configuring network for container communication

1. **docker run**: creates a container from an image
2. **docker pull**: pull images from the docker repository
3. **docker start**: starts one or more stopped container
4. **docker stop**: stops a running container
5. **docker images**: lists all the locally stored docker images
6. **docker ps:** lists the running containers
7. **docker ps -a:** show all the running and exited containers

docker logs: fetch logs of a container

docker exec -it: creates a new bash session in the container


Container
Chrome
Kernel
Hard Drive
Network
RAM
CPU
Portion of each made available to process

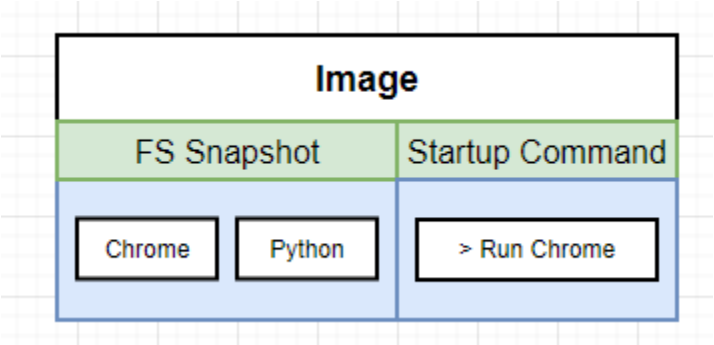
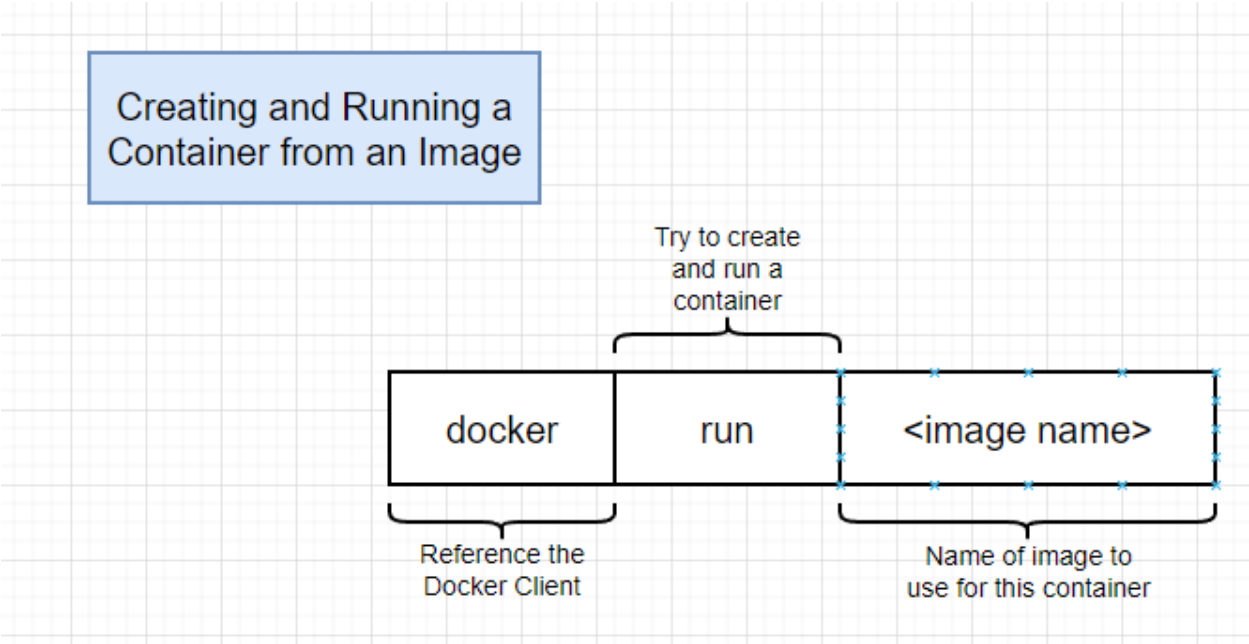
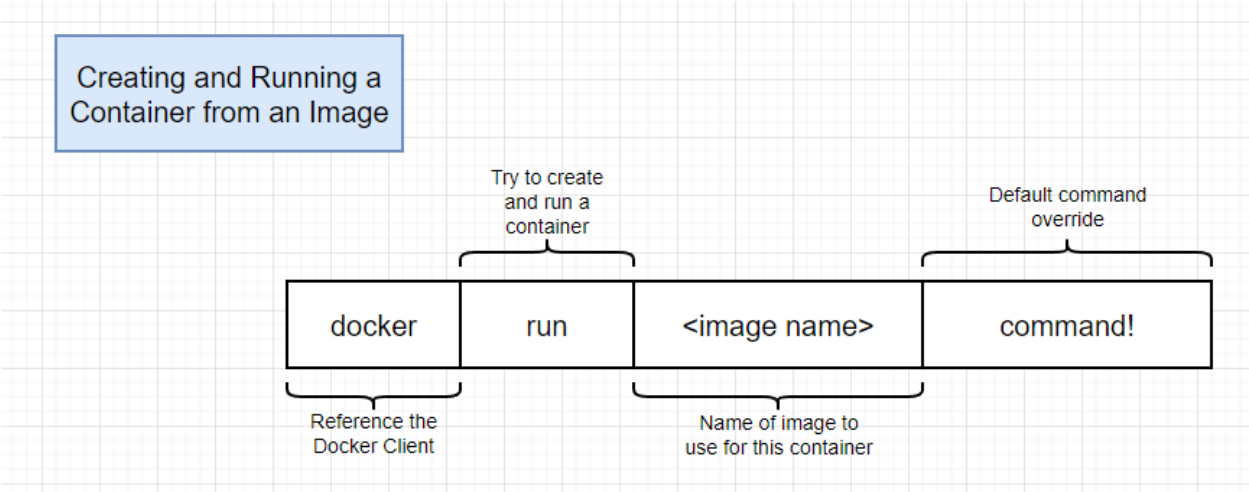
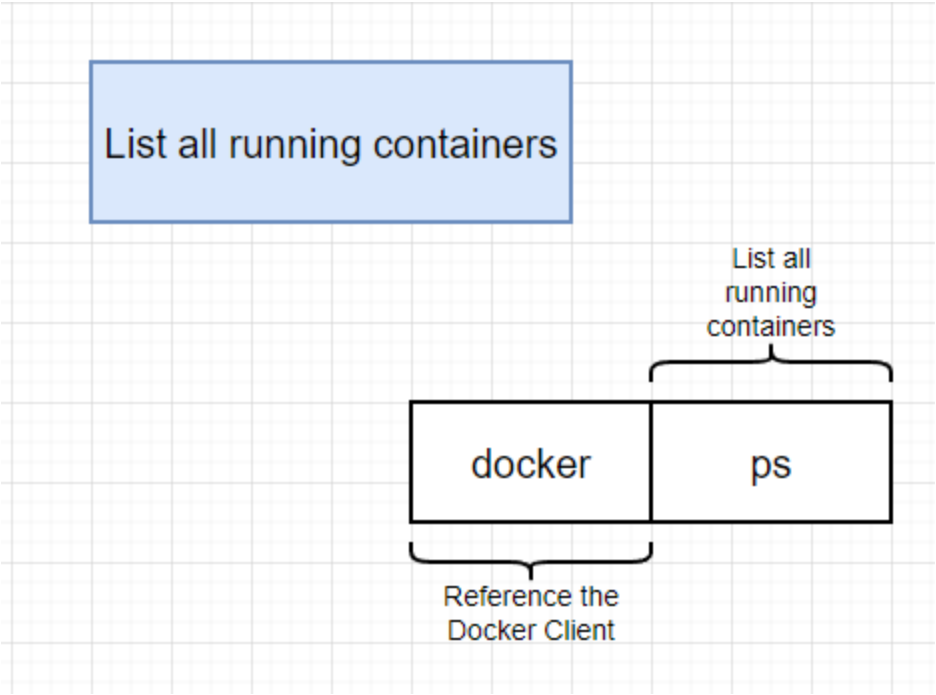
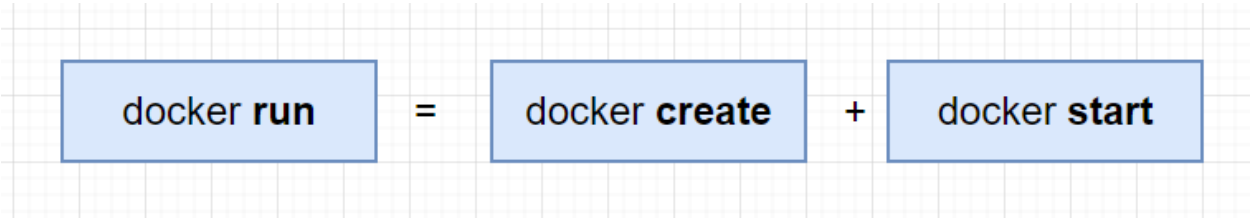


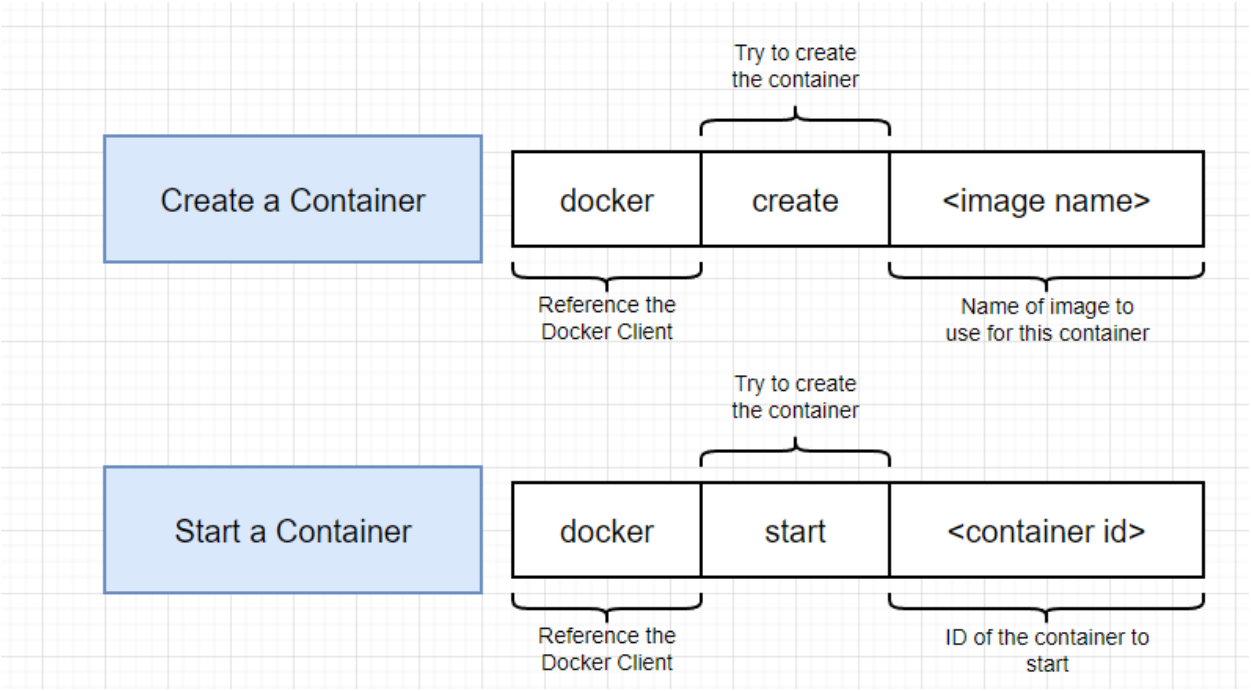
Image
FS Snapshot
Startup Command
Chrome
Python
> Run Chrome

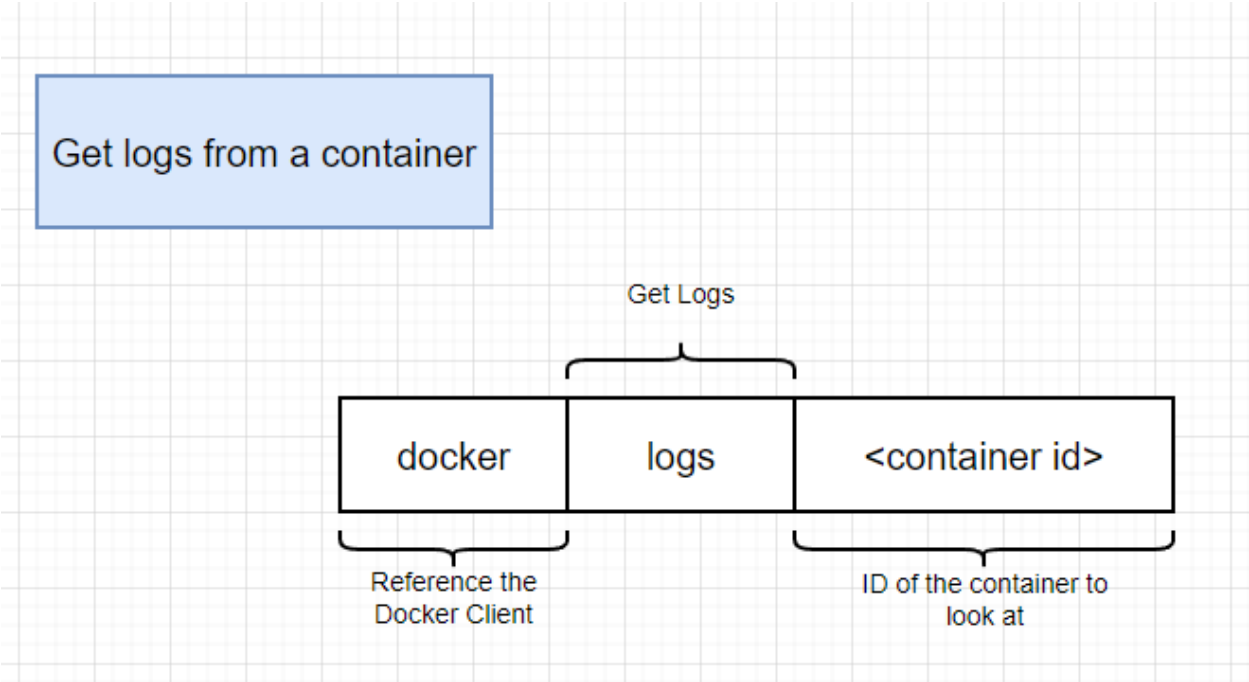

Creating and Running a Container from an Image
Try to create and run a container
docker
run
<image name>
Reference the Docker Client
Name of image to use for this container

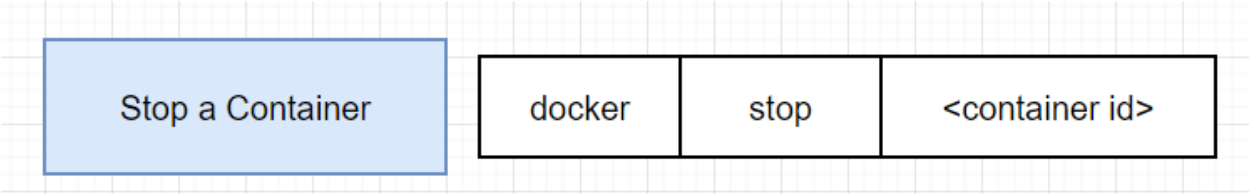

Creating and Running a Container from an Image
Try to create and run a container
Default command override
docker
run
<image name>
command!
Reference the Docker Client
Name of image to use for this container

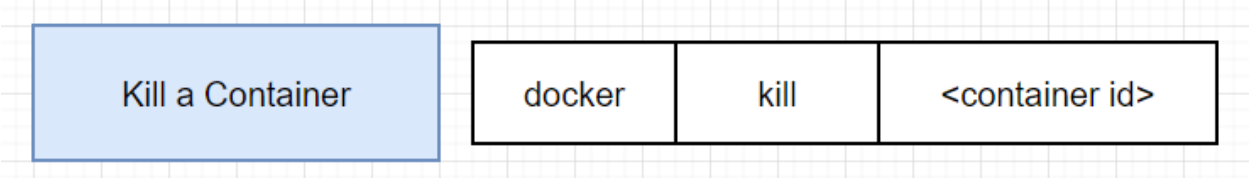

List all running containers
List all running containers
docker
ps
Reference the Docker Client

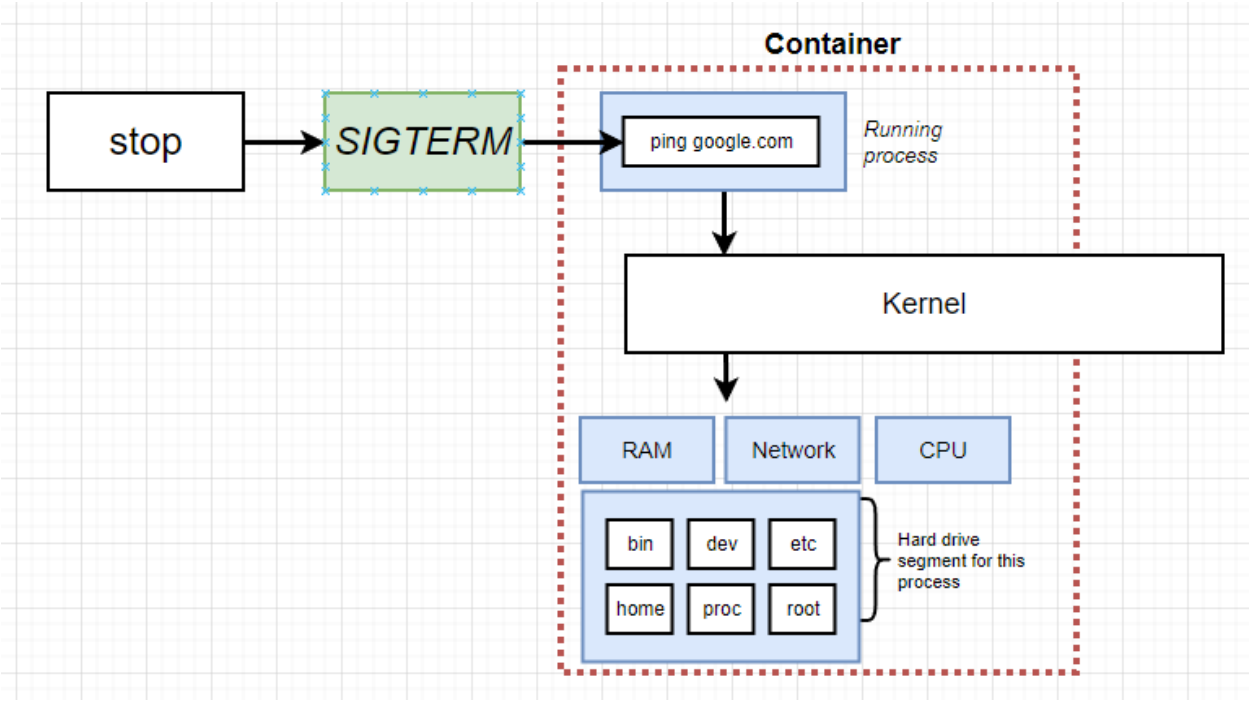

docker **run**
=
docker **create**
+
docker **start**

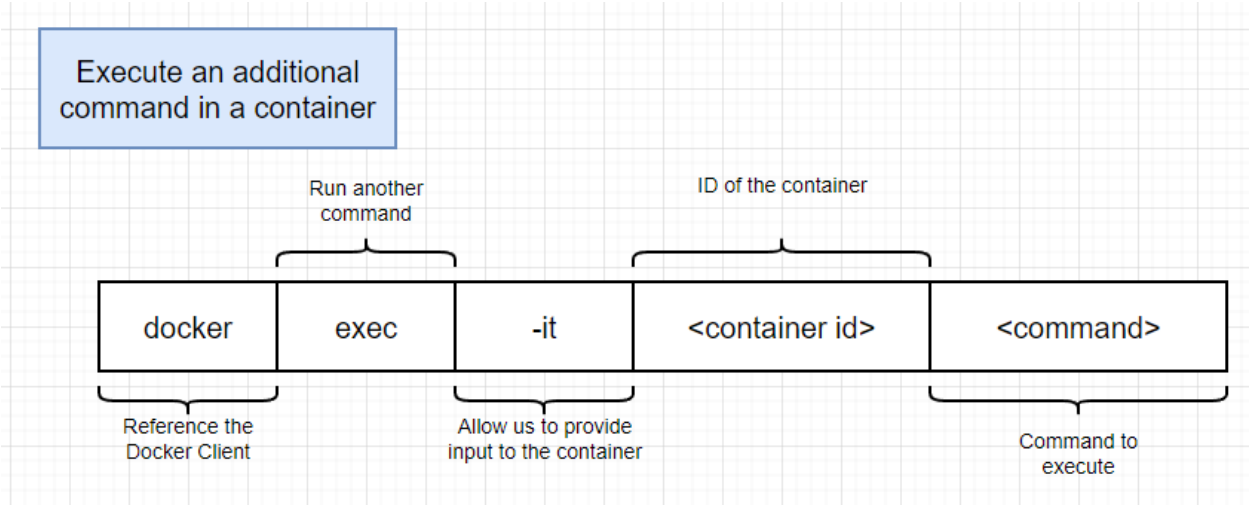

Create a Container
Try to create the container
docker
create
<image name>
Reference the Docker Client
Name of image to use for this container
Start a Container
Try to create the container
docker
start
<container id>
Reference the Docker Client
ID of the container to start

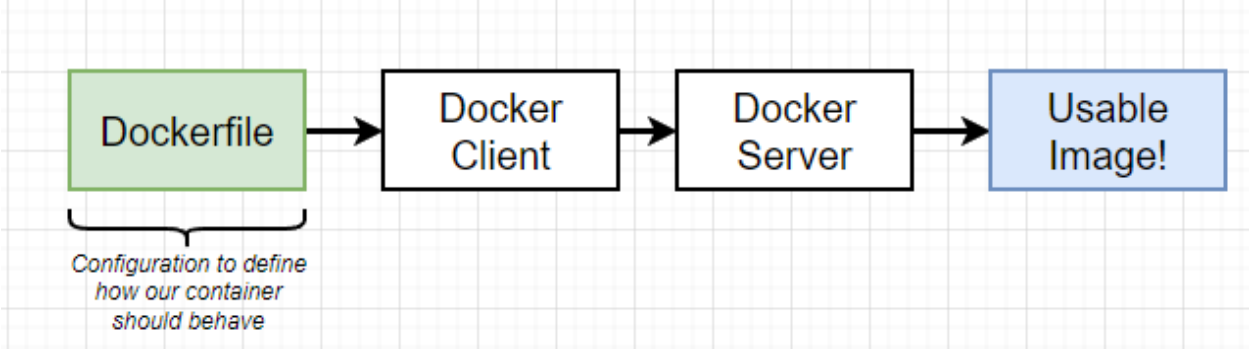

Get logs from a container
Get Logs
docker
logs
<container id>
Reference the Docker Client
ID of the container to look at

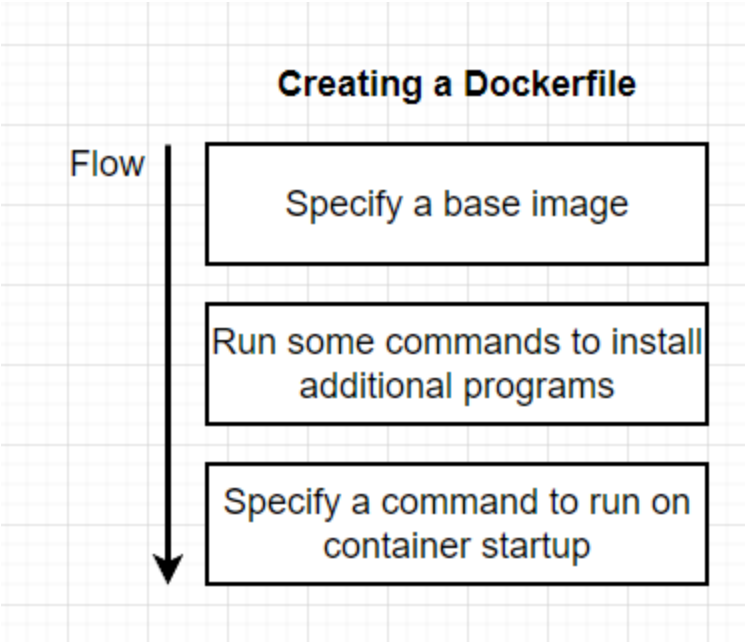

Stop a Container
docker
stop
<container id>

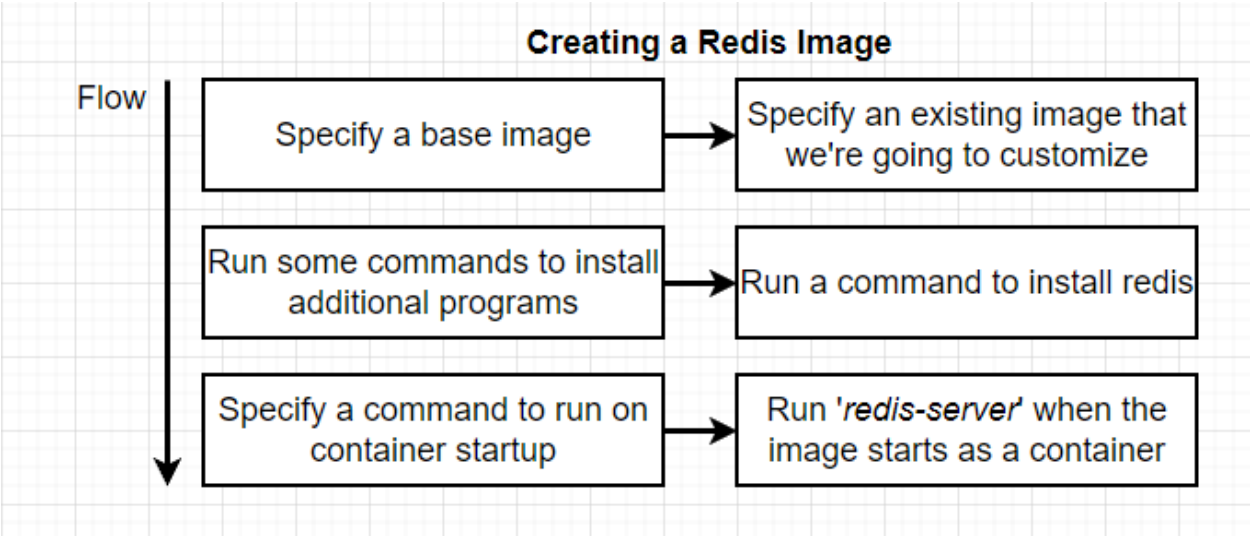

Kill a Container
docker
kill
<container id>

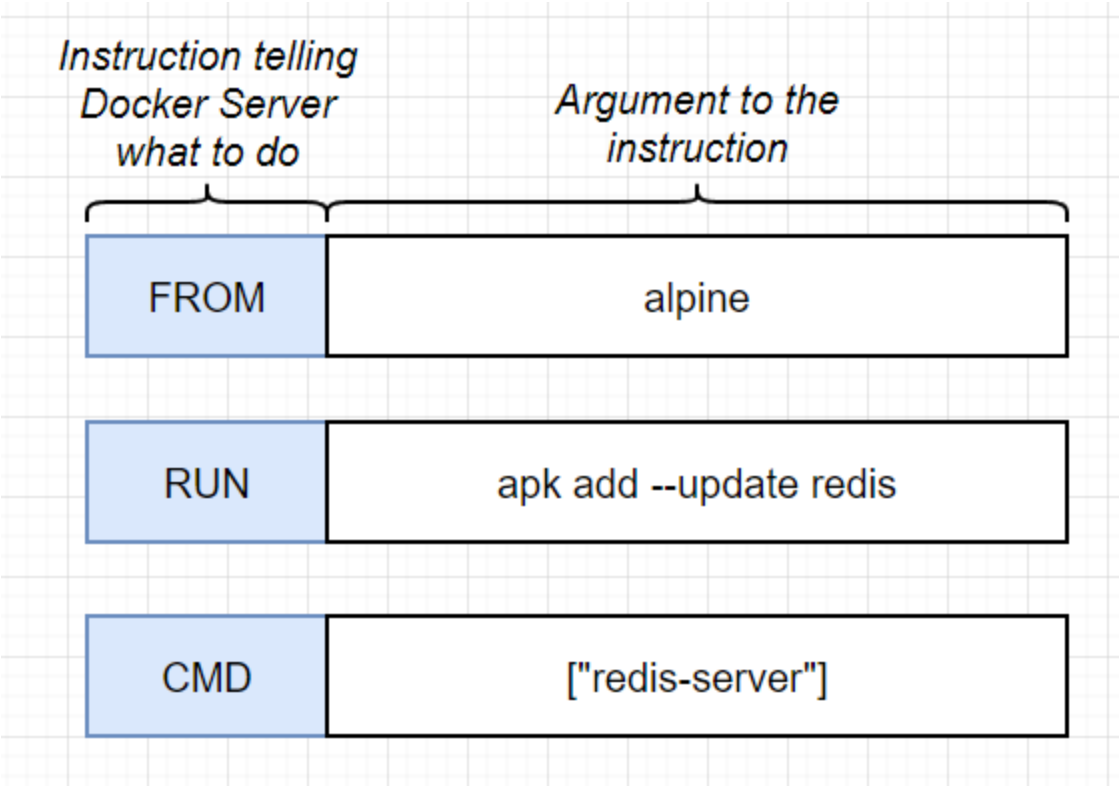

Container
stop
SIGTERM
ping google.com
Running process
Kernel
RAM
Network
CPU
bin
dev
etc
home
proc
root
Hard drive segment for this process

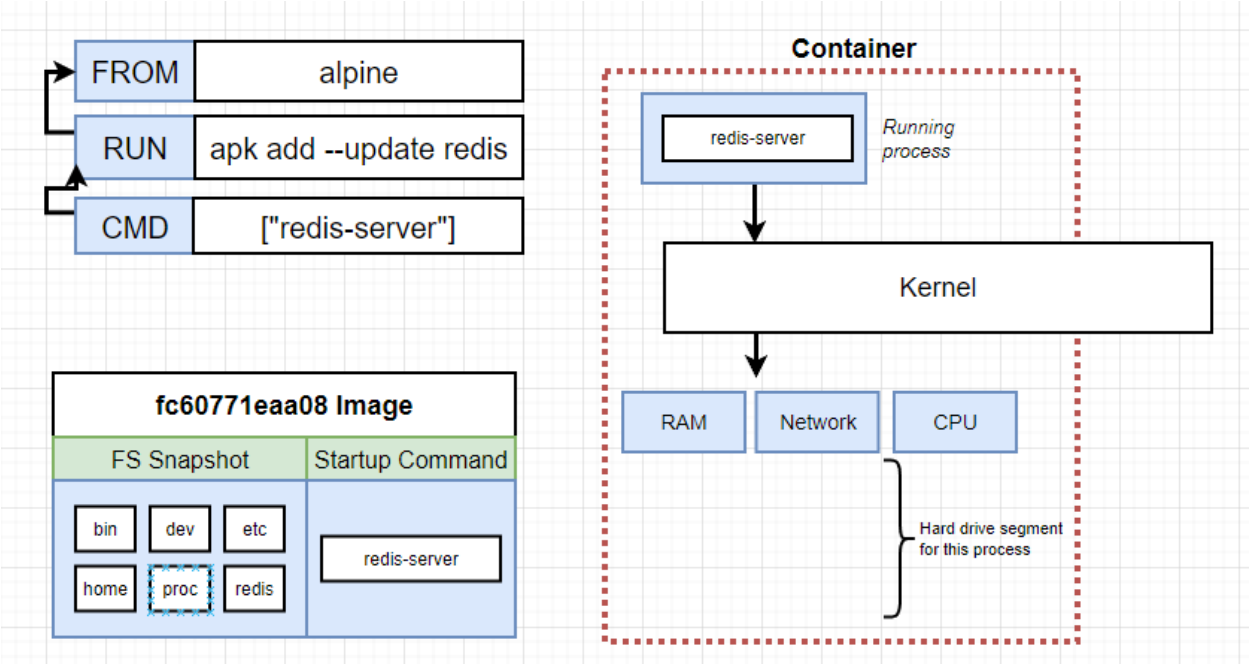

Execute an additional command in a container
Run another command
ID of the container
docker
exec
-it
<container id>
<command>
Reference the Docker Client
Allow us to provide input to the container
Command to execute

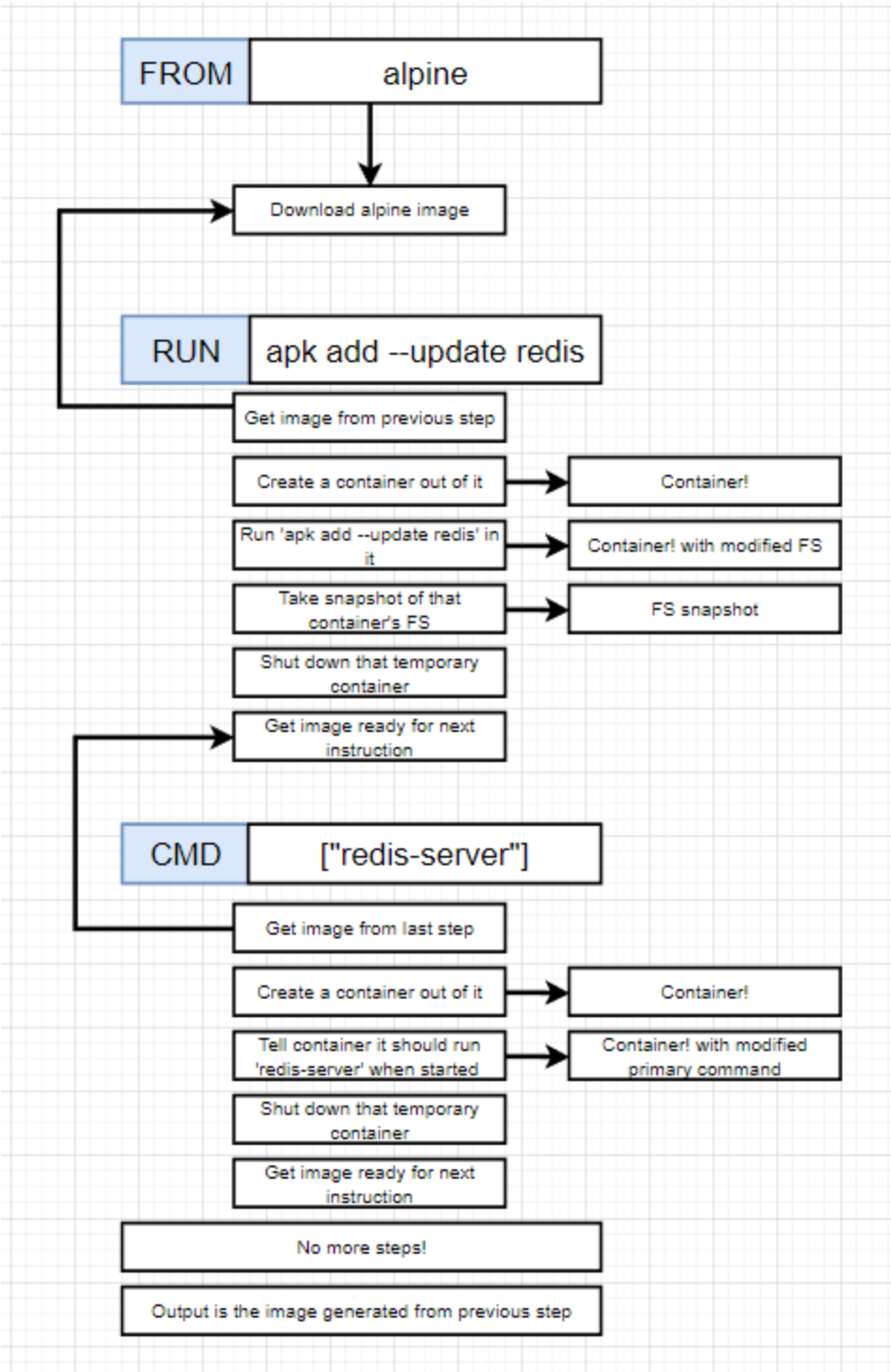

Dockerfile
Docker Client
Docker Server
Usable Image!
Configuration to define how our container should behave

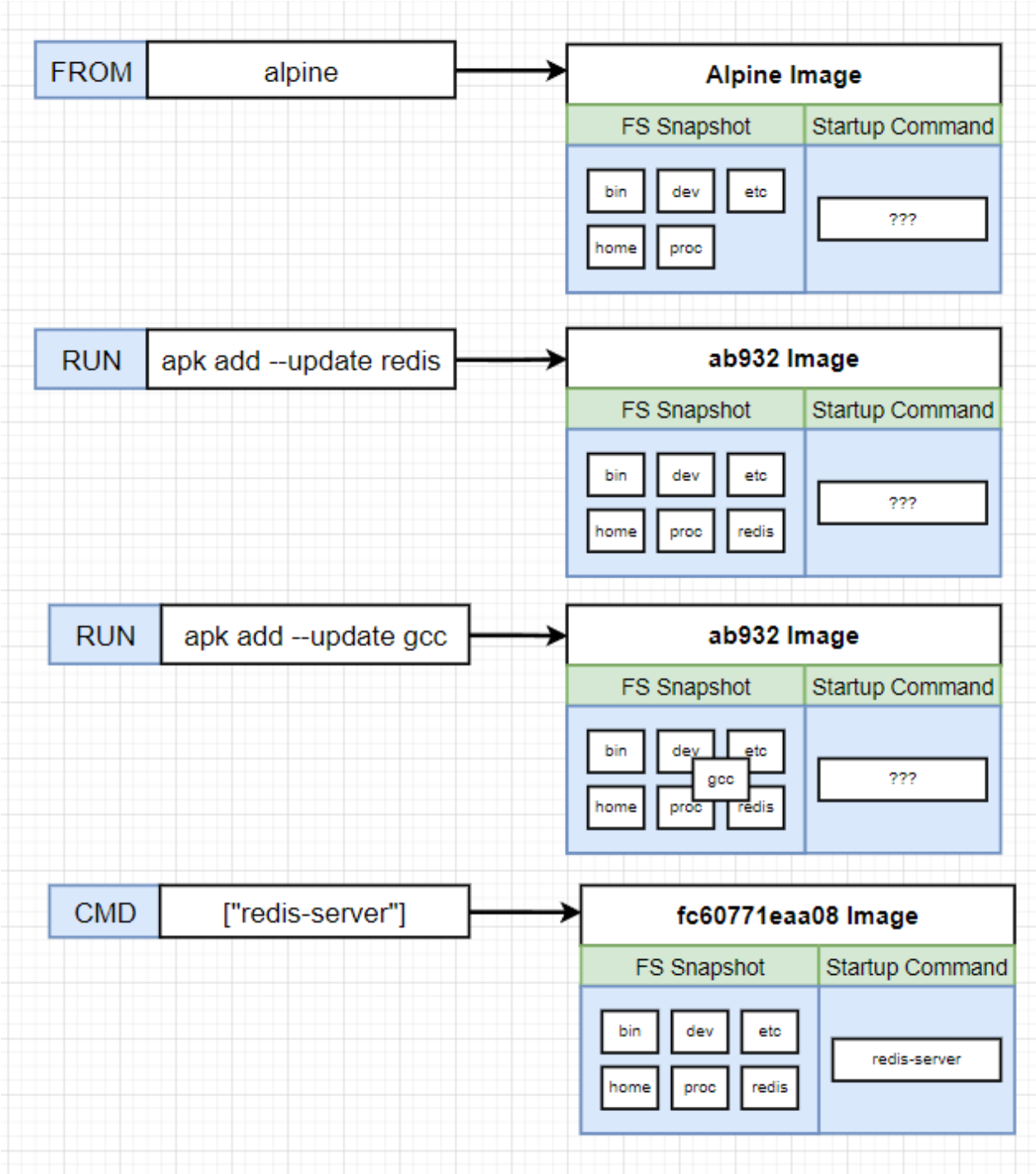

Creating a Dockerfile
Flow
Specify a base image
Run some commands to install additional programs
Specify a command to run on container startup

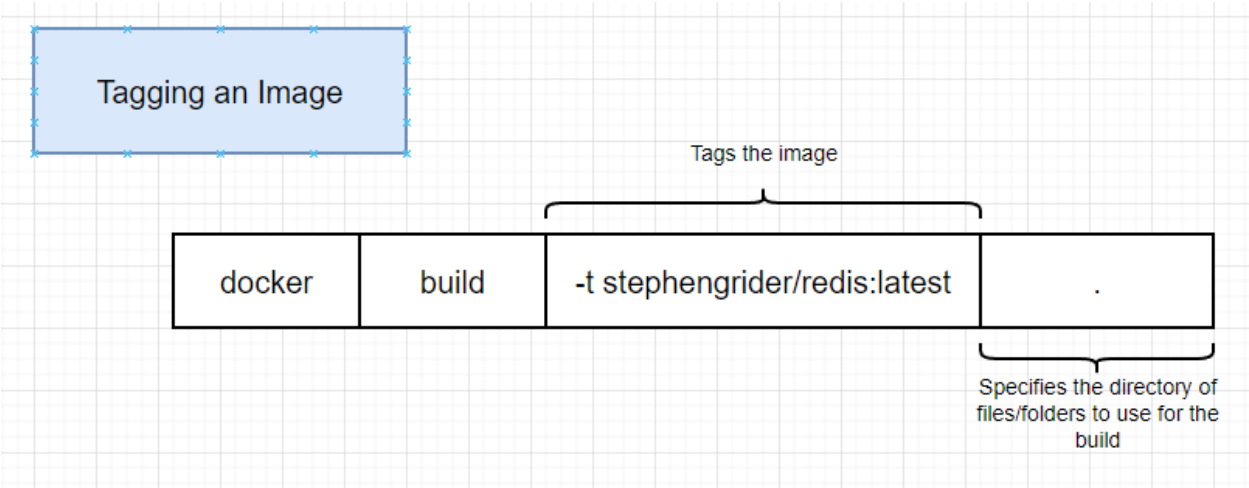

Creating a Redis Image
Flow
Specify a base image
Specify an existing image that we're going to customize
Run some commands to install additional programs
Run a command to install redis
Specify a command to run on container startup
Run '*redis-server*' when the image starts as a container

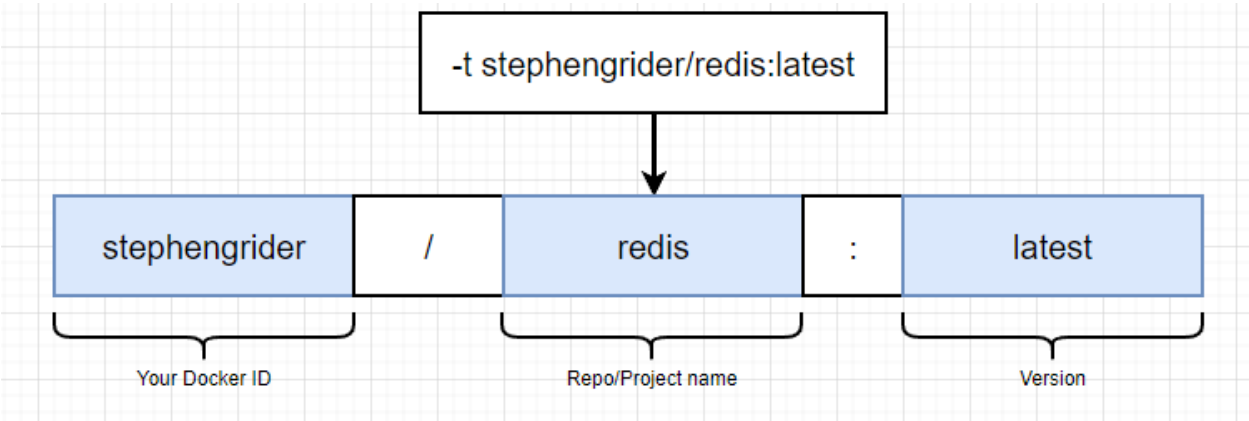

Instruction telling Docker Server what to do
Argument to the instruction
FROM
alpine
RUN
apk add --update redis
CMD
["redis-server"]

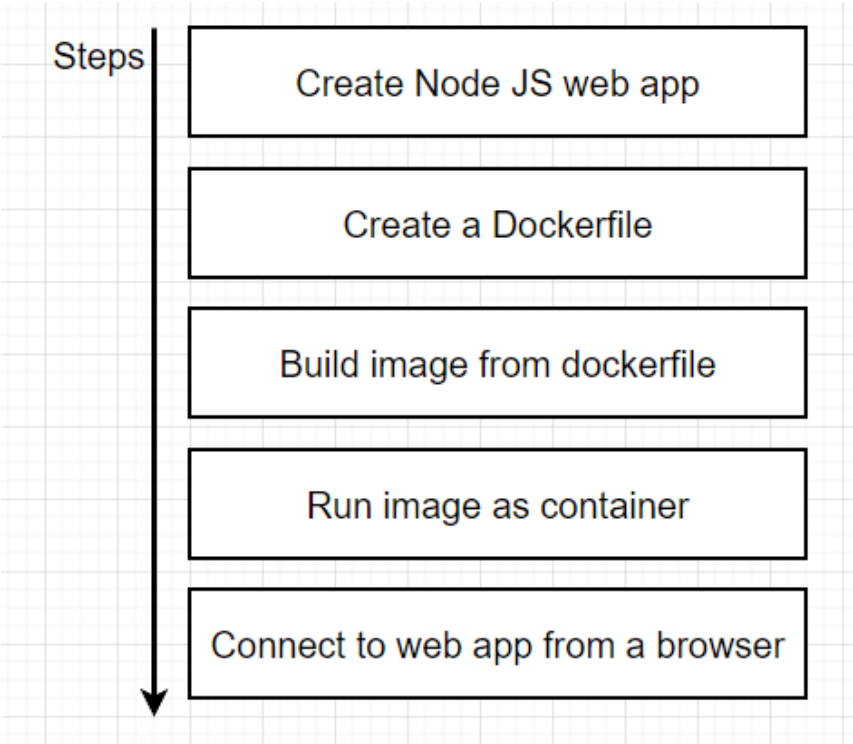

FROM
alpine
RUN
apk add --update redis
CMD
["redis-server"]
fc60771eaa08 Image
FS Snapshot
Startup Command
bin
dev
etc
home
proc
redis
redis-server
Container
redis-server
Running process
Kernel
RAM
Network
CPU
Hard drive segment for this process

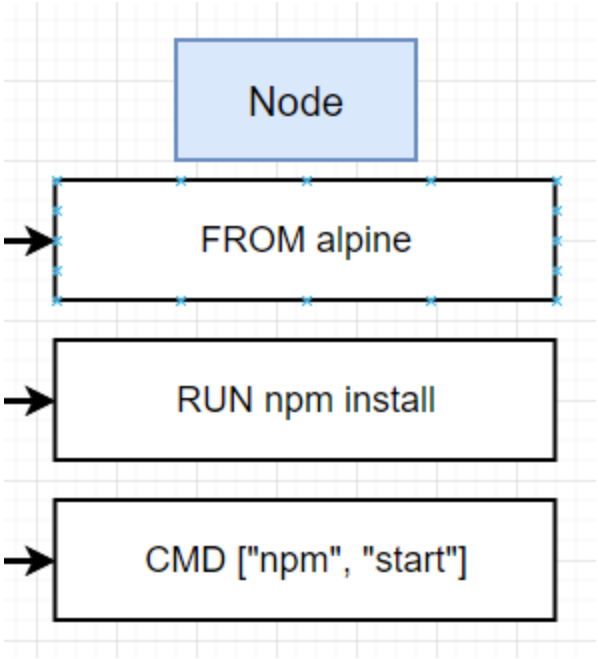

FROM
alpine
Download alpine image
RUN
apk add --update redis
Get image from previous step
Create a container out of it
Container!
Run 'apk add --update redis' in it
Container! with modified FS
Take snapshot of that container's FS
FS snapshot
Shut down that temporary container
Get image ready for next instruction
CMD
["redis-server"]
Get image from last step
Create a container out of it
Container!
Tell container it should run 'redis-server' when started
Container! with modified primary command
Shut down that temporary container
Get image ready for next instruction
No more steps!
Output is the image generated from previous step

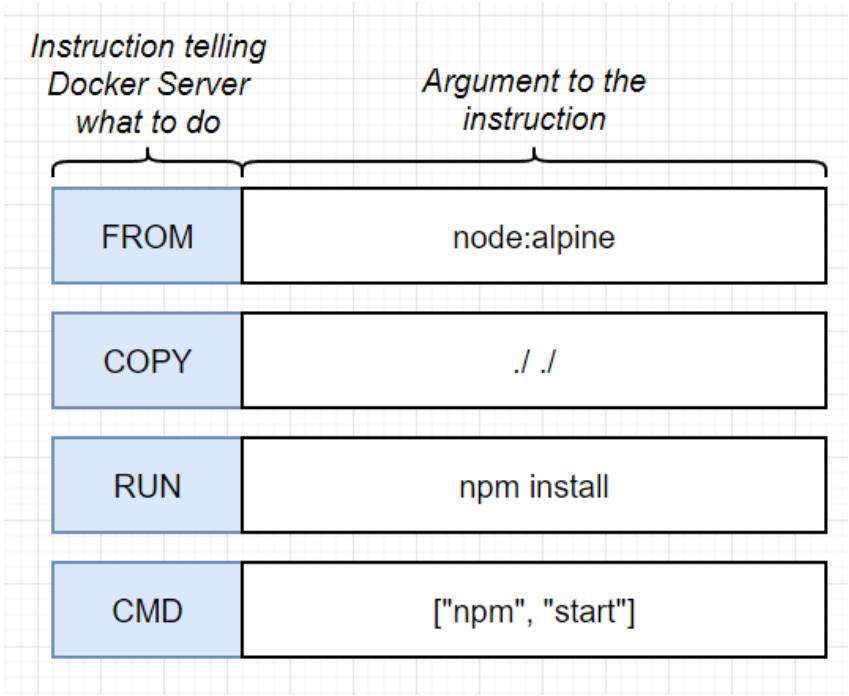

FROM
alpine
Alpine Image
FS Snapshot
Startup Command
bin
dev
etc
home
proc
???
RUN
apk add --update redis
ab932 Image
FS Snapshot
Startup Command
bin
dev
etc
home
proc
redis
???
RUN
apk add --update gcc
ab932 Image
FS Snapshot
Startup Command
bin
dev
etc
gcc
home
proc
redis
???
CMD
["redis-server"]
fc60771eaa08 Image
FS Snapshot
Startup Command
bin
dev
etc
home
proc
redis
redis-server

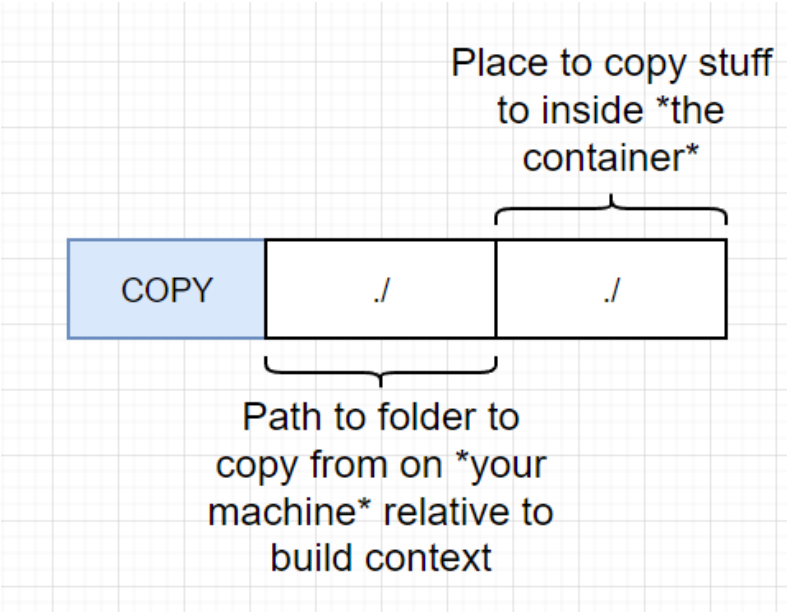

Tagging an Image
Tags the image
docker
build
-t stephengrider/redis:latest
.
Specifies the directory of files/folders to use for the build

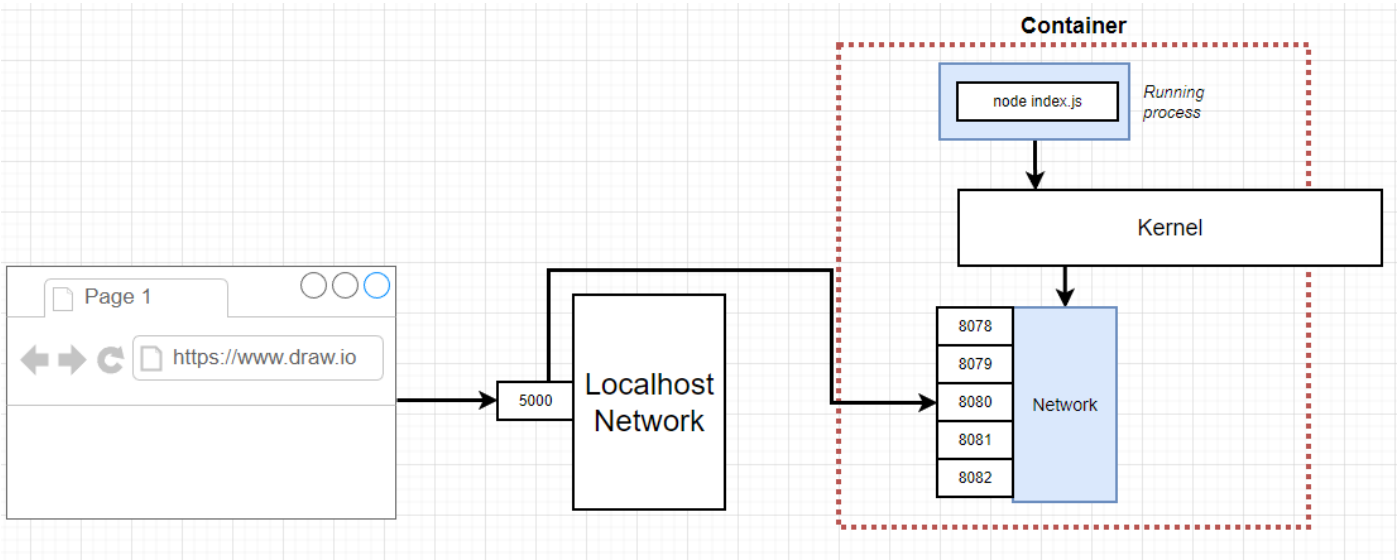

-t stephengrider/redis:latest
stephengrider
/
redis
:
latest
Your Docker ID
Repo/Project name
Version

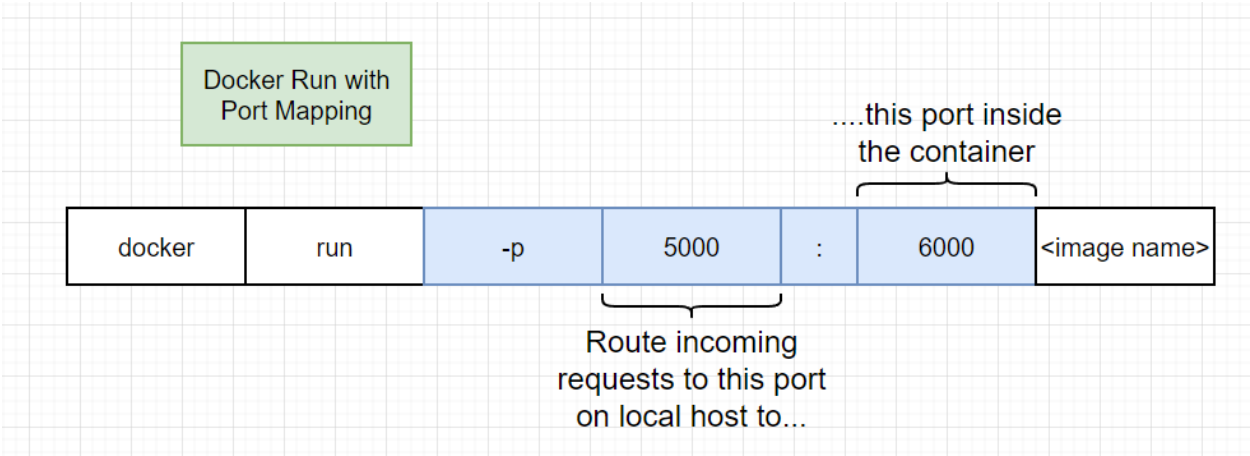

Steps
Create Node JS web app
Create a Dockerfile
Build image from dockerfile
Run image as container
Connect to web app from a browser

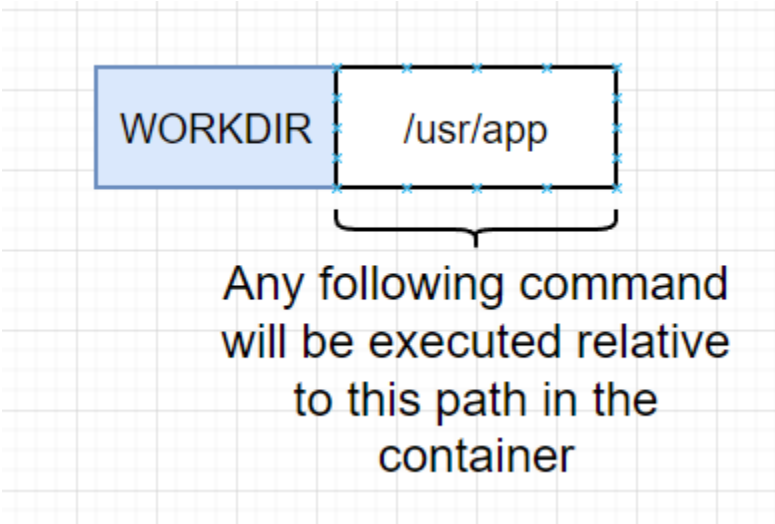

Node
FROM alpine
RUN npm install
CMD ["npm", "start"]

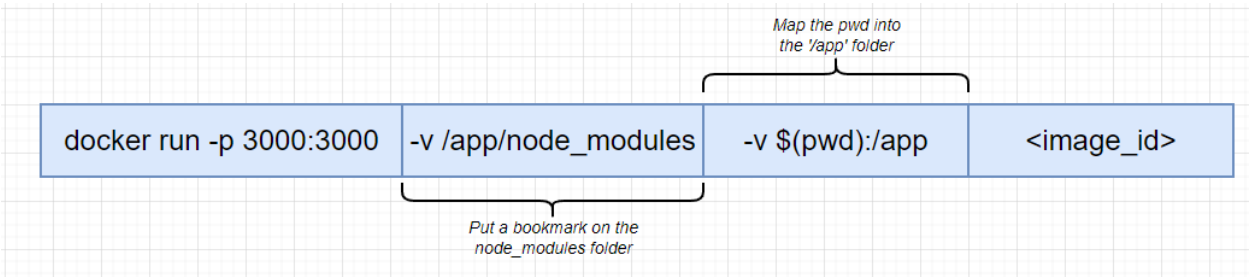

Instruction telling Docker Server what to do
Argument to the instruction
FROM
node:alpine
COPY
./ ./
RUN
npm install
CMD
["npm", "start"]

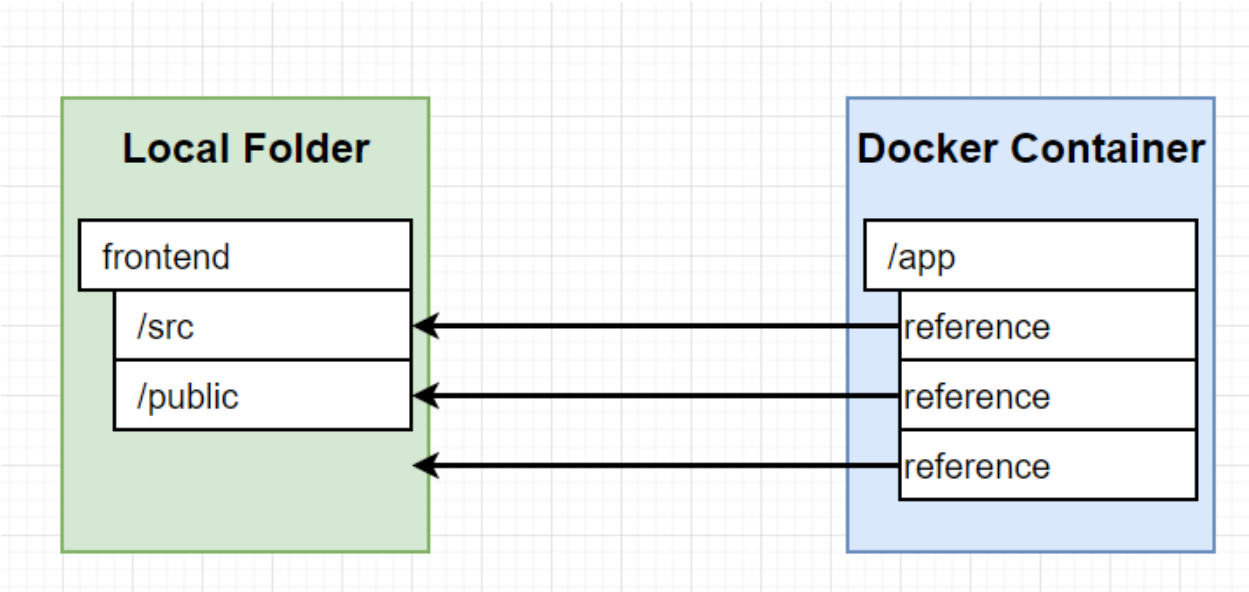

Place to copy stuff to inside *the container*
COPY
./
./
Path to folder to copy from on *your machine* relative to build context


Container
node index.js
Running process
Kernel
Page 1
https://www.draw.io
5000
Localhost Network
8078
8079
8080
8081
8082
Network


Docker Run with Port Mapping
....this port inside the container
docker
run
-p
5000
:
6000
<image name>
Route incoming requests to this port on local host to...


WORKDIR
/usr/app
Any following command will be executed relative to this path in the container


Map the pwd into the '/app' folder
docker run -p 3000:3000
-v /app/node_modules
-v $(pwd):/app
<image_id>
Put a bookmark on the node_modules folder


Local Folder
frontend
/src
/public
Docker Container
/app
reference
reference
reference

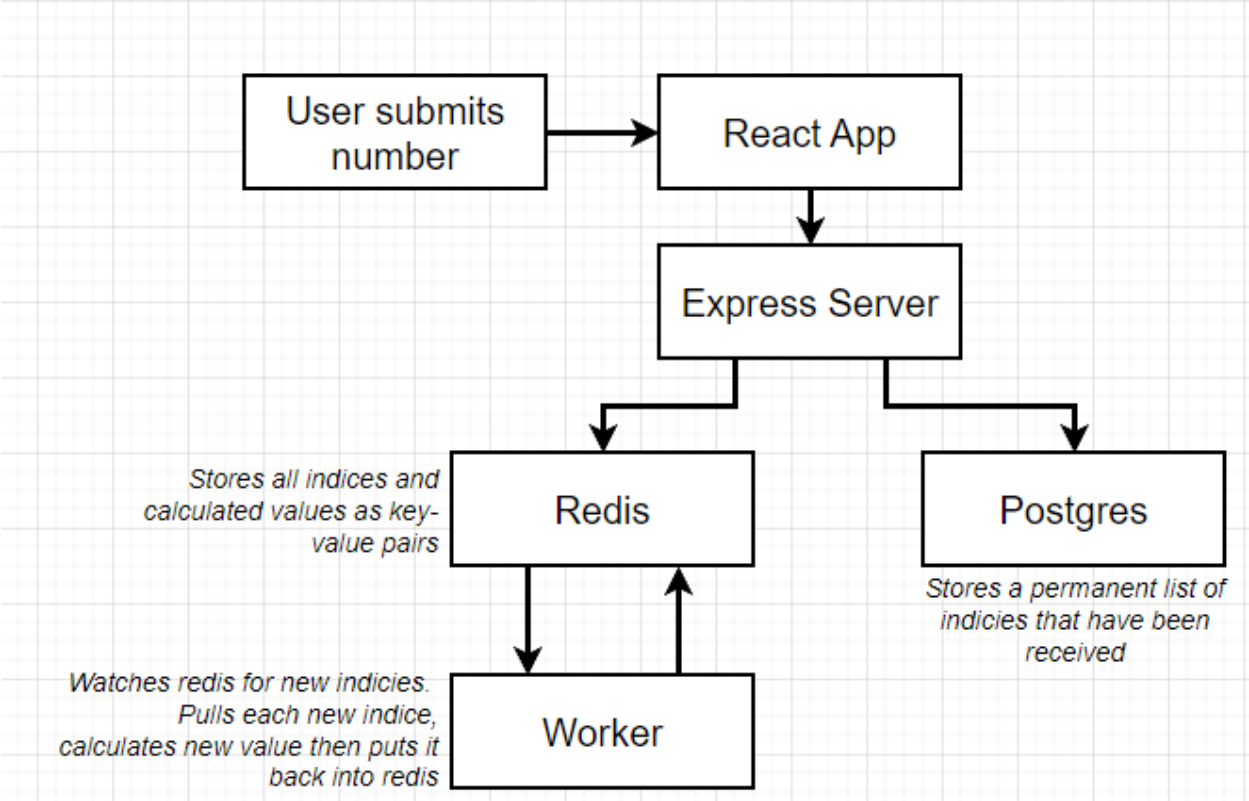

User submits number
React App
Express Server
Stores all indices and calculated values as key-value pairs
Redis
Postgres
Stores a permanent list of indicies that have been received
Watches redis for new indicies. Pulls each new indice, calculates new value then puts it back into redis
Worker

Image Environment Blueprint	DOCKERFILE
install node	**FROM** node
set MONGO_DB_USERNAME=admin set MONGO_DB_PWD=password	**ENV** MONGO_DB_USERNAME=admin \ MONGO_DB_PWD=password
create /home/app folder	**RUN** mkdir -p /home/app
copy current folder files to /home/app	**COPY** . /home/app
start the app with: "node server.js"	**CMD** ["node", "server.js"]

CMD = entrypoint command

You can have multiple RUN commands

blueprint for building images

- Each instruction in a Dockerfile results in an Image Layer:

- Command to build an image from a Dockerfile and a context
- The build's context is the set of files at a specified location

- When you work in a company, you will be working with a private docker registry
- **Public = DockerHub**, Example for **Private** = Amazon Elastic Container Registry "**AWS ECR**"

Difference to DockerHub

1. You need to **login,** so authenticate with the registry before fetching or pushing the image
2. **Tag** your image with the **registry address and name**,
3. Push the tagged image

```
docker login
- reponame
- username
- password

docker tag {repo-name}:{image-version}
docker push {tagged-image}
```

registryDomain/imageName:tag

Environment Variables

variableName=value	variableName
*Sets a variable in the container at *run time**	*Sets a variable in the container at *run time** *Value is taken from *your computer**

Restart policies

Using the --restart flag on Docker run you can specify a restart policy for how a container should or should not be restarted on exit.

When a restart policy is active on a container, it will be shown as either Up or Restarting in docker ps. It can also be useful to use docker events to see the restart policy in effect.

```
docker run --always
```

> Always restart the container regardless of the exit status. When you specify always, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.

I recommend you this documentation about **restart-policies**

[Documentation - Restart policies]

Update Docker v19.03

> **Restart policies (--restart)**
>
> Use Docker's --restart to specify a container's restart policy. A restart policy > controls whether the Docker daemon restarts a container after exit. Docker supports the following restart policies:
>
> **always** Always restart the container regardless of the exit status. When you specify always, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.

```
$ docker run --restart=always redis
```

I am trying to understand the actual reason for mounting `docker.sock` in `docker-compose.yml` file. Is it for auto-discovery?

```
volumes:
  - /var/run/docker.sock:/var/run/docker.sock
```

`docker.sock` is the UNIX socket that Docker daemon is listening to. It's the main entry point for Docker API. It also can be TCP socket but by default for security reasons Docker defaults to use UNIX socket.

Docker cli client uses this socket to execute docker commands by default. You can override these settings as well.

There might be different reasons why you may need to mount Docker socket inside a container. Like launching new containers from within another container. Or for auto service discovery and Logging purposes. This increases attack surface so you should be careful if you mount docker socket inside a container there are trusted codes running inside that container otherwise you can simply compromise your host that is running docker daemon, since Docker by default launches all containers as root.

Docker socket has a docker group in most installation so users within that group can run docker commands against docker socket without root permission but actual docker containers still get root permission since docker daemon runs as root effectively (it needs root permission to access namespace and cgroups).

```
denied: requested access to the resource is denied
```

```
Password:
Login Succeeded

[redacted]>docker push firstimage
The push refers to a repository [docker.io/library/firstimage]
5f70bf18a086: Preparing
d061ee1340ec: Preparing
d511ed9e12e1: Preparing
091abc5148e4: Preparing
b26122d57afa: Preparing
37ee47034d9b: Waiting
528c8710fd95: Waiting
1154ba695078: Waiting
denied: requested access to the resource is denied
```

to **log out**, then **log in** from the command line to your docker hub account

```
# you may need log out first `docker logout` ref. https://stackoverflow.com/a/538358
docker login
```

According to the docs:

```
You need to include the namespace for Docker Hub to associate it with your account.
The namespace is the same as your Docker Hub account name.
You need to rename the image to YOUR_DOCKERHUB_NAME/docker-whale.
```

So, this means you have to **tag** your image before pushing:

```
docker tag firstimage YOUR_DOCKERHUB_NAME/firstimage
```

and then you should be able to push it.

```
docker push YOUR_DOCKERHUB_NAME/firstimage
```

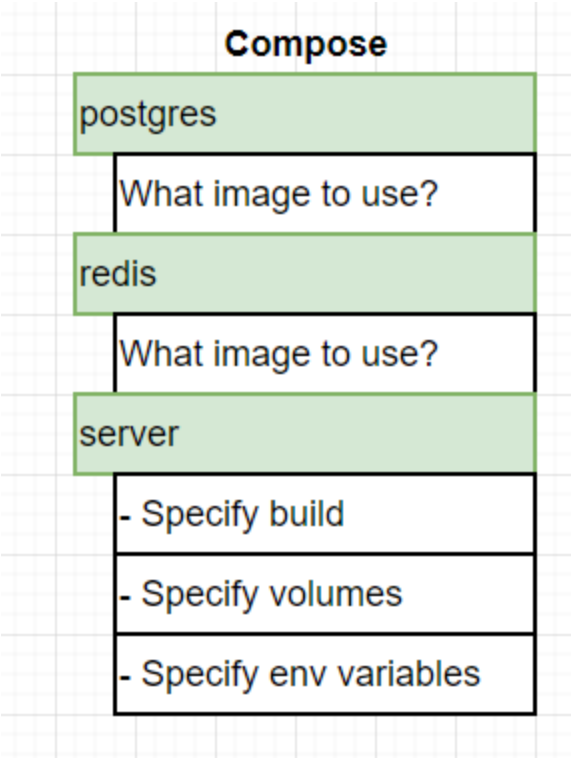
Environment variables are supported by the following list of instructions in the `Dockerfil`

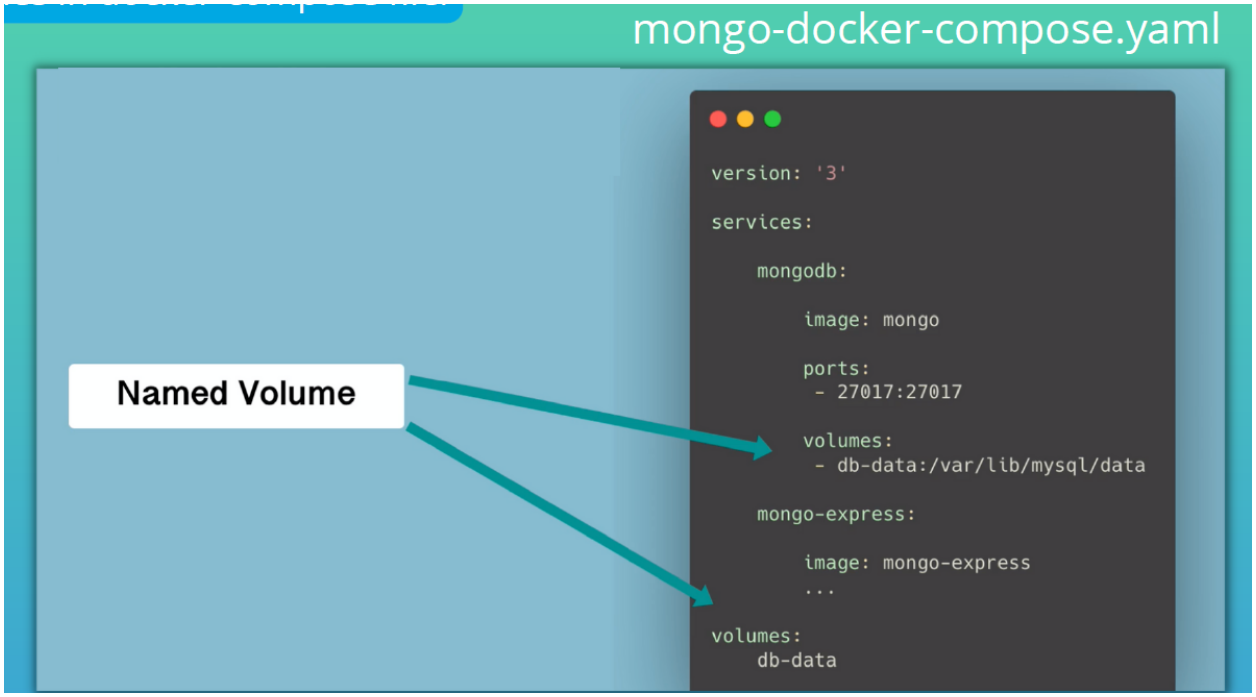
- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`
- `ONBUILD` (when combined with one of the supported instructions above)

Docker-Compose

Example docker-compose.yaml file

```yaml
version: '3'
services:
  mongodb:
    image: mongo
    ports:
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
      - 8080:8081
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER=mongodb
```

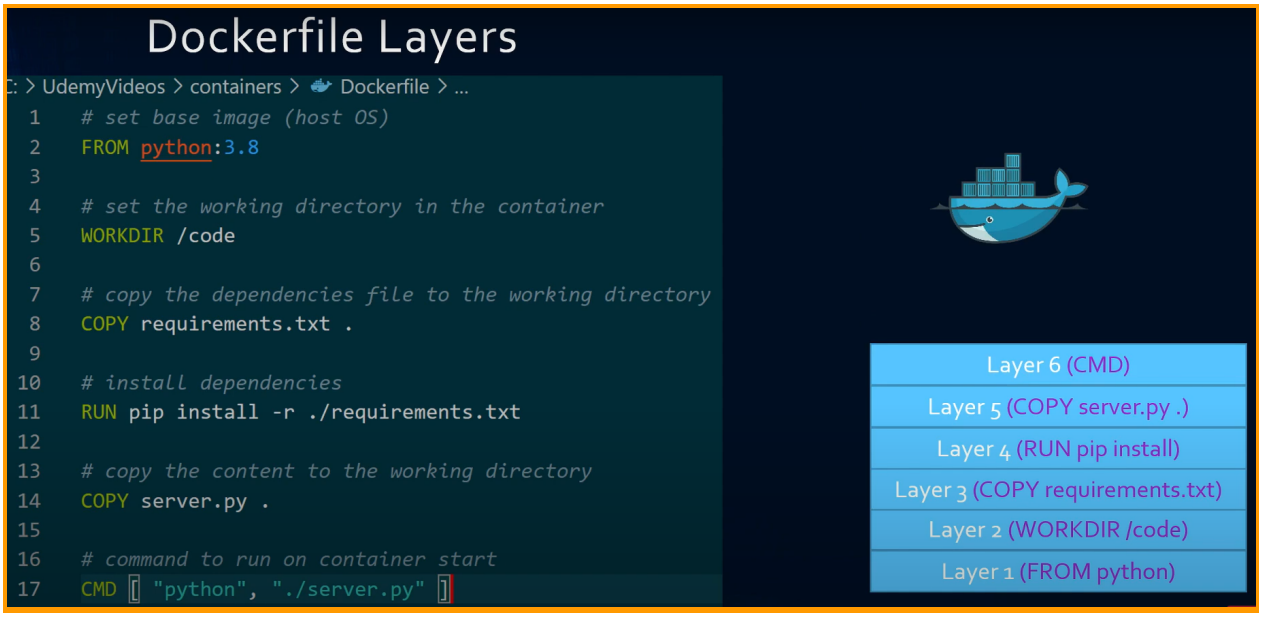




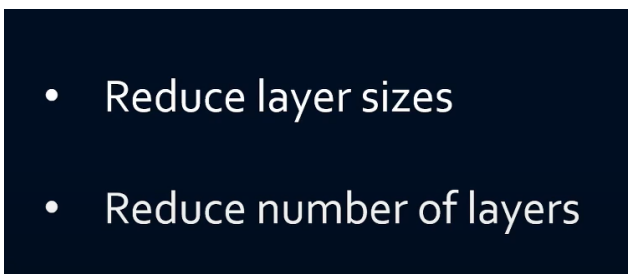
When the dockerfile has any name other than Dockerfile, we can use --file to write its name:

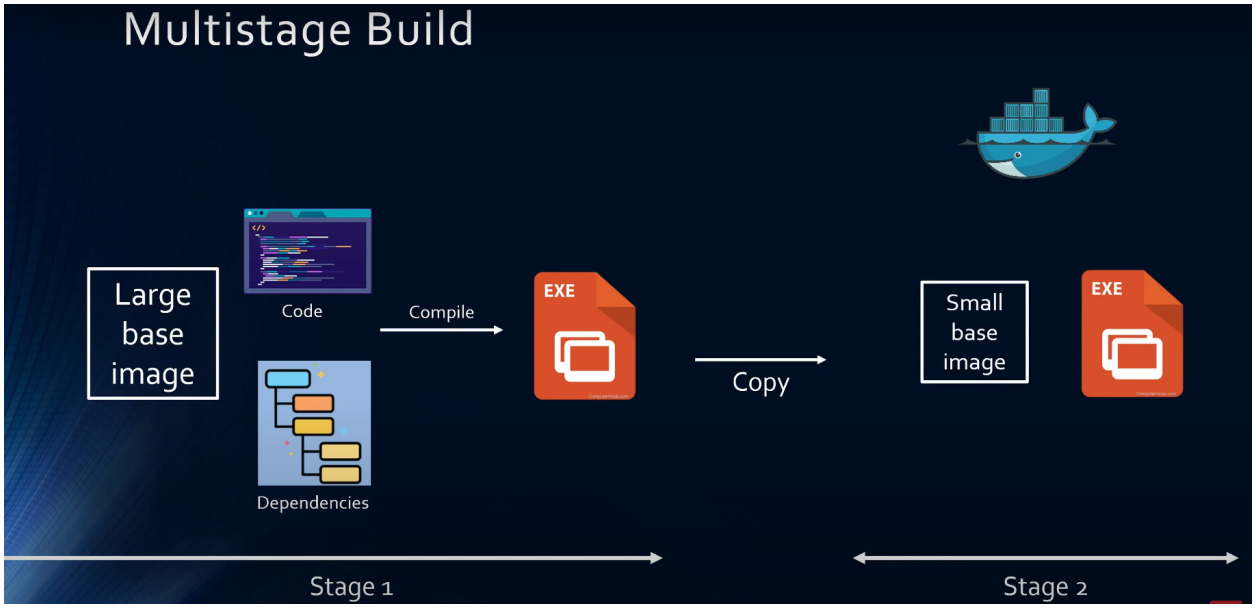
```
$ docker image build --tag demo --file Dockerfile-simple .
```

Multi-layer Docker Images



docker history <IMAGE-NAME> shows all the layers of an image






```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Stage Name

Compile

New stage starts

Copy compiled code from "builder" stage

The first stage will NOT be copied to the file docker image, only what what we copy from it to the last stage will be saved in the image.

** **Docker Security checks** ⇒ https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

.dockerignore file