Topic 3: Containers
part 1

Dr. Daniel Yellin

Overview

- Docker Overview
- Docker images
- Docker containers
- Union File System
- Docker architecture
- Docker summary

Docker overview

What problem does it solve?

Images, Containers, Docker files

What problems does Docker solve?

1. Packaging all the software dependencies together with the application to avoid conflicts
   - You want to run multiple applications or multiple modules of a single application on a computer.
   - Each application uses *different versions* of the same package.
   - How do you avoid conflicts when building and running the application?

   Dependency management: package your application with all the required packages and dependencies into one container.

Multiple programs running on top of OS

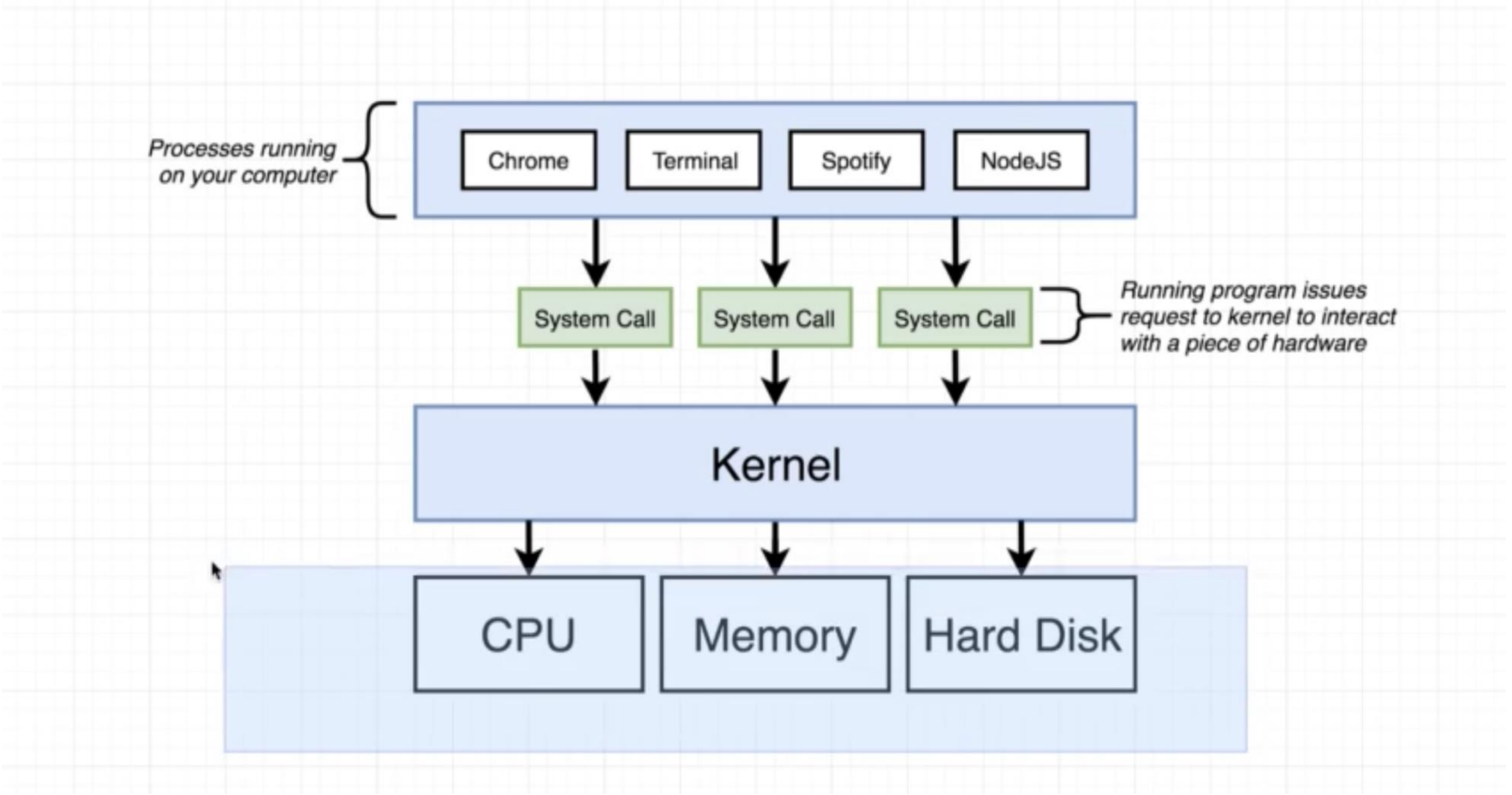
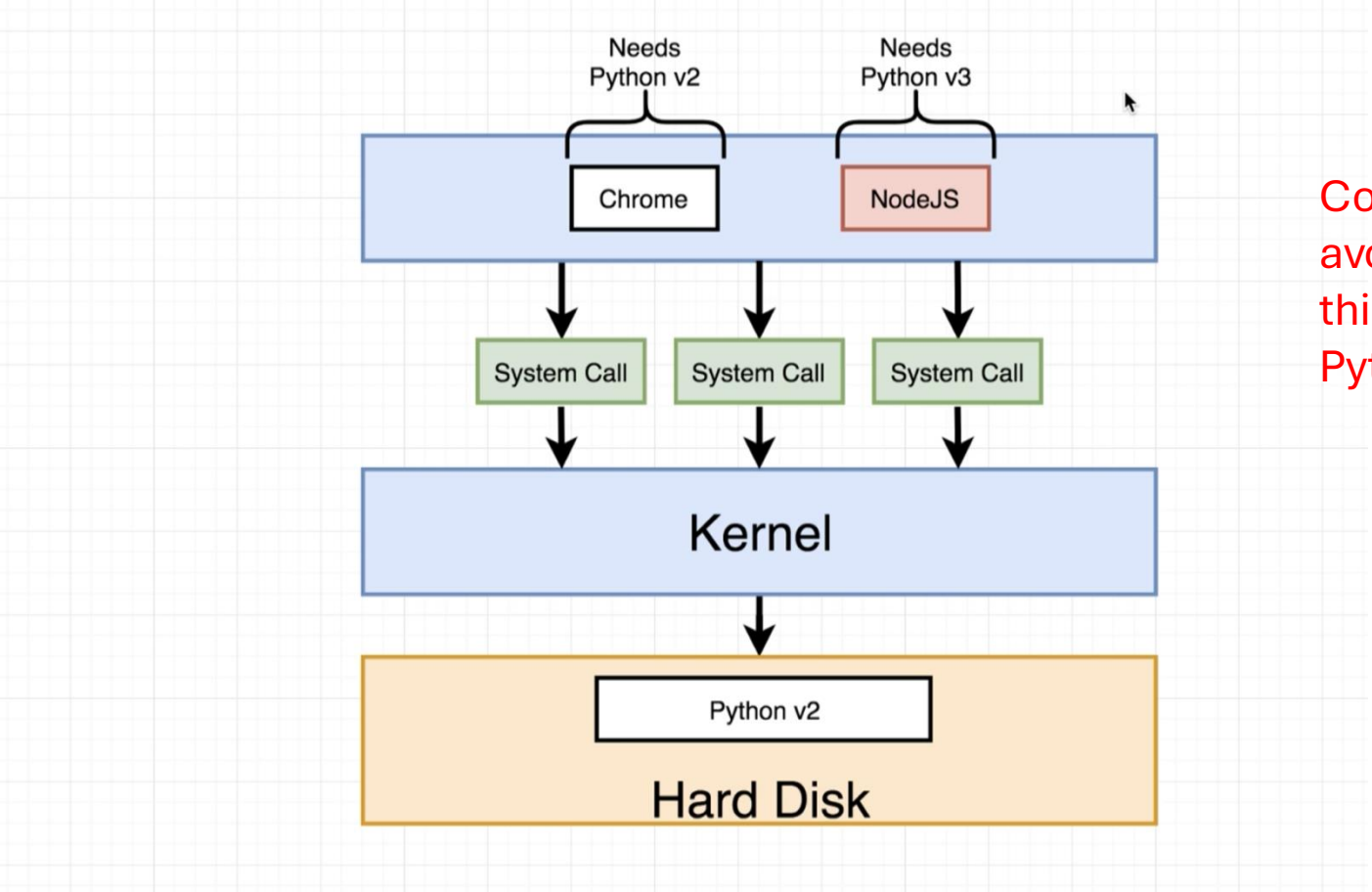


Diagram from: [Docker and Kubernetes: The Complete Guide]
Udemy Course by Stephen Grider

Multiple programs running on top of OS by default use the same versions of programs



Containers help to avoid conflicts like this, between Python v2 and v3.

Diagram from: Docker and Kubernetes: The Complete Guide
Udemy Course by Stephen Grider

A container partitions system resources so different containers are isolated from one another

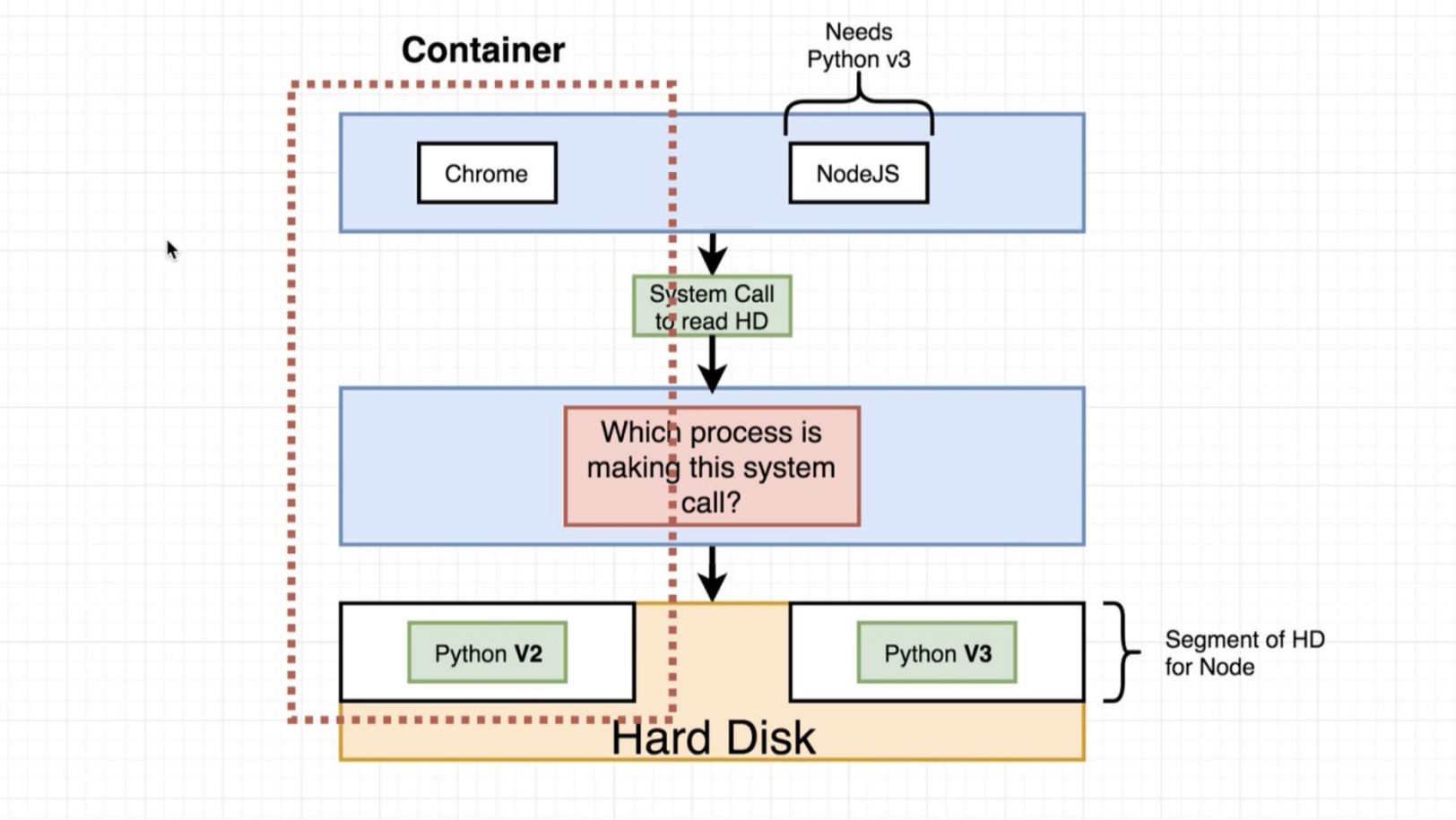


Diagram from: [Docker and Kubernetes: The Complete Guide]
Udemy Course by Stephen Grider

What problems does Docker solve?

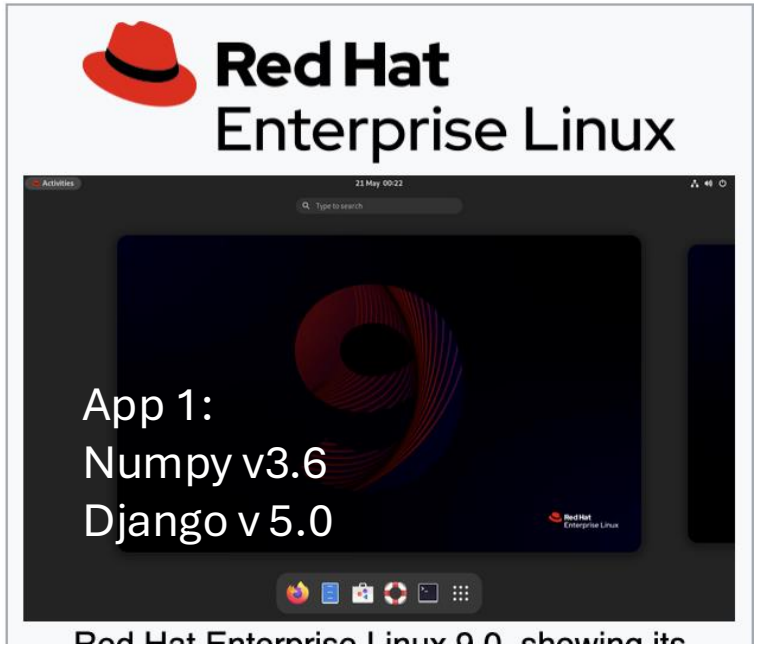
2. Portability

- You often want to run your application on your desktop, on your company's test environment, and on your company's production environment in the Cloud.
- They run on different hardware and different operating systems.
- How can you port code from one environment to another without having to make a separate version for each one?
- How can multiple apps built on different hardware with different operating systems run on the same server?

Heterogeneous applications can run on a single server with different hardware, different operating system (with same kernel), and different packages installed.

Built on RedHat
Run on Linux

Built on Ubunto
Run on Linux

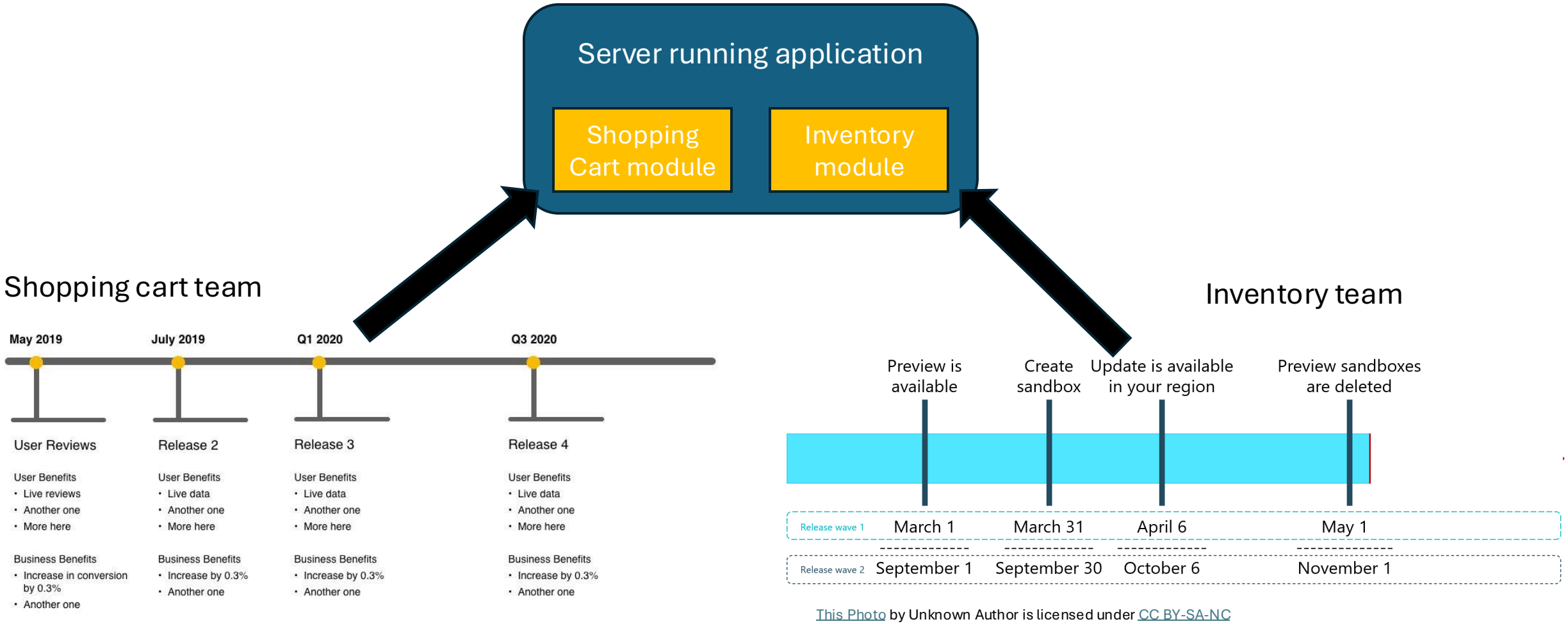


What problems does Docker solve?

3. Independent lifecylce management of application modules

- Whenever you fix a bug or add a feature to one module, you do not want to rebuild/test/deploy the entire application. You want to make the change to a small unit, then build, test and deploy just that unit.
- You want an independent lifecycle (build/test/deploy) for each "unit" of your application that still integrates with the entire application. This follows the *single responsibility* principle, which we will discuss more when we discuss microservices.

Independent lifecyle management



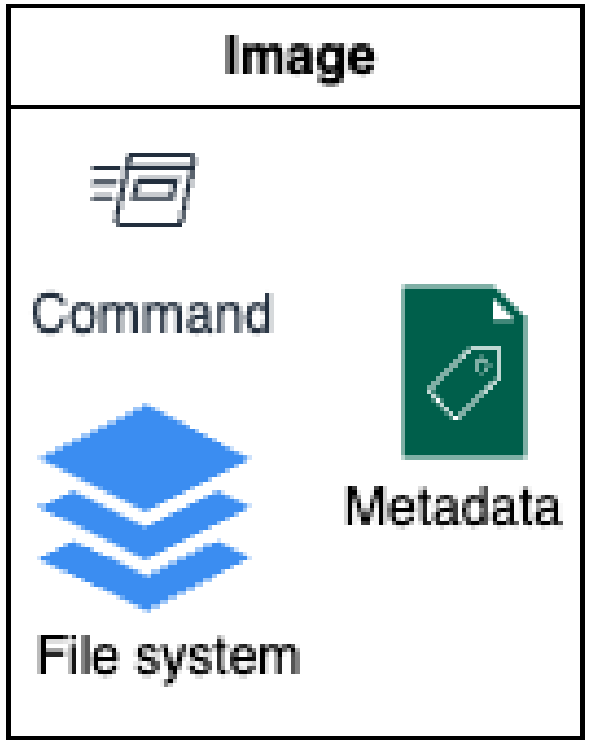
What problems does Docker solve?

4. Sharing and reusability
   - Reusing code from others can be complex: lengthy install process and error-ridden build process. GIT helps but does not solve the problem. It focusses on source code.
   - Containers allows others to take your code and update it, without having to tamper with your source code.
5. Scalability
   - You want to deploy your SaaS application. You do not have a lot of users yet, but your user-base will grow rapidly. You do not want to spend a ton of money on Cloud hardware or even VMs. Containers allow you to easily increase the number of instances handling requests as traffic to your application grows.

Images

Three fundamental concepts in Docker are *images*, *containers*, and *Dockerfiles*.

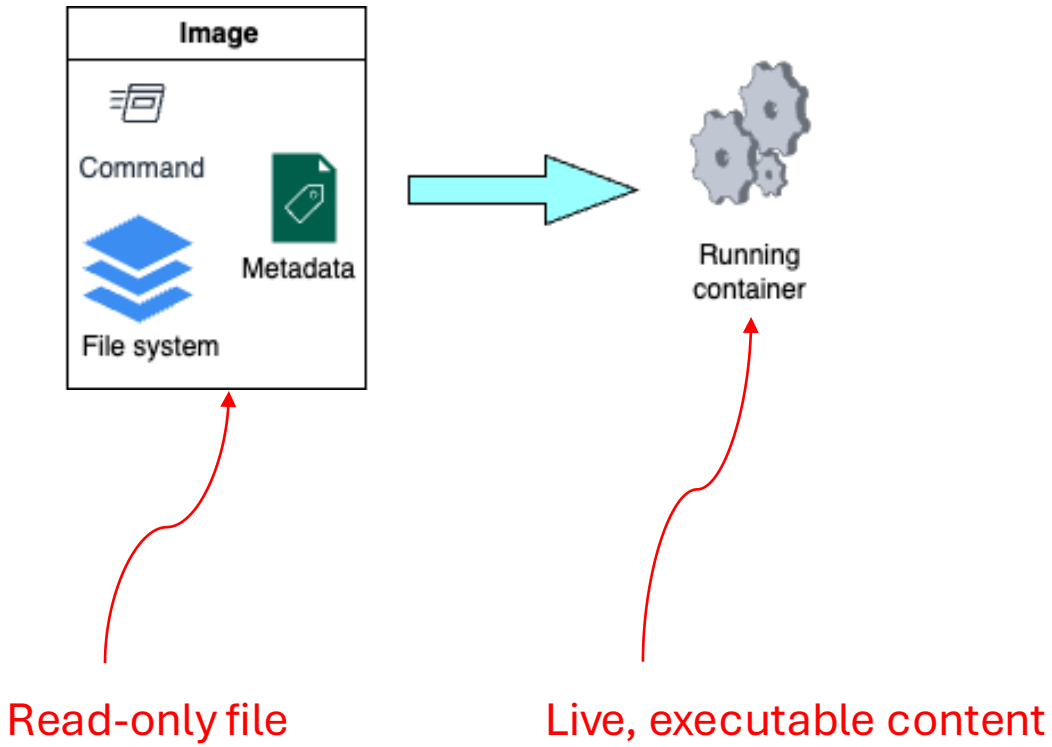
- An *image* is a collection of files, metadata, and a command. It contains executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container.



Images and containers

A *container* is created from an image.

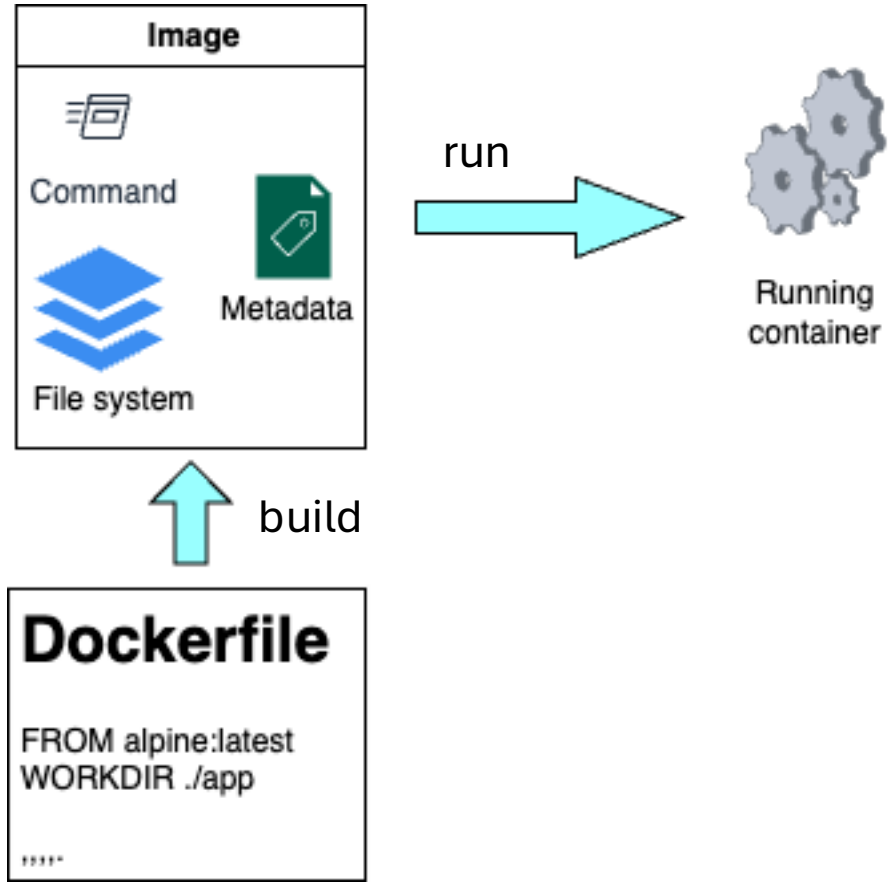
- It is an isolated running process (or processes) started using the command, with access only to the files in the image. The metadata defines properties of the process such as network ports accessible to the process.
- You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.



Dockerfiles

A *Dockerfile* automates the process of Docker image creation. It's a list of command-line interface (CLI) instructions that Docker Engine will run in order to assemble the image.

Often, an image is based upon another image. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create your desired image "on top of" the other image.



Docker images

Build, share, run

Docker layers

You can download docker from here: https://www.docker.com

4 docker commands

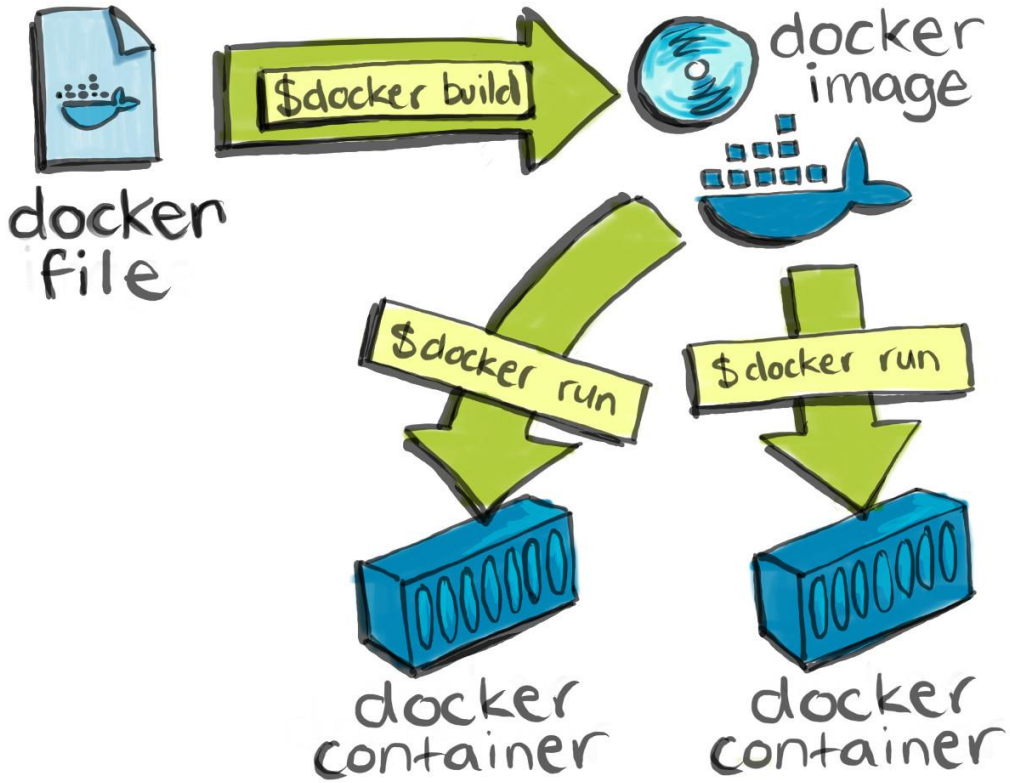
1. docker build --tag <img> <cntx>

2. docker run --name <cntr> <img>

3. docker images

4. docker ps

More Docker commands in the appendix



Source

A Dockerfile to containerize our toy.py app

```
FROM python:alpine3.17
WORKDIR ./app
COPY toys.py .
RUN pip install Flask
ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
#ENV NINJA_API_KEY - if required
EXPOSE 8001
CMD ["flask", "run", "--host=0.0.0.0"]
```

- You start with a *base image*. In this case, a “slim” python image (alpine).
- Every command creates a new *layer* on top of the previous layer.
- When you finish building the image using this Dockerfile, you have a new image.

Dockerfile: instructions on making an image

```
FROM python:alpine3.17

WORKDIR ./app

COPY toys.py .

RUN pip install flask

ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
ENV NINJA_API_KEY ...

EXPOSE 8001

CMD [”flask", ”run”, “--
host=0.0.0.0”]
```

FROM python:alpine3.17

Image to use as base. In this case, we chose the official Python image that has all the tools and packages we need to run Python (e.g., pip).

WORKDIR ./app

Creates and sets the working directory inside the container to be ./app. All further commands are executed in this directory.

COPY toys.py .

Copies python file from the host into the working directory of the container (in this case, ./app)

RUN pip install flask. RUN is executed at build time.

Runs these cmds to install these packages needed by our application into the container.

Dockerfile: instructions on making an image

```
FROM python:alpine3.17

WORKDIR ./app

COPY toys.py .

RUN pip install flask

ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
ENV NINJA_API_KEY ...

EXPOSE 8001

CMD ["flask", "run", "--host=0.0.0.0"]
```

ENV FLASK_APP=toys.py

ENV FLASK_RUN_PORT=8001

ENV NINJA_API_KEY ...

These commands set environment variables inside the Docker container.

EXPOSE 8001

Port 8001 is the port the container will listen on.

This is documentation as the actual port for listening is expressed by the "FLASK_RUN_PORT" variable and/or is expressed when starting the container.

CMD ["flask", "run", "--host=0.0.0.0"]

Executes the command "flask run --host=0.0.0.0" when the container startups. The parameters to the command are given, with each one parenthesis and in separated by a comma.

Build image

Remember to always add the “build context”. “.” means current directory. That’s where the resources for building an image (referenced by the Dockerfile) are located, unless specified otherwise explicitly in the Dockerfile.

```
docker build --tag toys .
```

This command tells docker to build a new image and give it the name (tag) “toys”.

You have to tell docker where to find the Dockerfile. The “.” at the end of the cmd tells docker to use the Dockerfile in the current directory.

```
danielyellin@Daniels-Air app % docker build --tag toys .
[+] Building 5.1s (9/9) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 280B
 => [internal] load metadata for docker.io/library/python:alpine3.17
 => [1/4] FROM docker.io/library/python:alpine3.17
 => [internal] load build context
 => => transferring context: 6.66kB
 => CACHED [2/4] WORKDIR ./app
 => [3/4] COPY toys.py .
 => [4/4] RUN pip install Flask
 => exporting to image
 => => exporting layers
 => => writing image sha256:056d6e5499b4be68bd3bee9afbd3e145e7391c107d
 => => naming to docker.io/library/toys

What's Next?
  1. Sign in to your Docker account → docker login
  2. View a summary of image vulnerabilities and recommendations → doc
```

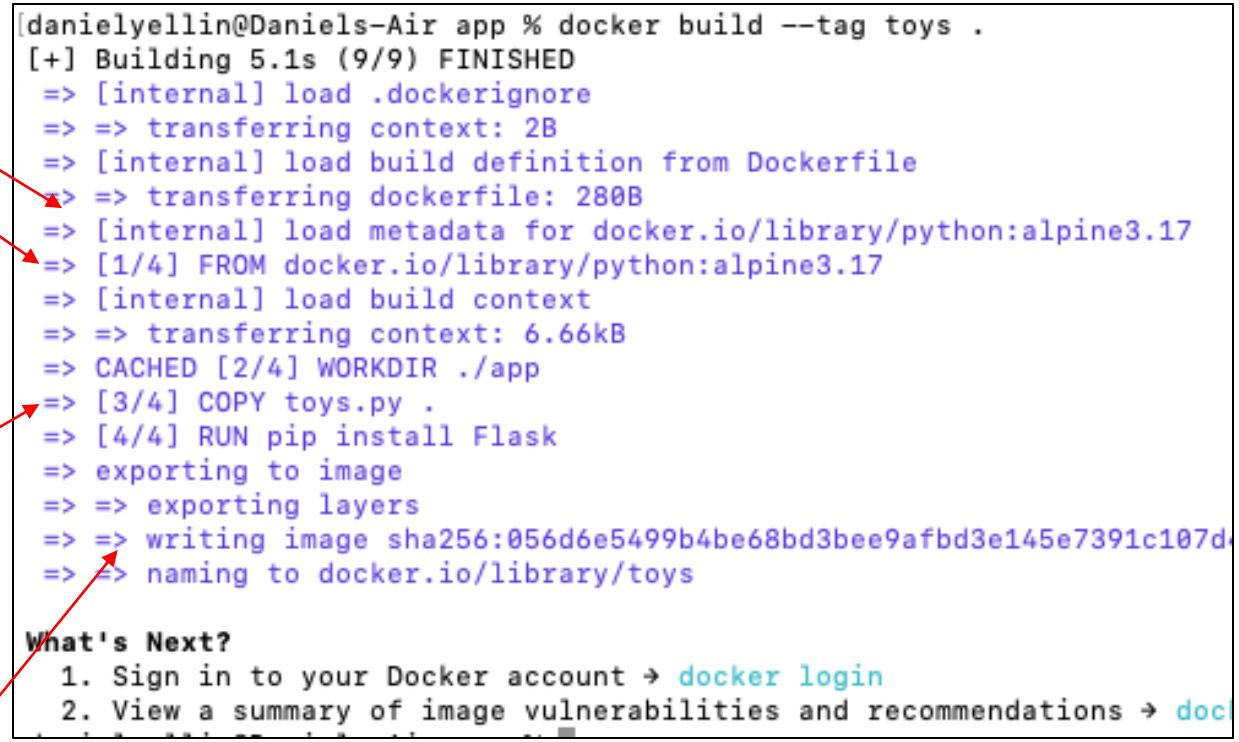
Explaining the `docker build` cmd messages

Docker uploads files to the Docker Builder.

The FROM cmd loads the python:alpine3.17 to the image file system. Each layer in the image is uploaded.

The rest of the Dockerfile commands are run. The commands are numbered. Each command creates a new layer in the image. If there are cached images (e.g., from a pull cmd, or a previous build), then they will be loaded and not rebuilt from scratch.

The final image is created with a unique ID, and then given the name:tag specified in the cmd.



Show images

`docker images`

This command tells docker to list all the images it is storing on the host. You can see listed the newly created docker image toys.

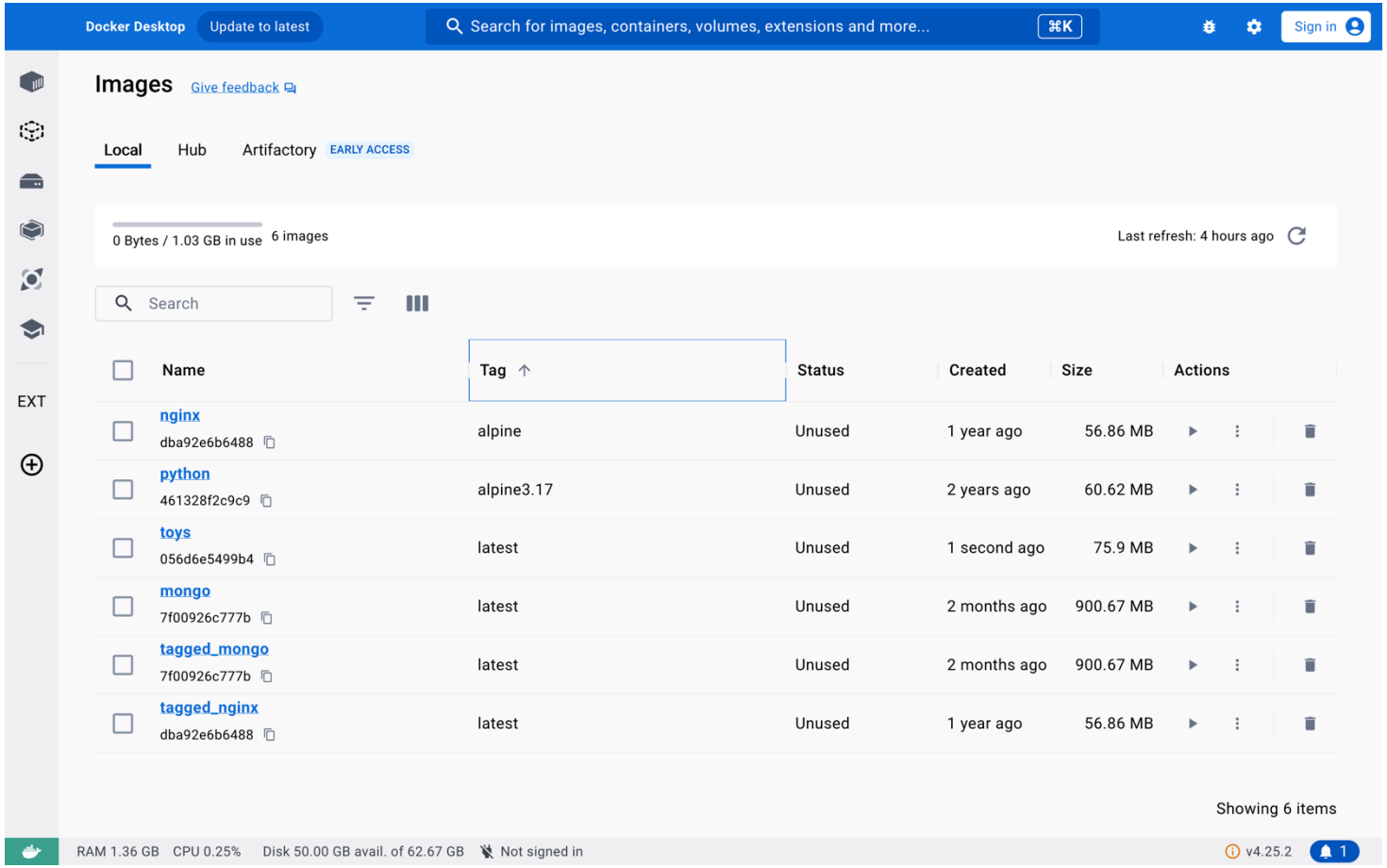
```
danielyellin@Daniels-Air app % docker images
REPOSITORY     TAG           IMAGE ID       CREATED          SIZE
toys           latest        056d6e5499b4   5 minutes ago    75.9MB
tagged_mongo   latest        7f00926c777b   6 weeks ago      901MB
mongo          latest        7f00926c777b   6 weeks ago      901MB
nginx          alpine        dba92e6b6488   11 months ago    56.9MB
tagged_nginx   latest        dba92e6b6488   11 months ago    56.9MB
python         alpine3.17    461328f2c9c9   2 years ago      60.6MB
```

Docker images

Docker images residing in my local repository. You can view your local images either using the cmd:

```
docker images
```

or by opening Docker Desktop→ Go to the Dashboard.



Dockerfiles and layers

Each instruction that modifies the filesystem in a Dockerfile creates a *layer* in the image.

Each layer encodes the changes to the filesystem by executing that instruction.

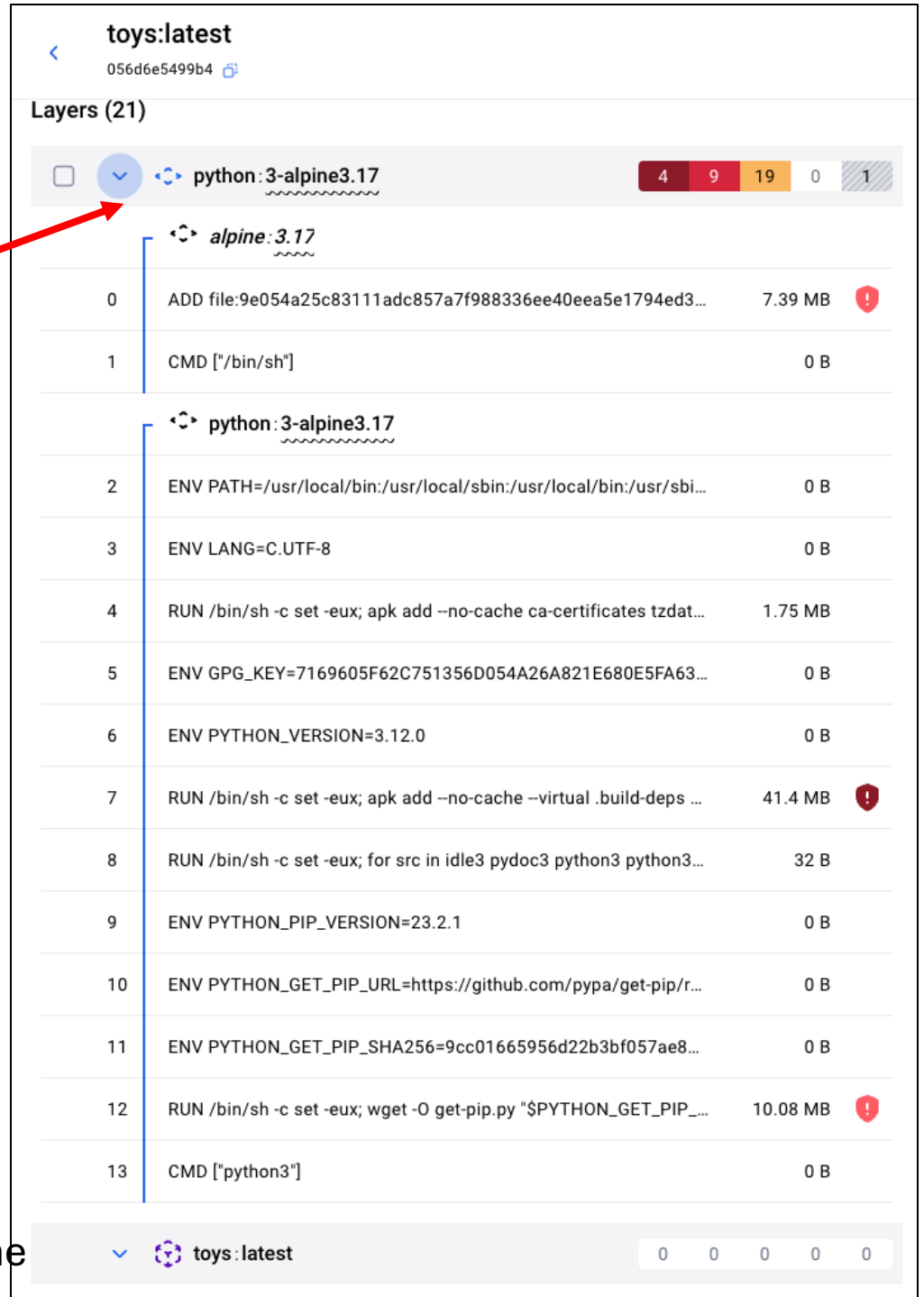
Other instructions just modify the image's metadata.

This view is available on Docker Desktop

There are 21 layers in our toys image. Some of them from the python:alpine image and some from commands in our Dockerfile.

Python:alpine layers

```
FROM python:alpine3.17
WORKDIR ./app
COPY toys.py .
RUN pip install Flask
ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
#ENV NINJA_API_KEY - if required
EXPOSE 8001
CMD ["flask", "run", "--host=0.0.0.0"]
```



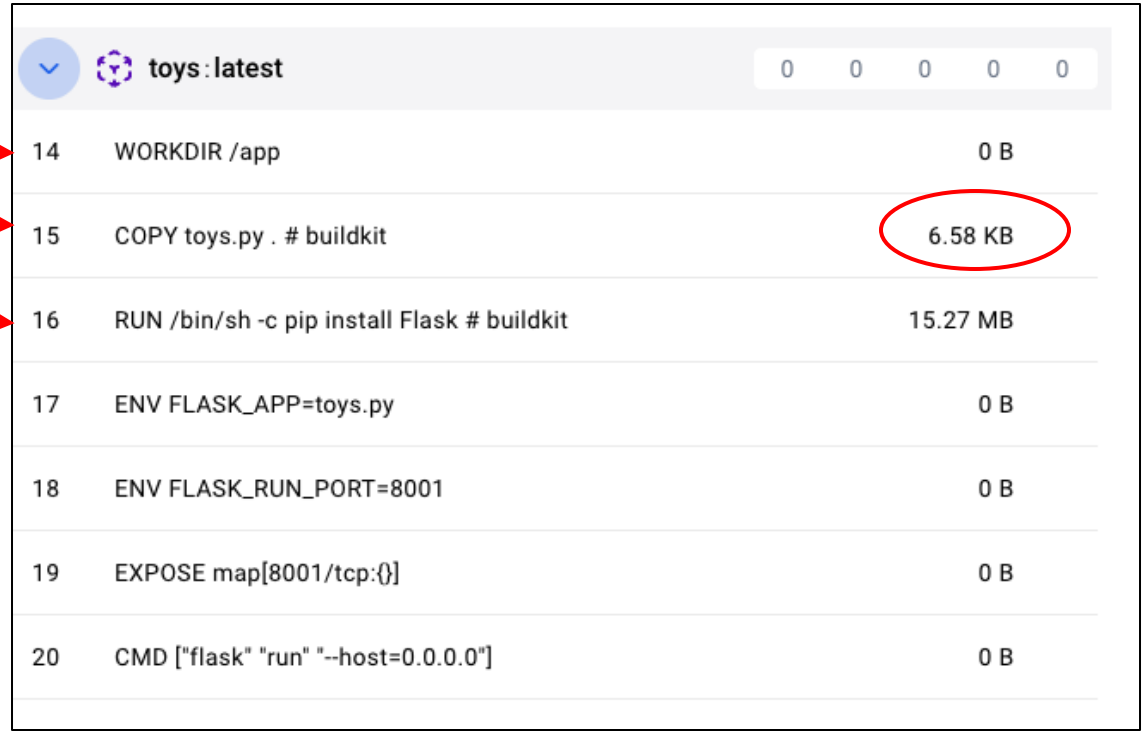
Layers from python:alpine

toys layers

Note that each layer has an associated size.

```
FROM python:alpine3.17
WORKDIR ./app
COPY toys.py .
RUN pip install Flask
ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
#ENV NINJA_API_KEY - if required
EXPOSE 8001
CMD ["flask", "run", "--host=0.0.0.0"]
```

...



Layers from toys dockerfile

Looking inside an image

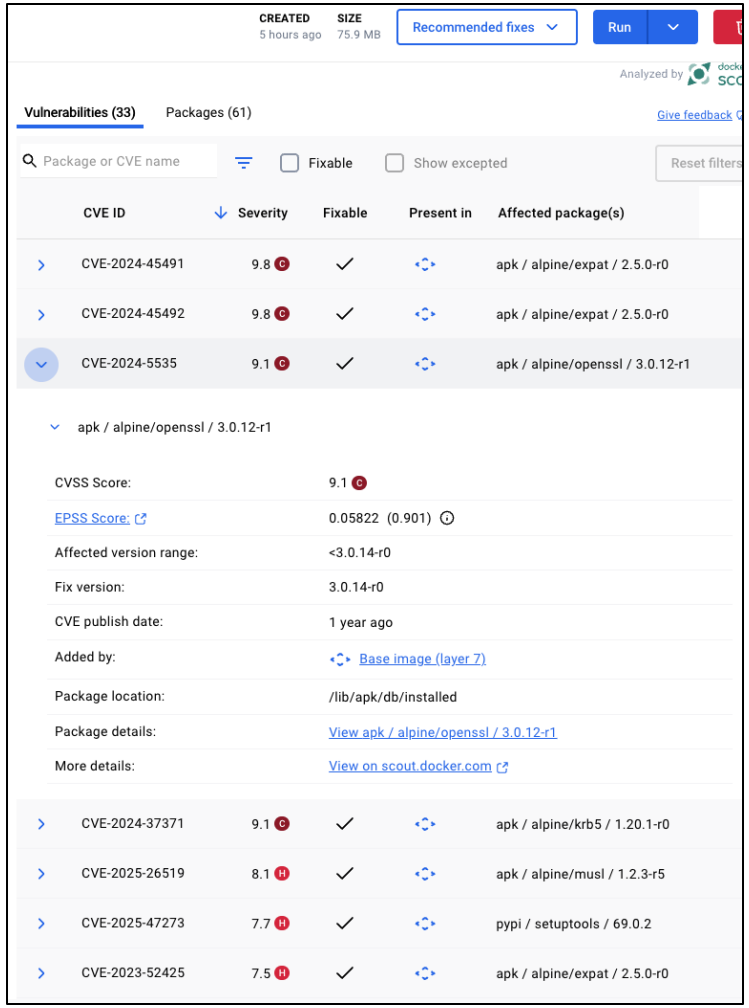
```
danielyellin@Daniels-Air app % docker sbom toys
Syft v0.43.0
 ✔ Loaded image
 ✔ Parsed image
 ✔ Cataloged packages      [47 packages]

NAME                    VERSION           TYPE
.python-rundeps         20231129.033204   apk
Flask                   3.1.2             python
Jinja2                  3.1.6             python
MarkupSafe              3.0.3             python
Werkzeug                3.1.3             python
alpine-baselayout       3.4.0-r0          apk
alpine-baselayout-data  3.4.0-r0          apk
alpine-keys             2.4-r1            apk
apk-tools               2.12.10-r1        apk
blinker                 1.9.0             python
busybox                 1.35.0-r29        apk
busybox-binsh           1.35.0-r29        apk
ca-certificates         20230506-r0       apk
ca-certificates-bundle  20230506-r0       apk
click                   8.3.0             python
gdbm                    1.23-r0           apk
itsdangerous            2.2.0             python
keyutils-libs           1.6.3-r1          apk
krb5-conf               1.0-r2            apk
krb5-libs               1.20.1-r0         apk
libbz2                  1.0.8-r4          apk
libc-utils              0.7.2-r3          apk
libcom_err              1.46.6-r0         apk
libcrypto3              3.0.12-r1         apk
libexpat                2.5.0-r0          apk
libffi                  3.4.4-r0          apk
libintl                 0.21.1-r1         apk
libnsl                  2.0.0-r0          apk
libssl3                 3.0.12-r1         apk
libtirpc                1.3.3-r0          apk
libtirpc-conf           1.3.3-r0          apk
libuuid                 2.38.1-r1         apk
libverto                0.3.2-r1          apk
musl                    1.2.3-r5          apk
musl-utils              1.2.3-r5          apk
ncurses-libs            6.3_p20221119-r1  apk
ncurses-terminfo-base   6.3_p20221119-r1  apk
pip                     23.2.1            python
readline                8.2.0-r0          apk
scanelf                 1.3.5-r1          apk
setuptools              69.0.2            python
sqlite-libs             3.40.1-r0         apk
ssl_client              1.35.0-r29        apk
tzdata                  2023c-r0          apk
wheel                   0.42.0            python
xz-libs                 5.2.9-r0          apk
zlib                    1.2.13-r0         apk
```

`docker sbom toys` Gets the *software bill of materials* for this image. These are all the files making up this image.

In the Docker Desktop you can get a breakdown on what packages are included *per layer* of your image.

The Docker Desktop also shows vulnerabilities in each layer, giving the Common Vulnerabilities & Exposures (CVE) score for each vulnerability.

Docker Desktop, CVE vulnerabilities



A **CVSS** (Common Vulnerability Scoring System) score is a numerical value from 0 to 10 that represents the severity of a software vulnerability. It is calculated using a framework that considers a vulnerability's exploitability and impact. It is used to prioritize remediation efforts.

EPSS estimates the probability of observing any exploitation attempts against a vulnerability in the next 30 days.

Docker containers

Image to container, running a container, Port publishing

You can download docker from here: https://www.docker.com

4 docker commands

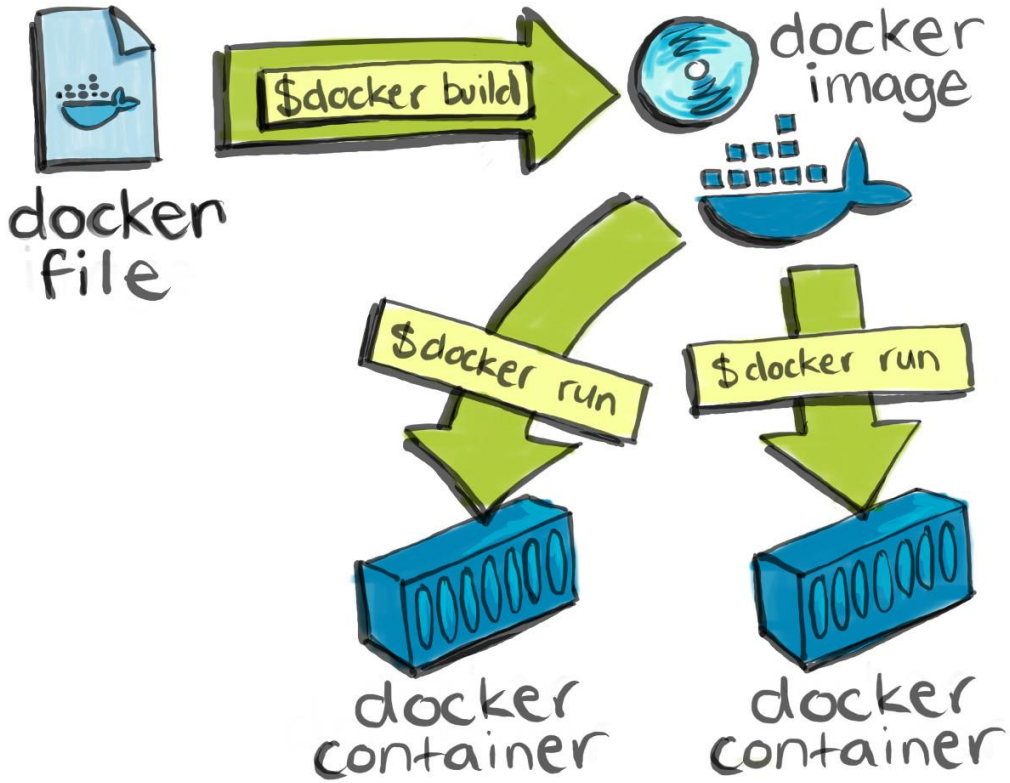
1. docker build --tag <img> <cntx>

2. docker images

3. docker run --name <cntr> <img>

4. docker ps

More Docker commands in the appendix



Source

Create and run container

The second port is the container port

```
docker run <image-name>
```

The docker `run command` = `docker create` + `docker start` cmds. It tells docker to create a container from the given image and start the container execution (using its init command).

```
docker run --publish 80:8001 --name toys-cont-v1 toys
```

This cmd creates and starts a docker container based upon the image “toys”. It gives the container the name “toys-contain-v1”. It maps the container port 8001 to the host port 80.

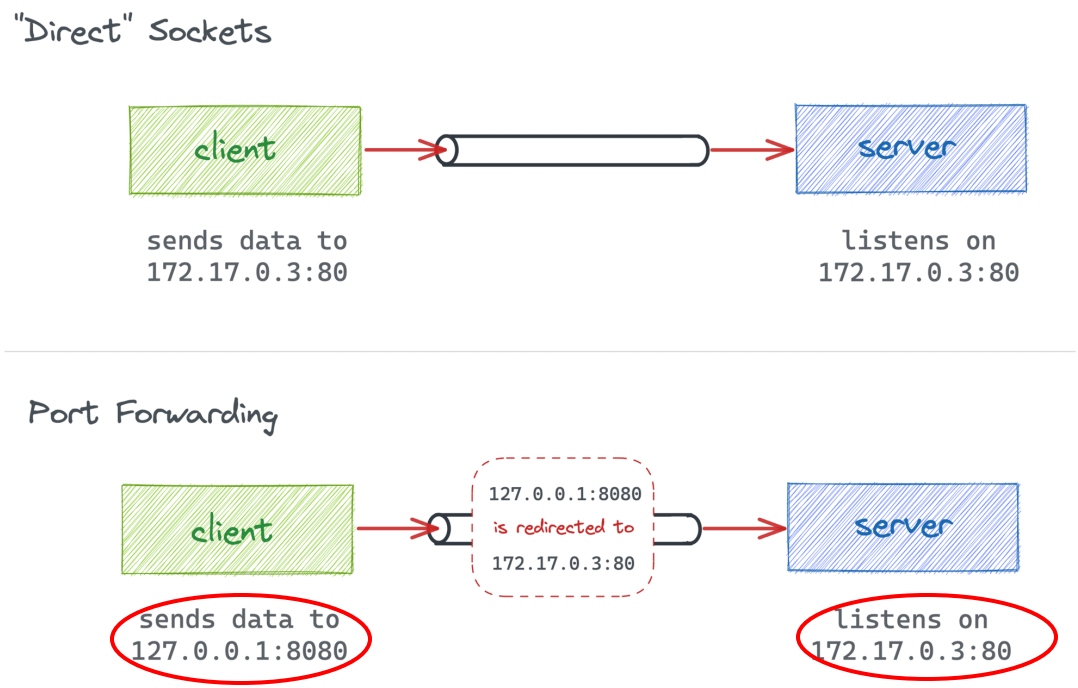
The first port is the host port

```
danielyellin@Daniels-Air app % docker run --publish 80:8001 --name toys-cont-v1 toys
 * Serving Flask app 'toys.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8001
 * Running on http://172.17.0.2:8001
```

Ports and port forwarding

What are ports?

- A **port** is a number assigned to uniquely identify a connection endpoint and to direct data to a specific service. At the software level, within an operating system, a port is a logical construct that identifies a specific process or a type of network service.
- **Port forwarding** or **port mapping** is an application of network address translation (NAT) that redirects a communication request from one address and port number combination to another.



Source

Wikipedia on port

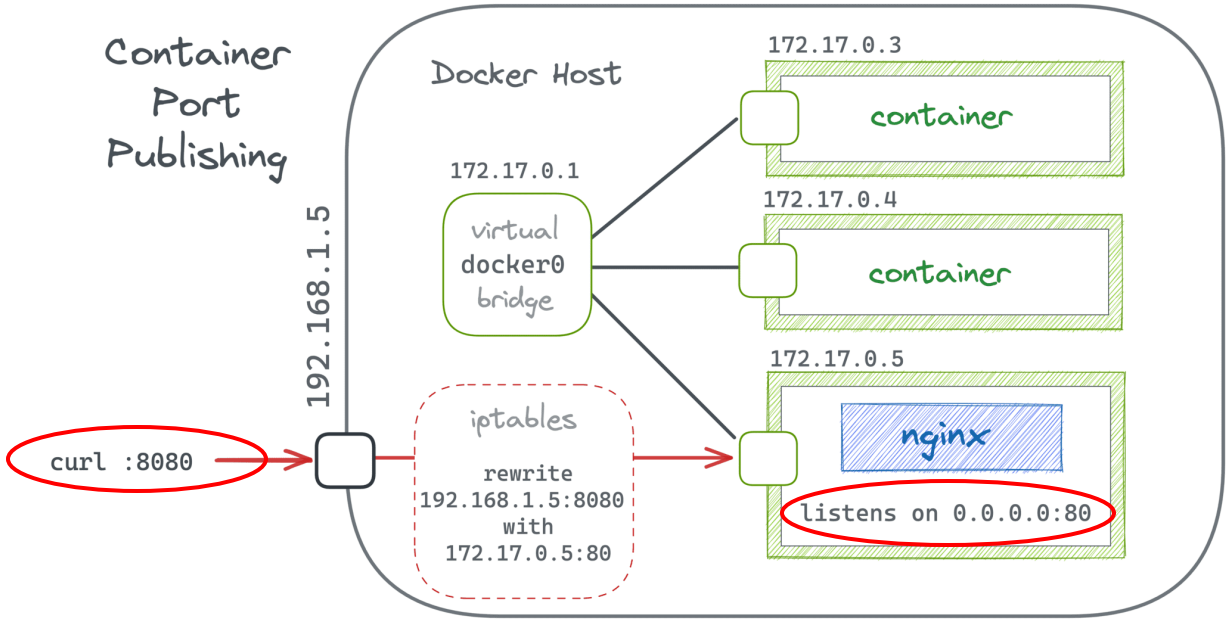
Wikipedia on port forwarding

Docker containers have their own IP addresses and ports

- Each container has its own internal IP address.
  - Normally this is not visible to you and it may change if the container is restarted.
- Each container may listen on specific ports.
  - For instance, our example toys app in the container will listen on port 8001 of the container for HTTP requests (API calls).
- What IP address and port does a client use if the client wishes to talk to the container ?
  - The client sends messages to the IP address of the host running the container
  - Docker provides a mechanism to specify which port on the host to use in order to reach the appropriate container port. This is called *port publishing*.

Port publishing

- Port publishing
  - To make a port available to services outside of Docker, or to Docker containers running on a different network, use the **--publish** or **-p flag**
- Example:
  - **`docker run --publish 8080:80 nginx`**
  - The client would send to the host IP at **port 8080**.
  - Docker uses port forwarding to send the message to the container running nginx listening on container **port 80**.

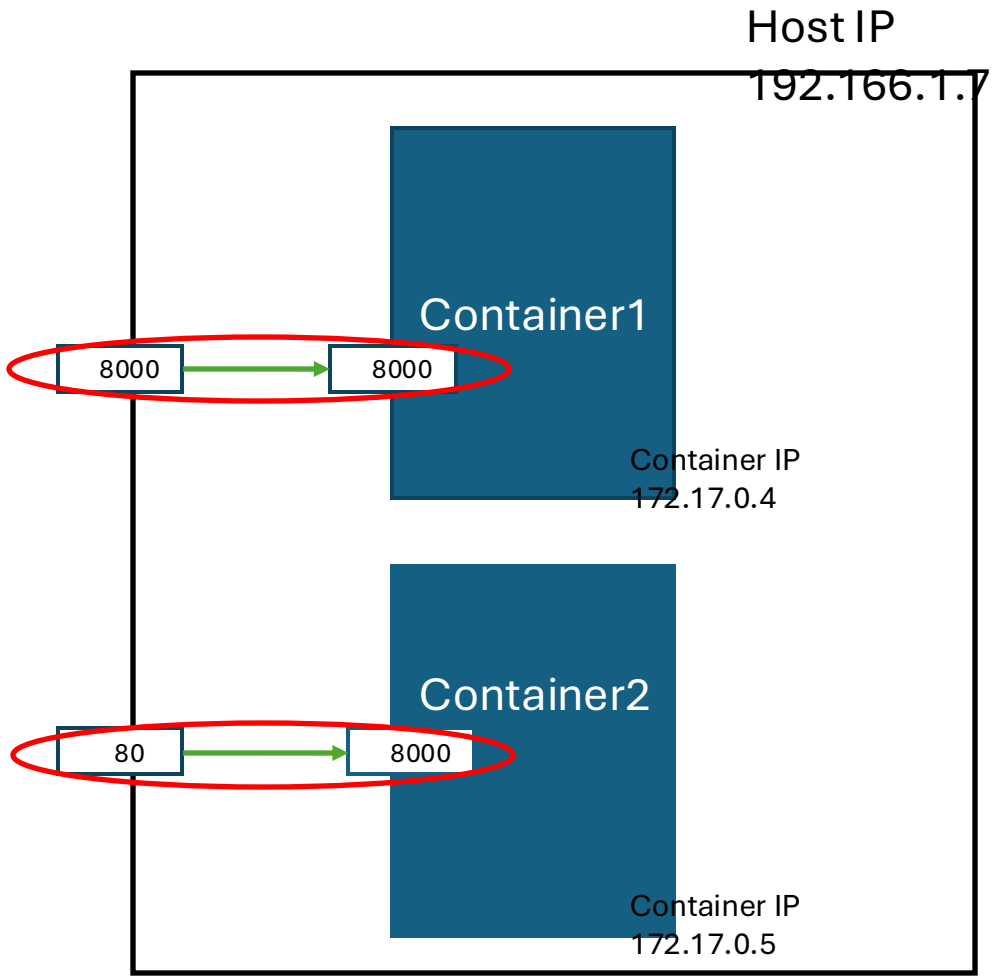


Source

Port mapping in Docker

```
docker run --p8000:8000 --name
container1 \ <img1>
```

```
docker run -p80:8000 --name
container2 \
                <img2>
```



List containers

`docker ps`

This cmd lists all running containers. We can see that the “toys-contain-v1” is up an running. The host is listening on port 80 and forwards request to the container port 8001. The docker container is based upon the image “toysimage:v1”.

```
danielyellin@Daniels-Air ~ % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                    NAMES
e5e3cdbc75e9   toys      "flask run --host=0.…"   6 minutes ago   Up 6 minutes   0.0.0.0:80->8001/tcp, [::]:80->8001/tcp   toys-cont-v1
```

`docker ps -a`

This cmd lists all containers, even those not running currently.

Note that the host (0.0.0.0) port of 80 will forward requests to the container port 8001 as we specified in the Dockerfile.

Testing the toys application running in a container

Our toys application is up and running in a container. REST requests to server port 80 get forwarded to container port 8001.

We can test our application by sending REST requests to port 80. We can do this by:

- Writing a program to issue REST requests to the server,
- Use *postman* to issue requests, or
- Use the *curl* program, which allows you to issue HTTP requests from your terminal. Curl stands for "Client URL". We issue 3 curl cmds on the next slide:
  - The first two POST words to the toys server
  - The last one GETs the collection of toys posted so far

If you are not familiar with curl, see here

Curl requests to toys server

```
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:80/toys' --header 'Content-Type: application/json' \
--data '{
    "name": "puzzle",
    "descr": "30 piece puzzle",
    "age": 4,
    "price": 25.00
}'
{"age":4,"descr":"30 piece puzzle","features":[],"id":"1","name":"puzzle","price":25.0}
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:80/toys' --header 'Content-Type: application/json' \
--data '{
    "name": "blocks",
    "descr": "12 building blocks",
    "age": 3,
    "price": 18.00
}'
{"age":3,"descr":"12 building blocks","features":[],"id":"2","name":"blocks","price":18.0}
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:80/toys'
[{"age":4,"descr":"30 piece puzzle","features":[],"id":"1","name":"puzzle","price":25.0},{"age":3,"descr":"12 building blocks","featur
```

What does “docker run” really do?

- It *finds* the image. If it does not exist in the local registry, it will search the public registry (or other registries you specify) DockerHub and pull it down into your local registry.
- It creates a *new container* from the image.
- It allocates a *file system* to the container. It mounts a read-write layer.
- Attaches stdout, stderr to the terminal
- It creates a *network interface* that allows the container to talk to the host.
- Sets up an *IP address* for the container.
- Invokes the *start cmd* that you specified for the image.

Source

Common container lifecyle commands

- docker remove <cnt>

Removes the container. All container resources (including container file system) are released.

- docker stop -<cntr>

Stops the running container(s). The main process inside the container will receive SIGTERM, and after a grace period, SIGKILL. The container's memory (not file system) will be released.

- docker start <cntr>

Starts a stopped container(s).

- docker kill <cnt>

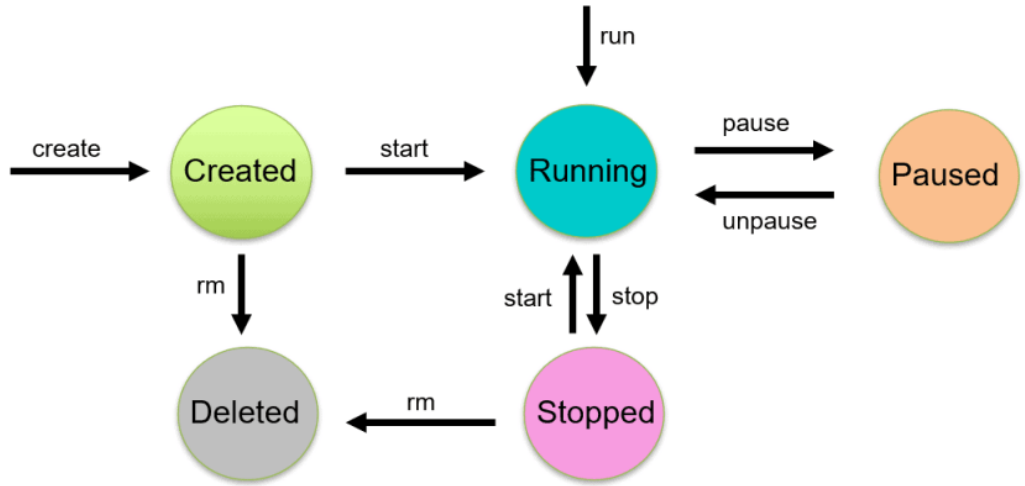
Kills the container(s). The main process inside the container is sent SIGKILL signal.

- docker pause <cnt>

Suspends all processes in the specified container(s) by sending the SIGSTOP to the main process. The container's resources are not released, except for the CPU.

- docker unpause <cnt>

Un-suspends all processes in the specified container(s).



Source

For complete list of docker container commands and explanation, see Docker command line documentation.

Question

How would you run two separate instances of our `toys image` on your host?

You must be able to communicate with each one separately!
Hence the host must listen for each one on a *separate port*.

Running two separate toys service instances

1. Issue a command to run the image toysimage:v1 in a container named toys-cont-v1 (just like we did before)

```
docker run --publish 80:8001 --name toys-cont-v1 toys
```

2. Issue another command to run the image toysimage:v1 in a container named toys-cont-v1-n2

```
docker run --publish 85:8001 --name toys-cont-v1-n2 toys
```

While they are both are listening on the container port 8001, requests to the first container must be addressed to host port 80, and the second one to host port 85.

Running two separate toys service instances (cont)

3. `docker ps`

You see both containers running and mapped to different host ports.

```
danielyellin@Daniels-Air ~ % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                                   NAMES
f84092aae902   toys      "flask run --host=0.…"   2 minutes ago    Up 2 minutes    0.0.0.0:85->8001/tcp, [::]:85->8001/tcp   toys-cont-v1-n2
e5e3cdbc75e9   toys      "flask run --host=0.…"   4 hours ago      Up 4 hours      0.0.0.0:80->8001/tcp, [::]:80->8001/tcp   toys-cont-v1
danielyellin@Daniels-Air ~ %
```

4. Run some requests on each container and see that we get the right results

(Next slide)

Issues requests (curl) to the two containers

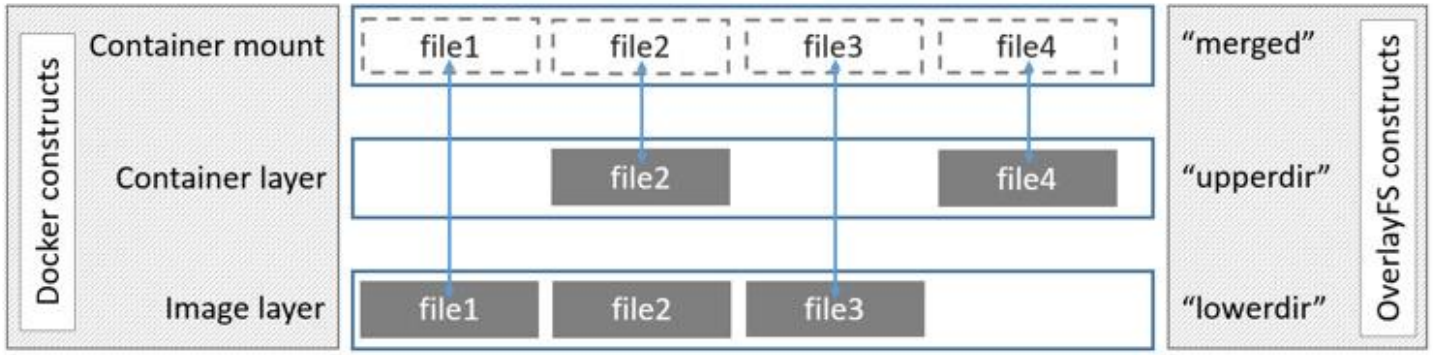
```
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:85/toys' --header 'Content-Type: application/json'
--data '{
    "name": "blocks",
    "descr": "12 building blocks",
    "age": 3,
    "price": 18.00
}'
{"age":3,"descr":"12 building blocks","features":[],"id":"1","name":"blocks","price":18.0}
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:80/toys' --header 'Content-Type: application/json'
--data '{
    "name": "puzzle",
    "descr": "30 piece puzzle",
    "age": 4,
    "price": 25.00
}'
{"age":4,"descr":"30 piece puzzle","features":[],"id":"1","name":"puzzle","price":25.0}
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:85/toys'
[{"age":3,"descr":"12 building blocks","features":[],"id":"1","name":"blocks","price":18.0}]
danielyellin@Daniels-Air ~ % curl --location 'http://0.0.0.0:80/toys'
[{"age":4,"descr":"30 piece puzzle","features":[],"id":"1","name":"puzzle","price":25.0}]
```

Union File System

How containers share and modify images

Union file system

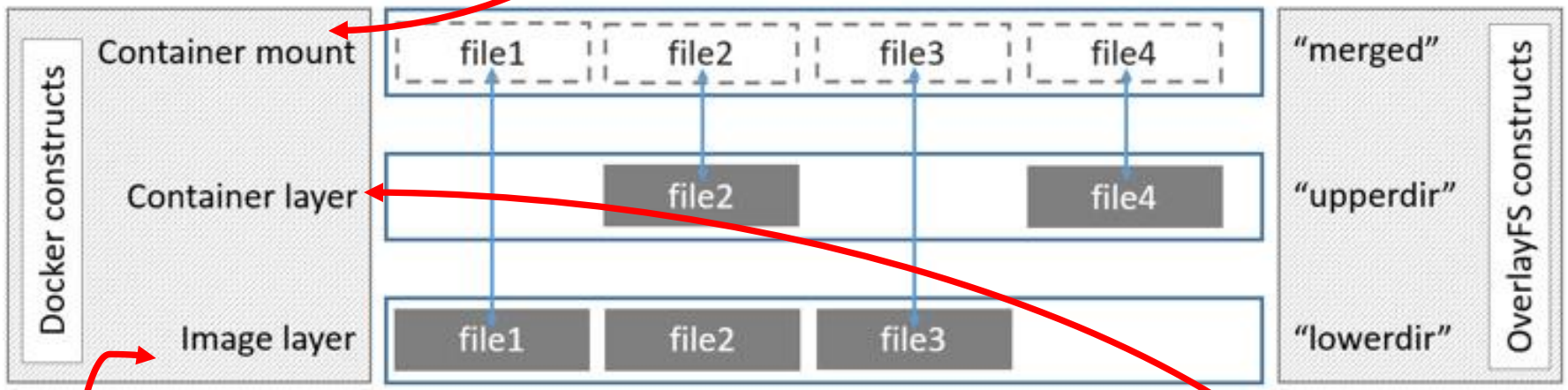
- Union file systems:
  - The Union File System (in the Linux kernel) allows contents from one file system to be merged with the contents of another, while keeping the "physical" content separate. Different “branches” can thereby share files.
  - When you have multiple layers in an image, the Union File System combines these into a single image.
  - UnionFS, AUFS, OverlayFS are implementations of a Linux Union Filesystem.
  - Docker uses **OverlayFS** by default.



Source

Union file system

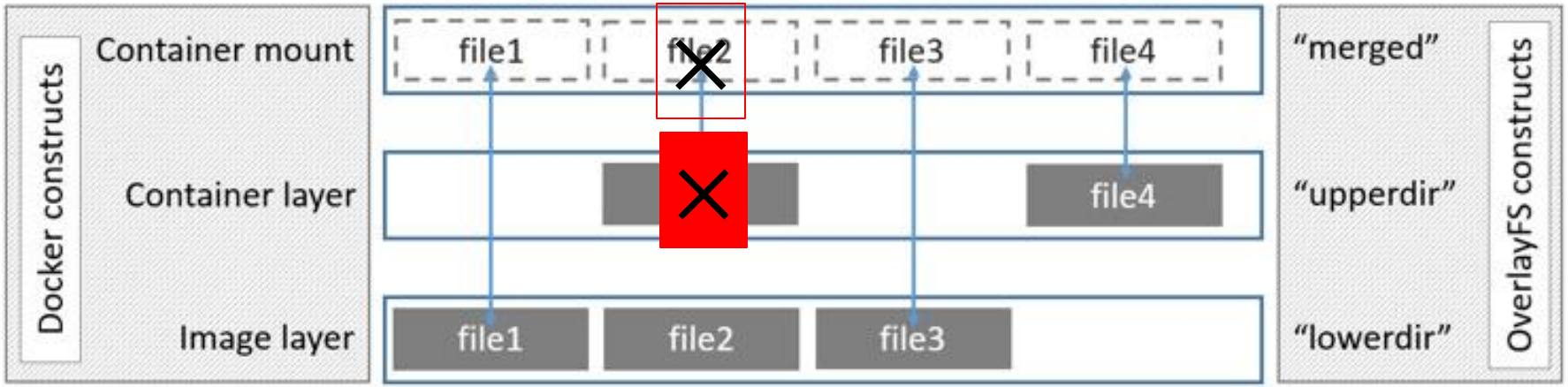
The layer reflects the merged image layer and container layer. It contains those files visible to the container during execution.



Think of this as the filesystem of the image after you build it from the Dockerfile. It is **read only**.

The **writeable** container layer contains any new files added,modified ,or deleted when you run your image in a container.

Union file system



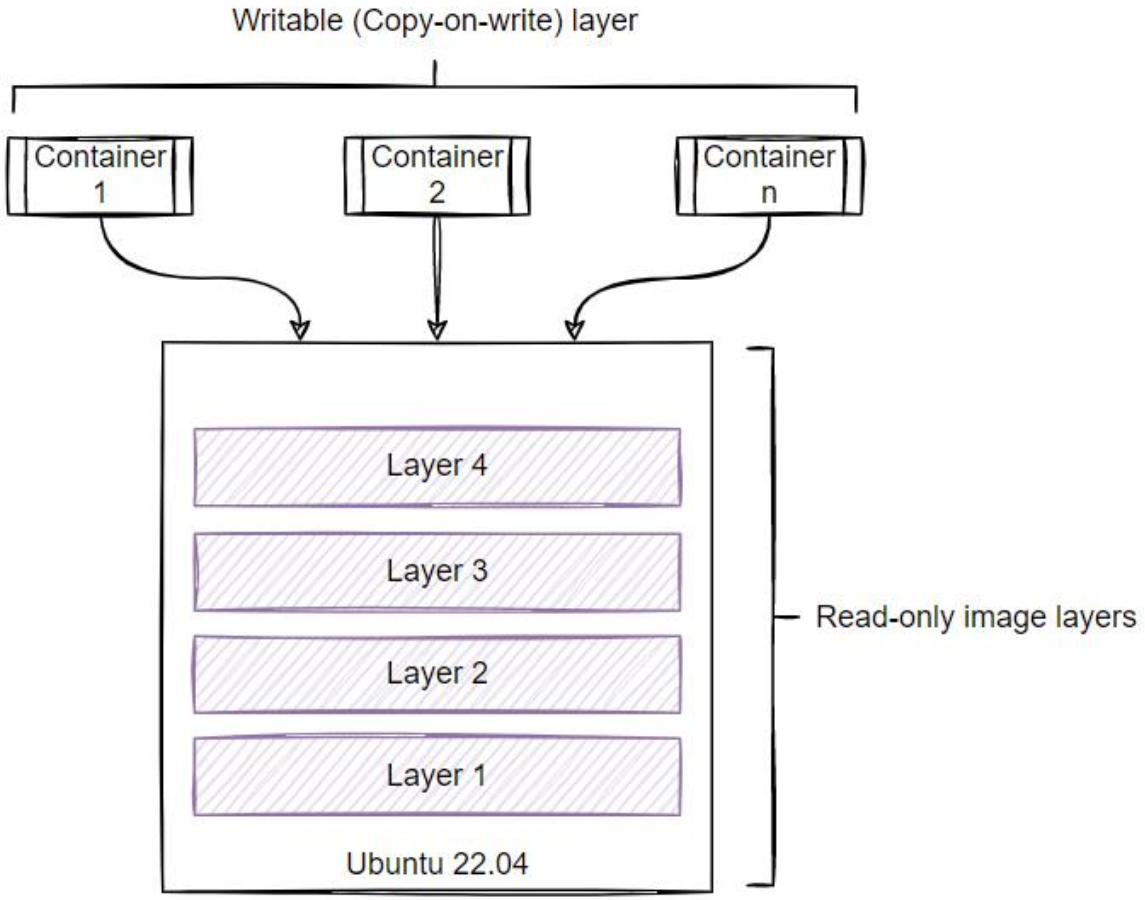
When a *file* is deleted within a container, a *whiteout* file is created in the container (upperdir). The version of the file in the image layer (lowerdir) is not deleted (because the lowerdir is read-only). However, the whiteout file prevents it from being available to the container.

For more information on how Docker uses OverlayFS, see here.

Copy-on-write

Containers share their filesystems. But when a container modifies a file, then it gets its own modified copy file. This is written in the writeable container layer. Hence that container will see the changes but other containers pointing to the same image will see the original file.

This enables a lot of *sharing* between containers, and fast container start-up time.



Source

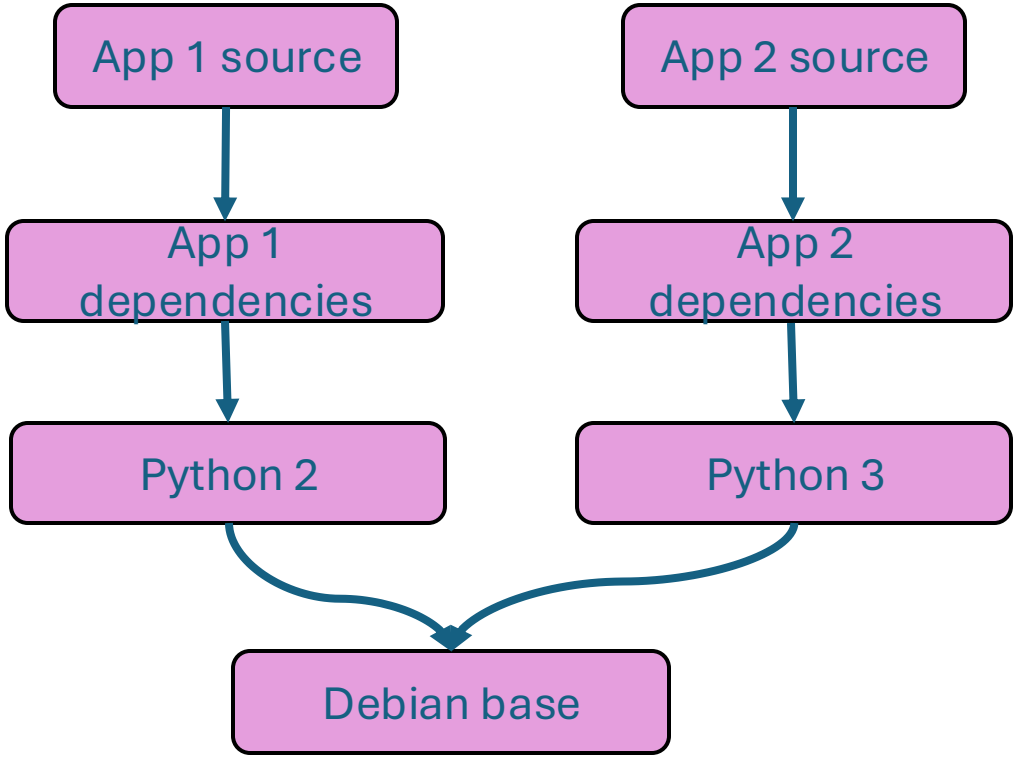
Container file system is not persistent

- The container file system exists only as long as the container exists.
  - For persistent storage, as we shall see, use Docker volumes
- If the container is killed or removed, any changes it made to its file system is lost.

Union file system helps avoid dependency conflicts

Hence each container has its own filesystem, and this avoids dependency conflicts between containers.

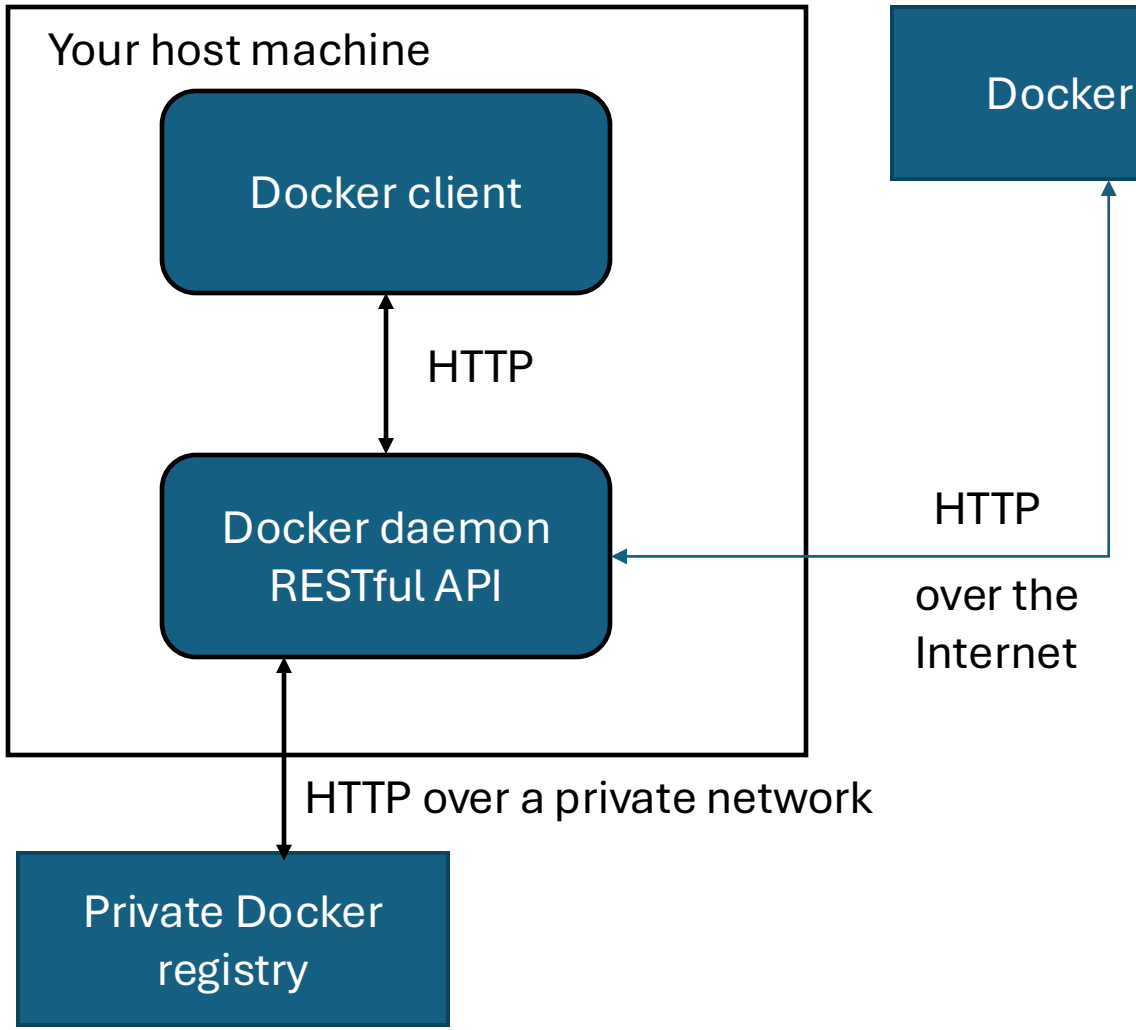
In this case, both use the same Debian base image but have different versions of python.



Docker Architecture

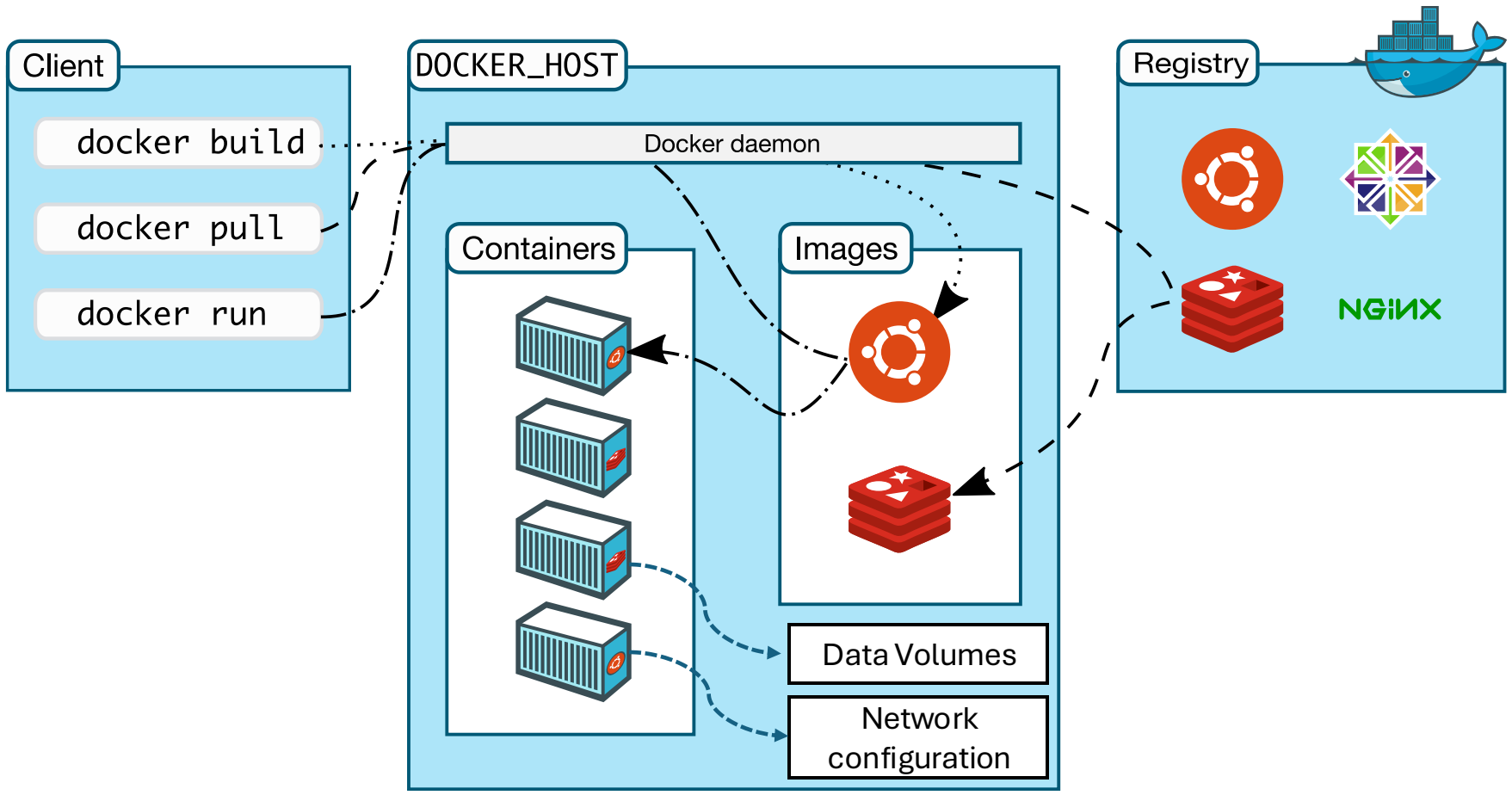
The main components comprising Docker

Docker Architecture



- You talk to the Docker client, a CLI that enables you to talk to the Docker *daemon*. A *daemon* is a process that runs in the background. It is part of the Docker Engine.
- The Docker Engine daemon is a *server* supporting a RESTful API. It receives requests, processes them, and sends responses.
- The daemon may communicate with other services. It manages the host's images and containers.

Docker Architecture



Source

docker docs

Search...

- Errors
- Versioning
- Authentication
- Containers
  - GET List containers
  - POST Create a container
  - GET Inspect a container
  - GET List processes running inside a container
  - GET Get container logs
  - GET Get changes on a container's filesystem

List containers

Returns a list of containers. For details on the format, see the inspect endpoint.

Note that it uses a different, smaller representation of a container than inspecting a single container. For example, the list of linked containers is not propagated .

QUERY PARAMETERS

all — boolean
Default: `false`
Return all containers. By default, only running containers are shown.

limit — integer
Return this number of most recently created containers, including non-running ones.

size — boolean
Default: `false`
Return the size of container as fields `SizeRw` and `SizeRootFs`.

filters — string
Filters to process on the container list, encoded as JSON (a `map[string][]string`). For example, `{"status":["paused"]}` will only return paused containers.

Available filters:

- `ancestor`=(`<image-name>[:<tag>]`, `<image id>`, or `<image@digest>`)
- `before`=(`<container id>` or `<container name>`)
- `expose`=(`<port>[/<proto>]`|`<startport-endport>/[<proto>]`)
- `exited=<int>` containers with exit code of `<int>`
- `health`=(`starting`|`healthy`|`unhealthy`|`none`)

GET /containers/json

Response samples

200 400 500

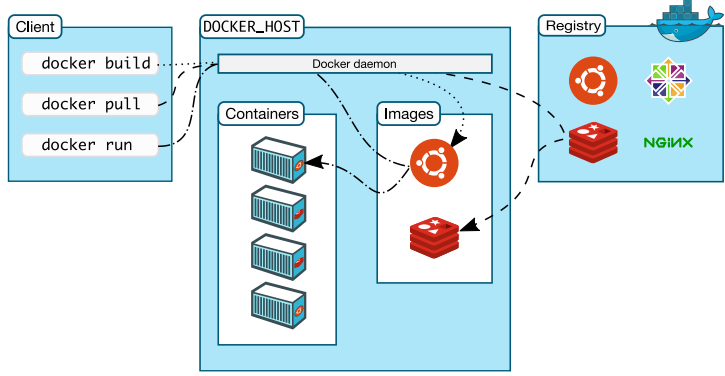
Content type
application/json

Copy Expand all Collapse all


```
[
  - {
      "Id": "8dfafdbc3a40",
    + "Names": [ … ],
      "Image": "ubuntu:latest",
      "ImageID": "d74508fb6632491cea586a
      "Command": "echo 1",
      "Created": 1367854155,
      "State": "Exited",
      "Status": "Exit 0",
    + "Ports": [ … ],
    + "Labels": { … },
      "SizeRw": 12288,
      "SizeRootFs": 0,
    + "HostConfig": { … },
    + "NetworkSettings": { … },
    + "Mounts": [ … ]
    },
  - {
      "Id": "9cd87474be90",
    + "Names": [ … ],
      "Image": "ubuntu:latest",
```


Docker Engine provides an OpenAPI spec https://docs.docker.com/reference/api/engine/version/v1.51/

Docker Architecture (cont)

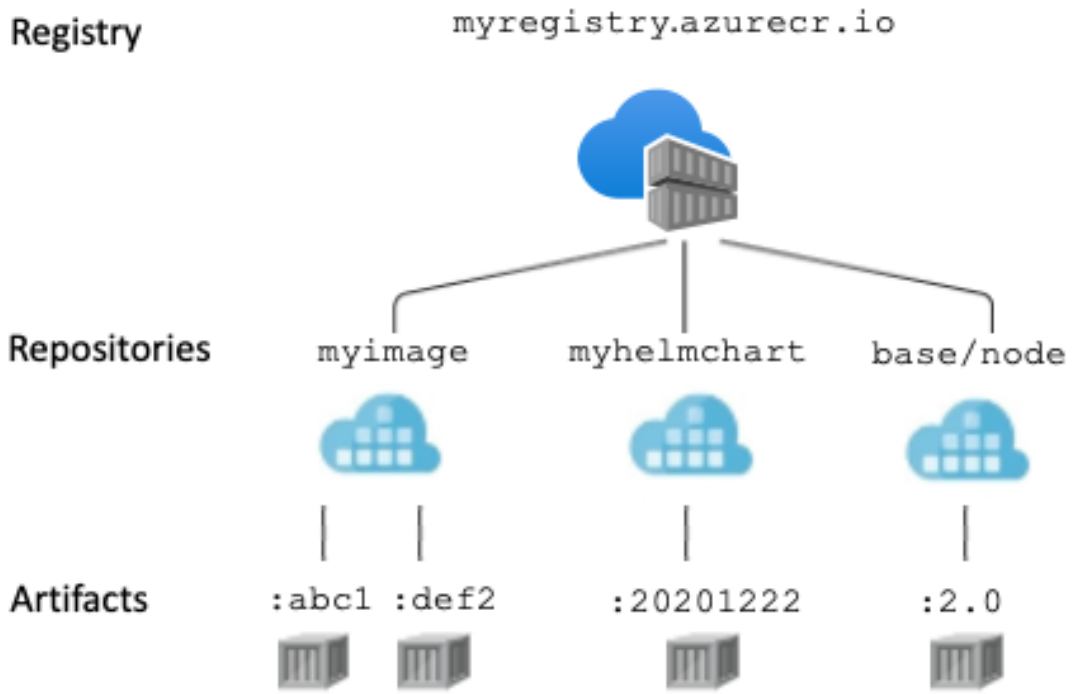


- A *Docker registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.

- When you use the docker pull or docker run commands, the required images are pulled from your local daemon or from a configured registry. When you use the docker push command, your image is pushed to your configured registry or to a specified registry.

Registries and Repositories

- A *Docker registry* is a storage and distribution system for named Docker images. The registry allows Docker users to pull images locally, as well as push new images to the registry (given adequate access permissions when applicable).
- A Docker registry is organized into *Docker repositories* , where a repository holds all the versions of a specific image. E.g., images that have the same name, but different tags.



An image may have 0, 1, or many tags.

Image Source

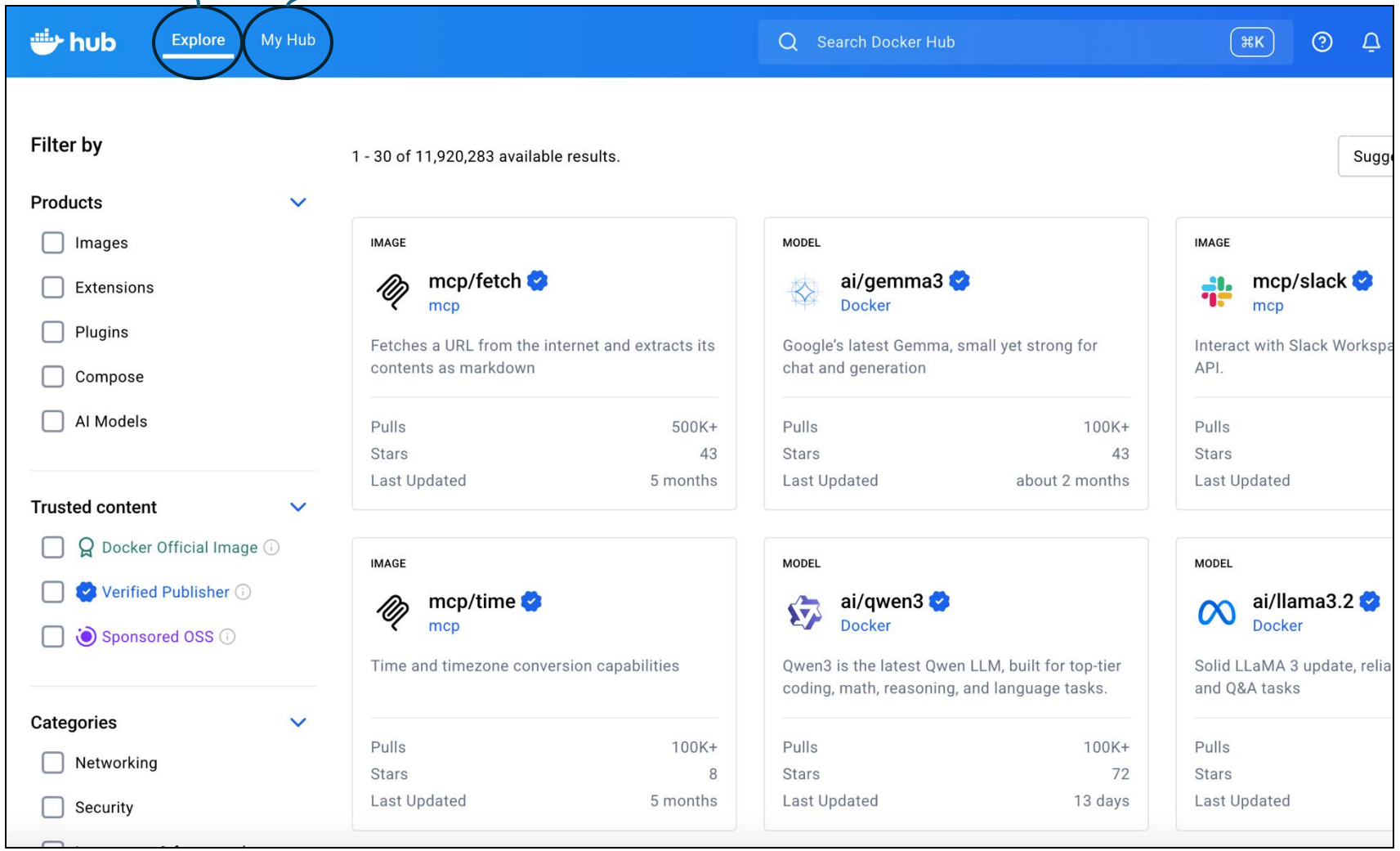
Docker Hub and private registries

Docker Hub hosts a large library of pre-built images (operating systems, development stacks, etc.). It is a great way to accelerate development as one can almost instantly get started using an image by downloading it from Docker Hub and running it.

Many organizations, however, deploy **private registries** for enhanced security control. These private registries contain public and proprietory images,all of which have been scanned for vulnerabilities and checked for conformance to corporate standards. The private registry faciliates incorporation of these images into development pipelines.

Public images

Images belonging to my account



By default, the Docker engine interacts with *DockerHub* , Docker's public registry instance. However, it is possible to run on-premise the open-source Docker registry/distribution, as well as a commercially supported version called Docker Trusted Registry. There are other public registries available online.

hub.docker.com

Sign up for a free account – you will need it later in the course

Docker best practices

1. *Run a single process per container*. Helps to build loosely coupled applications. Use container linking or *container networking* to communicate between containers.
2. *Treat containers as ephermal*. They should be immutable entities, capable of being stopped and restarted. Store runtime configuration and data outside the image; e.g., use *Docker volumes*.
3. *Build your image with minimum files required*. Use .dockerignore or other techqnies to keep the *build context minimal*.
4. *Use prebuilt images from Dockerhub* whenever you can.
5. *Minimize the layers* in your Docker image, to minimize the number of layers in your image.

Source: Docker Cookbook by Sébastien Goasguen

How to share using Docker

Create an image & put it in a in a public repository

- Quick to download and startup.
- Users will always get the exact replica of the image. There are pros and cons to this.*
- Does not require availability of any source files.
- Does not provide a repeatable way to reproduce the image.
- Hard to know exactly what is in an image and what dependencies it uses

Create a Dockerfile and share with others

- Need to build image and may be slow for large projects.
- Users create new image. May not be exact replica of original. There are pros and cons to this.*
- May require source files not available to others.
- Provides a repeatable way to reproduce the image.
- Easy to know what is in the image and what dependencies it has

***Do you want the latest version of a dependency or the original version? Do you specify in the Dockerfile nginx: 3.4 or nginx:latest? What about security vulnerabilty patches?**

Docker Summary

The main value of containers

Docker issues

What we have learnt about Docker so far

- Docker *images* consist of a file system, some metadata, and a startup cmd. Metadata includes things like exposed ports, environment variables, etc.
- When you *run* an image the Docker engine creates a *container*. The container is isolated, having its own filesystem and system settings.
- Because images are built using a *union filesystem*, containers share files, and only get their own copy when they modify a file. This keeps containers *lightweight* as they share files with other containers.
- It also makes it easy to make your own image by *modifying an existing image* – you just add or modify files on top of the existing image.

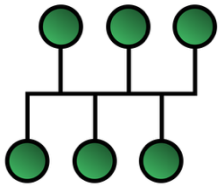
What we have learnt about Docker so far (cont)

- A *Dockerfile* is a recipe for making a new image. You specify an image to start with, and then issue commands to change that image. You also specify the startup command.
  - We saw how easy it is to take our rest-word-svr-v1.py code and use it to create a Docker container.
- There are many commands that operate on images and containers.
  - See the official Docker documentation or other Internet documentation.

Docker is an effective way to build and deploy apps

1. Simplicity of reusing & customizing pre-built images

2. Portable and platform independent by encapsulating dependencies



3. Lightweight environment for executing small loosely coupled microservices

4. Facilitates automation of CI/CD processes

But Docker does not solve all problems

Dependencies and isolation:

- When importing multiple packages into a single container, there can still be dependency conflicts
  - This is mitigated if each container has a single responsibility and uses just a few integrated packages
- When a Dockerfile updates packages or is based upon an image that is tagged “latest”, it may break previous working image
  - And one often needs to use the latest version; e.g., to make sure latest vulnerabilities are patched
  - Still need to test!

Docker or containers?

Docker is not the only container technology. We focus on Docker because

- It is industry leader currently.
- The reasons people use containers are largely the same across all container technologies.
- The concepts you learn for Docker will apply to many other container technologiess.

Nonetheless, be aware that technologies like Podman are gaining traction. This is partially due to desire for better security and performance.

References

References

- Docker
  1. https://docs.docker.com/get-started/overview/
  2. Docker in Practice, by Ian Miell and Aidan Hobson Sayers, Manning, 2nd edition, 2019
  3. Docker Cookbook, by Sébastien Goasguen, O'Reilly, 2016
  4. High level overview and value description: https://www.ibm.com/in-en/cloud/learn/containers
  5. A good overview of Docker. https://www.freecodecamp.org/news/what-is-docker-used-for-a-docker-container-tutorial-for-beginners/
  6. Examples of Dockerfiles. https://github.com/topics/dockerfile-examples
  7. Explanation of 4 key Linux technologies that Docker makes use of: https://opensource.com/article/21/8/container-linux-technology
  8. Explanation of Linux namespaces and cgroups: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/ https://www.codementor.io/blog/docker-technology-5x1kilcbow

References

- Docker

9. https://pythonspeed.com/articles/base-image-python-docker-images
10. Overlay filesystem: https://jvns.ca/blog/2019/11/18/how-containers-work--overlayfs
    https://gdevillele.github.io/engine/userguide/storagedriver/overlayfs-driver/
    https://martinheinz.dev/blog/44

    https://docs.docker.com/engine/storage/drivers/#images-and-layers
9. Dockerfiles depend on containers. https://iximiuz.com/en/posts/you-need-containers-to-build-an-image/
10. Docker and DevOps. https://learn.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/docker-application-lifecycle/containers-foundation-for-devops-collaboration

References

- Docker
  11. Publishing ports: https://iximiuz.com/en/posts/docker-publish-container-ports/
  12. Differences between a VM and Docker: https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine
  13. “Navigating the Docker Ecosystem: A comprehensive survey”, https://arxiv.org/pdf/2403.17940.pdf
  14. Research on how to automate the construction/optimization of a Dockerfile. “Automatic Service Containerization with Docker”, João Carlos Maduro, https://repositorio-aberto.up.pt/bitstream/10216/135486/2/487218.pdf
- Docker with VMs, Applications integrated into VMs
  - “Experimental Assessment of Containers Running on Top of Virtual Machines”, https://arxiv.org/pdf/2401.07539.pdf
  - “Live Objects All The Way Down”, https://arxiv.org/pdf/2312.16973.pdf
- Management of Docker containers across multi-cloud environments
  - “Containerization in Multi-Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation”, https://arxiv.org/pdf/2403.12980.pdf (surveys many papers on using containers in complex multicloud environments)

Additional reference material

Comes in handy when building docker containers

DELIVER FASTER DELIVER BETTER https://devopscycle.com/

DOCKER CHEAT SHEET

Create Dockerfiles

To build Docker images, you first need a Dockerfile. A Dockerfile is a text file named Dockerfile and has no file extension.

```
FROM <baseimage> # get them from https://hub.docker.com

# set the working directory inside the image
WORKDIR /app

# copy the source code from the host machine to the image
# (host: your development machine or your server)
# the first dot (.) is the current working directory (CWD)
# of the host, the second dot (.) is the CWD of the image
COPY . .

# run any shell command inside the image
RUN <command> # e.g., [npm run build]

# does not really expose the port 80 (only documentation)
# so that we know which port to expose
EXPOSE 80 # we expose ports with the --publish flag

# this command gets executed when the container starts
CMD <command> # e.g., ["node", "server.js"]
```

Create Multistage Dockerfiles

Multistage Dockerfiles are used to optimize Dockerfiles. One use case is to create a builder and a serve stage with separate base images. This strategy can be used to make the final Image smaller and have a lower attack because it has less system libraries. Each stage starts with FROM.

```
# with as, you can give the current stage a variable name
FROM <baseimage> as builder

# now do something, install dependencies, build your code
RUN <command> # e.g., g++ main.cpp

# the second stage could use a smaller image
# small images are based on alpine or you can build
# FROM scratch (this is a Docker image)
# if you do not need any system libraries
FROM scratch as serve

# you can now copy files from the builder stage
# e.g., the binary file that you build in that stage
COPY --from=builder ./hello-world ./hello-world

# this command gets executed when the container starts
CMD ["./hello-world"]
```

Create Docker Images

```
# <path> sets the context for the docker build command
# set to dot (.) it will use the CWD of the hostmachine
# to find the Dockerfile
$ docker build <path>

# if the Dockerfile is located somewhere else
# or the Dockerfile is named differently use --file
# remember docker will still use the <path> as context
$ docker build --file ./path/to/Dockerfile
./path/to/context

# name images: <image-name> equals <name>:<tag> where
# name is separated into <username>/<repository>:<version>
$ docker build --tag user/repo:0.1.0 .

$ docker image ls # list all images
```

Create Docker Containers

```
# Docker containers are running images
$ docker run <image-name>

# you can run public images from Docker Hub
# or images from a private registry
$ docker run https://privateregistry.com/<image-name>

# containers are started in the foreground. as soon as
# you kill the process e.g., the terminal, it will stop
# the container
# to run a container in the background, you need to run
# it in detached mode
$ docker run --detached <image-name>

# list all containers
$ docker container ls
# or shorthand syntax
$ docker ps
# list all containers, including stopped ones
$ docker container ls --all

# stop a container
$ docker stop <container-id>
# remove a container
# only stopped containers can be removed
$ docker rm <container-id>
# start a stopped container
$ docker start <container-id>
# restart a running container
$ docker restart <container-id>

# automatically remove a container when it is stopped
$ docker run --rm <image-name>
```

Access Docker Containers

```
# publish ports, e.g., forward container port to host port
$ docker run --publish <host-port>:<container-port>
  <image-name>

# execute shell command in a container
$ docker exec --interactive --tty <container-id> <command>

# open an interactive shell (like connecting to a server)
$ docker exec --interactive --tty <container-id> sh

# exit the container
$ exit
```

Create Docker Volumes

To persist data from Docker containers between starts of containers you need volumes. When a container is removed all data from the container will be lost if you do not use volumes.

```
# using a named volume (docker handles location on host)
$ docker run --volume <volume-name>:/path/in/container
  <image-name>

# using a mounted volume (you handle location on host)
$ docker run --volume /path/on/host:/path/in/container
  <image-name>

# list all volumes incl. metadata
$ docker volume ls
```

Blog: https://devopscycle.com/blog/the-ultimate-docker-cheat-sheet/
GitHub: https://github.com/aichbauer/the-ultimate-docker-cheat-sheet/
Consulting: https://devopscycle.com/

BY LUKAS AICHBAUER https://devopscycle.com/

https://devopscycle.com/blog/the-ultimate-docker-cheat-sheet/

Some useful commands on images

- When you create a new image, it gets a unique ID. An image can be referred to either by
  - its ID (unique prefix is good enough), or
  - by its name.

Hence any command requiring an image, the image can be specified by its ID or its name.

- docker images

Lists all existing docker images in the local registry

- docker rmi <img>

Removes an existing image from the local registry

- docker tag <src-img> <trg-img>

Tags the image <src-img> with a new tag <trg-img>. The <src-img> may be the image's unique ID or an already specified name:tag for the image. In the latter case, the original tag remains – this command creates an additional tag for the same image.

- docker run <img>
  docker run --name <cntr> <img>
  docker run --publish hport:cport python-docker

Build a container based upon <img> and start the container.

The --name (-n) flag gives a user friendly name to the container.

The --publish (-p) flag maps the host's port hport to the container's port cport.

- docker history <img>

Shows the history of an image – the layers comprising the image.

- docker sbom <img>

Displays the *Software Bill of Materials* of a Docker image. New experimental cmd

Common Dockerfile commands

Command	Purpose
FROM	To specify the parent image.
WORKDIR	To set the working directory for any commands that follow in the Dockerfile.
RUN	To install any applications and packages required for your container.
COPY	To copy over files or directories from a specific location.
ADD	As COPY, but also able to handle remote URLs and unpack compressed files.
ENTRYPOINT	Command that will always be executed when the container starts. If not specified, the default is /bin/sh -c
CMD	Arguments passed to the entrypoint. If ENTRYPOINT is not set (defaults to /bin/sh -c), the CMD will be the commands the container executes.
EXPOSE	To define which port through which to access your container application.
LABEL	To add metadata to the image.

Documentation to get started using Docker with Python

Docker overview

- https://docs.docker.com/get-docker/

Install Docker

- https://docs.docker.com/get-docker/

Build

- https://docs.docker.com/language/python/build-images/

Run

- https://docs.docker.com/language/python/run-containers/

These links basically cover the steps we just went over.

You can also use these links as a starting point to explore a lot more rich documentation on Docker.