UNIVERSITÀ
DEGLI STUDI
DI PADOVA

J. Pazzini
PADOVA UNIVERSITY, INFN

M. Migliorini
PADOVA UNIVERSITY, INFN, CERN

X - CONTAINERS AND DOCKER

Management and Analysis of Physics Datasets - Module B

Physics of Data

A.A. 2023/2024

DOCKER…? CONTAINERS…? WHAT? WHY?

Containers are processes running on our machine in completely isolated environments that allows us to build and run software packages without conflicting with other applications

DOCKER…? CONTAINERS…? WHAT? WHY?

Containers are processes running on our machine in completely isolated environments that allows us to build and run software packages without conflicting with other applications

Let's imaging a very common situation…

Multiple people/teams working on the development of the same application/program/analysis

- someone implements a new solution or feature
- tests it within own machine and environment
- and shares it with all others
- and…

DOCKER…? CONTAINERS…? WHAT? WHY?

Containers are processes running on our machine in completely isolated environments that allows us to build and run software packages without conflicting with other applications

Let's imaging a very common situation…
Multiple people/teams working on the development of the same application/program/analysis

- someone implements a new solution or feature
- tests it within own machine and environment
- and shares it with all others
- and…

Nothing works on someone else's machine… Why would that be?

DOCKER…? CONTAINERS…? WHAT? WHY?

Containers are processes running on our machine in completely isolated environments that allows us to build and run software packages without conflicting with other applications

Let's imaging a very common situation…
Multiple people/teams working on the development of the same application/program/analysis

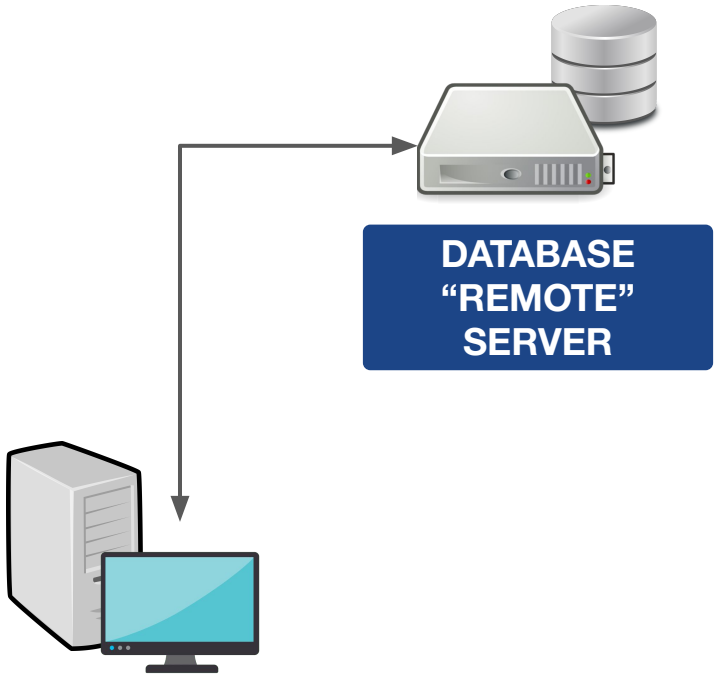
- someone implements a new solution or feature
- tests it within own machine and environment
- and shares it with all others
- and…

Nothing works on someone else's machine… Why would that be?

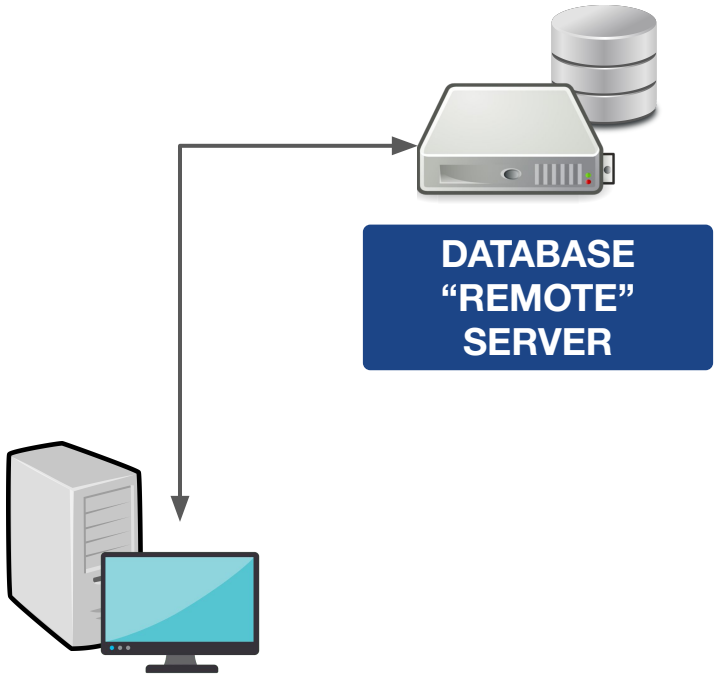
Too many things might actually have gone wrong…

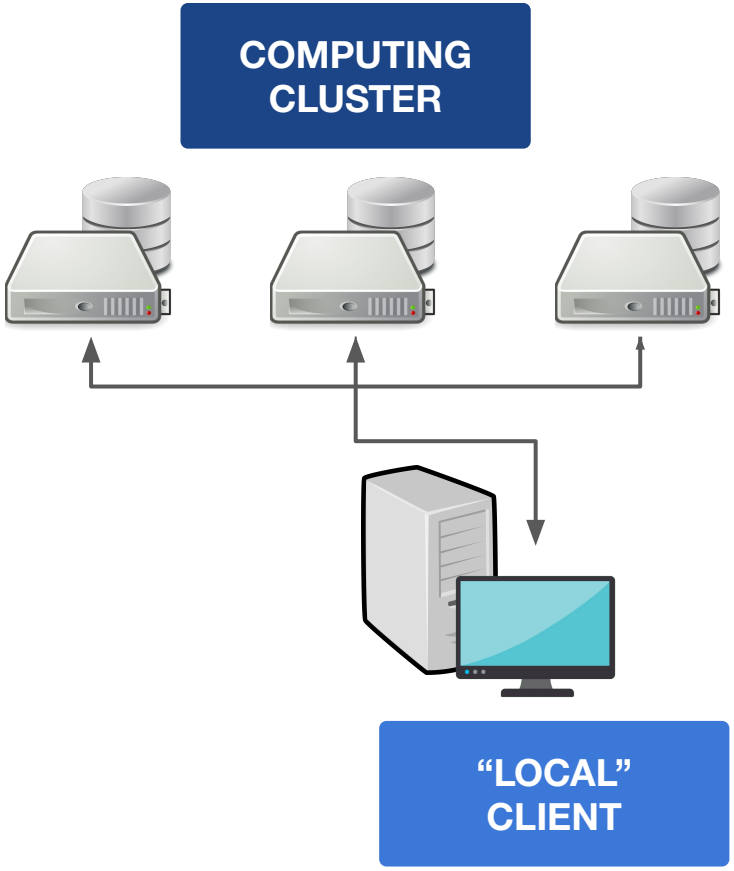
- Missing packages/libraries
- Wrong versions of a given package (*but*… installing the correct ones does break other dependencies)
- Different OS (e.g. Win-Linux)
- …

WHAT WILL WE USE CONTAINERS FOR?

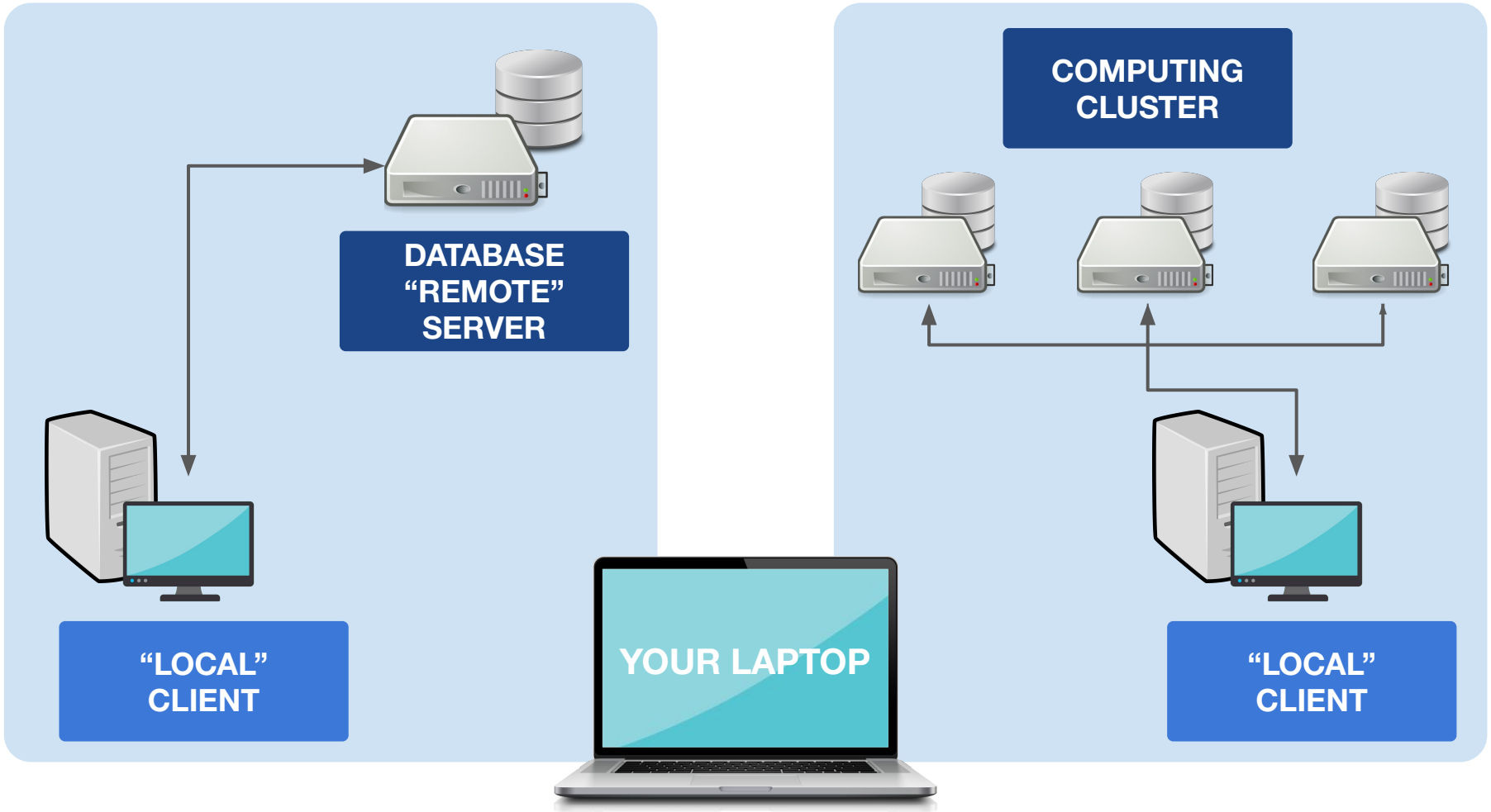


WHAT WILL WE USE CONTAINERS FOR?





WHAT WILL WE USE CONTAINERS FOR?



SO… A VIRTUAL MACHINE?

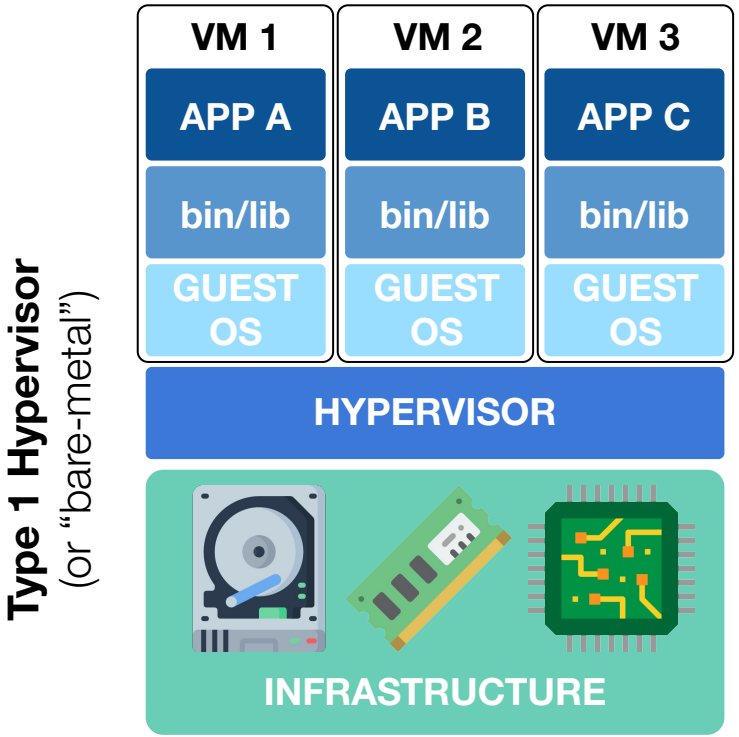
Virtual Machines (VMs) are an abstraction of the physical hardware of a computer system

Virtualization is provided by a named ***Hypervisor*** → one server can be turned into many ("smaller") servers, each with its own fraction of memory, disk, CPU resources, its own OS, and running its own application(s)

SO… A VIRTUAL MACHINE?

Virtual Machines (VMs) are an abstraction of the physical hardware of a computer system

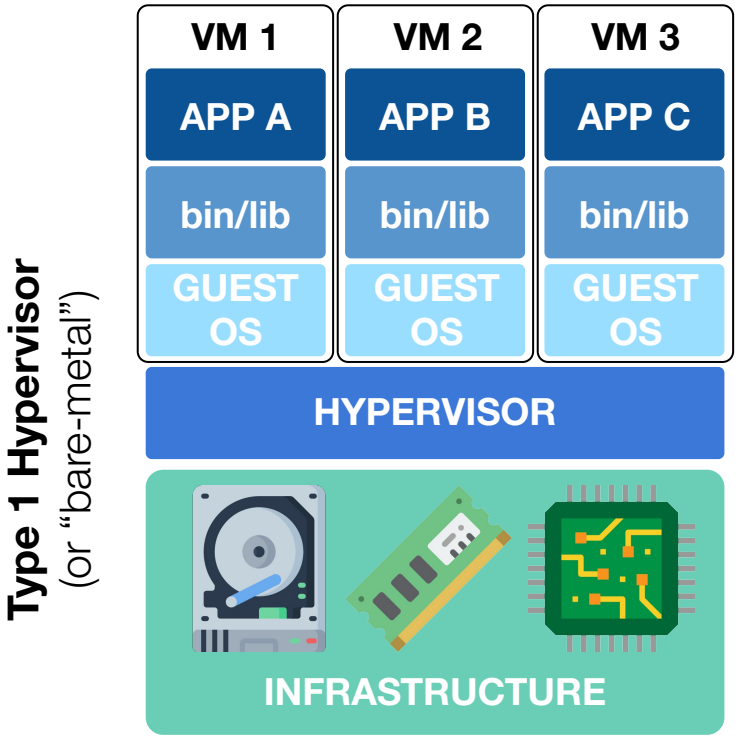
Virtualization is provided by a named ***Hypervisor*** → one server can be turned into many ("smaller") servers, each with its own fraction of memory, disk, CPU resources, its own OS, and running its own application(s)

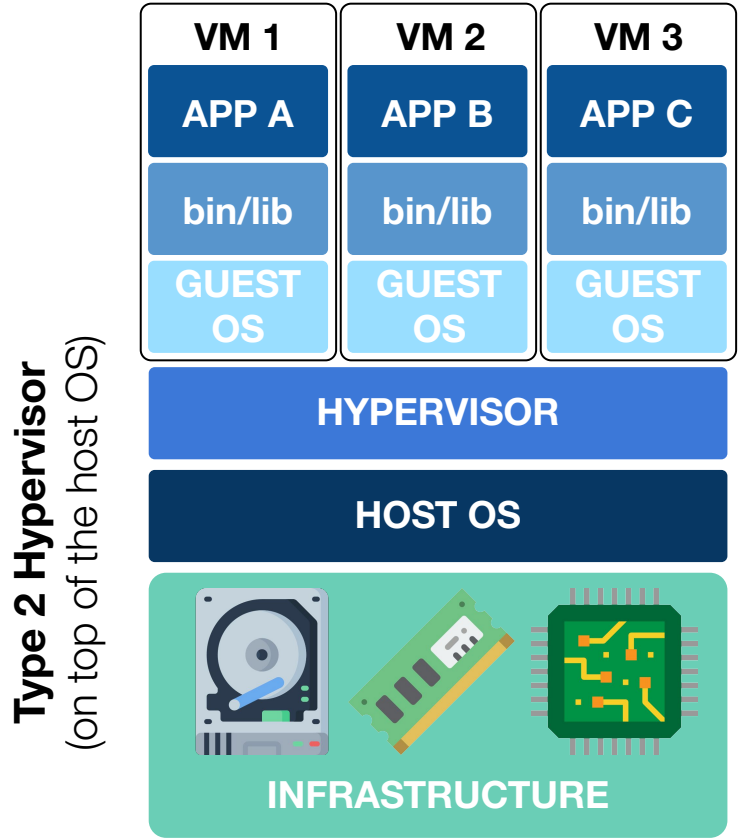


SO… A VIRTUAL MACHINE?

Virtual Machines (VMs) are an abstraction of the physical hardware of a computer system

Virtualization is provided by a named ***Hypervisor*** → one server can be turned into many ("smaller") servers, each with its own fraction of memory, disk, CPU resources, its own OS, and running its own application(s)





SO… A VIRTUAL MACHINE?

Each running VM includes a full copy of the guest OS (Win/Linux/…), all the necessary binaries and libraries, and finally the application you want to run.

All this might take up to tens of GBs, and makes VMs also quite slow to start (boot).

VMs provide full process isolation for applications: the software running in the guest operating system does not interfere with the host OS, and vice-versa.

But this isolation comes at the cost of the overhead spent virtualizing the hardware.

NOT REALLY…

Containers are instead “software packages” that include the code, libraries and all dependencies required to run your applications, without the need to bring along a Guest OS as in VMs.

NOT REALLY…

Containers are instead “software packages” that include the code, libraries and all dependencies required to run your applications, without the need to bring along a Guest OS as in VMs.

Containerization is still a type of virtualization that allows to run applications independently and in complete isolation, but it is more efficient than Virtual Machines because they all share the Host OS kernel.

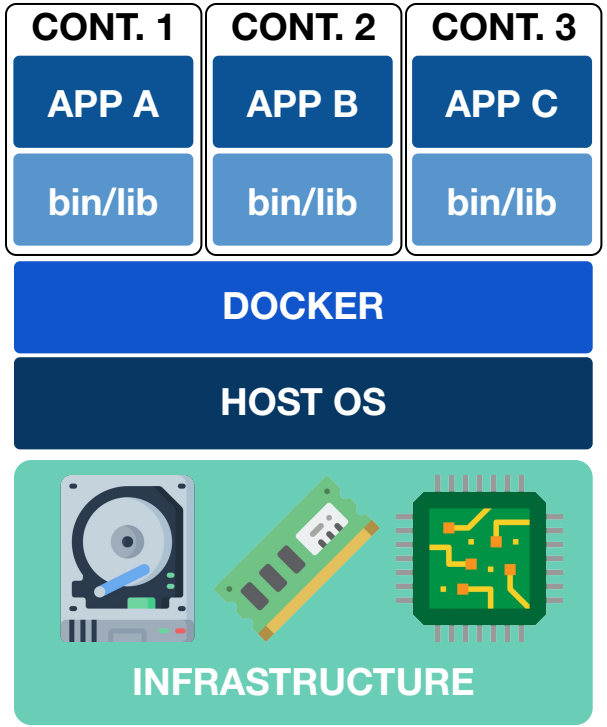
The resulting “software packages” (the containers) are lightweight compared to VMs, faster to run, and using much less resources.

NOT REALLY…

Containers are instead “software packages” that include the code, libraries and all dependencies required to run your applications, without the need to bring along a Guest OS as in VMs.

Containerization is still a type of virtualization that allows to run applications independently and in complete isolation, but it is more efficient than Virtual Machines because they all share the Host OS kernel.

The resulting “software packages” (the containers) are lightweight compared to VMs, faster to run, and using much less resources.



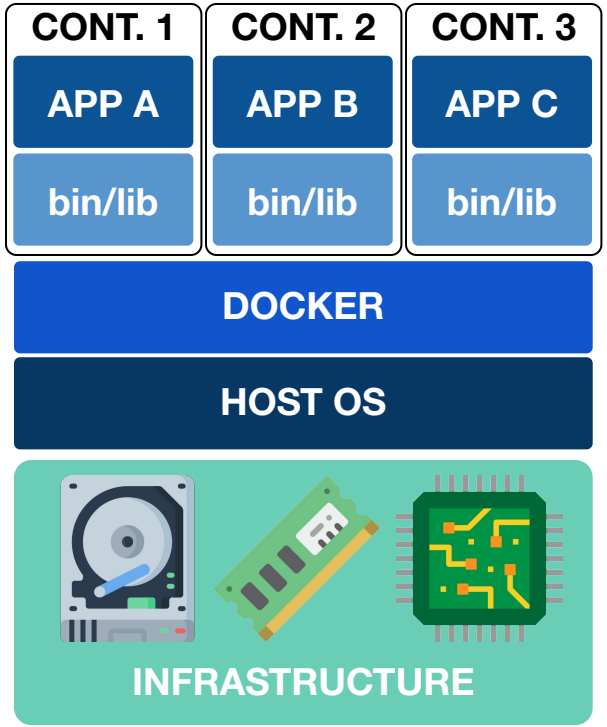
NOT REALLY…

Containers are instead “software packages” that include the code, libraries and all dependencies required to run your applications, without the need to bring along a Guest OS as in VMs.

Containerization is still a type of virtualization that allows to run applications independently and in complete isolation, but it is more efficient than Virtual Machines because they all share the Host OS kernel.

The resulting “software packages” (the containers) are lightweight compared to VMs, faster to run, and using much less resources.

Docker is a containerization platform that offers a way to create and run containers



BASICS OF DOCKER

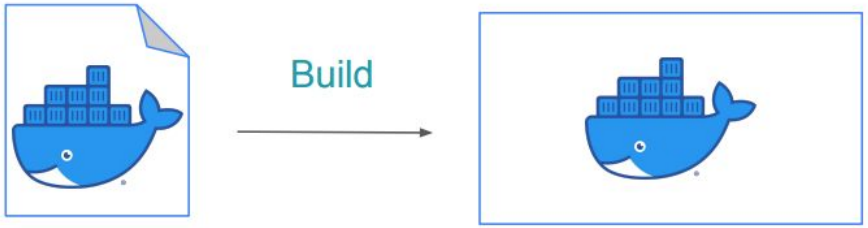
Set of instructions to create a docker image

Dockerfile

BASICS OF DOCKER

Set of instructions to create a docker image

Blueprint for creating containers



Dockerfile

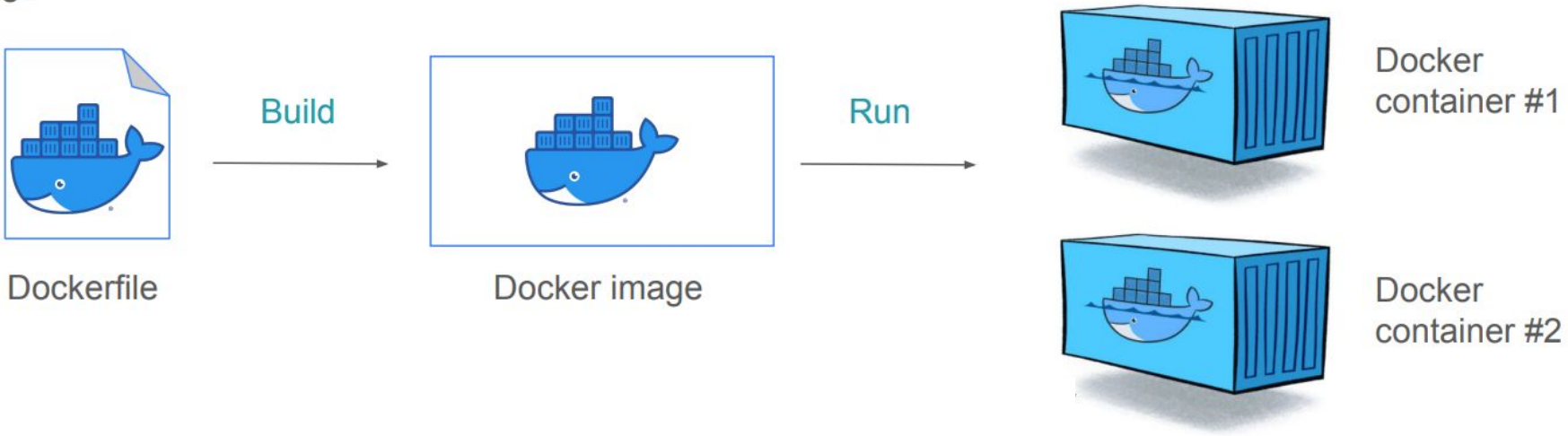
Docker image

BASICS OF DOCKER

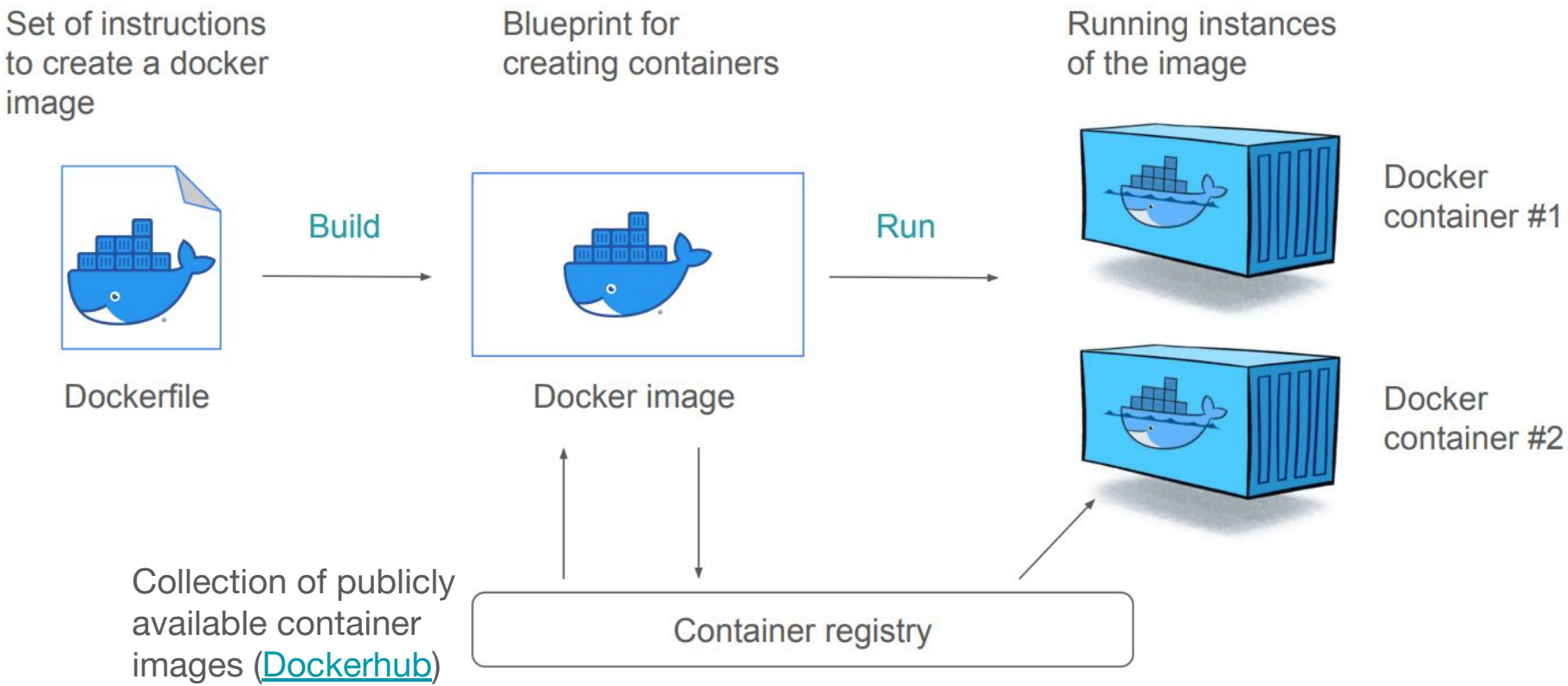
Set of instructions to create a docker image

Blueprint for creating containers

Running instances of the image



BASICS OF DOCKER



INSTALLING DOCKER

The Docker engine is available for almost all Linux distributions, MacOS and Windows (using the WSL2 backend) versions

You are asked to install Docker on your own, by following the instructions and installation guides provided in the official documentation

Installation guides for Linux/MacOS/Windows can be found at the link

A couple of summary notes on the installation instructions are also provided on Moodle, but please (as always) take some time to read the official documentation first.

At the end of the installation, you must cross-check if Docker is correctly installed by running the command (not run as a super-user!)

```
$ docker version
```

On the terminal for Linux and MacOS, on the WSL terminal on Windows (not the power shell)

USING DOCKER (PRE-BUILT) IMAGES

To pull an image (e.g. Ubuntu version 22.04) from docker hub

```
$ docker pull ubuntu:jammy
```

To list all existing images

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mapd_notebook	latest	e781fa1a99e2	41 hours ago	994MB
mysql_test	latest	74328f98a01d	5 days ago	1.15GB
ubuntu	jammy	74f2314a03de	8 days ago	77.8MB
mapd-b-exam	latest	fc7f5ce9a203	9 days ago	1.28GB
...	...	...	...	...

To delete a local image

```
$ docker image rm <imageID>
```

To remove ALL unused images

```
$ docker image prune
```

RUNNING CONTAINERS FROM IMAGES

To create and launch a container from an existing image

```
$ docker run --rm -i -t -d --name myubuntu ubuntu:jammy /bin/bash
```

- Create a container from the Ubuntu image, with name `myubuntu`
- `/bin/bash` is a (optional!) command that will be run inside the container
- `--rm` specifies Docker to remove the container once stopped executing
- `-i` specifies that the command is Interactive (it starts the bash shell)
- `-t` specifies the allocation of a Terminal
- `-d` instructs the container to run in the background (Detached)

You can list all the containers currently running or (with -a) also exited

```
$ docker ps -a
```

You can attach to a running container (to the running process, /bin/bash in previous example)

```
$ docker attach <container-name>
```

Or create a new shell inside the container

```
$ docker exec -it <container-name> /bin/bash
```

MANAGING CONTAINERS

Docker containers are applications that could be started/restarted/stopped/removed

```
$ docker start/restart/stop/rm <container-name>
```

Please do remember… containers are **isolated environments**

- Host's (your computer) files aren't visible inside the container
- When the container is deleted, data created inside the container will be lost
- The container by default don't accept connections over any port (e.g. the 8888)

MANAGING CONTAINERS - VOLUMES

Docker containers are applications that could be started/restarted/stopped/removed

```
$ docker start/restart/stop/rm <container-name>
```

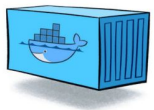
Please do remember… containers are **isolated environments**

- Host's (your computer) files aren't visible inside the container
- When the container is deleted, data created inside the container will be lost
- The container by default don't accept connections over any port (e.g. the 8888)

We can however create Volumes to persist data produced and used by the container

```
$ docker run --rm -v $PWD/test_volumes:/mnt -itd --name myubuntu ubuntu:jammy /bin/bash
```

- `-v` is the option to create a volume
- PATH_IN_YOUR_COMPUTER : PATH_INSIDE_THE_CONTAINER



- Mount the local directory `test_volumes` into `/mnt/the_volume` (in this example)

MANAGING CONTAINERS - PORTS

Docker containers are applications that could be started/restarted/stopped/removed

```
$ docker start/restart/stop/rm <container-name>
```

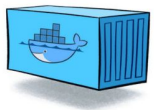
Please do remember… containers are **isolated environments**

- Host's (your computer) files aren't visible inside the container
- When the container is deleted, data created inside the container will be lost
- The container by default don't accept connections over any port (e.g. the 8888)

We can also open Ports to allow communication between the host and the container

```
$ docker run --rm -p 1234:8888 -itd --name myubuntu ubuntu:jammy /bin/bash
```

- `-p` is the option to connect ports
- PORT_IN_YOUR_COMPUTER : PORT_INSIDE_THE_CONTAINER



- Maps the local port `1234` into the container port `8888` (in this example)

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas



EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas



EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

All containers must start **FROM** an existing base-image containing properties we want to inherit

→ In this case we use an official Python image taken from the Docker-hub remote registry as a base installation platform

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas


EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

WORKDIR creates a working directory inside the container (the default Path once started the container)

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
    PYTHONUNBUFFERED=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1 \
    PIP_NO_CACHE_DIR=1

RUN pip install notebook \
                matplotlib \
                SQLAlchemy==2.0.27 \
                ipython-sql==0.5.0 \
                mysql-connector-python==8.3.0 \
                pandas



EXPOSE 8888

CMD jupyter notebook \
    --ip=0.0.0.0 \
    --port=8888 \
    --no-browser \
    --allow-root \
    --NotebookApp.token=
```

ENV declare and assign one or more environmental variables (for instance in this case the default timeout for `pip`).
These variables are **only valid inside the container**

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas


EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

RUN allows to execute shell commands.
In this case, installing a bunch of libraries from pip

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
    PYTHONUNBUFFERED=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1 \
    PIP_NO_CACHE_DIR=1

RUN pip install notebook \
                matplotlib \
                SQLAlchemy==2.0.27 \
                ipython-sql==0.5.0 \
                mysql-connector-python==8.3.0 \
                pandas


EXPOSE 8888

CMD jupyter notebook \
    --ip=0.0.0.0 \
    --port=8888 \
    --no-browser \
    --allow-root \
    --NotebookApp.token=
```

RUN allows to execute shell commands.
In this case, installing a bunch of libraries from pip

You can have multiple **RUN** calls inside one DockerFile
For instance… if you wanted to break down the `pip install` procedure into multiple calls, you could have written:

```
RUN pip install notebook
RUN pip install matplotlib
RUN pip install SQLAlchemy==2.0.27
[...]
```

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas


EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

EXPOSE the container port 8888 (the one we can use with Jupyter-notebooks).
It does not actually ***publish*** the port (i.e. attach the container port to your computer one), as there's no "outside" port specified

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
              matplotlib \
              SQLAlchemy==2.0.27 \
              ipython-sql==0.5.0 \
              mysql-connector-python==8.3.0 \
              pandas



EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

Execute the command **CMD** right after the creation of the container.
In this case, start a Jupyter server on port 8888 (the one previously exposed)

WRITING A CUSTOM DOCKERFILE

```
FROM python:3.11-slim

WORKDIR /mapd-workspace

ENV PIP_DEFAULT_TIMEOUT=100 \
   PYTHONUNBUFFERED=1 \
   PIP_DISABLE_PIP_VERSION_CHECK=1 \
   PIP_NO_CACHE_DIR=1

RUN pip install notebook \
               matplotlib \
               SQLAlchemy==2.0.27 \
               ipython-sql==0.5.0 \
               mysql-connector-python==8.3.0 \
               pandas



EXPOSE 8888

CMD jupyter notebook \
   --ip=0.0.0.0 \
   --port=8888 \
   --no-browser \
   --allow-root \
   --NotebookApp.token=
```

N.B.: we can **RUN** several commands but we can have only one **CMD**.

The **RUN** commands will be executed during the creation of the container.
The **CMD** action is the single one the container is started (e.g. create a shell/start a program/etc)

Execute the command **CMD** right after the creation of the container.
In this case, start a Jupyter server on port 8888 (the one previously exposed)

BUILDING YOUR DOCKER IMAGE

To build the image starting from the Dockerfile

```
$ docker build --tag my_image -f my_dockerfile.dockerfile .
```

- Create an image from the custom dockerfile `my_dockerfile.dockerfile`
- `--tag` tags the image with name `my_image`
- `-f` specifies the Dockerfile
- `.` specifies that the context of the image is the current directory

[to avoid issues for the Apple Silicon M1/M2 chip users, also include in the command the additional option `--platform=linux/amd64`]

Once built, the image appears in the list of available images

```
$ docker image ls
```

And can be used to run a container

```
$ docker run --rm -i -t (-d) --name my_container my_image
```

BUILDING YOUR DOCKER IMAGE

To build the image starting from the Dockerfile

```
$ docker build --tag my_image -f my_dockerfile.dockerfile .
```

- Create an image from the custom dockerfile `my_dockerfile.dockerfile`
- `--tag` tags the image with name `my_image`
- `-f` specifies the Dockerfile
- `.` specifies that the context of the image is the current directory

Images can be pushed to Dockerhub to be stored and shared:

1. Create a repository on Dockerhub, and login from your machine with:
   ```
   $ docker login
   ```
2. Tag the image with your username, a repository name and the image name:
   ```
   $ docker image tag my_image username/repo-name:image_name
   ```
3. Push the image to the remote repository:
   ```
   $ docker push username/repo-name:image_name
   ```

RUNNING MULTIPLE CONTAINERS

In this course we will use Docker to "simulate" computing systems with >1 servers

⇒ Each container is going to be viewed as an independent server running a given service

(BTW, this is also true in practice in several applications, where each container is run on a different physical server or VM)

Docker-compose is a Docker tool that can be used to manage running multiple containers.

A single Docker-compose `.yaml` file is used to define all running services (containers), as well as volumes, networks, etc

To start all services described by a Docker-compose file:

```
$ docker compose up
```

To stop and remove all resources instantiated by Docker-compose:

```
$ docker compose down
```

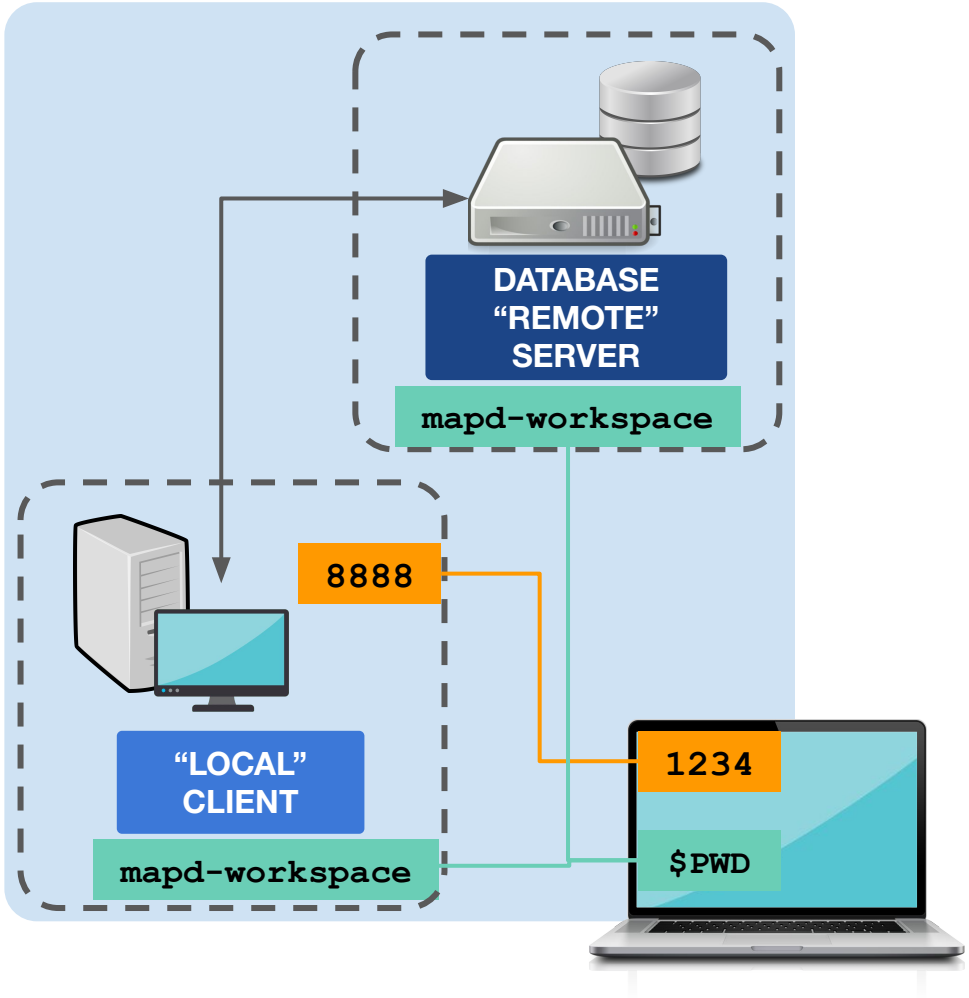
THE DOCKER-COMPOSE YAML FILE

```yaml
version: '3.9'

services:

 db:
   image: mysql:8.0.32
   environment:
     MYSQL_USER: "my_user"
     MYSQL_PASSWORD: "my_pwd"
     MYSQL_ROOT_PASSWORD: "root_pwd"
   volumes:
     - $PWD:/mapd-workspace
   command: --secure_file_priv="/mapd-workspace"

 jupyter:
   depends_on:
     - db
   image: mapd_notebook
   ports:
     - 1234:8888
   volumes:
     - $PWD:/mapd-workspace
```



THE DOCKER-COMPOSE YAML FILE

```yaml
version: '3.9'

services:

 db:
   image: mysql:8.0.32
   environment:
     MYSQL_USER: "my_user"
     MYSQL_PASSWORD: "my_pwd"
     MYSQL_ROOT_PASSWORD: "root_pwd"
   volumes:
     - $PWD:/mapd-workspace
   command: --secure_file_priv="/mapd-workspace"

 jupyter:
   depends_on:
     - db
   image: mapd_notebook
   ports:
     - 1234:8888
   volumes:
     - $PWD:/mapd-workspace
```

All containers contributing to a Docker-compose are referred to as **services**

THE DOCKER-COMPOSE YAML FILE

```yaml
version: '3.9'

services:

 db:
   image: mysql:8.0.32
   environment:
     MYSQL_USER: "my_user"
     MYSQL_PASSWORD: "my_pwd"
     MYSQL_ROOT_PASSWORD: "root_pwd"
   volumes:
     - $PWD:/mapd-workspace
   command: --secure_file_priv="/mapd-workspace"

 jupyter:
   depends_on:
     - db
   image: mapd_notebook
   ports:
     - 1234:8888
   volumes:
     - $PWD:/mapd-workspace
```

The first service is named **db** and uses an image from Dockerhub (a MySQL server).
Environment variables (usernames and passwords to use the database) and volumes specific to this container are defined directly inside the `.yaml` file

THE DOCKER-COMPOSE YAML FILE

```yaml
version: '3.9'

services:

 db:
   image: mysql:8.0.32
   environment:
     MYSQL_USER: "my_user"
     MYSQL_PASSWORD: "my_pwd"
     MYSQL_ROOT_PASSWORD: "root_pwd"
   volumes:
     - $PWD:/mapd-workspace
   command: --secure_file_priv="/mapd-workspace"

 jupyter:
   depends_on:
     - db
   image: mapd_notebook
   ports:
     - 1234:8888
   volumes:
     - $PWD:/mapd-workspace
```

The second service is named **`jupyter`** and uses the image we built locally.
It depends on the **`db`** container, thus it starts only after the first one is up and running.
Ports and volumes specific to this second container are defined with the same logic already discussed (port 8888 inside the container will be mapped to port 1234 in your laptop)

COMMANDS TO MONITOR/RECLAIM RESOURCES

Docker will use your computer resources to run containers. You can check the resource usage and (most important) free up some of them with the following commands:

Check the disk used by Docker

```
$ docker system df
```

Live monitor the computing resource used by running docker containers (similar to `top`)

```
$ docker stats
```

Reclaim resources by removing **all stopped** containers

```
$ docker container prune
```

Reclaim resources by removing all dangling images (older builds and currently unused)

```
$ docker image prune
```

Reclaim resources by removing **stopped** containers, images and volumes

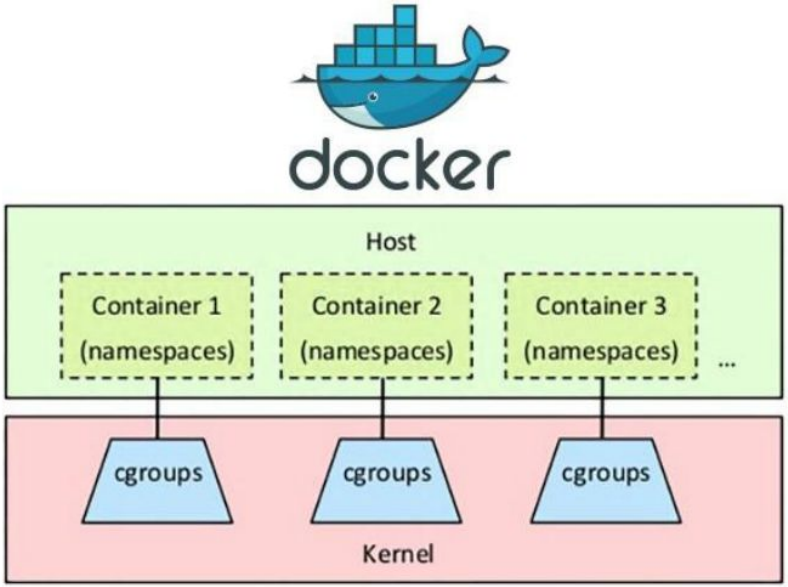
```
$ docker system prune
```

Reclaim all system resources by removing **all(!)** containers, images and volumes

```
$ docker system prune -a
```

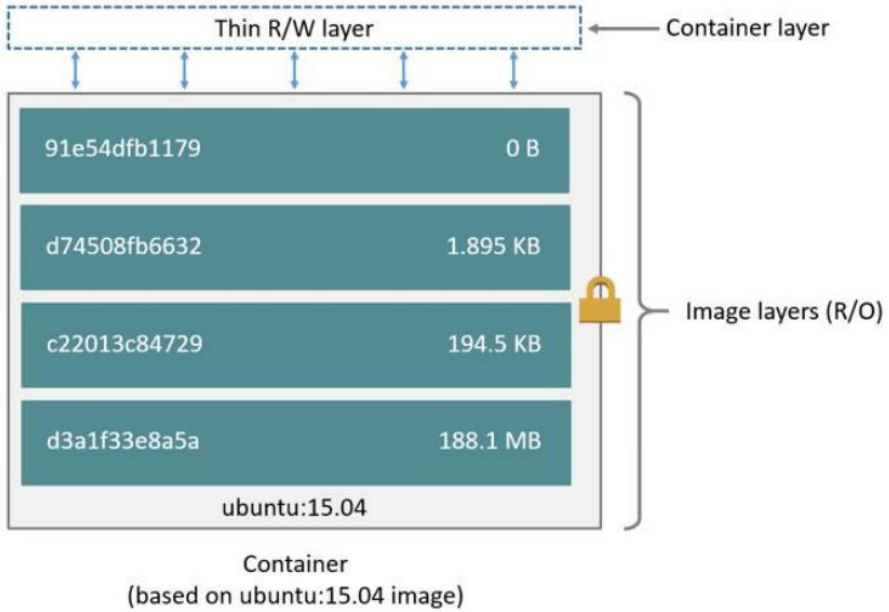
Containers = combination of namespaces & cgroups

- Namespaces: Control what you can see
- Control Group: Control what you can use



Docker images

- Read-only layers built on top of each other
- Union File System (UFS) used to build images
  - Type of file system creating the “illusion” of merging the content of several directories into a single one without modifying the content of them
- Image shared across containers
- Each time Docker launches a container from an image, it adds a writable layer (container layer), which stores all changes to the container through its runtime



Docker architecture

Client-server architecture:

- Server with a daemon process called dockerd
- Command Line Interface (CLI) to interact with the API