Introduction to Containers

(Oltre Lo) Sviluppo Software

Alessandro Costantini, Doina Cristina Duma

INFN – CNAF

April 27th 2021, Bologna

Overview

- Containers
  - Containers vs VMs
  - Working with containers
  - Networking
  - Management
- And…. Hands-on/demos, while we go …
  - https://baltig.infn.it/corso-olss-2020/corso_olss_2020

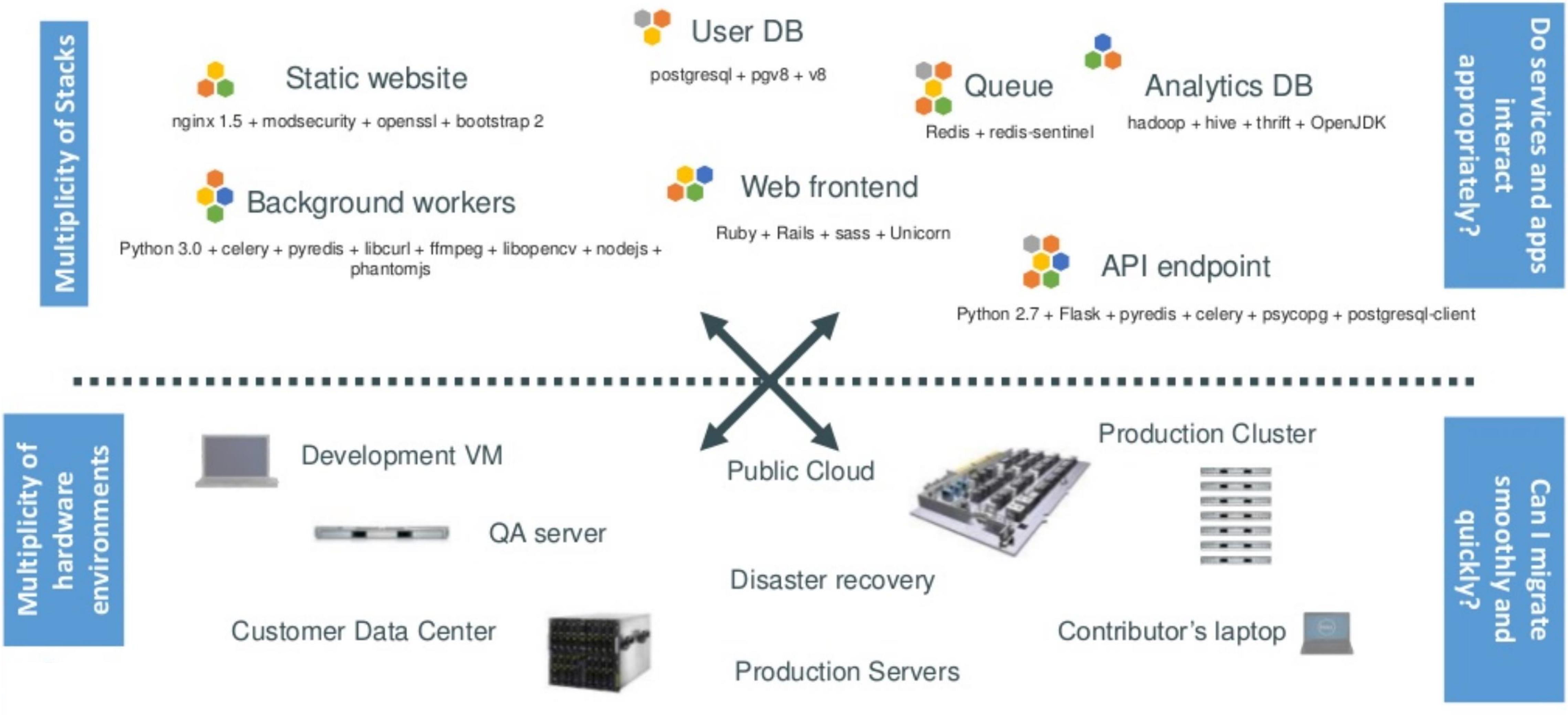
Background

- Building a web service on a Ubuntu machine
- Code works fine on local machine
- Moved to a remote server …. does not work

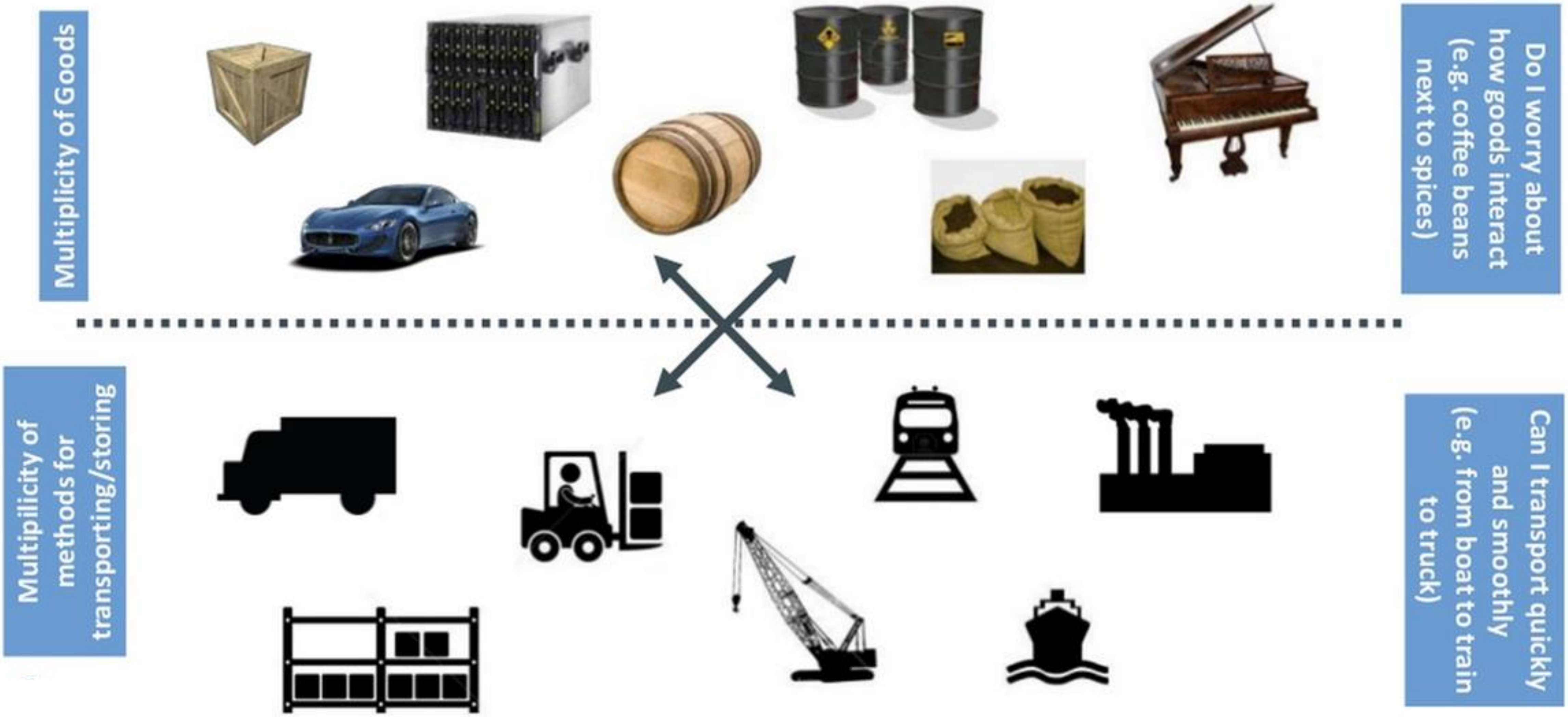
- Reasons:
  - Different OS => missing libraries or files for the runtime
  - Incompatible version of software (python, java)

It is essential to find a solution to these problems

The Challenge



Cargo Transport Pre-1960



Solution: Intermodal Shipping Container

Intermodal Shipping Container Ecosystem

- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalizations

- 5000 ships deliver 200M containers per year

Intermodal Shipping Container Ecosystem

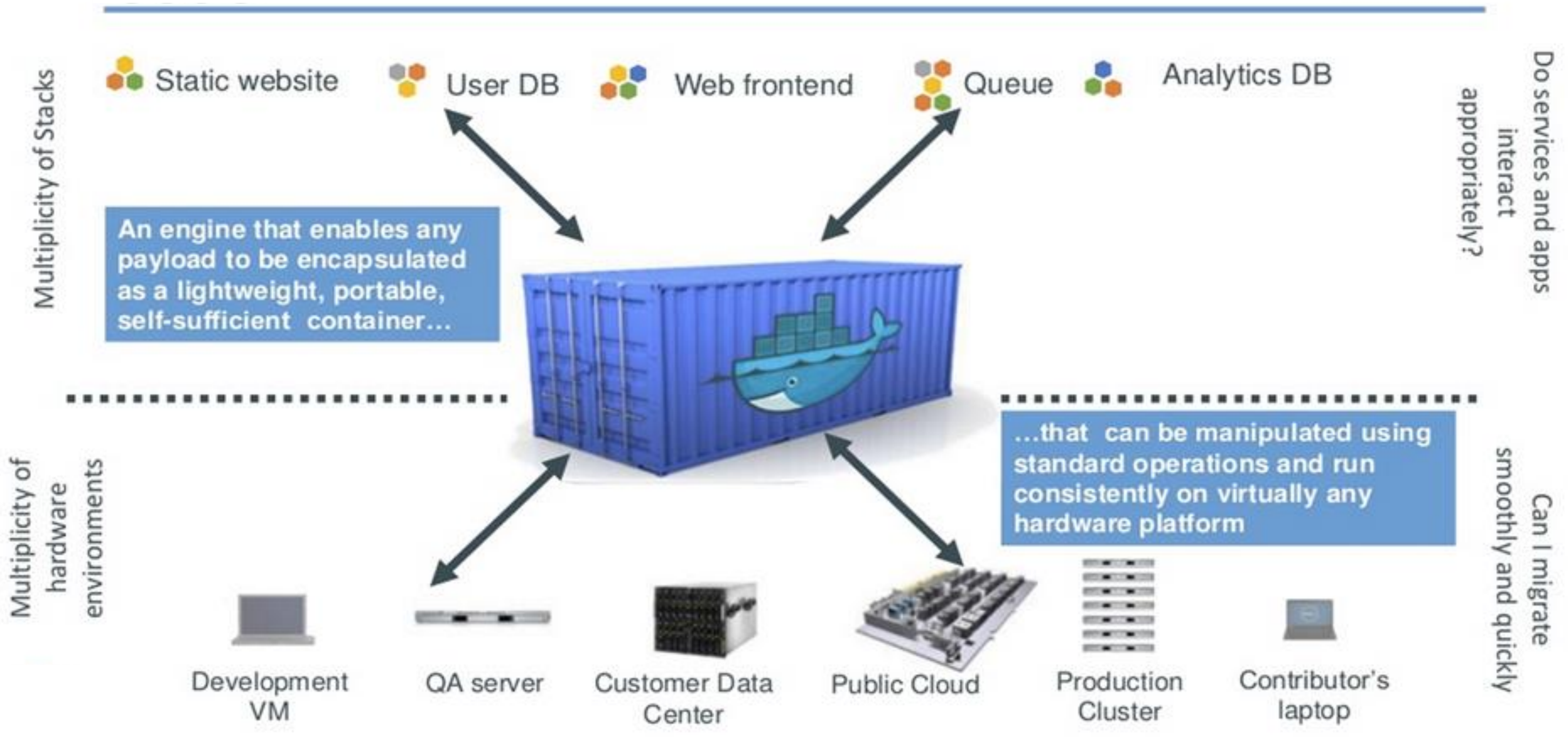
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalizations

- 5000 ships deliver 200M containers per year

OK, not everything always goes as planned…

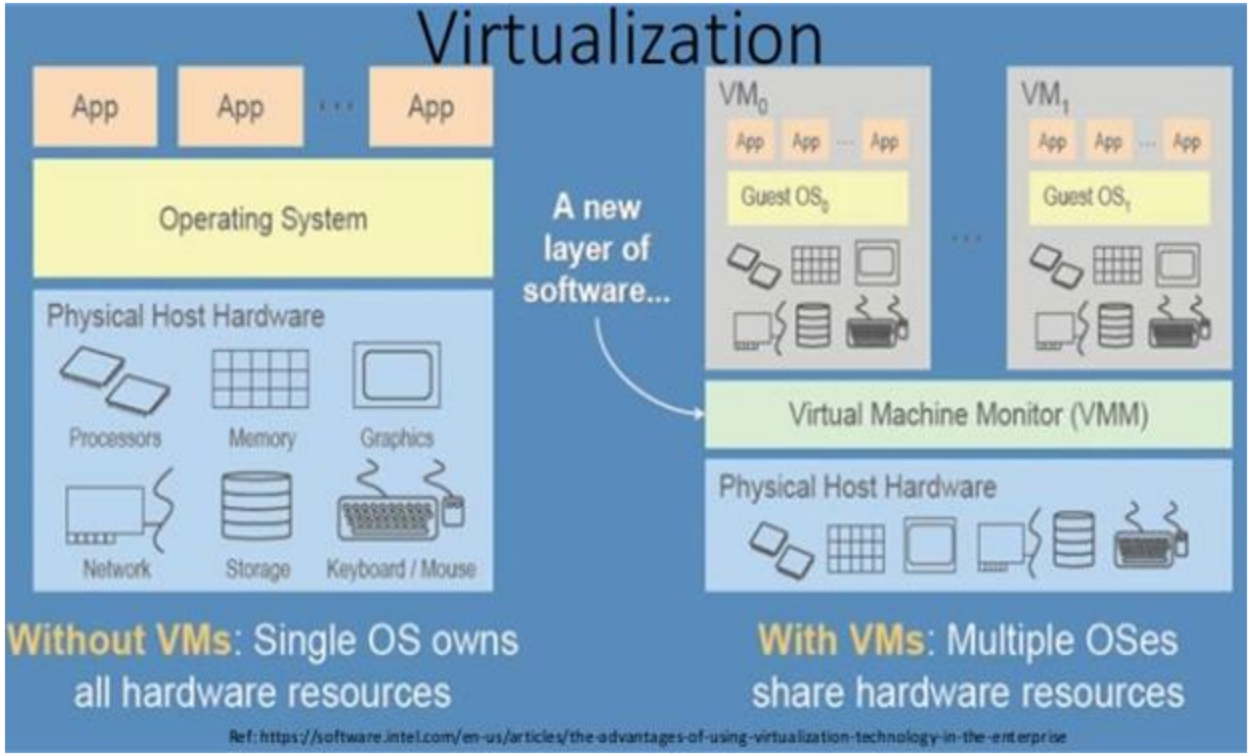
Analogue solution: virtual containers



Virtualization

- What is “Virtualization” in general?

- It is **the creation of a virtual version of *something***: an Operating System, a storage device, a network resource: pretty much almost anything can be made virtual.
    - This is done through an abstraction, that hides and simplifies the details underneath.

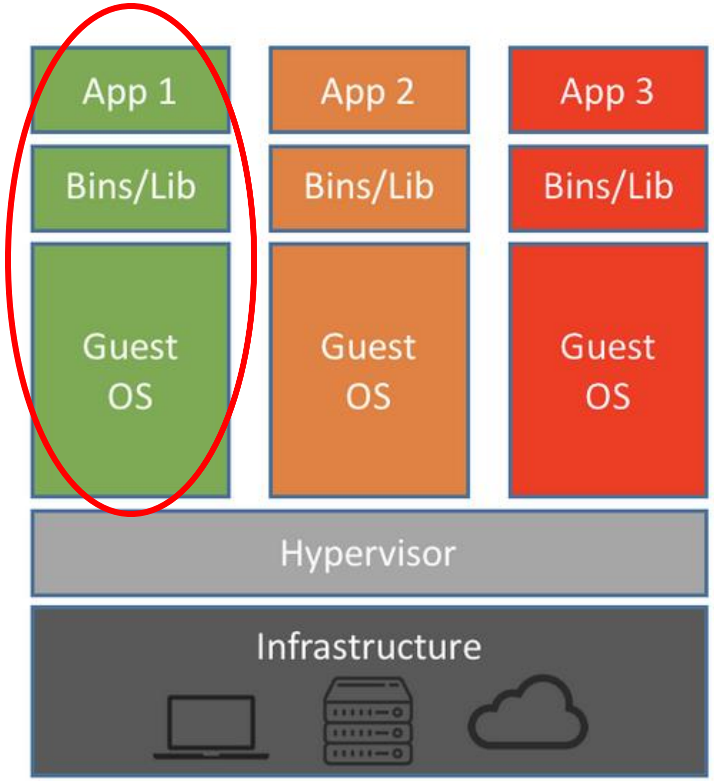


Containers

- **Containers** are an operating system virtualization technology used to **package applications** and **their dependencies** and **run them in isolated environments**.
  - They provide a lightweight method of packaging and deploying applications in a **standardized** way across many different types of infrastructure.
- Based on 2 main features of Linux kernel
  - "*control groups*" or "***cgroups***":
    - a kernel feature that allow processes and their resources to be grouped, isolated, and managed as a unit
    - cgroups provide the functionality needed to **bundle processes together as a group** and limit the resources they can access
  - **Namespaces** limit what processes can see of the rest of the system.
    - Processes running inside namespaces are not aware of anything running outside of their namespace.
- **How Do Containers Work?**
  - To understand how containers work, it is sometimes helpful to discuss **how they differ from virtual machines.**

What are VMs?

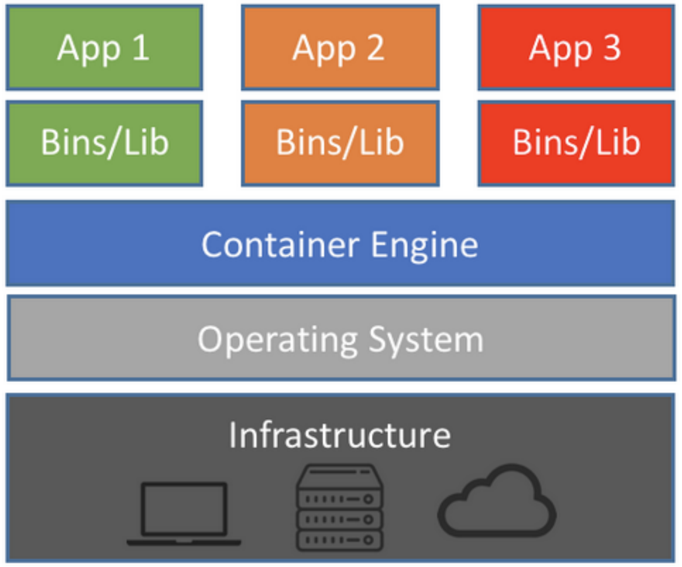
- As server processing **power and capacity increased**, bare metal applications **weren't able to exploit** the new abundance in resources.
  - Thus, **VMs were born**, designed by **running software on top of physical servers** to emulate a particular hardware system.
  - A **hypervisor (VMM)** - > is software, firmware, or hardware that creates and runs VMs.
    - sits between the hardware and the virtual machine and is necessary to virtualize the server.
- Within each VM runs a **unique guest** OS.
  - VMs with **different operating systems** can run on the same **same physical server**



Machine Virtualization

What are containers?

- **Operating system (OS) virtualization** has grown in popularity over the last decade to enable software to run predictably and well when **moved from one server environment to another**.
  - containers provide a way to run these isolated systems on a single server or host OS.
  - containers sit on top of a physical server and its host OS
    - shares the host OS kernel, the binaries and libraries
    - Shared components are read-only =>"light"
    - reduce management overhead as they share a common OS operating system

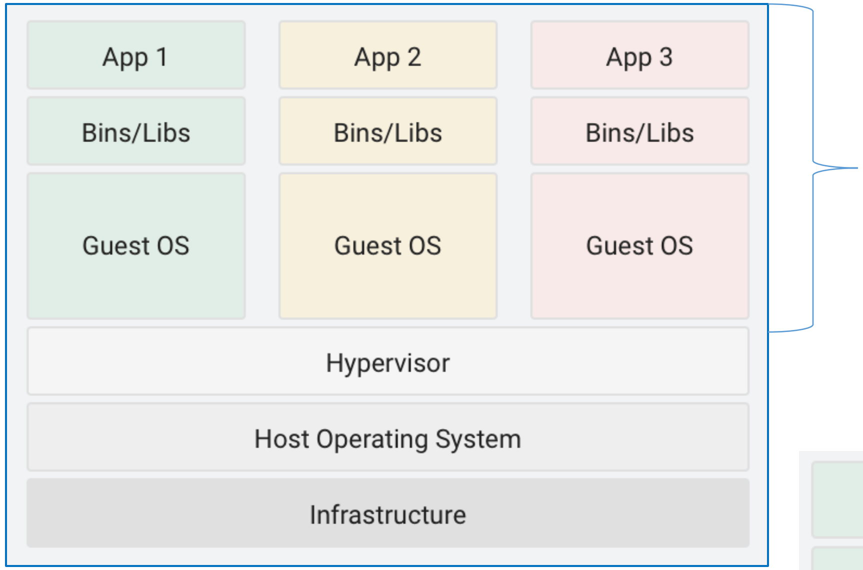


Containers

- Differences
  - **Containers** provide a way to **virtualize an OS** so that multiple workloads can run on a single OS instance
  - VMs, the hardware is being virtualized to run multiple OS instances

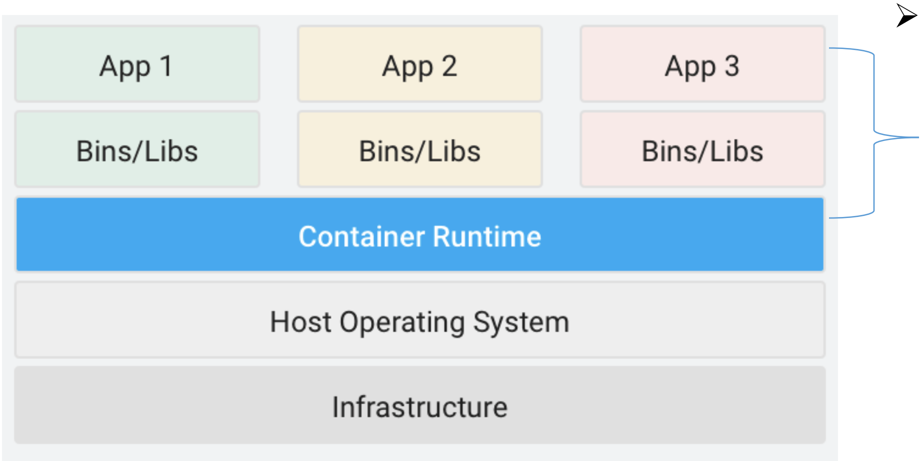
Going beyond …. Virtual Machines

Virtual Machines (VMs) carry quite some overhead with them -> introducing **Containers**



Virtual Machine

- Each virtualized application includes not only the **application** — which may be only 10s of MB — and the necessary **binaries** and **libraries**, but also an **entire guest operating system** — which may weigh 10s of GB.

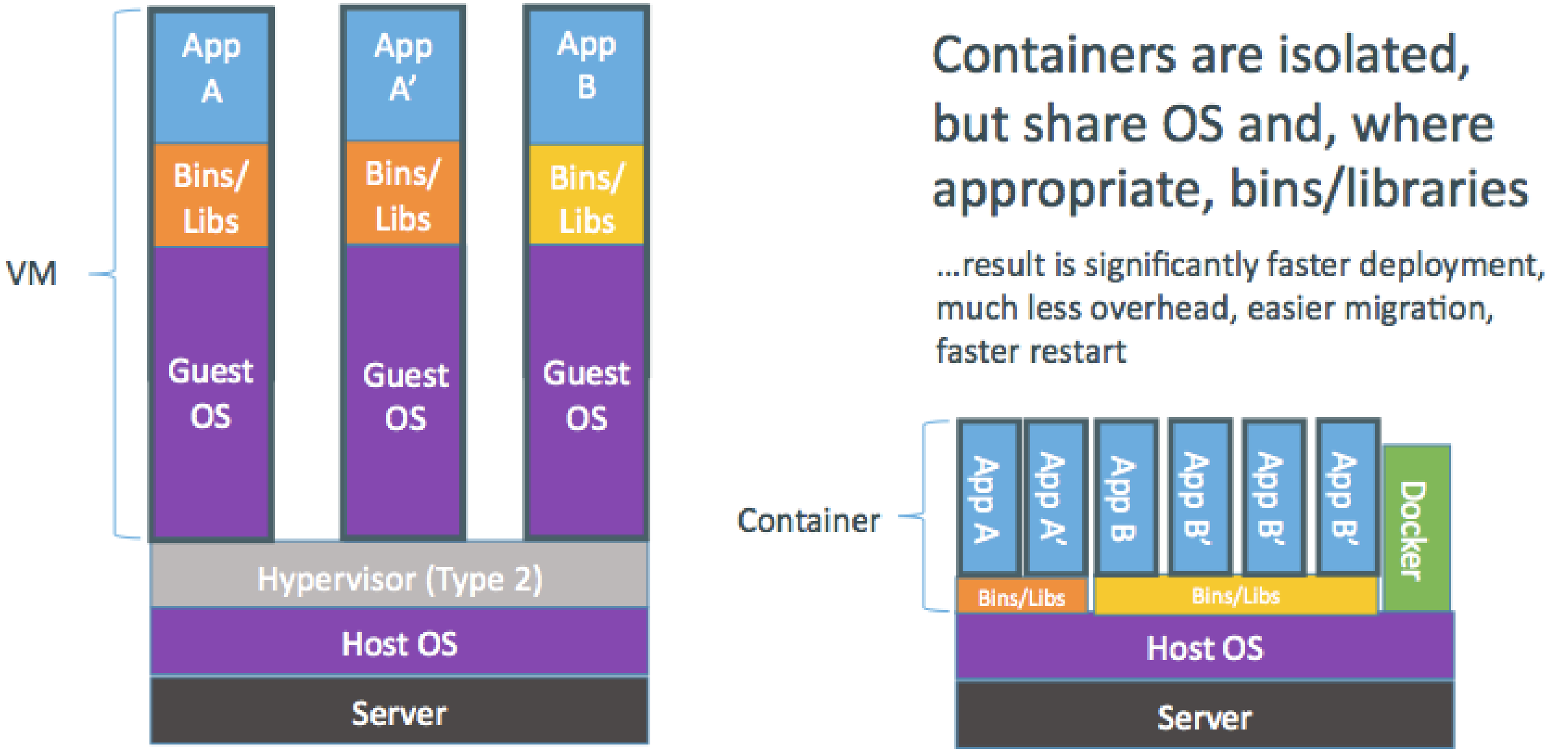


Container

- comprises just the **application** and its **dependencies**. It runs as an **isolated process** in userspace on the host operating system, **sharing the kernel** with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

Containers are «lightweight VMs»

*A container is a **standard unit of software** that packages up **code and all its dependencies,** so the application runs quickly and reliably from one computing environment to another*



Source: http://goo.gl/4jh8cX

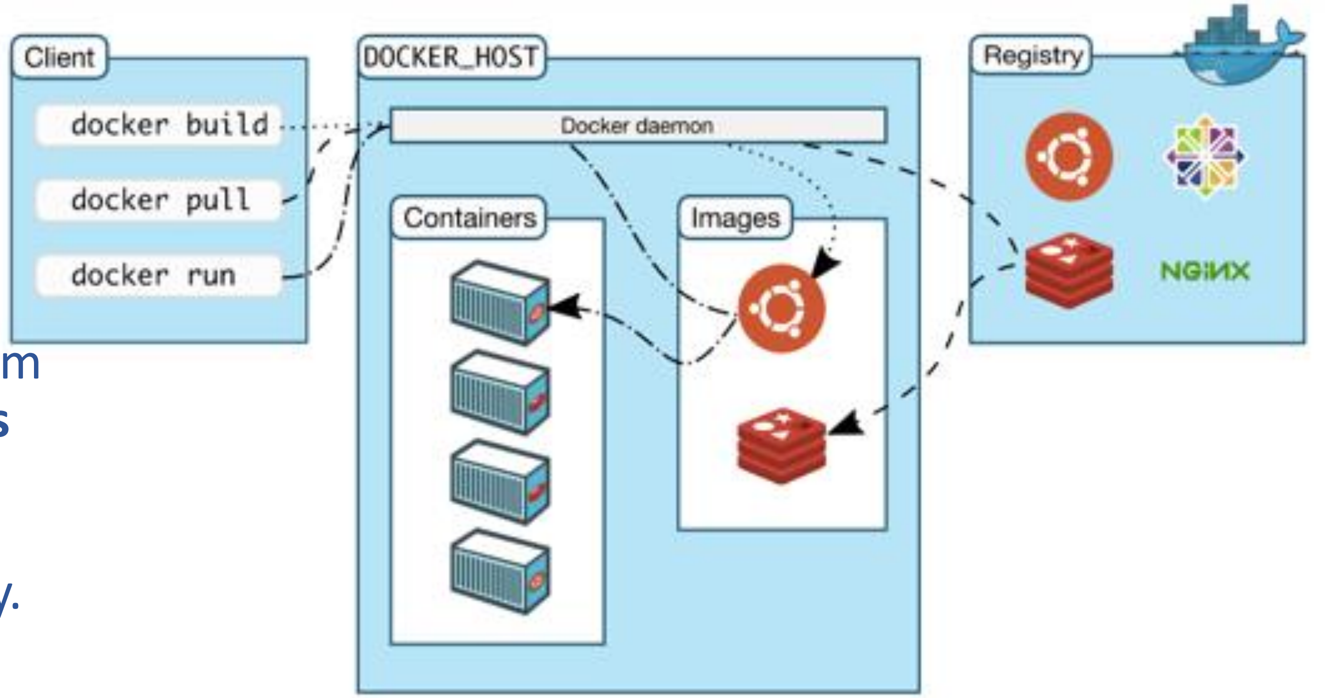
“Lightweight”, in practice

- **Containers require less resources**: they start faster and run faster than VMs, and you can fit many more containers in a given hardware than VMs.
- **Very important**: they provide enormous simplifications to software development and deployment processes, because they allow to simply encapsulate applications in a controlled and extensible way.
- Provide a uniformed wrapper around a software package:
  - *«Build, Ship and Run Any App, Anywhere»*

“Similar to shipping containers: The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships, …”

Docker (1)

- Docker is an open-source platform that automates the development and deployment of applications inside portable and self-sufficient software “containers”.
  - Like virtualenv for Python
- Main components:
  - ***Docker Engine***
    - portable **runtime** and packaging system that gives **standardized environments** for the development and flexibility for workload deployment so that it is not restricted by infrastructure technology.
  - ***Docker Hub***
    - Docker Hub is a cloud solution for sharing apps and automating workflows.



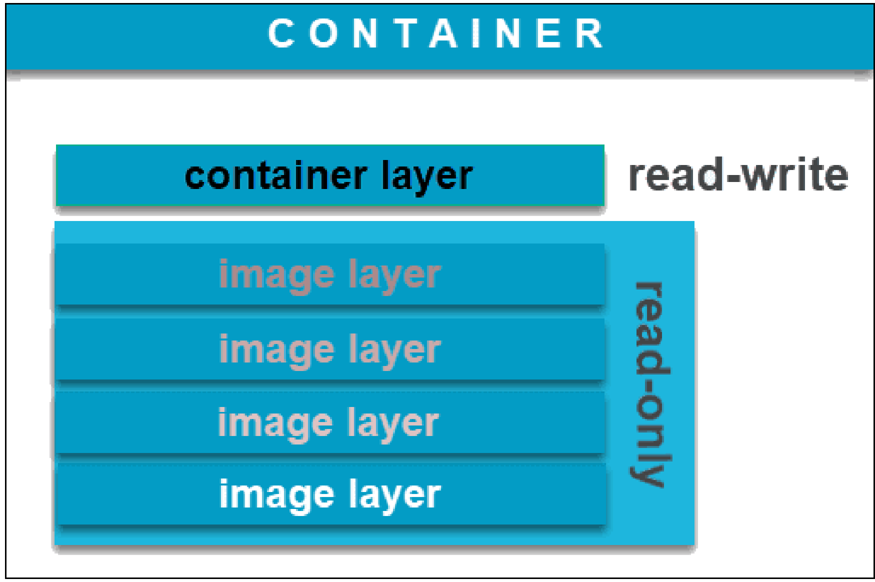
Docker (2)

- Enhanced to orchestrate multi-container distributed application:
  - ***Docker Compose*** - a tool for defining and running **multi-container** *Docker* applications.
    - transforms complex and time-consuming procedure of deployment into a simple one.
    - A small YAML configuration file allows assembling apps from discrete Docker containers and deploy them very quickly, independently of any underlying infrastructure.
  - ***Docker Machine*** a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them)
    - Flexibility in host provision provides quicker iterations and compressing the development-to-deployment cycle.
  - ***Docker Swarm* -** is a group of either physical or virtual machines that are running the **Docker** application and that have been configured to join together in a cluster
    - activities of the cluster are controlled by a **swarm** manager, and machines that have joined the cluster are referred to as nodes
    - *It* is an orchestration management tool that runs on Docker application

Containers vs. Images

- “A ***container image*** *is a lightweight*, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.”
  - A **Docker image** is an **immutable (unchangeable)** file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.
    - They are **templates, read-only, cannot run**
    - **Container is a running image**

Images can exist without containers, whereas a container needs to run an image to exist



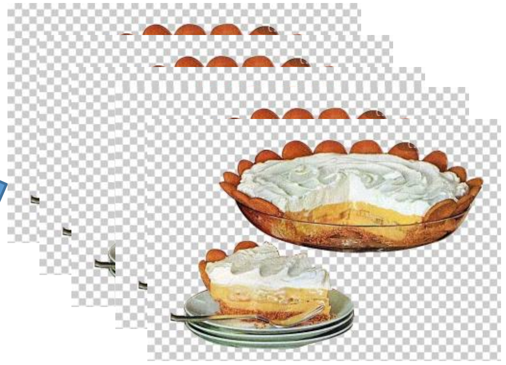
Docker container

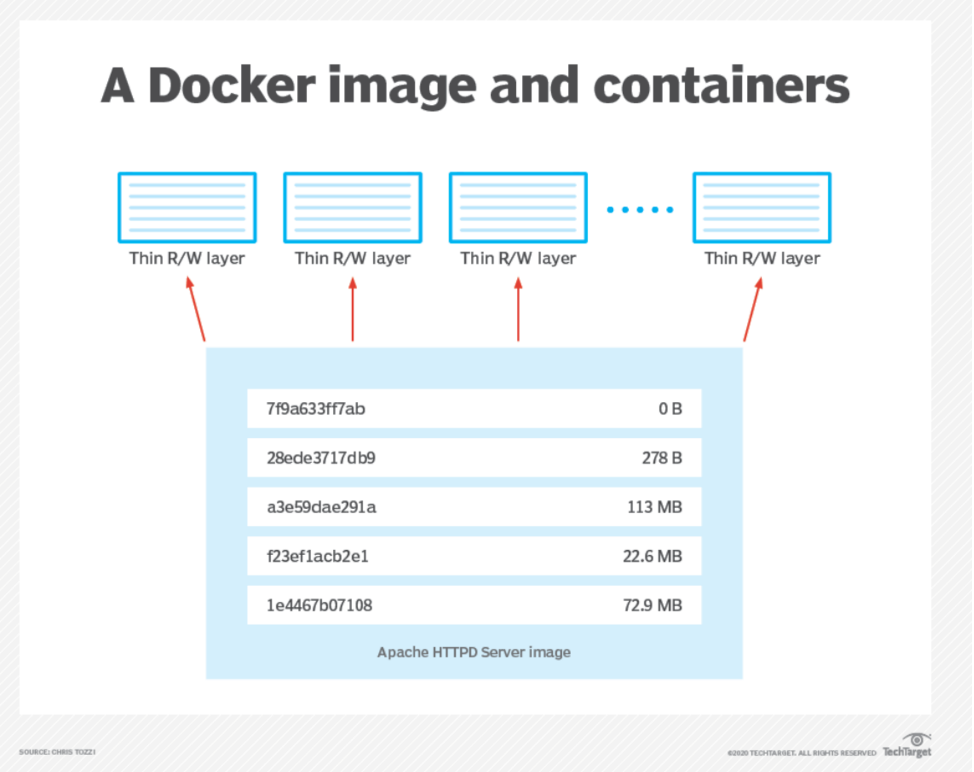
- **From** a container **image**, you can **start a container** based on it. Docker containers are the way to execute that package of instructions in a runtime environment
- Containers run until they fail and crash, stopped.
  - does not change the image on which it is based
- Docker image = **recipe** for a cake
- and a container = **cake** you baked from it.


Simple Cake Recipe

225g (8 oz) self-raising flour.
225g (8 oz) soft butter (i.e. room temperature).
225g (8 oz) caster sugar.
4 eggs.
1 teaspoon baking powder.

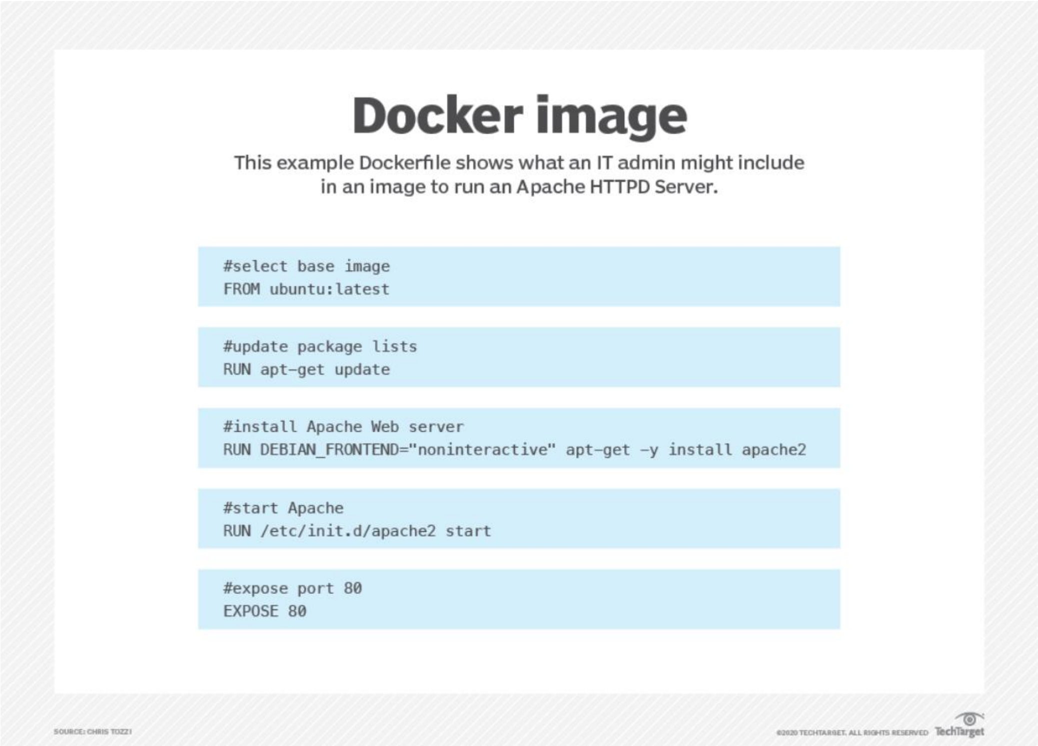
Mix the ingredients well in a large bowl using an electric whisk.
Halve the mixture and pour into 2 non-stick 18cm (7 inch) cake tins.
Cook till golden brown (15-25 minutes) in a preheated oven at 180 degrees C (gas mark 4).
Cool on a wire rack before serving, add jam between the two halves and optionally top with butter cream.





Docker Image

- It is a **set of instructions** that defines what should run inside a container.
- A Docker image typically specifies:
  - Which **external image to use as the basis** for the container, unless the container image is written from scratch;
  - **Commands** to run when the container starts;
  - How to **set up the file system** within the container; and
  - **Additional instructions**, such as which ports to open on the container, and how to import data from the host system.



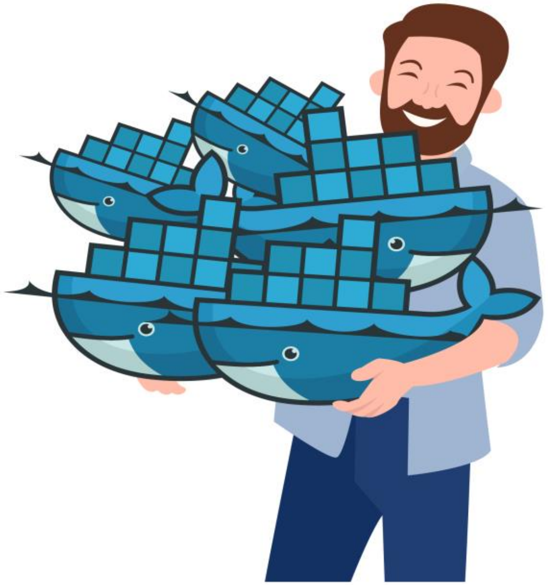
Container terminology

- **Container:**
  - In Linux, containers are an ***operating system virtualization technology*** used to package applications and their dependencies and run them in isolated environments.
- **Container Image:**
  - Container images are ***static files*** that define the filesystem and behavior of specific container configurations. Container images are used as a template to create containers.
- **Docker:**
  - Docker was the first technology to successfully popularize the idea Linux containers.
  - Among others, Docker's ecosystem of tools includes ***docker***, a container runtime with extensive container and image management features, ***docker-compose***, a system for defining and running multi-container applications, and ***Docker Hub***, a container image registry.
- **Linux cgroups:**
  - or control groups, are a kernel feature that **bundles processes together** and **determines their access to resources**. Containers in Linux are implemented using cgroups in order to manage resources and separate processes.
- **Linux namespaces:**
  - a kernel feature designed to **limit the visibility** for a process or cgroup to the rest of the system. Containers in Linux use namespaces to help isolate the workloads and their resources from other processes running on the system.
- **LXC:**
  - LXC is a form of Linux containerization that predates Docker and many other technologies while relying on many of the same kernel technologies. Compared to Docker, LXC usually virtualizes an entire operating system rather than just the processes required to run an application, which can seem more similar to a virtual machine.
- **Virtual Machines:**
  - Virtual machines, or VMs, are a hardware virtualization technology that emulates a full computer. A full operating system is installed within the virtual machine to manage the internal components and access the computing resources of the virtual machine.

Docker for different OS

Supported OS:

- https://docs.docker.com/engine/install/
  - **Windows**: https://docs.docker.com/docker-for-windows/install/
  - **Linux**:
    - for RedHat see https://docs.docker.com/install/linux/docker-ce/centos
  - **MacOS**: https://docs.docker.com/docker-for-mac/



Some… hands-on

The test infrastructure for this course

- Each of you has access to a VM with CentOS 7, a public IP address, 4GB RAM, a 20GB disk and 2 Virtual CPU.
  - You have root permission on that machine
- Link to the material in Baltig related to the hands-on
  - https://baltig.infn.it/corso-olss-2020/corso_olss_2020/-/tree/master/containers
- **You should now log on to your VM.**
  - We will use it for the hands-on on containers and in other lectures during this course.
    - Linux/Mac OS:
      - ***ssh –i*** *<private_key> -l centos devopsX.cloud.cnaf.infn.it*
    - Windows:
      - https://devops.ionos.com/tutorials/use-ssh-keys-with-putty-on-windows/

Check hands-on environment

- To avoid specifying `sudo` before each docker command, the user "centos" was added to the docker Unix group. Check it:

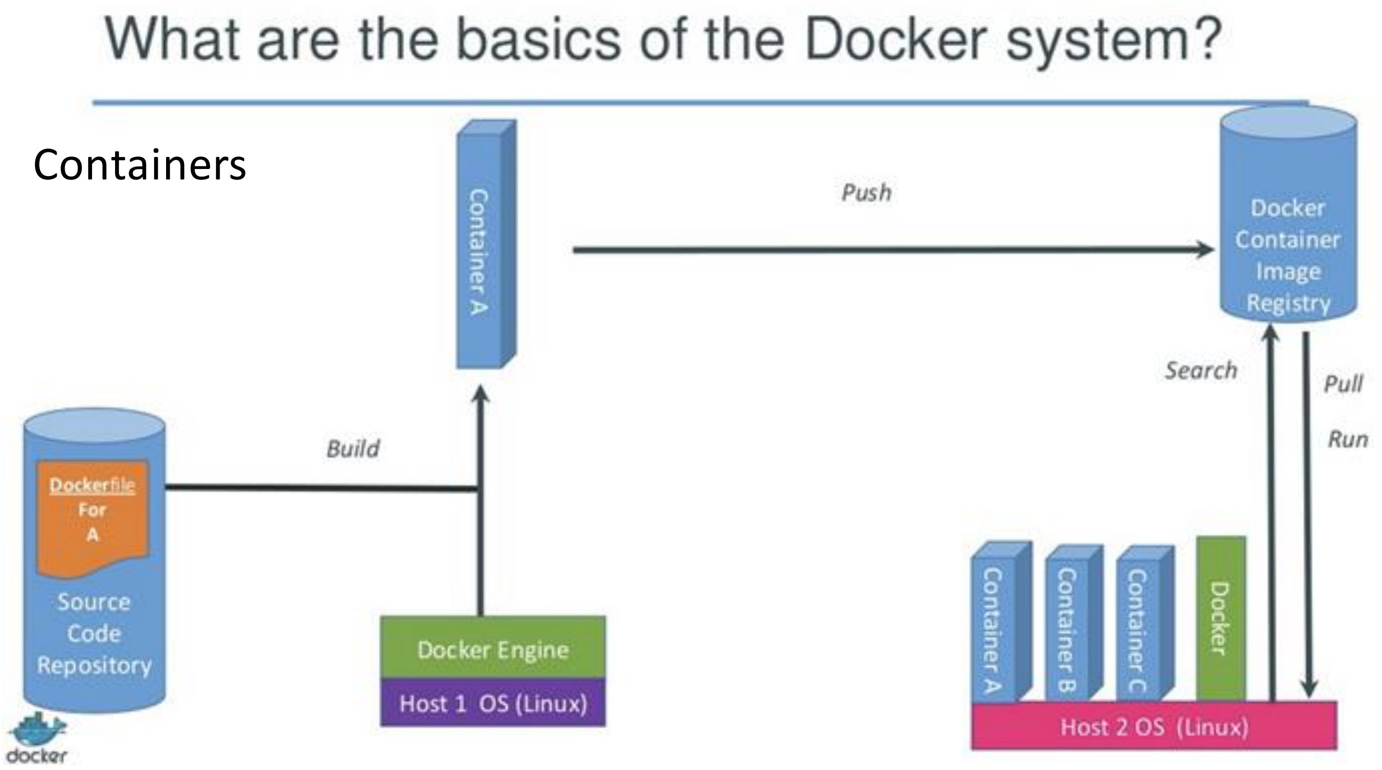
```
centos@VM1:~$ id
(where do you see it?)

centos@VM1:~$ docker info
[…]
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
[…]
```

The first docker commands

- By default, the "container image registry" on the left is the service running at https://hub.docker.com (called "**Docker Hub**"). It stores more than 100,000 container images.

- **To download** a container **image** from Docker Hub, use the command "***docker pull***".
- **To run** a **container**, use the command "***docker run***".



Search, pull, run

- **Try these commands on your VM**:
  - **Search** for a container image at Docker Hub:
    - `$ docker search ubuntu` (or e.g. `docker search rhel` – what would this do?)
  - Fetch (**pull**) a Docker image (in this case, an Ubuntu container image):
    - `$ docker pull ubuntu`
  - List images
    - `$ docker images`
  - Execute (**run**) a docker container:
    - Run the "echo" command inside a container and then exit:
      - ```
        $ docker run ubuntu echo "hello from the container"
        hello from the container
        ```
    - Run a container in interactive mode:
      - `$ docker run -it ubuntu /bin/bash`
        ```

How efficient is docker?

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              7698f282e524        2 weeks ago         72.9MB
```

=> the latest Ubuntu image takes about 70MB of disk space *as a container*. If you had just to download a full Ubuntu (server) distribution, it would be more in the range of 900MB.

```
$ time docker run ubuntu echo "hello from the container"
hello from the container

real	0m1.384s
user	0m0.069s
sys	0m0.106s
```

=> The total time it takes on this system (not a really powerful one) to start a container, execute a command inside it and exit from the container is about half a second. How long would it take if we used a full VM?

How to extend a docker container (1)

- Suppose you need a command inside a container, but it is not installed in the image you pulled from Docker Hub. For example, you would like to use the `ping` command but by default it's not available:
  - ```
    $ docker run ubuntu ping www.google.com
    docker: Error response from daemon: OCI runtime create failed:
    container_linux.go:345: starting container process caused "exec: \"ping\":
    executable file not found in $PATH": unknown.
    ```
- We can install it ourselves; it is in the package `iputils-ping`:
  - `$ docker run ubuntu /bin/bash -c "apt update; apt -y install iputils-ping"`
- But it still doesn't work!?
  - ```
    $ docker run ubuntu ping www.google.com
    docker: Error response from daemon: OCI runtime create failed:
    container_linux.go:345: starting container process caused "exec: \"ping\":
    executable file not found in $PATH": unknown.
    ```
- Why? The ping command was successfully installed!

How to extend a docker container (2)

- Whenever you issue a `docker run <container>` command, a **new container** is started, based on **the original container image**.
  - Check it yourself with `$ docker ps -a` command.
- If you modify a container and then want to reuse it (which is often the case!), **you need to save the container, creating a new image**.
- So, install what you need to install (e.g. the `iputils-ping` package, using the same command as before) , and then issue a commit command like

```
$ docker commit xxxx ubuntu_with_ping
```

- This **locally commits** a container, creating an image with the name `ubuntu_with_ping` (or any other name you like). Take `xxxx` from the container ID shown by the `docker ps -a` output.
- **Do it now**.

How to extend a docker container (3)

- Verify that the `ping` command inside our new image now works:
  - **$ docker run ubuntu_with_ping ping -c 3 www.google.com**

```
PING www.google.com (216.58.216.100) 56(84) bytes of data.
64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=1 ttl=43 time=18.5 ms
64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=2 ttl=43 time=18.5 ms
64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=3 ttl=43 time=18.5 ms

--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 18.501/18.539/18.586/0.035 ms
```

- **To recap**: we have an original image (called **"ubuntu"**), downloaded from Docker Hub, and a new image (called **"ubuntu_with_ping"**), created by us extending the "ubuntu" image (i.e. installing some packages). Let's check:
  - $ docker images

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest              3e7a8818665f        11 minutes ago      97.2MB
ubuntu              latest              7698f282e524        7 days ago          69.9MB
```

Cleaning up container space

- When you don’t need some containers anymore, it’s wise to check and clean up some disk space. This is done with the **`docker system`** commands.
- Check disk space used by containers with `$ docker system df`:
  - `$ docker system df`

```
TYPE            TOTAL      ACTIVE      SIZE        RECLAIMABLE
Images          2          2           97.22MB     69.86MB (71%)
Containers      4          0           27.36MB     27.36MB (100%)
Local Volumes   0          0           0B          0B
Build Cache     0          0           0B          0B
```

- Reclaim disk space with `$ docker system prune`, then check again:
  - `$ docker system df`

```
TYPE            TOTAL      ACTIVE      SIZE        RECLAIMABLE
Images          2          0           97.22MB     97.22MB (100%)
Containers      0          0           0B          0B
Local Volumes   0          0           0B          0B
Build Cache     0          0           0B          0B
```

Removing unused images

- Besides containers, you can also remove images you don't need anymore with `docker rmi <image>`:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest              3e7a8818665f        29 minutes ago      97.2MB
ubuntu              latest              7698f282e524        7 days ago          69.9MB

$ docker rmi ubuntu_with_ping
Untagged: ubuntu_with_ping:latest
Deleted: sha256:3e7a8818665fc7eb1be20e8d633431ad8c0bdfba05d6d11d40edd32a915708bb
Deleted: sha256:a4c24b3590e4e95c30d4d0e82d3f769cde94436a5dd473b4e7ec7bd4682ce1b7

$ docker rmi ubuntu
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:f08638ec7ddc90065187e7eabdfac3c96e5ff0f6b2f1762cf31a4f49b53000a5
Deleted: sha256:7698f282e5242af2b9d2291458d4e425c75b25b0008c1e058d66b717b4c06fa9
Deleted: sha256:027b23fdf3957673017df55aa29d754121aee8a7ed5cc2898856f898e9220d2c
Deleted: sha256:0dfbdc7dee936a74958b05bc62776d5310abb129cfde4302b7bcdf0392561496
Deleted: sha256:02571d034293cb241c078d7ecbf7a84b83a5df2508f11a91de26ec38eb6122f1

$ docker system df
TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
Images              0                   0                   0B                  0B
Containers          0                   0                   0B                  0B
Local Volumes       0                   0                   0B                  0B
Build Cache         0                   0                   0B                  0B
```

Working with Docker Hub

Pushing images to Docker Hub (1)

- The command `$ docker push <image>`. This writes an image **to Docker Hub**.
  - In order to issue that command, **you first need an account on Docker Hub**: go to https://hub.docker.com and sign up (or sign in, if you already have an account there) – it's free.
  - **Do it now.**
- Click on **Create Repository**, make it public (careful: <u>everybody will be be able to see the images you upload there!</u>) and give it a name, for example **olss_*2021*** **(only lowercase is allowed)**, a description, and click on "Create". This will create your public repository, called e.g. "olss_2021".

Pushing images to Docker Hub (2)

- To push an image (for example the `ubuntu_with_ping` image we created earlier) to your new repository, we **must** give a **tag** to the image *and* specify **our Docker Hub username and repository** as part of the image name.
  - The full image name should be **<username>/<repository>:<tag>**.
  - In my case, the first part should be "*caifti/olss_2021*". As tag, you can put any string; let's set it to "ubuntu_with_ping_1.0".
  - In order to assign this tag to our existing image, find out its "image id" with the `docker images` command:

Images before the new tag

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest              7c45b9ad4de6        45 minutes ago      97.2MB
ubuntu              latest              7698f282e524        7 days ago          69.9MB
```

```
$ docker tag 7c45b9ad4de6 caifti/olss_2021:ubuntu_with_ping_1.0
```

Images after the new tag

```
$ docker images
REPOSITORY          TAG                   IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest                7c45b9ad4de6        About an hour ago   97.2MB
alexcos/olss_2021   ubuntu_with_ping_1.0  7c45b9ad4de6        About an hour ago   97.2MB
ubuntu              latest                7698f282e524        7 days ago          69.9MB
```

Pushing images to Docker Hub (3)

- Now login to Docker Hub with your username and password:
  - `$ docker login`

```
Login with your Docker ID to push and pull images from Docker
Hub. If you don't have a Docker ID, head over to
https://hub.docker.com to create one.
Username:
Password:
WARNING! Your password will be stored unencrypted in
/home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

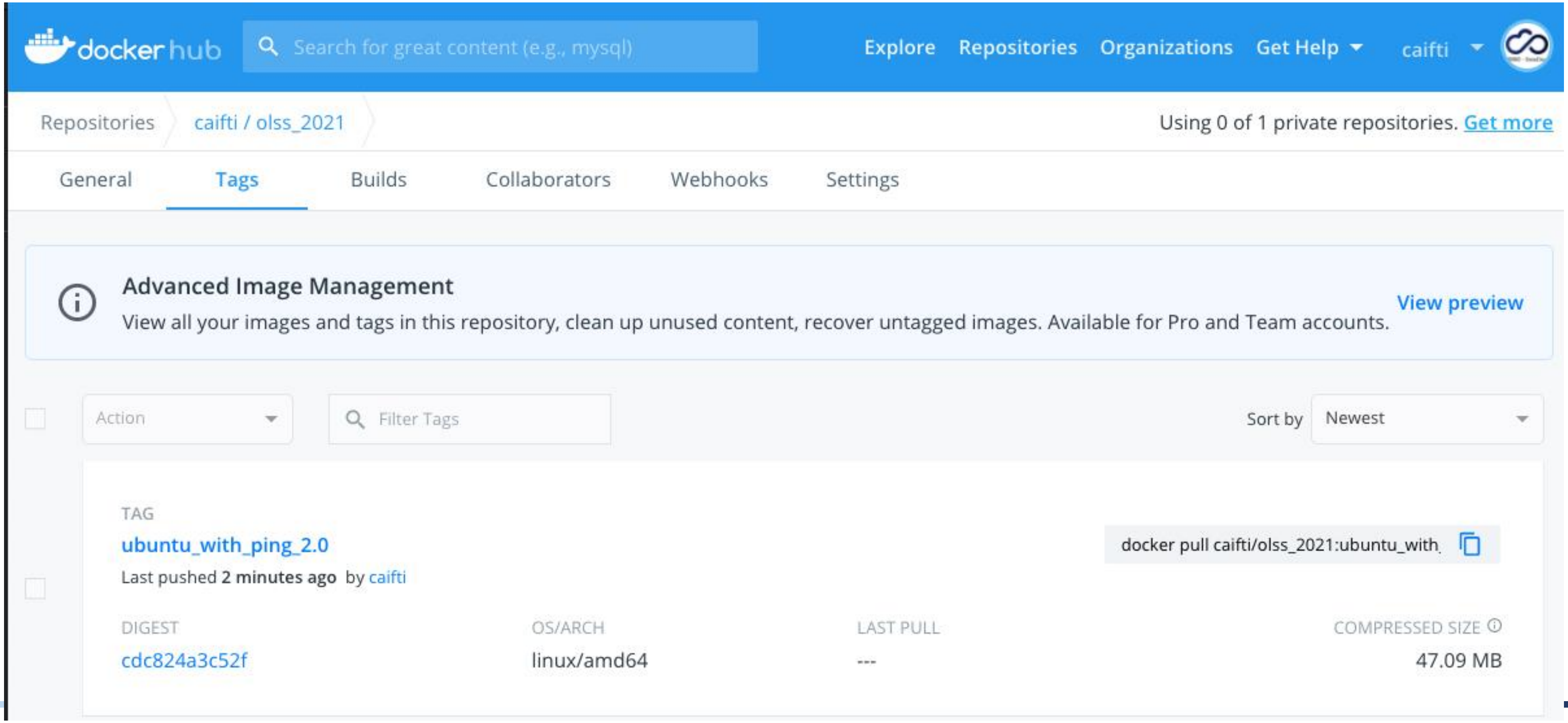
Login Succeeded
```

We'll disregard this warning here. For more info, see the URL in the message.

- Finally, we can push our image to Docker Hub:
  - `$ docker push caifti/olss_2021:ubuntu_with_ping_1.0`

Verifying our Docker Hub repository

- Go to Docker Hub (https://hub.docker.com/), login with **your username**, click on the ***"olss_2021"*** repository, and then on "Public View". You should see something like this:



Handling multiple commands

- If you have **several commands to apply to a container** (for example, you want to install many applications), you could run the container in interactive mode as shown earlier (use the “-i” switch), and then issue the various commands at the prompt once you are in the container.
  - For example, when you are running a **container interactively**, you could issue a sequence of commands such as

```
# apt update
# apt install -y wget unzip
# wget <some_file>
# unzip <some_other file>
…
```

- Once you exit from the container, remember to **commit** the container, or your modifications to the container will be lost (like in our “ping” example earlier).

Build images

Dockerfiles

- Rather than modifying a container “by hand”, connecting interactively and then installing packages as previously shown, it is often much more convenient to **put all the required commands in a text file (called by default Dockerfile)**, and then **build** an image executing these commands.
- As an example, through the following `Dockerfile` we **create** an image starting from an Ubuntu image, **installing** a web server (through the `apache2` package) and telling the image to **serve** a simple html page (`index.html`), which we copy from our system:

```
$ cat Dockerfile

FROM ubuntu
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update
RUN apt install -y apache2
COPY index.html /var/www/html/
EXPOSE 80
CMD ["apachectl", "-D", "FOREGROUND"]
```

This Dockerfile:
- Starts from the Ubuntu container
- Updates all installed packages
- Installs the apache2 web server
- Copies an index.html file from our system
- Exposes port 80 (the standard web port)
- Starts the apache2 web server through the "apachectl" command

The `index.html` file

- This is the `index.html` file we used in the previous Dockerfile. It will just show a greeting message:
- ```
  $ cat index.html
  <!DOCTYPE html>
  <html>
  <h1>Hello from a web server running inside a container!</h1>
  This is an exercise for the OLSS2021 course.
  </html>
  ```
- **Create (or download)** both the previous `Dockerfile` and the `index.html` file in your home directory.
  - `$ wget https://baltig.infn.it/corso-olss-2020/corso_olss_2020/-/raw/master/containers/Dockerfile`
  - `$ wget https://baltig.infn.it/corso-olss-2020/corso_olss_2020/-/raw/master/containers/index.html`
  ```

Build images via Dockerfiles

- Once we have a Dockerfile, we can create (”build”) an image and name it for example “web_server” with the command

```
$ docker build -t web_server .
```

  - **Note**: the **.** at the end the line above is important!

- We can now run our new container in the background (flag `-d`) simply with

```
$ docker run -d -p 8080:80 --name=web_server web_server
```

- The `-p 8080:80` part redirects port 80 *on the container* (the port we exposed in the Dockerfile) to port 8080 *on the host system* (that is, VM1).

- Check that everything works opening in a browser the page http://<VM1_ip_address>:8080/
- **Try it now!**

Check that our web server is running

- Check with:

```
$ docker ps
CONTAINER
ID          IMAGE                    COMMAND                      CREATED              STATUS                 P
ORTS                    NAMES
f9dc164be001        web_server           "apachectl -D FOREGR…"   12 minutes ago       Up 12
minutes        0.0.0.0:8080->80/tcp   laughing_pare
```

- Stop the container with:

```
$ docker stop f9dc164be001
```

- You can now type

```
$ docker run -d -p 8080:80 web_server
```

to instantiate a new web server.

What happens if you type

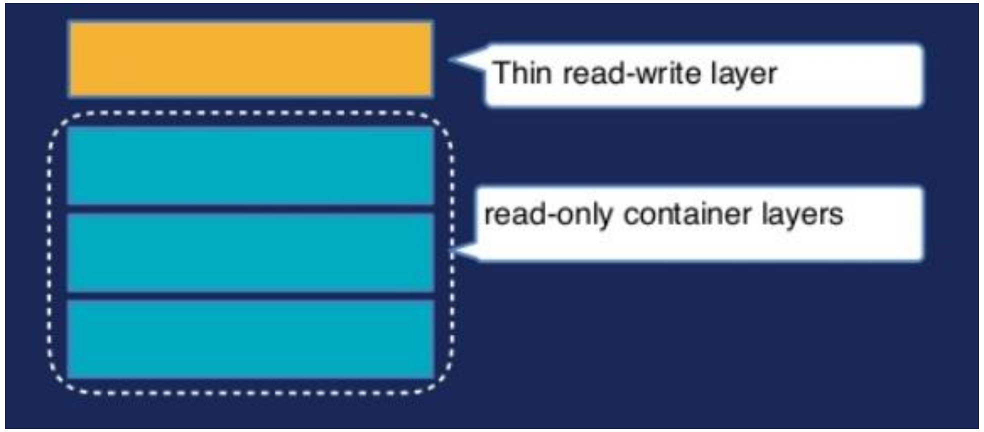
```
$ docker run -d -p 8081:80 web_server
```

?

Docker layers

Container Layers

- Dokerfile
  - A series of instruction for building images
  - Each Dockerfile command creates a Layer
  - Only ADD, RUN and COPY influence the size of the image
- Container layers
  - From image to container



- $ cat Dockerfile

```
FROM ubuntu
  ENV DEBIAN_FRONTEND=noninteractive
  RUN apt update
  RUN apt install -y apache2
  COPY index.html /var/www/html/
  EXPOSE 80
  CMD ["apachectl", "-D", "FOREGROUND"]
```

Image building process

```
$ docker build -t web_server .
Sending build context to Docker daemon  3.072kB
Step 1/7 : FROM ubuntu
latest: Pulling from library/ubuntu
a70d879fa598: Pull complete
c4394a92d1f8: Pull complete
10e6159c56c0: Pull complete
Digest:
   sha256:3c9c713e0979e9bd6061ed52ac1e9e1f246c9495aa06361
   9d9d695fb8039aa1f
Status: Downloaded newer image for ubuntu:latest
---> 26b77e58432b
Step 2/7 : ENV DEBIAN_FRONTEND=noninteractive
---> Running in 02d8ddcd78de
Removing intermediate container 02d8ddcd78de
---> be8e0da46ada
Step 3/7 : RUN apt update
---> Running in bd5a1f1828c9
Removing intermediate container bd5a1f1828c9
---> fdfd77a871e9
Step 4/7 : RUN apt install -y apache2
---> Running in 0af7fad77c22Removing
intermediate container 0af7fad77c22
---> e7748d3c4880
Step 5/7 : COPY index.html /var/www/html/
---> 1e624144130e
Step 6/7 : EXPOSE 80
---> Running in dc2da127dfd4
Removing intermediate container dc2da127dfd4
---> 0e61f433ab18
Step 7/7 : CMD ["apachectl", "-D",
   "FOREGROUND"]
---> Running in a5829e92006a
Removing intermediate container a5829e92006a
---> 62d58cfe544e
Successfully built 62d58cfe544e
Successfully tagged web_server:latest
```

Inspect image building

```
$ docker images
REPOSITORY                     TAG       IMAGE ID       CREATED          SIZE
web_server                     latest    62d58cfe544e   5 minutes ago    214MB
ubuntu                         latest    26b77e58432b   2 weeks ago      72.9MB
hello-world                    latest    d1165f221234   6 weeks ago      13.3kB

$ docker history 62d58cfe544e
IMAGE          CREATED          CREATED BY                                       SIZE      COMMENT
62d58cfe544e   8 minutes ago    /bin/sh -c #(nop)  CMD ["apachectl" "-D" "FO…    0B
0e61f433ab18   8 minutes ago    /bin/sh -c #(nop)  EXPOSE 80                     0B
1e624144130e   8 minutes ago    /bin/sh -c #(nop) COPY file:6bbba72179c3da84…    137B
e7748d3c4880   8 minutes ago    /bin/sh -c apt install -y apache2                113MB
fdfd77a871e9   9 minutes ago    /bin/sh -c apt update                            27.9MB
be8e0da46ada   9 minutes ago    /bin/sh -c #(nop)  ENV DEBIAN_FRONTEND=nonin…    0B
26b77e58432b   2 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B
<missing>      2 weeks ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…    7B
<missing>      2 weeks ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]      0B
<missing>      2 weeks ago      /bin/sh -c set -xe    && echo '#!/bin/sh' > /…   811B
<missing>      2 weeks ago      /bin/sh -c #(nop) ADD file:27277aee655dd263e…    72.9MB
```

Reduce Layers

- More layers mean a larger image
  - The larger the image, the longer that it takes to build, push and pull
- Smaller images mean faster builds and deploys

- How reduce layers
  - Use shared base images (where possible)
  - Limit the data written on the container layers
  - Chain RUN statemets

- Some links
  - https://dzone.com/articles/docker-layers-explained
  - https://stackoverflow.com/questions/32738262/whats-the-differences-between-layer-and-image-in-docker

Docker volumes

Containers are *ephemeral*

- **An important point to remember** is that any data that is created within a running container is **only available within the container**, and only when the container is running.
- Let’s prove this. Run a container using the Ubuntu image in interactive mode:

```
$ docker run -it ubuntu /bin/bash
```

- Once in the container, create a file and verify it is there:

```
root@2000824922fb:/# touch my_new_file  # this creates an empty file in the container file system
root@2000824922fb:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  my_new_file  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@2000824922fb:/#
```

- Now exit from the container. Run it again with the same command as above `$ docker run -it ubuntu /bin/bash`
- Is the file still there? (it should not!)
  - It is not there because every time you do `docker run` above you start a new Ubuntu container.

Connect a container to a host file system

- So, what if we want to retain data within a container?
- We can map a directory that is available *on the host* (the system where we run the docker command, e.g. VM1), to a directory that is available *on the container*. This is done with the docker flag `-v`, like this:

```
$ docker run -v /host/directory:/container/directory <other docker arguments>
```

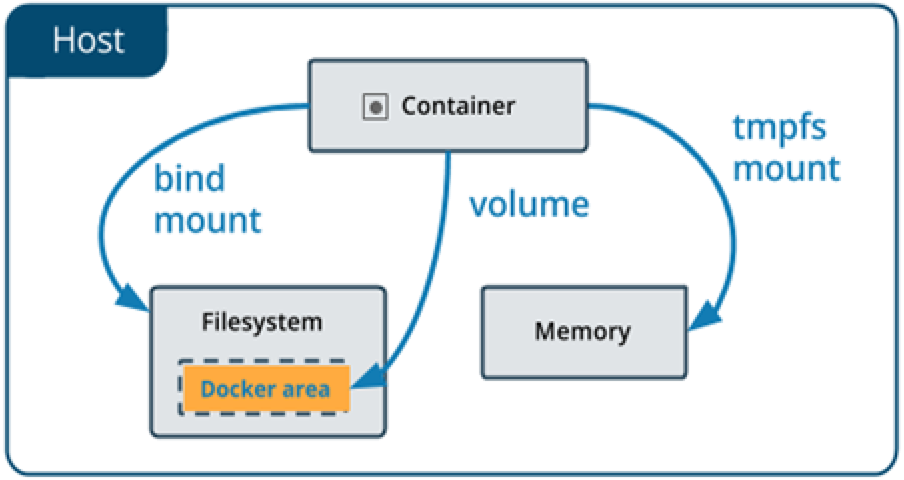
- So, for example, create a directory `local_data` . Let’s map this directory to the directory `/cointainer_data` on the container:

```
$ docker run -v /<path_to>/local_data/:/container_data -it ubuntu /bin/bash
```

- Now, when you are *within the container*, if you write `ls /container_data` you should see files from `local_data`. **Do it now**.

Docker volume (1)

- In the previous slide, we mapped a directory that was available *on the host* to a directory on the container.
- But what if we want to copy or move our docker container to a different host, with a different directory structure? Or perhaps with a different operating system? Remember that Docker promises to be system-independent.
- We can (and should generally prefer to) use **Docker volumes**.



Docker volume (2)

- Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.
- While **bind mounts** are dependent on the directory structure and OS of the host machine, **volumes** are completely managed by Docker. Volumes have several advantages over bind mounts:
  - Volumes are easier to back up or migrate than bind mounts.
  - You can manage volumes using Docker CLI commands or the Docker API.
  - Volumes work on both Linux and Windows containers.
  - Volumes can be more safely shared among multiple containers.
- Volumes are often a better choice than persisting data in a container’s writable layer
  - a volume does not increase the size of the containers using it.

Connect a container to a Docker volume

- You can create a new Docker volume with the command

```
$ docker volume create my-volume
```

  - Try these self-explanatory commands:

```
$ docker volume ls
$ docker volume inspect my-volume
```

- You can also start a container with a volume which does not exist yet with the `-v` flag. It will be automatically created:

```
$ docker run -it --name myname -v my-volume:/app ubuntu /bin/bash
```

  - Notice that we also introduced here **the flag `--name`** to give an explicit name (here: `myname`) to a container.
  - In this case, check the volume with the command `docker inspect myname` and look for the `Mounts` section. **Try it now**: what do you see?

Removing docker volumes

- As we said, Docker volumes are **directly managed by Docker**, in some Docker-specific area (see the `docker inspect` command we used earlier to know more). They use up space in the local file system.
- When you do not need a docker volume anymore, it is wise to reclaim its space:

  `$ docker volume rm <volume_name>`

  - Can you remove a volume which is being used by a container? Try.
  - You can also use (unused volumes)

    `$ docker volume prune`

    Note that the previous command `$ docker system prune` does not remove volumes!

Docker-compose

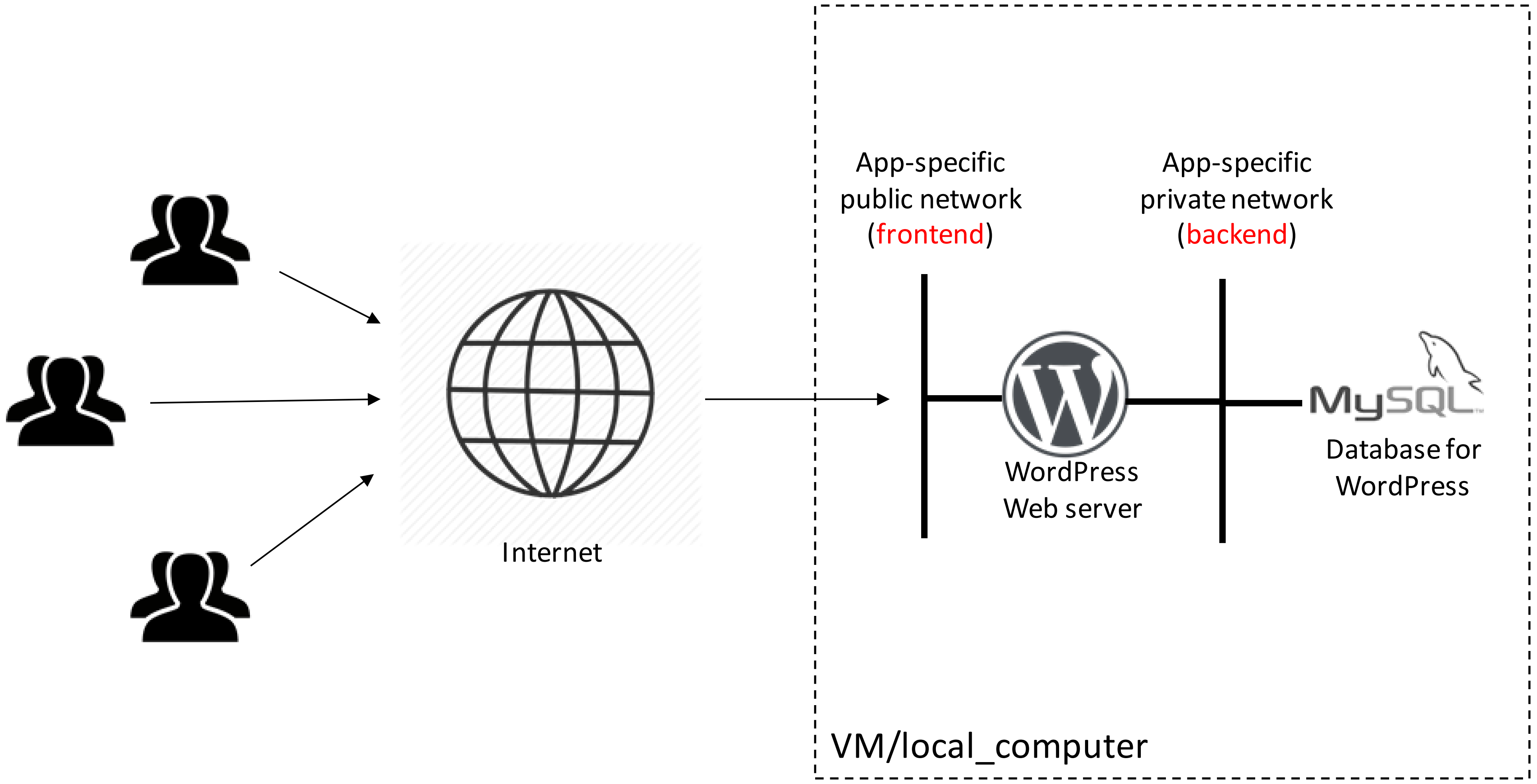
Application stacks: docker-compose

- We have seen how **easy** it is to create and run a Docker container, pulling images from Docker Hub.
  - We then learned how to **extend** an image, either manually adding packages to it (and then committing the changes), or writing a Dockerfile to automatize the process. We now also know how to export an image to a tar file, for example because we want to share it without using Docker Hub, or to save it for backup purposes.
- We will now move on to consider how to create "**application stacks**": that is, how to create multiple containers linked together to provide a multi-container service, all on a single VM.
- This is done via the `docker-compose` command.

A scenario for docker-compose

- `docker-compose` works by parsing a text file, written in the YAML language (see https://yaml.org for more info). This file, which is normally called `docker-compose.yml`, defines how our application stack is structured.
- We will now use `docker-compose` to create and launch an application stack made of two connected containers, both running on VM/your_local_computer:
  1. A **MySQL database**. It won't be accessible from the Internet.
  2. A **WordPress instance**. It will be accessible from the Internet. WordPress (https://wordpress.org) is a very popular (open source) software used to create websites or blogs.

Our app stack architecture



docker-compose.yml

```
version: '3'
services:
  database:
    image: mysql:5.7
    environment:
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=olss_passwd
      - MYSQL_DATABASE=wordpress
      - MYSQL_RANDOM_ROOT_PASSWORD=true
    networks:
      - backend
```

This builds the container for the database, with only the "backend" network

Container image for mySQL (from Docker Hub)

Configuration variables for the container software

"Obvious" note: although this is just for a demo, do not use the passwords shown in this screen!

```
  wordpress:
    image: wordpress
    depends_on:
      - database
    environment:
      - WORDPRESS_DB_HOST=database
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=olss_passwd
      - WORDPRESS_DB_NAME=wordpress
    ports:
      - 8080:80
    networks:
      - backend
      - frontend

networks:
  backend:
    driver: bridge
  frontend:
    driver: bridge
```

This builds the container for WordPress, with both the "backend" and "frontend" networks

Container image for WordPress (from Docker Hub, latest)

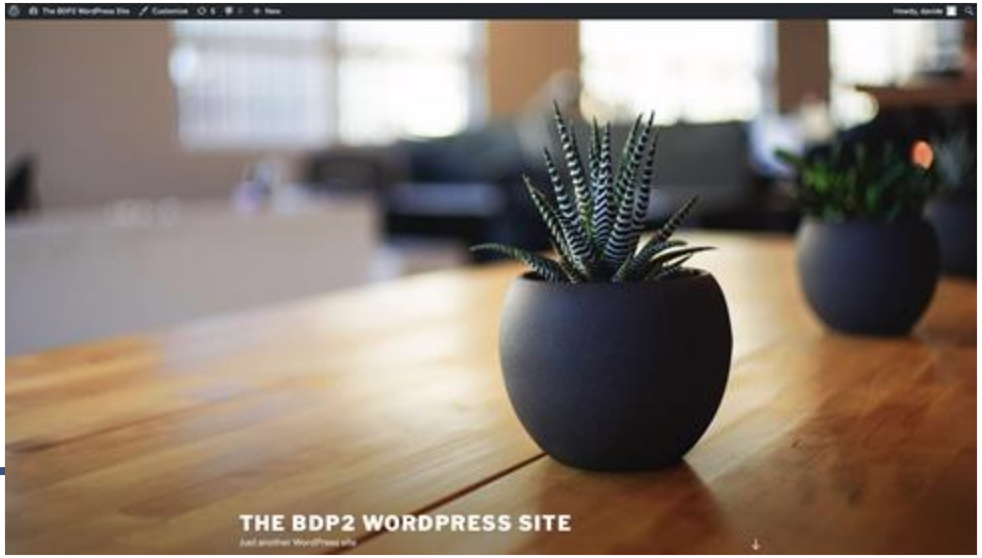
Note that here we refer to the other container

Port 8080 on the host (VM) is mapped to port 80 on the container

Build & run the application stack

- Check if docker-compose is available
  `$ docker-compose --version`
- On VM1, create or download the docker-compose.yaml
  `$ wget https://baltig.infn.it/corso-olss-2020/corso_olss_2020/-/raw/master/containers/docker-compose.yaml`
- On VM1, build the application stack:
  `$ docker-compose up --build --no-start`
- Now start it:
  `$ docker-compose start`
- *If you now open a browser pointing to VM1's public address on port 8080 (look at the previous `docker-compose.yml`), you should get the set up page for WordPress on the right. Go on and set it up.*

- Once WordPress is set up, you should see the default WordPress home page, similar to the one on the right (which of course you can graphically customize).
- Once the app stack is started, the running containers can be seen with the usual `docker ps` command.
- The application stack can be stopped with:
  `$ docker-compose stop`
- **Try it yourself now.**



Stop and Delete your services

- If you want to stop the services

```
$ docker-compose stop
Stopping costa_wordpress_1 ... done
Stopping costa_database_1  ... done
```

- If you want to delete the services

```
$ docker-compose down -v
Stopping costa_wordpress_1 ... done
Stopping costa_database_1  ... done
Removing costa_wordpress_1 ... done
Removing costa_database_1  ... done
Removing network costa_backend
Removing network costa_frontend
```

Specifying volumes in docker-compose

- If you wish to use docker volumes, they can also be specified in the docker-compose YAML file. For example:

```
version: ‘3’
volumes:
    wordpress:
    db:
services:
    wordpress:
            volumes:
                    - wordpress:/var/www/html
            [...]
     database:
            volumes:
                    - db:/var/lib/mysql
[...]
```

This automatically creates the Docker volume wordpress, mapping it to the directory /var/vvv/html on the container

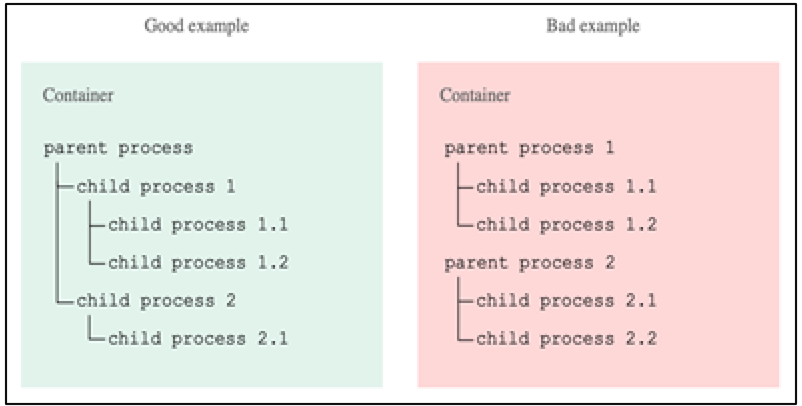
Limitations of `docker-compose`

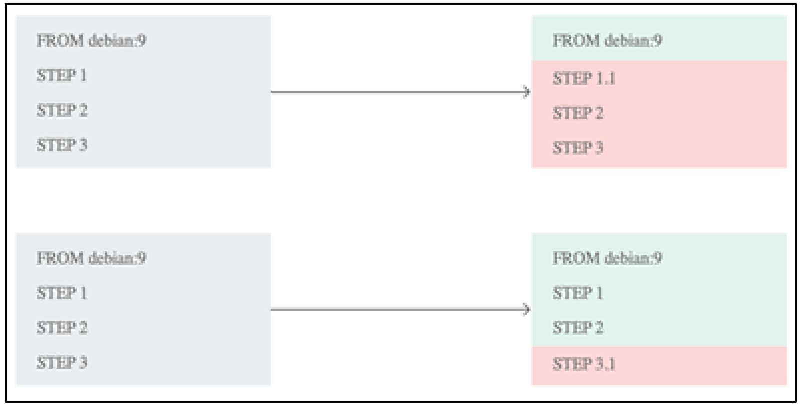
- As seen, `docker-compose` is very handy to create combinations of containers running on the same machine (VM1 in our case).
- It is best suitable **if you don't need automatic scaling of resources or multi-server environments**.
- For complex set ups, other tools such as Docker Swarm or Kubernetes are more appropriate. You'll see them in the next lectures.

Docker: best practices and security

Some best practices for writing containers

1. Put a **single application per container**. For example, do not run an application *and* a database used by the application in the same container.
2. **Do not confuse RUN with CMD**.
   - RUN runs a command and commits the result;
   - CMD does not execute anything at build time, it specifies the intended command for the image.
3. If in a Dockerfile you have **layers that change often, put them at the bottom of the Dockerfile**. This way, you speed up the process of building the image.
4. **Keep it small**:
   - use the smallest base image possible, remove unnecessary tools, install only what is needed.
5. Properly **tag your images**, so that it is clear which version of a software it refers to.
6. **Do you really want / can you use a public image?** Think about possible vulnerabilities, but also about potential license issues.





More (and more detailed) information available at https://bit.ly/2Zr6Hyq

A few words on Docker security (1)

- As seen so far, if you want to run Docker containers, you need to have Docker installed on your host system.
- If Docker is not installed, you can install it yourself, **but you must have root access**.
- Once you have installed Docker, you can download and execute containers from DockerHub or other sources.
  - Careful, because this is a potentially big security threat: some containers that you download might be compromised (e.g. include viruses or trojan)!
- Passwords, certificates, encryption keys, etc.
  - **Do not** embed them into the containers, and **do not** store them e.g. in GitHub repositories!

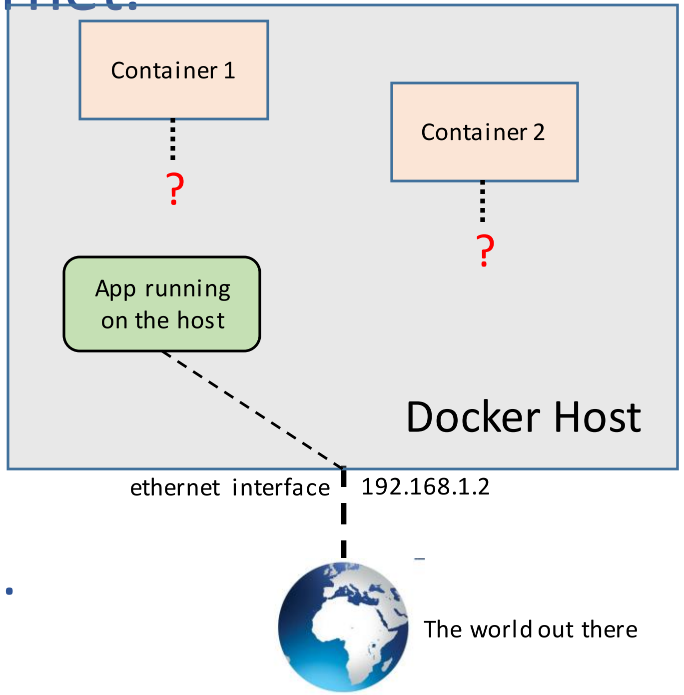
Recap of Containers

- We covered the basic concepts about **Containers**, comparing them to Virtual Machines.
- We executed a container, list docker images and extend them to create new containers.
- We then saw how to push containers to repositories on Docker Hub and simplified the building of containers via Dockerfiles.
- We created a container serving web pages and we then connected containers to volumes.
- We studied also how to combine multiple containers in an application stack with `docker-compose`.
- We then discussed about some Docker limitations, in particular with regard to security

Container Networking

Networking in containers

- Containers **isolate** applications from each other and from a physical infrastructures.
- But typically container may also need to connect to somewhere; for instance, to other containers, or in general to the internet.
- Remember that Docker containers live inside a host (called “Docker host”). That host normally has one or more IP addresses of its own, connected to a *physical or virtual network interface*, used e.g. by applications running on the host.
  - Docker containers, which are software appliances, use ***virtual network interfaces*** to connect to the outside world.
  - We will now see how.



Before we continue…

- In the hands-on exercises for this part, we will use the **Alpine** container. It is a Docker official image for the Alpine Linux distribution (https://www.alpinelinux.org/), a lightweight Linux distribution.

  Compare:

```
ubuntu@VM1:~$ docker images alpine
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              f70734b6a266        5 weeks ago         5.61MB
ubuntu@VM1:~$ docker images ubuntu
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              1d622ef86b13        5 weeks ago         73.9MB
```

- You may check the details of the interfaces on a Docker host or on a container with the command

```
$ ip address show
```

Check the network interfaces on VM1

`lo` is the "loopback interface" (we'll ignore it here)

`eth0` is the interface of the host

172.31.17.119 is the IPv4 address of the host

This `docker0` interface is an ethernet bridge device

```
centos@VM1:~$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether fa:16:3e:58:ca:ab brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.26/24 brd 192.168.1.255 scope global dynamic eth0
       valid_lft 63732sec preferred_lft 63732sec
    inet6 fe80::f816:3eff:fe58:caab/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:95:de:43:5e brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:95ff:fede:435e/64 scope link
       valid_lft forever preferred_lft forever
centos@VM1:~$
```

Docker networking options

- There are several ways to handle networking with Docker containers. We will discuss here the following:
  - No networking.
  - Bridge networking. This is the default if you don't specify anything else.
  - Host networking.
  - Overlay networking.
  - Macvlan.
- These options are selected using the flag

  `--network=<network_type>`

  in commands such as `docker run`.

The `--network=none` option

- Sometimes you just **don't need or want** to connect a Docker container to the network.
  - Maybe you just want to create a container and use it locally to your host to run some jobs, and that's it.
  - On VM1, type
    `$ docker run -it --network=none alpine /bin/sh`
    Once logged in, run `ip address show`. You will see that the container has no ip addresses other than the loopback IP address (which is always 127.0.0.1).
  - In this case, there is no way to connect to the container except than with docker commands such as `docker run` or `docker exec`.
  - Since there is no IP address on the container, no IP communications to/from the container are possible.

Bridge networking

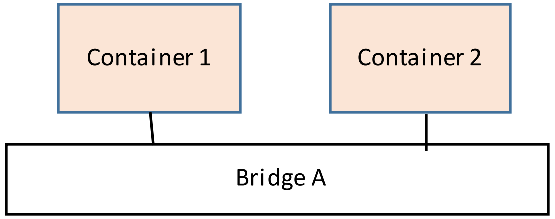
- This is the **default networking option for Docker**. A "bridge" is a type of network device making it possible to transfer packets between devices *on the same network segment*.
  - For example, if you have 2 laptops at home, you may connect them with each other via a physical "bridge" (sometimes called also a "switch") – we won't discuss the differences between bridges, switches and hubs here.
- With Docker, we deal with virtual (rather than physical) bridges. Docker always creates a default bridge called in fact `bridge`. You can see it if you issue the command

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
9b88500f1da1	bridge	bridge	local
320bf6394a48	host	host	local
a89c28f34f85	none	null	local

Multiple bridges (1)

- Containers connected **to the same bridge do communicate with each other**.



- Let’s start two Alpine containers **without** specifying any `--network` option. Open two separate ssh terminals on VM1 and run the following commands:

```
$ docker run -td --name test1 alpine
$ docker run -td --name test2 alpine
```

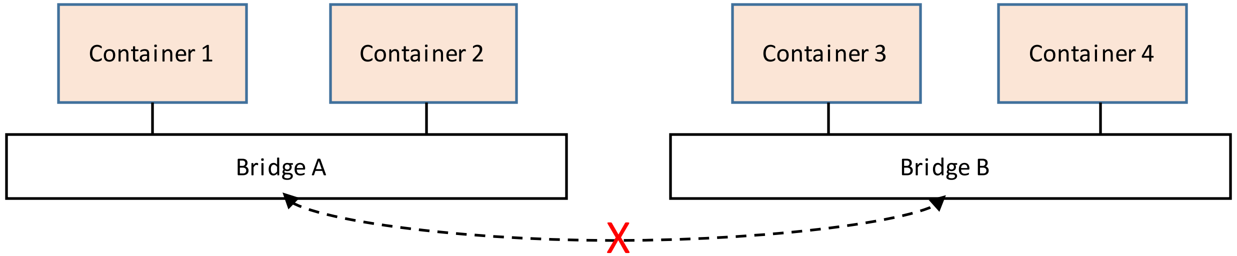
- Access the container

```
$ docker exec -it test1 /bin/sh
```

- Run `/# ip address show eth0` on each container. You will see that both have an IP address on the same network, something like 172.17.0.x.
- Are the two containers able to communicate with each other? (*hint: ping*)
  - Both containers are connected to the same default bridge (called `bridge`).

Multiple bridges (2)

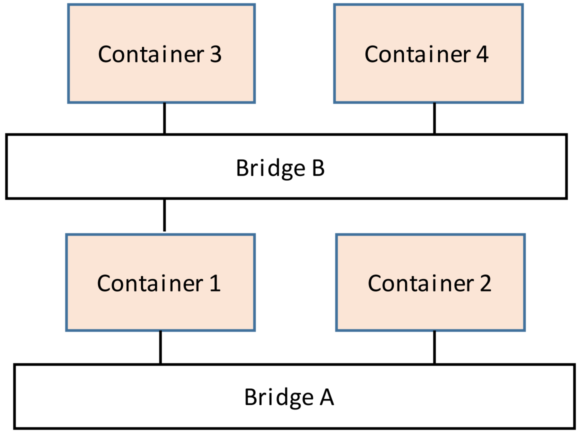
- Containers connected **to different bridges do <u>not</u> communicate with each other.**



- Now create a second bridge on VM1 (called a *user-defined* bridge):
  `$ docker network create my-bridge`
- List the bridges with `$ docker network ls` and confirm that `my-bridge` is there.
- Create the `test3` container, and connect it directly to `my-bridge`:
  `$ docker run -itd --network=my-bridge --name=test3 alpine`
- Are the containers still able to communicate with each other?
  - Two containera are connected to the bridge `bridge`, the other to the bridge `my-bridge`.
  - Check the IP address on `test3` now.

Connecting to multiple bridges

- You may also connect a container to more than one bridge. This is possible with the docker network connect <bridge> <container> command (note: not directly with the docker run command).
- Disconnect a container from a bridge with docker network disconnect <bridge> <container>
- Try it yourself with 3 or 4 containers, of which one is connected to two bridges.
What will happen in this case?



Inspecting bridges

- The configuration of a bridge can be shown with
  `$ docker network inspect <bridge>`
- This will emit some JSON output with information such as the IP range associated to the bridge and the containers (if any) connected to it.
- Try it out with `$ docker network inspect my-bridge`
- A single-line, nerdy way of parsing the output of this command to show just the containers connected to a bridge :

```
$ docker network inspect my-bridge | python -c "import
sys, json; print([v['Name'] for k,v in
json.load(sys.stdin)[0]['Containers'].items()])"
```

What is my IP address?

- We have seen that containers connected to different bridges do not see each other. But they can connect to the internet.
- Try it for yourself: from the `test1` container connected to `my-bridge`, issue the command `ping www.google.com` and verify that it works.
- Do the following on `test1`:
  ```
  /# apk update && apk add bind-tools
  ```
  This will install a utility called `dig` (for **d**omain **i**nformation **g**roper, used to query the DNS). Note that Alpine Linux uses the command `apk` (and not `apt` as in Ubuntu) to install packages.
- Now with the command
  ```
  /# dig +short myip.opendns.com @resolver1.opendns.com
  ```
  you will see the **real IP address** that your `test1` container uses to connect to the internet.

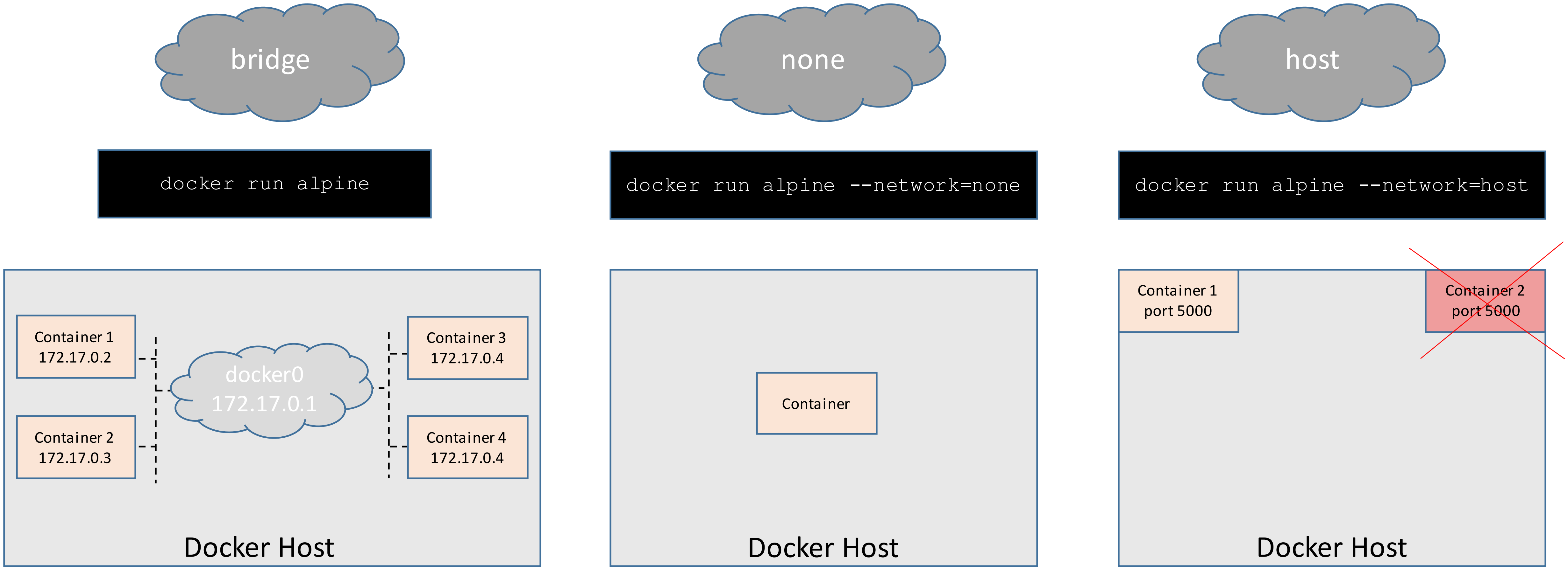
Network Address Translation (NAT)

- Our `test1` container was able to ping the internet. However, it was not able to ping another container on the same Docker host, but connected to a different bridge.
- We also just discovered that, when connecting to the internet, `test1` uses an IP address that is not its own.
- This is because the Docker engine on VM1 performs an automatic **Network Address Translation** (NAT) when `test1` wants to connect to the outside world, transparently mapping the `test1` IP address (the one you see with `#/ ip address show`) to the IP address of the Docker host.

Host networking

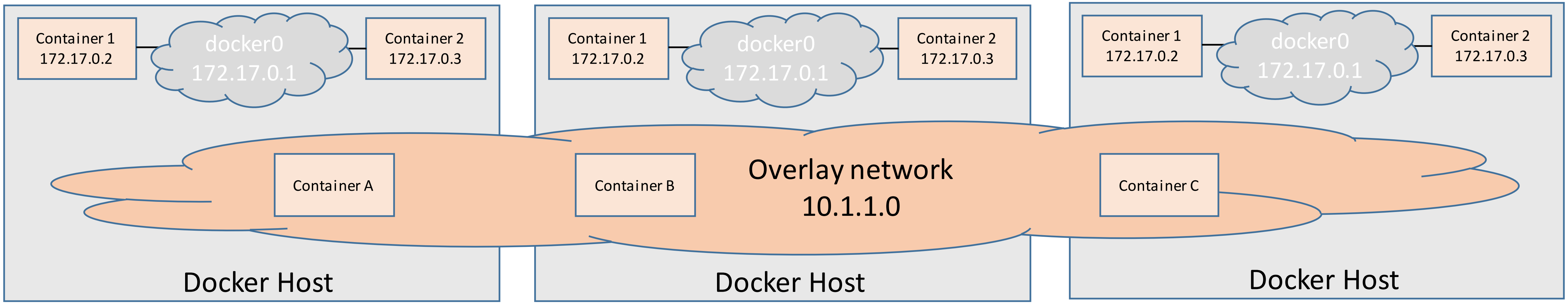
- We have seen the options `--network=none` and `--network=bridge`.
- Another option is **host networking**, specified with `--network=host`. This connects a container directly to the Docker host network interface and avoids using NAT (which could be useful e.g. for performance purposes).
  - The container does not get any IP addresses of its own and **uses directly the Docker host IP address**.
  - This means that *port mapping does not make sense with host networking* (the container shares the same ports of the Docker host). It also means that you cannot have two containers in host mode running a service on the same port.
  - Host networking is used in special cases. We won't discuss it more here.

The main types of Docker networks covered



Overlay networks

- So far, we have considered network configurations that were applicable to containers running on the same Docker host.
- Docker **overlay networks** connect Docker daemons running on multiple hosts.
  - VXLAN used for encapsulation as network virtualization technology



Macvlan networks

- *Macvlan* networks allow to assign a MAC address to a container, making it appear as a physical device on your network.
- The Docker daemon routes traffic to containers by their MAC addresses.
- Using the *`macvlan`* driver is sometimes the best choice when dealing with **legacy applications** that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.
- See Macvlan networks.

Docker management

Process management

- Once you start a container, you may want to check how it is going. For example, which are the running processes, or how much CPU of the Docker host it is using, or RAM, etc. Possibly you can log in to the container and issue commands there, but it is very useful to verify what containers are doing **directly from the Docker host**.
- Example: suppose we want to compute Pi using the Leibniz formula:

$$\pi = \sum_{i=0}^{\infty} \frac{4(-1)^i}{2i+1}$$

- Let's implement it with a very simple Python program, call it for example `mypi.py`:

```
pi = 0
accuracy = 1000000

for i in range(0, accuracy):
    pi += ((4.0 * (-1)**i) / (2*i + 1))
    print(pi)
```

Process management: `docker top`

- On VM1, create a container called e.g. test1 (if you had a `test1` container earlier, remember to delete it first with `docker rm test1`) with

```
$ docker run -it --name test1 alpine /bin/sh
```

- Connect to `test1` and install `python`:

```
$ apk update && apk add python2
```

- Now create the `mypi.py` program on `test1` (you may take it from IOL) and run it simply with `python mypi.py`. It will take some time to finish; the Leibniz formula is not very efficient to compute Pi.
  - `/# wget https://baltig.infn.it/corso-olss-2020/corso_olss_2020/-/raw/master/containers/mypi.py`

- Now open another terminal on VM1 and type `docker top test1`: you should see the **running processes** on `test1`, something like

```
$ docker top test1
PID        USER       TIME       COMMAND
50725      root       0:00       /bin/sh
50824      root       0:15       python mypi.py
```

Process management: `docker stats`

- While the `mypi.py` program is still running, type `docker stats test1` on VM1. It should output something like this:

```
$ docker stats test1
CONTAINER ID    NAME     CPU %    MEM USAGE / LIMIT     MEM %    NET I/O        BLOCK I/O    PIDS
3fa7f0adb613    test1    42.50%   44.19MiB / 3.848GiB   1.12%    17MB / 194kB   0B / 0B      2
```

This is the percentage **of the Docker host's CPU and memory** the container is using

- The `docker stats` command displays a **live stream of container resource usage statistics**. It is live, so it refreshes automatically. Interrupt with Ctrl-C.
- This is quite useful in order to check that a container is doing what it is supposed to do, but **how can we limit the resources available to a container**?

Why limit resources for containers

- By default, a container has **no resource constraints** and can therefore use up Docker hosts resources as much as it is allowed by the Docker host kernel scheduler.
- For example, if you do not limit the memory a container uses, the Docker host could run out of memory and throw an Out of Memory exception.
- In practice, if a Docker host runs out of memory for example because of a container misbehaving, the entire system could just crash (i.e, all the other processes or containers running on the host will crash).

Some ways to limit resources for containers

- Check first with `docker stats` what your container is doing with resources.
- You can then limit e.g. memory to 256MB for a nginx container with
  ```
  $ docker run -d -p 8080:80 --memory="256m" nginx
  ```
- Similarly, you can limit the number of CPU cores that a container may use with
  ```
  $ docker run -d -p 8080:80 --cpus=".5" nginx
  ```
  This will limit the container to use up to half a CPU core.
- More information on this topic can be found at https://docs.docker.com/config/containers/resource_constraints/

Logging container behavior

- Especially when a container is running in the background and you are not able to check its behavior interactively from within the container, it is very useful to checks what it is writing to STDOUT and STDERR.
- For example, suppose we run the following command on VM1:
  `$ docker run -d --name test1 alpine /bin/sh -c "while true; do $(echo date); sleep 1; done"`
- This creates the `test1` container using the Alpine image, running it in background (`-d`) and executing an infinite loop printing the current date to STDOUT every second.
- The container is running in the background, but you may check what it is printing with the command. Try it out.
  - `$docker logs --follow test1`
  - You may limit logs output to e.g. the last 10 lines with
    `$ docker logs --tail 10 test1`
- Once done, stop the `test1` container running in the background with
  `$ docker stop test1`

Graphical Docker management

- So far, we have seen ways to manage containers via the terminal. This is the normal and preferred way to do it, especially when learning Docker concepts.
- However, there are also ways to manage containers graphically. This can be very handy.
- We will briefly explore here the use of the open source tool called **Portainer**, https://www.portainer.io/. Its scope is to build and manage Docker environments directly from a browser.

Graphical Docker management

- Run Portrainer as a container in the local host

```
$ docker volume create portainer_data
$ docker run -d -p 8080:9000 --name=portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce
```