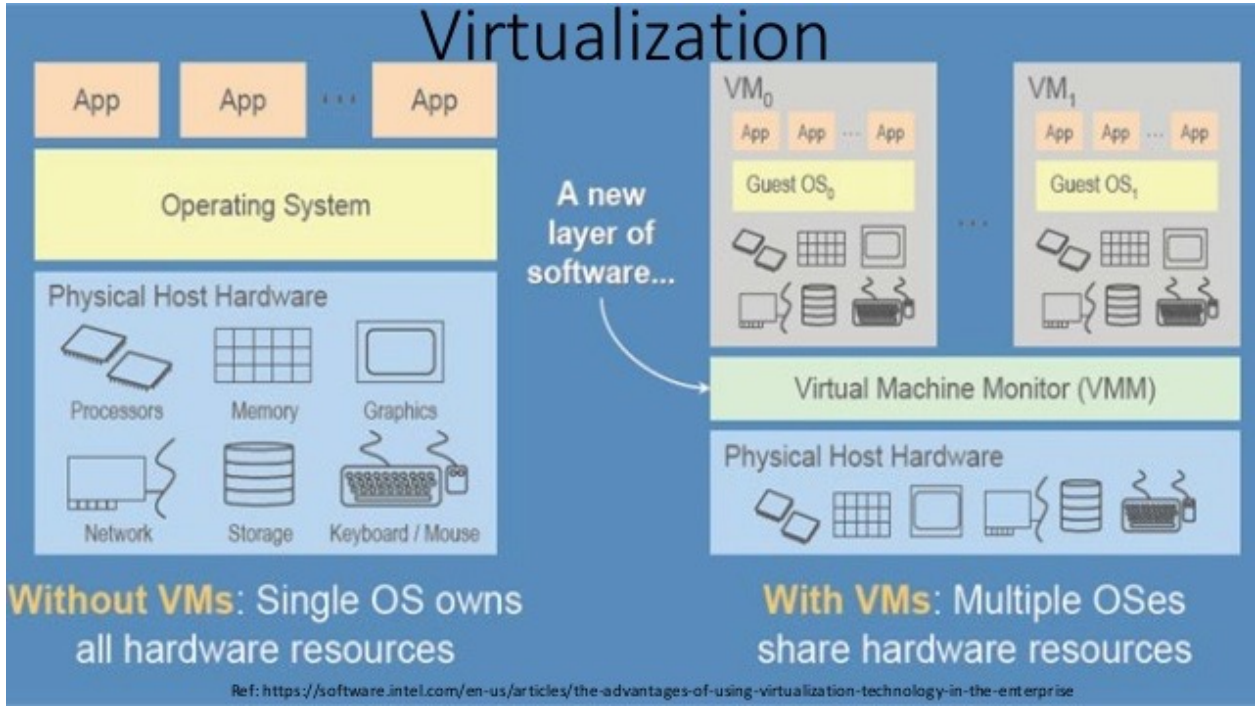
Introduction to Containers

SOSC22 – Perugia, 28/11/2022

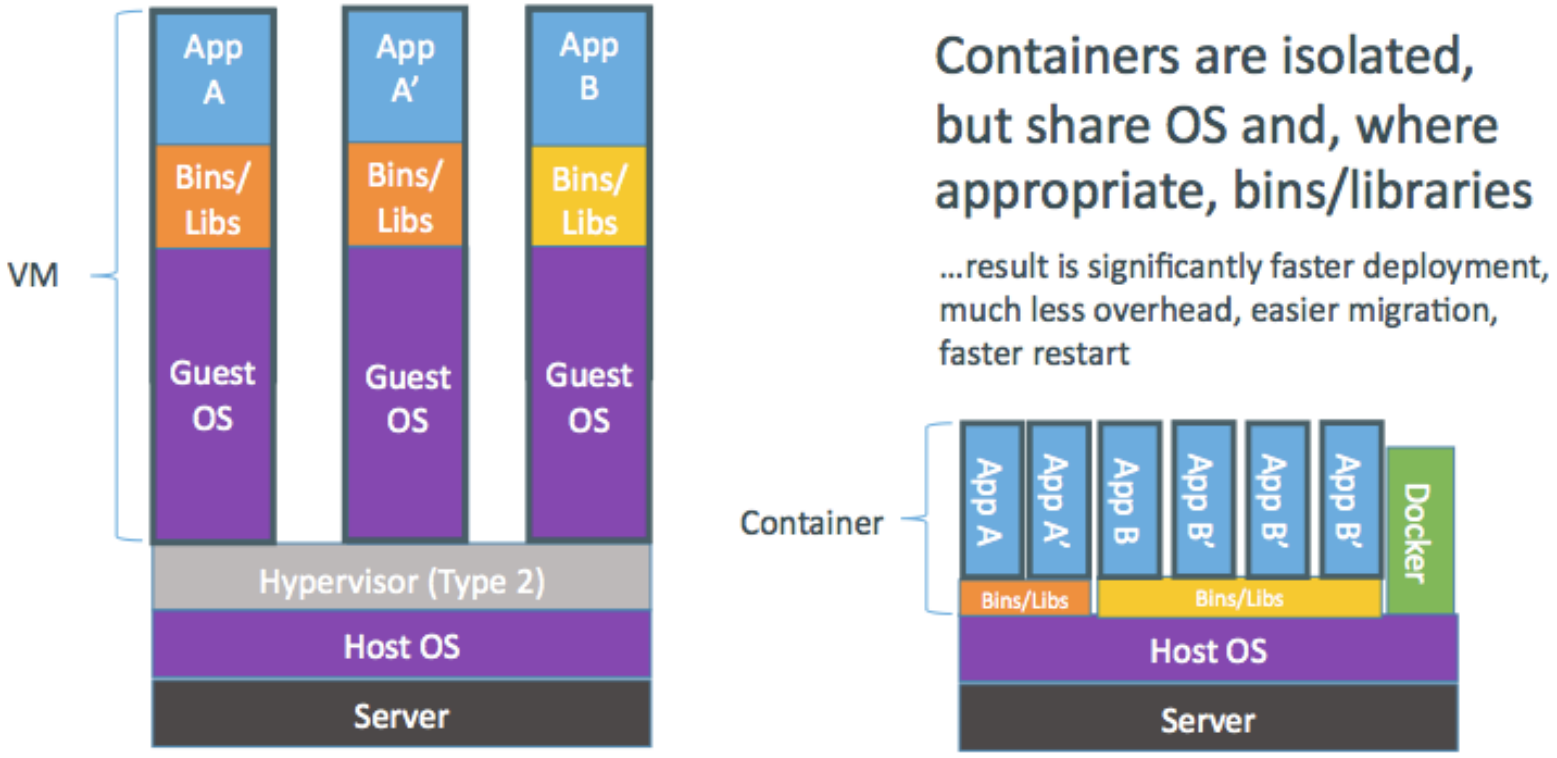
Davide Salomoni (davide@infn.it)

Virtualization and Virtual Machines

- Informally, a Virtual Machine (or **VM** for short) is a “virtual copy of a real machine”.
- But 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 are «lightweight VMs»



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

Cargo Transport Pre-1960

Multiplicity of Goods

Do I worry about how goods interact (e.g. coffee beans next to spices)

Multipilicity of methods for transporting/storing

Can I transport quickly and smoothly (e.g. from boat to train to truck)

docker

Solution: Intermodal Shipping Container

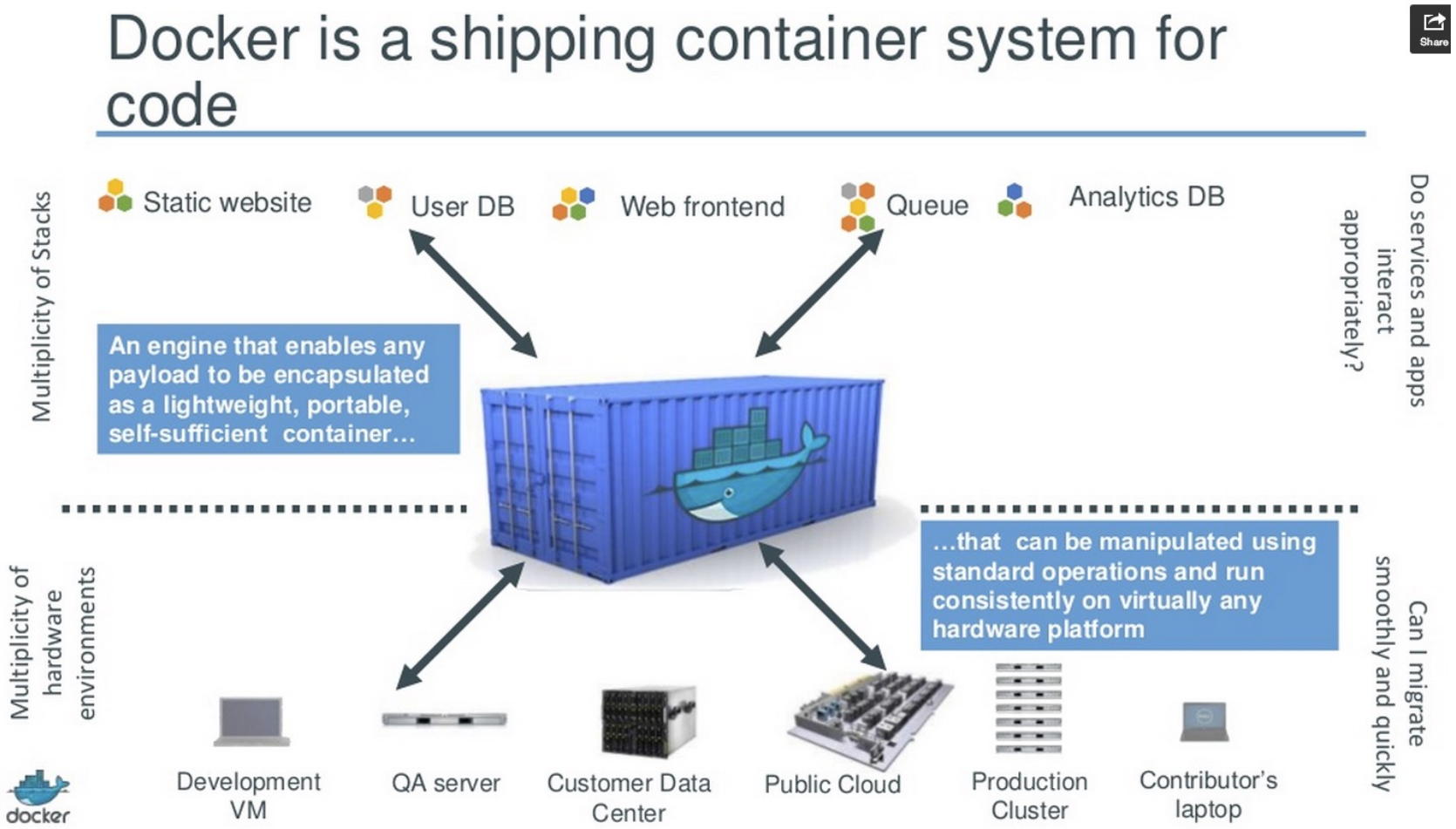
Intermodal Shipping Container Ecosystem

- 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

Docker Containers



Containers in practice (1)

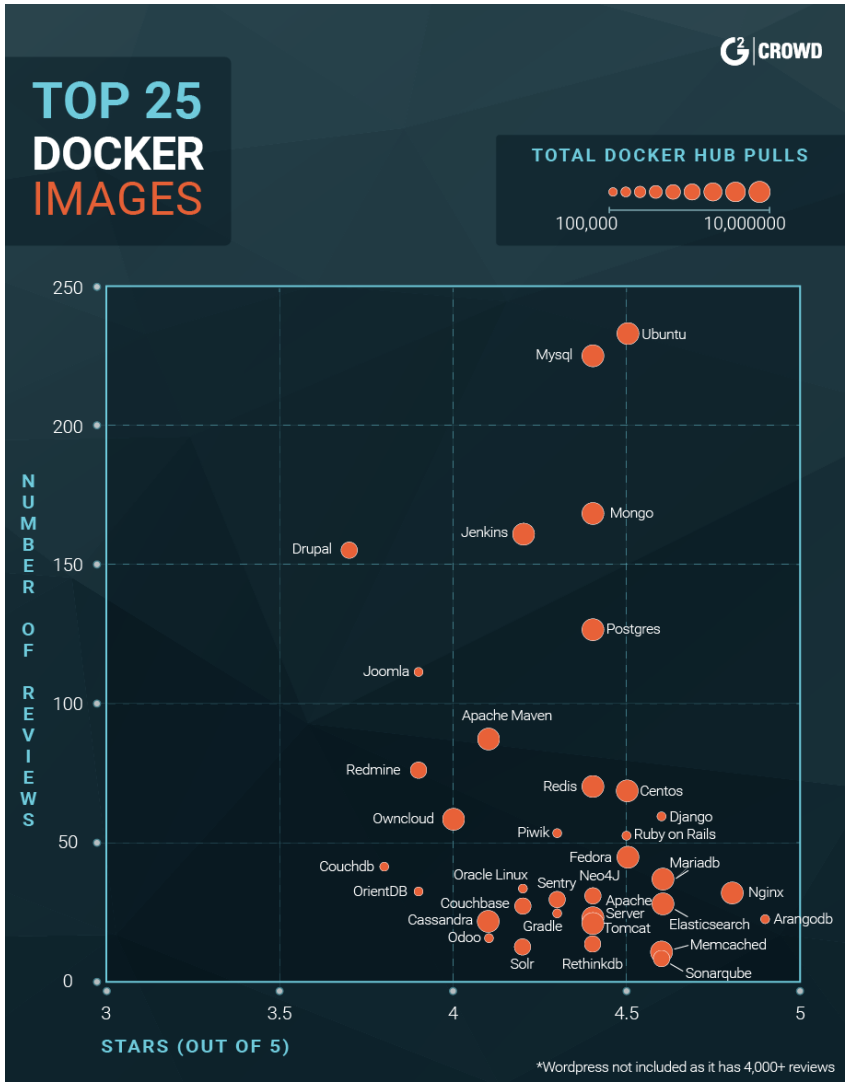
- Instead of installing multiple applications X_1,X_2,…,X_n on a system, **you just install the Docker application (or "Docker engine")**.
- Applications X_1,X_2,…,X_n then come *encapsulated in **Docker images***. You don't need to install anything else beyond the Docker engine to run them. A Docker image, when downloaded and executed, creates a container. This container "wraps" the application.
- Not only can you run different Docker images on your system at the same time, but you can also have *different instances of the same application* (same Docker image) running in different containers at the same time.
  - How many containers can you start on a system? Of course, it depends on how powerful your system is, but as a rule *many more* than the number of VMs you can run on the same system.

Containers in practice (2)

- Without going into details, each container is “isolated” from other containers. However, containers can optionally communicate among them and with the outside world.
- If, for instance, you develop an application, you could distribute it under the form of a Docker image and let everyone who has access to a system with the Docker engine installed run it.
  - Compare this with the traditional way of providing installation / deinstallation procedures for Windows, MacOS and various flavors of Linux, making sure that all the dependencies of your application are satisfied, etc.

What are containers used for?

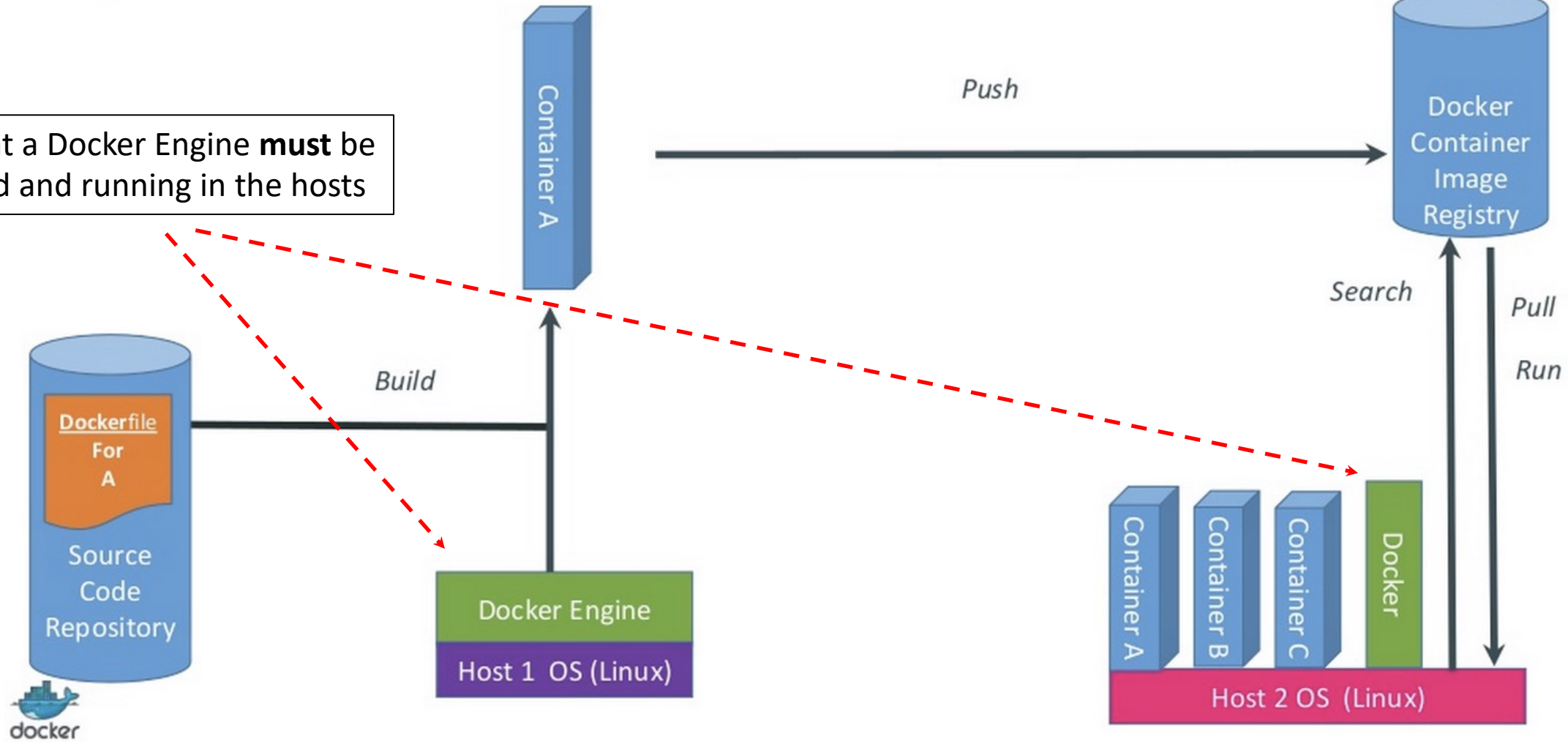
- Docker images have been created for many purposes. For example, images can hold:
  - Entire Linux distributions, such as Ubuntu, Debian, CentOS, Arch Linux, Fedora, etc.
  - Databases, such as MySQL, Redis, MongoDB, PostgreSQL, etc.
  - Applications, such as web servers, content management software (e.g., Wordpress), programming language environments (e.g., Python, C, C++, etc.), and tons of other stuff.
  - Complete scientific environments for data science (*we will work with one of these here at SOSC22*).



Source: https://learn.g2.com/best-docker-containers-repository

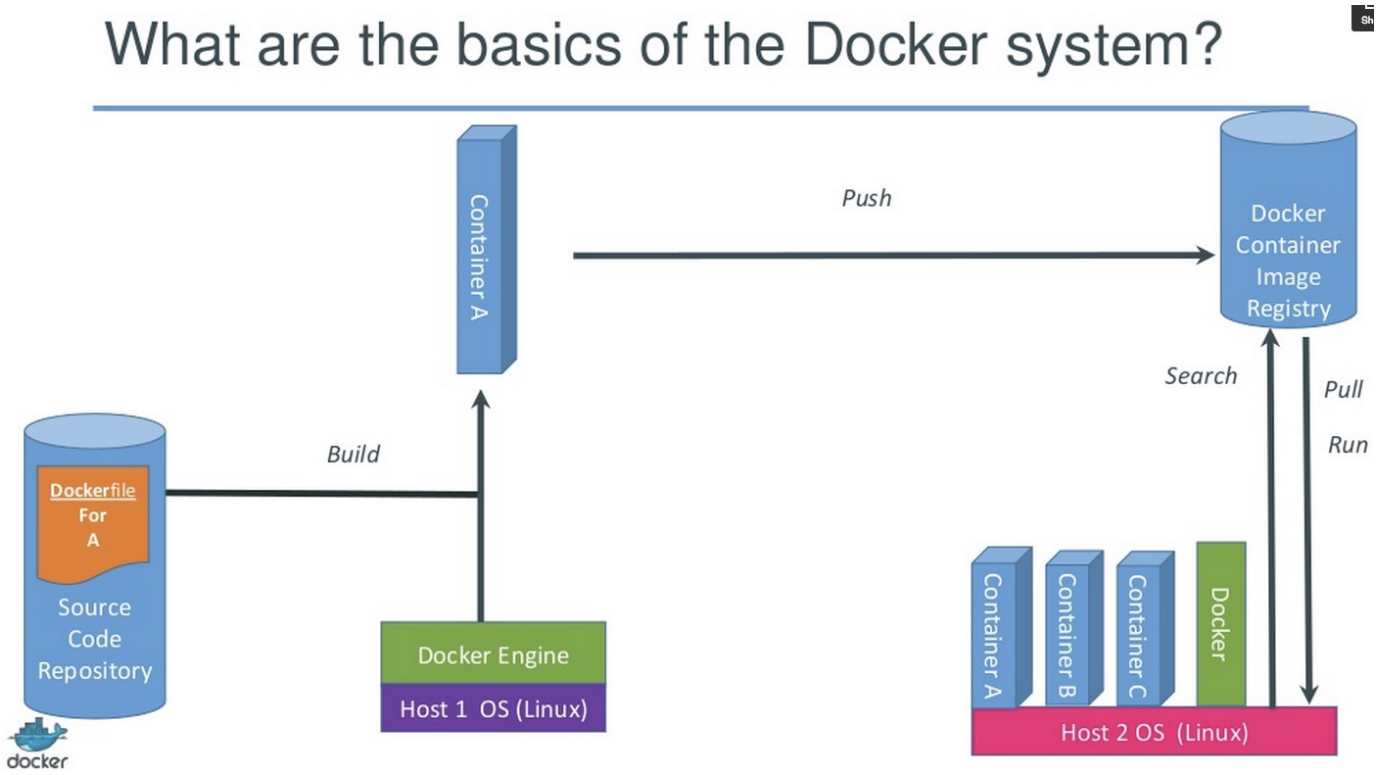
What are the basics of the Docker system?

Note that a Docker Engine **must** be installed and running in the hosts



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



Search, pull, run and push

- **Try these commands on the SOSC22 systems that will be made available to you**:
  - **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):
    - `docker pull ubuntu`
  - Execute (**run**) a docker container:
    - Run the “echo” command inside a container and then exit:
      - `docker run --rm ubuntu echo "hello from the container"`
        `hello from the container`
    - Run a container in interactive mode:
      - `docker run --rm -i -t ubuntu /bin/bash`

How efficient is Docker?

```
davide@iz4ugl-3:~$ docker images ubuntu
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
ubuntu       latest    a8780b506fa4   3 weeks ago   77.8MB
```

→ the latest Ubuntu image takes about 78MB 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 the GB.

```
davide@iz4ugl-3:~$ time docker run --rm ubuntu echo "hello from the container"
hello from the container

real    0m0.729s
user    0m0.069s
sys     0m0.118s
```

→ The total time it took on this system (not a very powerful one) to start a container, execute a command inside it and exit from the container is less than a second. How long would it take if we used a full VM?

An essential thing to know about containers

- There are many things to learn about Docker containers, but it is important to realize right from the start that

  Docker containers are **ephemeral**!

- By this, we mean that if you start a container, and write some data into it (for instance, in a folder visible only by the container), **this data will disappear when the container stops running**. In other words, if you stop the container and start it again, you will start from scratch, without the data you had written to the previous container.
  - This is by design! (let's discuss the reason offline if you want)
- We will see ways to make *data visible to containers* permanent (or "persistent"). But keep in mind that, as a rule, you should **not** expect a container to store permanent data.

Verify that containers are ephemeral

```
davide@iz4ugl-3:~/bdb/ubuntu$ docker run -it --rm alpine /bin/sh
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # touch test_file.txt
/ # ls
bin            home           mnt            root           srv            tmp
dev            lib            opt            run            sys            usr
etc            media          proc           sbin           test_file.txt  var
/ # exit
davide@iz4ugl-3:~/bdb/ubuntu$ docker run -it --rm alpine /bin/sh
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ #
```

1. Start a Linux container interactively
2. List the files in the home directory
3. Create an empty file called `test_file.txt`
4. List the files in the home directory: `test_file.txt` is present
5. Exit from the container
6. Start the same container again
7. List the files in the home directory: `test_file.txt` is **gone**

How to *extend* a docker container (1)

- Suppose you need a command inside a container, that 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:
  - **davide@iz4ugl-3**:~$ docker run --rm ubuntu ping www.google.com
    docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable to start container process: exec: "ping": executable file not found in $PATH: unknown.
- We can install it ourselves; it is in the package `iputils-ping`:
  - **davide@iz4ugl-3**:~$ docker run --rm ubuntu /bin/bash -c "apt update; apt -y install iputils-ping"
- But it still doesn't work!?
  - **davide@iz4ugl-3**:~$ docker run --rm ubuntu ping www.google.com
    docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable to start container process: exec: "ping": executable file not found in $PATH: unknown.
- Who can explain this? The ping command was successfully installed!

How to *extend* a docker container (2)

- Whenever you issue a `docker run <image>` command, a **new container** is started, based on the specified Docker image.
- If you modify a container (for instance, installing some software in it) and then want to reuse it, **you must "save the container", creating a new Docker 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 **commits** a container locally, creating an image with the name `ubuntu_with_ping` (you may give it any name you like). Take `xxxx` from the container ID shown by the `docker ps` output (or `docker ps -a` if the container is not running anymore).

How to extend a docker container (3)

- Verify that the `ping` command inside our new image now works:
  - **davide@iz4ugl-3**:~$ docker run --rm ubuntu_with_ping ping -c 3 www.google.com

```
PING www.google.com (142.251.209.4) 56(84) bytes of data.
64 bytes from mil04s50-in-f4.1e100.net (142.251.209.4): icmp_seq=1 ttl=37 time=108 ms
64 bytes from mil04s50-in-f4.1e100.net (142.251.209.4): icmp_seq=2 ttl=37 time=34.7 ms
64 bytes from mil04s50-in-f4.1e100.net (142.251.209.4): icmp_seq=3 ttl=37 time=29.4 ms

--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2007ms
rtt min/avg/max/mdev = 29.352/57.305/107.855/35.810 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:
  - **davide@iz4ugl-3**:~$ docker images ubuntu*

```
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
ubuntu_with_ping    latest    9a6a648de422   2 minutes ago    119MB
ubuntu              latest    a8780b506fa4   3 weeks ago      77.8MB
```

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`:
  - **ubuntu@VM1:~**$ 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:
  - **ubuntu@VM1:~**$ 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

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

```
ubuntu@VM1:~$ 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

ubuntu@VM1:~$ docker rmi ubuntu_with_ping
Untagged: ubuntu_with_ping:latest
Deleted: sha256:3e7a8818665fc7eb1be20e8d633431ad8c0bdfba05d6d11d40edd32a915708bb
Deleted: sha256:a4c24b3590e4e95c30d4d0e82d3f769cde94436a5dd473b4e7ec7bd4682ce1b7

ubuntu@VM1:~$ docker rmi ubuntu
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:f08638ec7ddc90065187e7eabdfac3c96e5ff0f6b2f1762cf31a4f49b53000a5
Deleted: sha256:7698f282e5242af2b9d2291458d4e425c75b25b0008c1e058d66b717b4c06fa9
Deleted: sha256:027b23fdf3957673017df55aa29d754121aee8a7ed5cc2898856f898e9220d2c
Deleted: sha256:0dfbdc7dee936a74958b05bc62776d5310abb129cfde4302b7bcdf0392561496
Deleted: sha256:02571d034293cb241c078d7ecbf7a84b83a5df2508f11a91de26ec38eb6122f1

ubuntu@VM1:~$ 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
```

Handling multiple commands

- If you want to **apply several commands 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>
…
```

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

Dockerfiles

- Rather than modifying a container “by hand”, connecting interactively, installing packages and the committing the container as previously shown, it is normally 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, where we install a web server (through the `apache2` package) and tell the image to serve a simple html page (`index.html`), which we copy from our system:

```
$ cat Dockerfile
FROM ubuntu
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:
- **ubuntu@VM1:~**`$ cat index.html`

```
<!DOCTYPE html>
<html>
<h1>Hello from a web server running inside a container!</h1>
This is an exercise for the SOSC22 course.
</html>
```

- **Create** both the previous `Dockerfile` and the `index.html` file on your test machine.

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 --rm -d -p 8080:80 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://<test-system-ip-address>:8080/

Check that your web server is running

- Check with:

```
ubuntu@VM1:~$ docker ps
CONTAINER
ID        IMAGE                 COMMAND                  CREATED              STATUS              PORT
S                   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:

```
ubuntu@VM1:~$ docker stop f9dc164be001
```

- You can now type `docker run --rm -d -p 8080:80 web_server` any time you want to instantiate a new web server.
- What happens if you then type `docker run --rm -d -p 8081:80 web_server` ?

Connect a container to a host file system

- Containers are ephemeral, but what if we want to persist data with containers?
- We can map a directory that is available *on the host* (the system where we run the docker commands), to a directory that is available *on the container*. This is called a "**bind mount**", and is done with the Docker flag `-v`, like this:

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

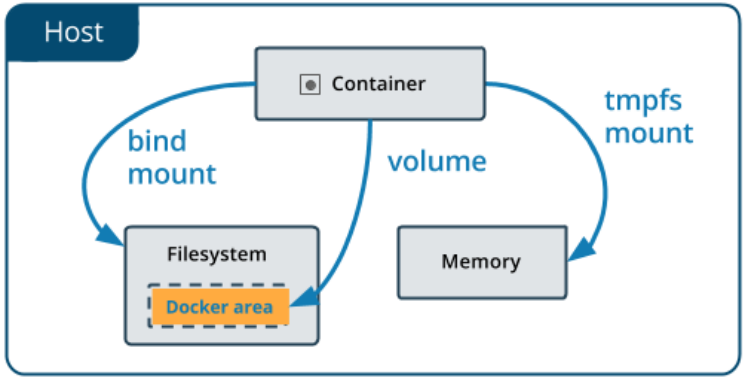
- So, for example, to map a local directory called `/local_data` to the directory `/cointainer_data` on the container:

```
docker run --rm -v /local_data/:/container_data -i -t ubuntu /bin/bash
```

- Now, when you are *within the container*, if you write `ls /container_data` you should see what is in `/local_data`.

Docker volumes (1)

- In the previous example, 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 wants to be system-independent.
- We can (and should generally prefer to) use **Docker volumes**.
- Docker volumes are persistent, but they are not tied to the specific filesystem of the host. They are completely managed by Docker itself.



We'll see what a `tmpfs` mount is later on

Docker volumes (2)

- You can create a new Docker volume with the command

```
docker volume create some-volume
```

  - Try these self-explanatory commands:

```
docker volume ls
docker volume inspect some-volume
docker volume rm some-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 --rm -i -t --name myname -v some-volume2:/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.
- More in general, you can remove all unused docker volumes with

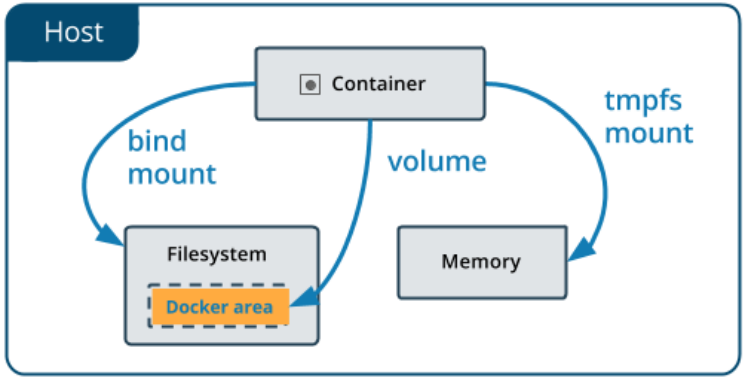
  `docker volume prune`

  - Note that the `docker system prune` command we showed previously **does not remove volumes**!

`tmpfs` mounts

- If you are running Docker on Linux, there is a third option to mount a volume on a container: the so-called **`tmpfs` mount** option.
- When you create a container with a `tmpfs` mount, the container can create files outside the container's writable layer, directly into the host system memory (RAM).
- This is a **temporary volume**, i.e., it will be automatically removed once the container exits. It is useful for example if you have sensitive data that you do not want to store neither in the container nor in a dedicated area (be it a bind mount or a Docker volume).
- An example of mounting the `/app` directory of a container under a `tmpfs` mount (whatever you write in that directory will only be stored in RAM):

```
docker run --rm -it --name mytmp --tmpfs /app ubuntu
/bin/bash
```



Hands-on: create *your* image

- A small assignment: take a Python program (any program you like) and/or any software you want to install, and embed these things into a Docker image, using a Dockerfile to build it.
- Verify that your new image works as expected using `docker run`.
- **This is something you should do on your own**.

Detour: using the `tar` command

- In Linux, `tar` (for “tape archive”: this tells you how old this command is) is one of the most useful commands to package several files or directories into a single file, often called `tarball`. It can be combined with the `gzip` tool to also compress the archived file (with this option, it is similar to the Windows `zip` and `unzip` tools).
- Typical extensions:
  - `.tar` → uncompressed archive file using tar
  - `.zip` → compressed archive file using zip
  - `.gz` → file (it can be an archive or not) compressed using gzip
  - `.tar.gz` or `.tgz` → a compressed archive file using tar
- Examples of some useful `tar` commands (see e.g. https://www.howtoforge.com/tutorial/linux-tar-command/ for more information):
  - Create an archive file called `my_devstuff.tar` with the directory `/home/davide/devstuff/` and its content:

```
tar -cvf my_devstuff.tar /home/davide/devstuff/ # my_devstuff.tar will be created in the current directory
tar -xvf my_devstuff.tar   # extract my_devstuff.tar in the current directory
tar -xvf my_devstuff.tar -C /home/davide/newdir   # extract my_devstuff in another directory
```

  - The same archive as above, but compressed:

```
tar -cvzf my_devstuff.tar.gz /home/davide/devstuff/  # note the z flag to enable compression
tar -xvf my_devstuff.tar.gz # note that the uncompress command is the same as above
```

  - List the content of an archive file, compressed or not:

```
tar -tf <tar_filename>
```

Copy an image somewhere else

- So far, we have stored our images locally. But what if we wanted to copy our images to another system?
  - We could publish them to Docker Hub of course, and then retrieve them from there. But what if *we do not want* to publish them on Docker Hub?
- Docker allows to export an image to a tar file specifying its name (you could also compress it, if you wanted to save space):

```
# docker save -o my_exported_image.tar my_local_image
```

- You can then copy the tar file (`my_exported_image.tar`) to another system via e.g. `scp`, and then import it to a Docker image on that system:

```
# docker load -i my_exported_image.tar
```

Copy a Docker volume somewhere else

- Recall that Docker volumes are independent of the local file system structure and are managed directly by the Docker engine.
- In order to transfer a Docker *volume* to another host, you must first **back it up to a tar file** using the `--volumes-from` flag. This flag must be applied to an *existing container* (even if not running) which mounted the volume you want to back up, with a command similar to the following one:

```
docker run --rm --volumes-from EXISTING_CONTAINER -v /tmp:/backup ubuntu tar cvf /backup/backup.tar /app
```

  - This command backs up a volume that was mounted by the EXISTING_CONTAINER under the directory `/app` into the file `backup.tar` in the `/tmp` directory of the local system.
- At this point, you can simply transfer the tar file to another machine and restore it to another running container.
- For example, once you have the tar in the `/tmp` directory of another machine, you can do:

  `docker run -it -v /app --name myname2 ubuntu /bin/bash` (this runs `myname2` interactively)

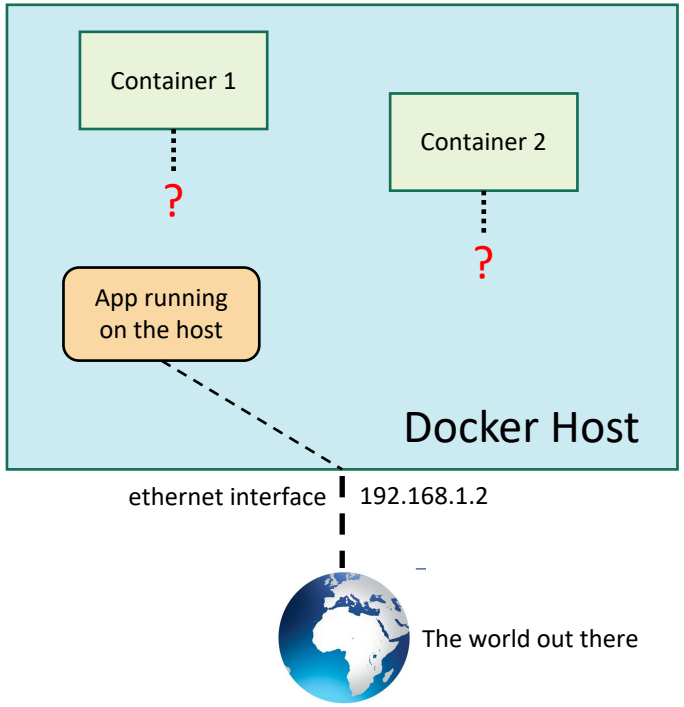
  (in another shell) `docker run --rm --volumes-from myname2 -v /tmp:/backup ubuntu bash -c "cd /app && tar xvf /backup/backup.tar --strip 1"`

Networking refresher

- You probably know that devices connect to the internet and to each other via **IP addresses**.
  - IP (Internet Protocol) is present on the internet today in two versions:
    - **IPv4**, based on a 32 bits address space. 8 bits = 1 byte, i.e. the IPv4 address space has 4 bytes. When representing an IPv4 address, these 4 bytes are normally expressed in decimal notation and separated by a dot. For example, 192.168.2.1 is an IPv4 address.
      - How many addresses can we represent with 32 bits? That is 2^{32} = 4,294,967,296 addresses (~4.3×10^9). Out of these, some are reserved for special purposes, e.g. for private IP addresses (~18M) or for multicast addresses (~270M).
    - **IPv6**, based on a 128 bits address space, i.e. 16 bytes. An IPv6 address is typically referred to in hexadecimal notation, with its 16 bytes separated two by two with a colon. For example, 2001:0DB8:AC10:FE01:0000:0000:0000:0000 is an IPv6 address.
      - How many addresses can we represent with 128 bits? That is $2^{128} \sim 3.4 \times 10^{38}$ addresses. Like in IPv4, some of these are reserved for special purposes.
- Today, we will only deal with IPv4 addresses, and will just say “IP addresses” to mean IPv4 addresses.

Networking in containers

- Containers *isolate* applications from other applications and from a physical infrastructures.
- But typically, containers may also need to *connect* to somewhere; for instance, to other containers, or in general to the internet.
  - How, and with which IP address?
- 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 for instance 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** docker image. 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              6dbb9cc54074        4 weeks ago         5.61MB
ubuntu@VM1:~$ docker images ubuntu
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              7e0aa2d69a15        3 weeks ago         72.7MB
```

- You may check the details of the interfaces on a Docker host or on a container with the command `ip address show`.
  - Warning: the command `ip` used in `ip address show` **is not installed by default** on the Ubuntu docker image (it is however on the Alpine image). If on an Ubuntu system the `ip` command is not installed, you can install it yourself with `apt update && apt install -y iproute2` (`iproute2` is the Ubuntu package containing the `ip` command).

Check the network interfaces *on the host*

`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

fe80::85f:efff:fef0:e96f is the IPv6 address of the host

Note this `docker0` interface (we'll explain it later)

```
ubuntu@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 9001 qdisc fq_codel state UP group default qlen 1000
    link/ether 0a:5f:ef:f0:e9:6f brd ff:ff:ff:ff:ff:ff
    inet 172.31.17.119/20 brd 172.31.31.255 scope global dynamic eth0
       valid_lft 2228sec preferred_lft 2228sec
    inet6 fe80::85f:efff:fef0:e96f/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
ubuntu@VM1:~$
```

Check the network interface *on a container*

- Let's log in *into a container* running the `alpine` image and check its network interfaces:

Run the `alpine` container in *interactive mode* (`-it`)

The `alpine` image is not found on the local system, so it will be downloaded from DockerHub

This is the virtual ethernet interface of the container

This is the IP address of the container

```
ubuntu@VM1:~$ docker run --rm  -it alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
540db60ca938: Pull complete
Digest: sha256:69e70a79f2d41ab5d637de98c1e0b055206ba40a8145e7bddb55ccc04e13cf8f
Status: Downloaded newer image for alpine:latest
/ # ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN 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
12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # exit
ubuntu@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" networking.
- 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 your test system, type
    `docker run --rm -it --network=none alpine 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.

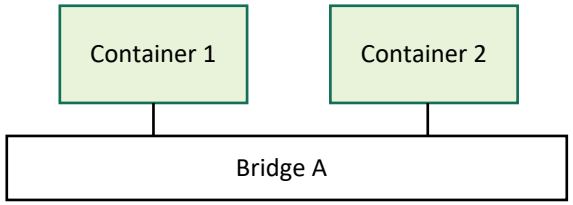
Bridge networking

- 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 here the differences between bridges, switches and hubs.
- With Docker, we deal with *virtual* and not physical bridges. Docker always creates a default bridge called in fact `bridge`. You can see it if you issue the command `docker network ls` for instance on your test system:

```
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**.



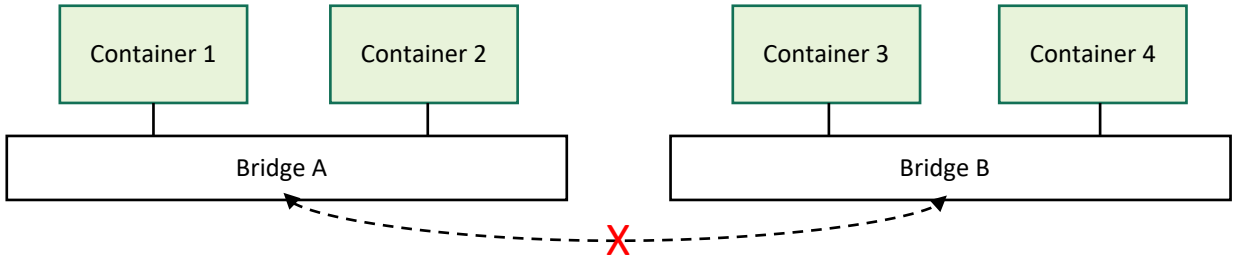
- Let's start two Alpine containers without specifying any `--network` option. **Open two separate ssh terminals on your test system** and run the following commands:

```
docker run --rm -it --name=test1 alpine sh
docker run --rm -it --name=test2 alpine 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? Try!
  - Both containers are connected to the same default bridge (called `bridge`).

Multiple bridges (2)

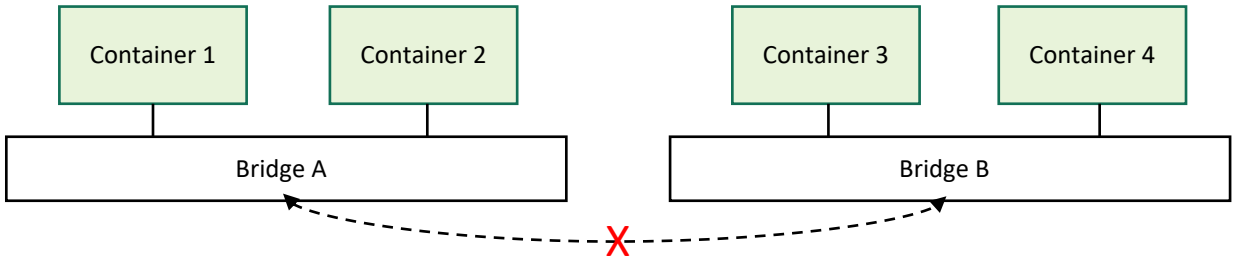
- Containers connected **to different bridges do <u>not</u> communicate with each other**.



- Now exit from the `test1` container. Then create a second bridge on your test system (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 `test1` container again, but this time connect it directly to `my-bridge`:
  `docker run --rm -it --network=my-bridge --name=test1 alpine sh`
- Are the two containers still able to communicate with each other?
  - One container is connected to the bridge `bridge`, the other to the bridge `my-bridge`.
  - Check the IP address on `test1` now.

Why create user-defined bridges then? (1)

- What could be the use of creating bridges other than the default `bridge`, if containers attached to them are not able to communicate with containers on other bridges?



Why create user-defined bridges then? (2)

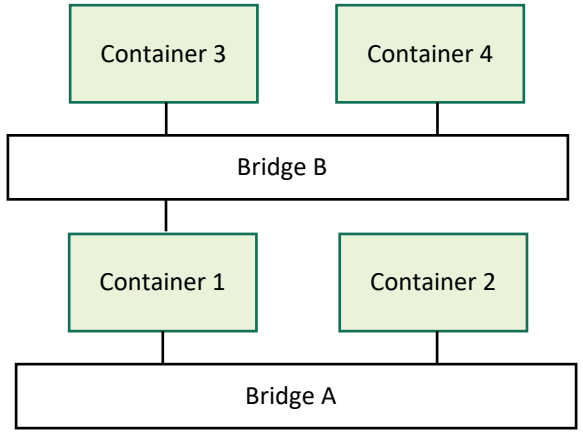
- What could be the use of creating bridges other than the default `bridge`, if containers attached to them are not able to communicate with containers on other bridges?
  - Security! You can easily create many containers on a Docker host. You just may not want to allow all of them to see each other.
  - A container on a given bridge automatically sees all the ports of all the other containers on the same bridge.
    - How can you then make a certain port on a container available to containers attached to other bridges?

Why create user-defined bridges then? (3)

- What could be the use of creating bridges other than the default `bridge`, if containers attached to them are not able to communicate with containers on other bridges?
  - Containers on a bridge different from the default `bridge` have automatic name resolution (DNS). Try this:
    - On `test2`, issue the command `ping test1`. Does it work?
    - Exit from `test2`, and connect it this time to `my-bridge`: `docker run --rm -it --network=my-bridge --name=test2 alpine sh`
    - Now, from `test2`, type `ping test1` again. You will see that this time Docker automatically resolves the name `test1` to the actual IP address of `test1`.

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 the containers connected to a certain 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
  ```

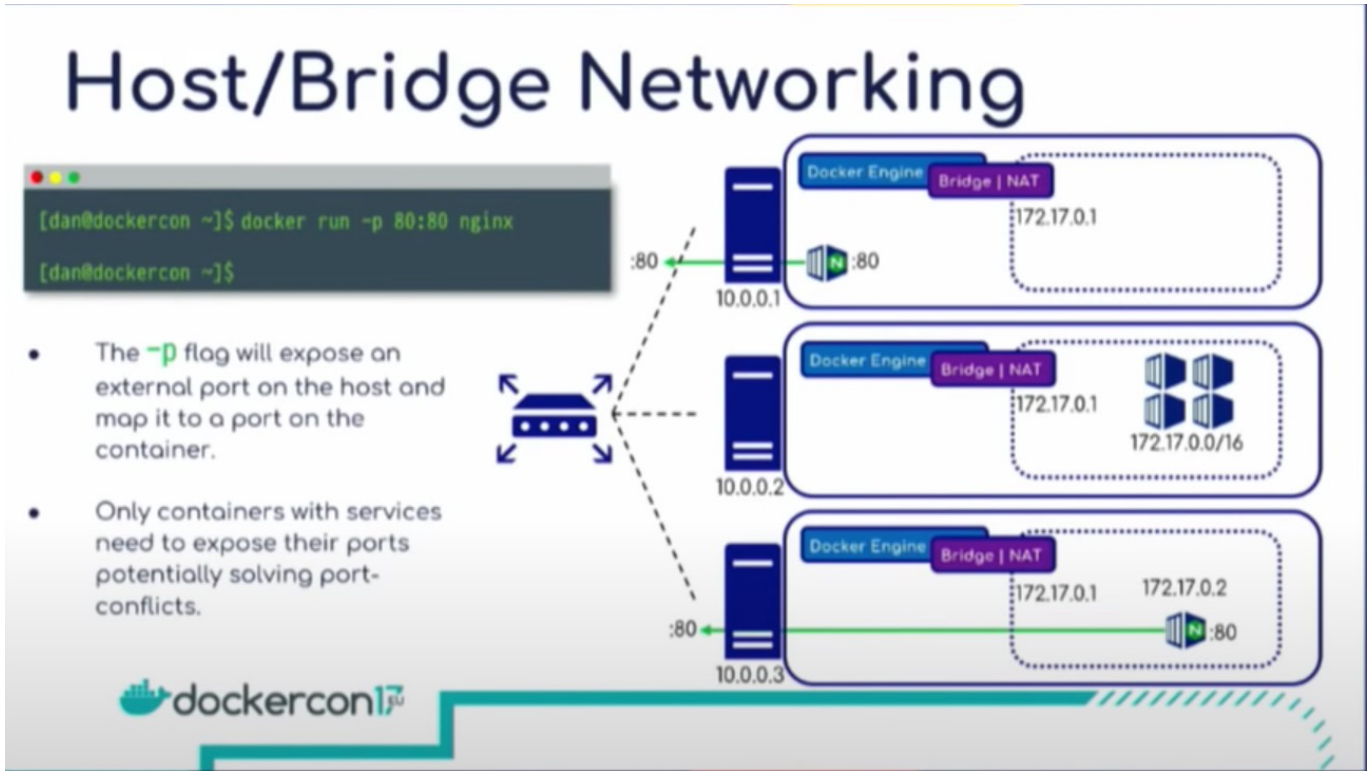
  you will see the **real IP address** that your `test1` container uses to connect to the internet. This is not the IP address shown by `ip address show` on `test1`!

- Where does this address come from?

Network Address Translation (NAT)

- Our `test1` container was able to ping the internet. However, it was not able to ping another container instantiated 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 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 public IP address of the Docker host.

Recap of bridged networks



Source: https://www.youtube.com/watch?v=PpyPa92r44s

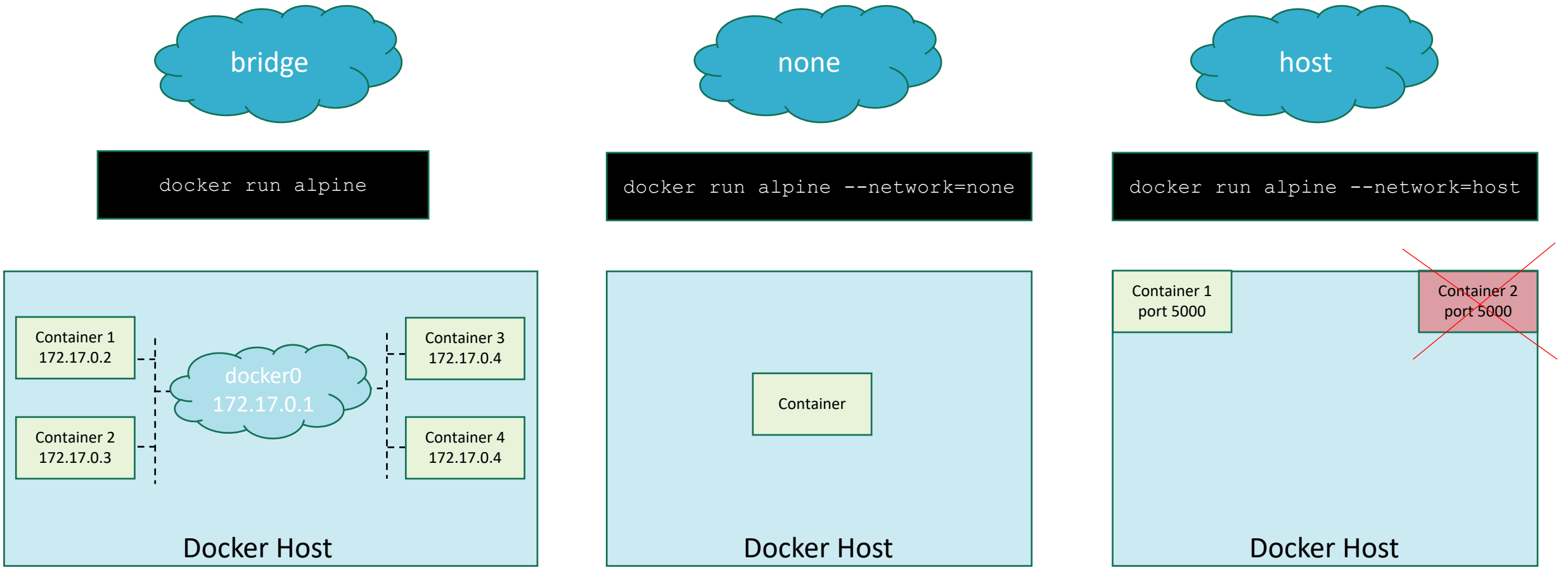
Host networking

- 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 for example 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.

macvlan networking

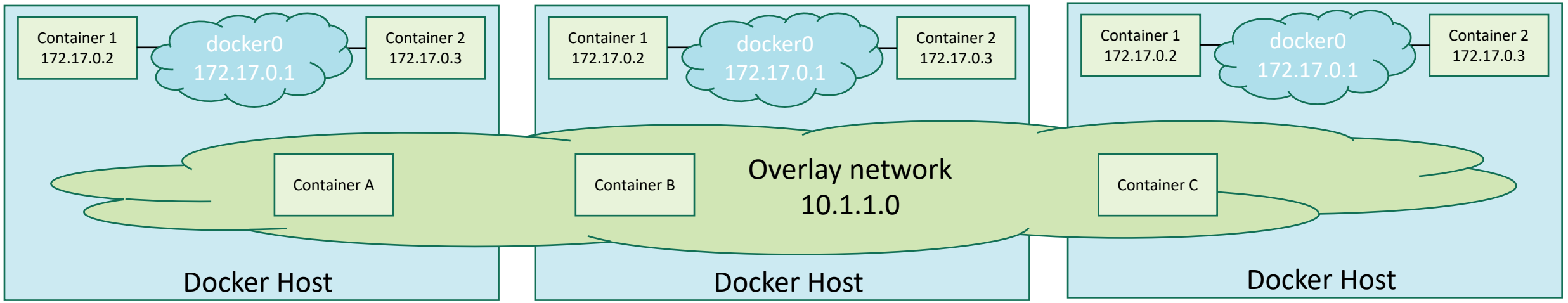
- Another type of Docker networking is the so-called **macvlan** mode (it has nothing to do with Apple Macs). With this mode, it is possible to assign an individual MAC address (MAC = Media Access Control is a unique identifier normally assigned to a *physical* network interface).
  - A container in macvlan mode has its very own MAC address and IP address. No NAT is used.
  - The usage of this mode is also very specific and should be carefully considered. Using macvlan, if you instantiate for example 100 containers on a single Docker host, you will have 100 MAC addresses and 100 IP addresses to allocate. This may easily lead to IP address exhaustion.

The three main types of Docker networks covered so far



Overlay networks

- So far, we have considered network configurations that were applicable to containers running on the same Docker host.
- **Overlay networks** connect Docker daemons running on multiple hosts. Due to time constraints, we won't discuss them here at SOSC22.



Process management

- Once you start a container, you may want to check how it is performing. For example, what are the running processes, or how much CPU of the Docker host it is using, how much RAM, etc. You can typically 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**.
- As an 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 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 your test system, create a container called `test1` (in case you had created a `test1` container without specifying the docker option `--rm`, remember to delete it first with `docker rm test1`):
  `docker run --rm -it --name=test1 alpine sh`
- Install `python` on the `test1` container:
  `# apk update && apk add python3`
- Now create the `mypi.py` program on the `test1` container and run it with `python3 mypi.py`. It will take some time to finish (the Leibniz formula is not a very efficient way to compute Pi).
- Now open another terminal on your test system and type `docker top test1`. You should see the **running processes** on `test1`, something like this:

```
ubuntu@VM1:~$ docker top test1
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                4300                4279                0                   09:43               pts/0               00:00:00            sh
root                4441                4300                26                  09:46               pts/0               00:00:05            python3 mypi.py
ubuntu@VM1:~$
```

Process management: `docker stats`

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

```
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 the resources on the Docker host as much as it is allowed by the Docker host kernel scheduler.
- For example, if you do not limit the memory that a container can use, the Docker host could run out of memory and throw an *Out of Memory* exception. When this happens, the kernel starts killing processes to free up memory. The problem is that you don't know in advance *which processes* the kernel is going to kill.
- In practice, if a Docker host runs out of memory, for example because of a container misbehaving, the entire system could crash (that is, 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 for instance memory to 256MB for a container running the `nginx` image with
  `docker run --rm -d -p 8080:80 --memory="256m" nginx`
- Similarly, you can limit the number of CPU cores that a container is allowed to use with
  `docker run --rm -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 --rm -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 the standard output (`echo date`) every second.
- The container is running in the background, but you may check what it is printing with the command `docker logs --follow test1`. Try it out.
  - 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`

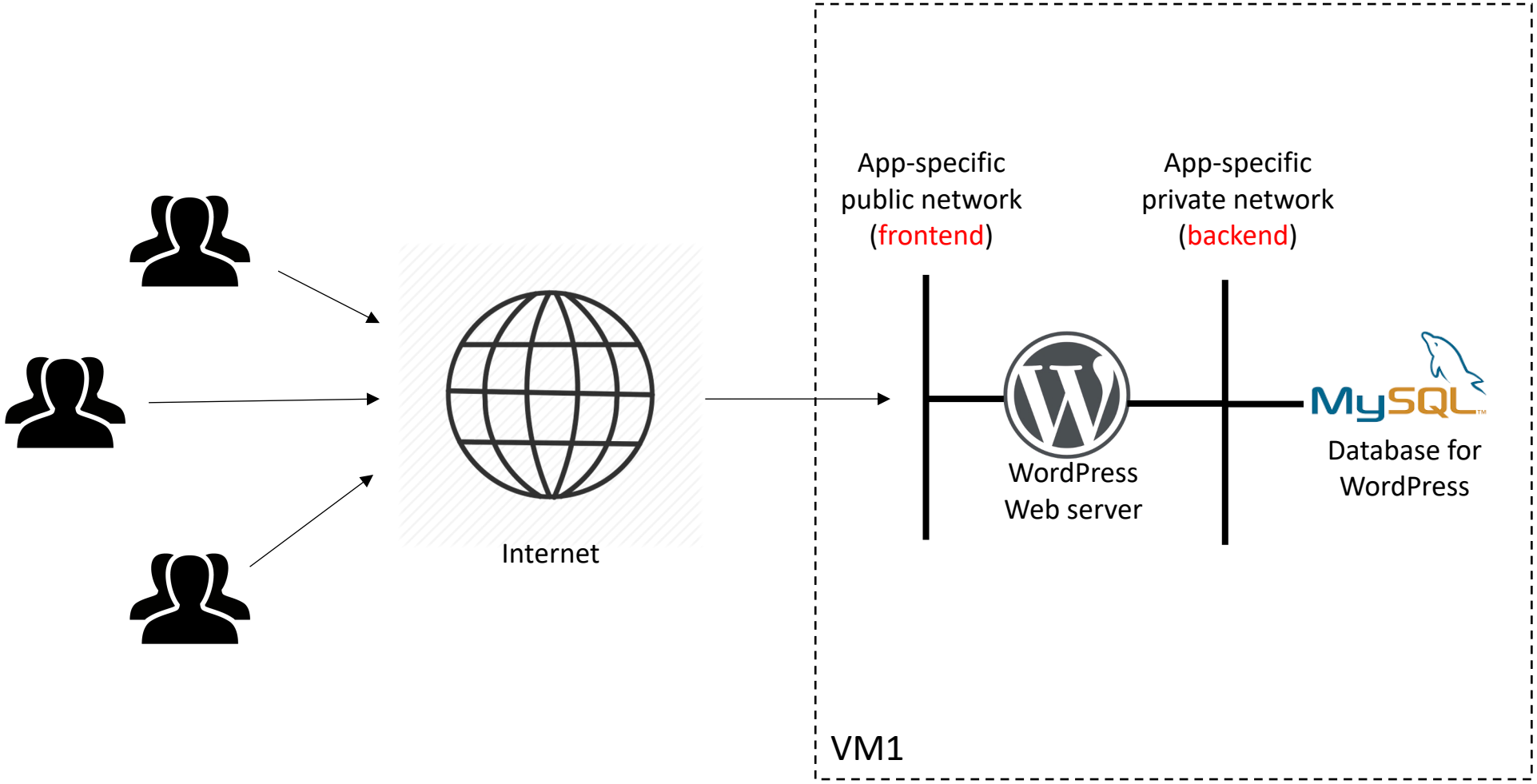
Application stacks: docker-compose

- 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.

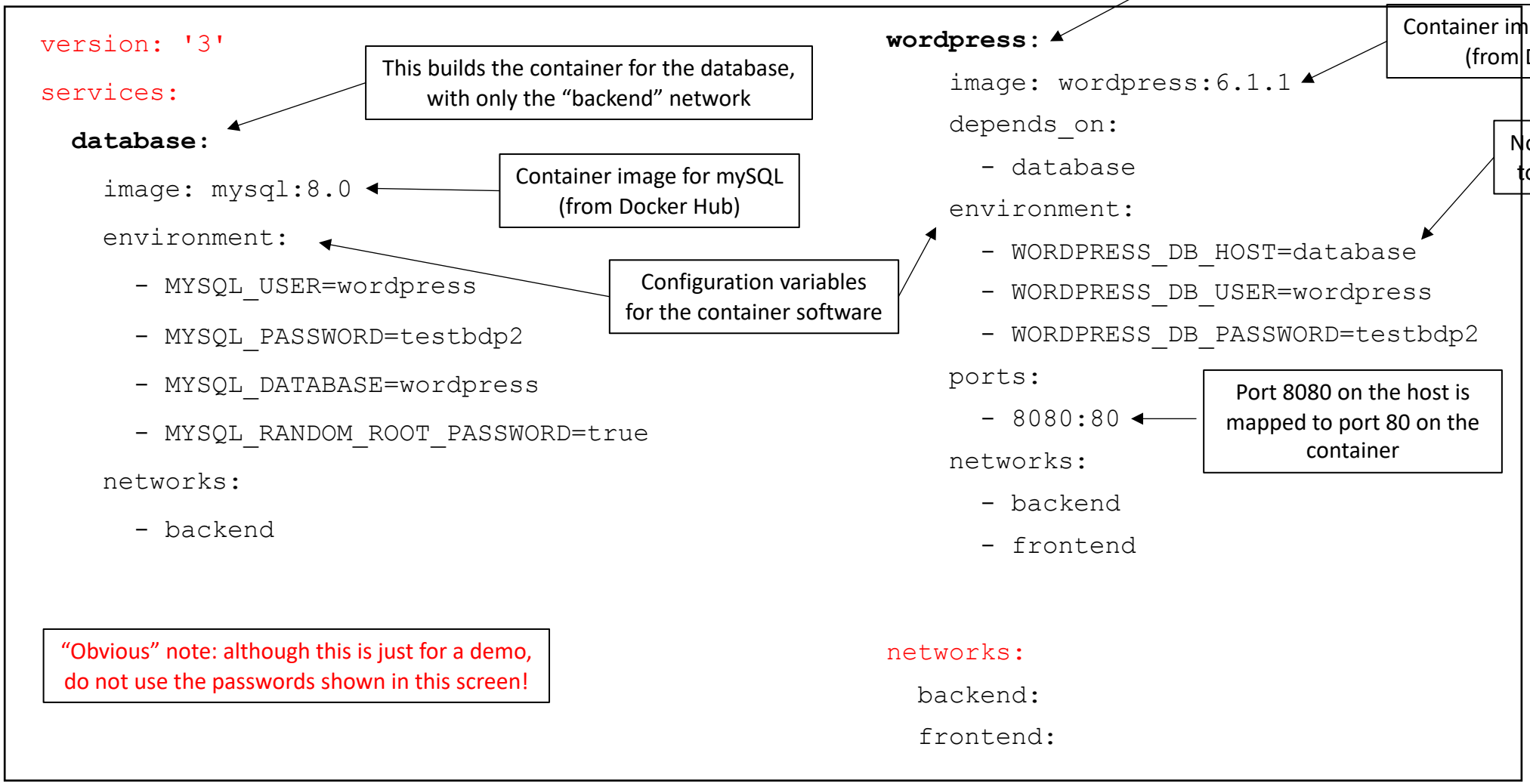
A scenario for docker-compose

- `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 your test system:
  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

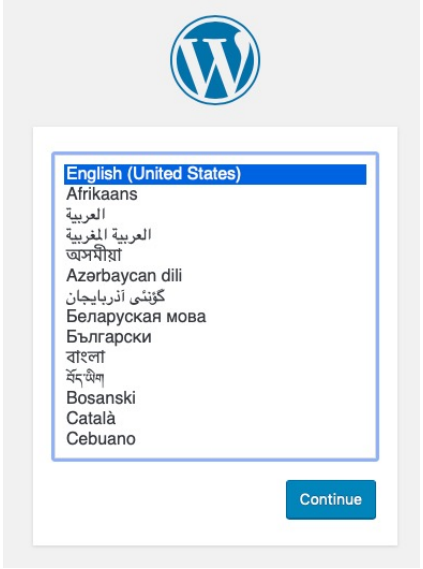


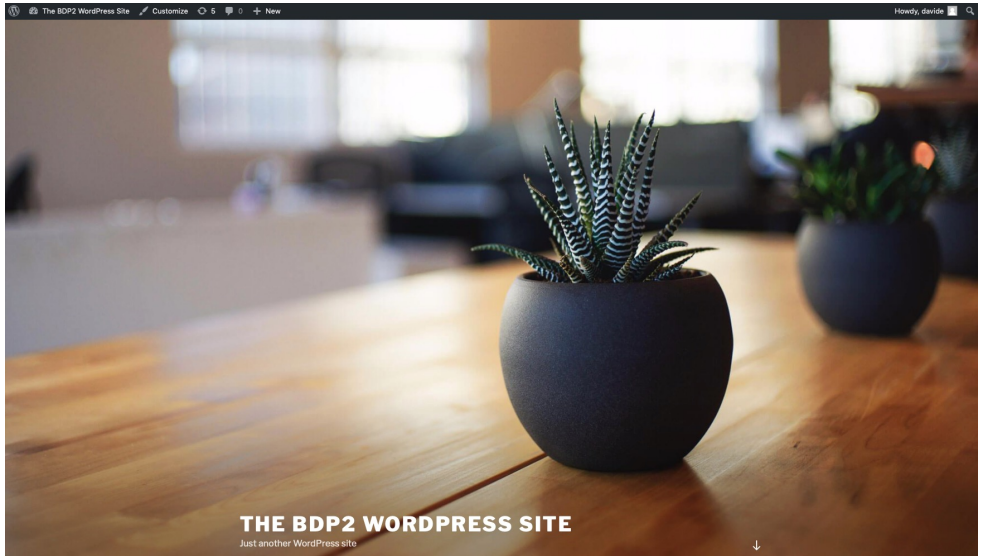
docker-compose.yml



Build & run the application stack

- Build the application stack:
  ```
  # docker-compose up --build --no-start
  ```
- Now start it:
  ```
  # docker-compose start
  ```
- If you now open a browser pointing to the public address of the test system on port 8080 (look at the previous `docker-compose.yml`), you should get the set-up page for WordPress shown on the right. Go on and set it up.
- Once WordPress is set up, you should see the default WordPress home page, like 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
  ```





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:
    my_volume_1:
    my_volume_2:
services:
    application_1:
            volumes:
                    - my_volume_1:/app1/dir
            […]
    application_2:
            volumes:
                    - my_volume_2:/app2/dir
[…]
```

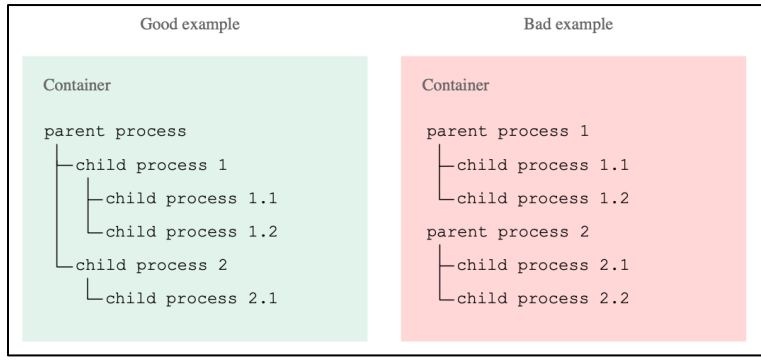
This automatically creates the Docker volume `my_volume_1`, mapping it to the directory `/app1/dir` on the container

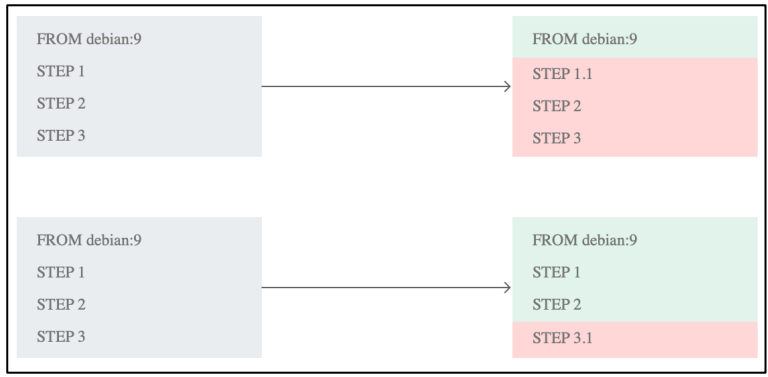
Limitations of `docker-compose`

- As seen, `docker-compose` is very handy to create combinations of containers running on the same machine (your test system).
- 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. This part deserves substantial time of its own, and we won't cover it here.

Some best practices for creating 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. Explicitly **define the entry point in the container** with the CMD command in the Dockerfile.
3. If in a Dockerfile you have **commands that change often, put them at the bottom of the Dockerfile**. This way, you speed up the process of building the image out of the Dockerfile.
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**.
  - A largely Docker-compatible tool which does not require root access for the installation is `udocker` (https://github.com/indigo-dc/udocker). Feel free to explore its use on your own.
- 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 (and include for example viruses or trojans)!
- How can you send passwords, certificates, encryption keys, etc. to tasks / applications in a Docker swarm cluster? **Do not** embed them into the containers, and **do not** store them for instance in GitHub repositories!
  - Docker has a "Secrets Management" feature, which is a standardized interface for accessing secrets. See https://dockr.ly/2H4M5SU for details.
  - Other resource orchestrations, such as Kubernetes, have similar solutions.

A few words on Docker security (2)

- If the *host* where the Docker daemon is running gets compromised, container isolation is gone. So, it is important to make sure that the **Docker host system is properly secured** (that is, you should regularly update it!).
- On other hand, there could be exploits that make it possible for **containers to bypass isolation** (remember that the Docker daemon requires root privileges) and get access in privileged mode to the host system.
- Since you can so easily start up containers on a system, there is the possibility of a **Denial-of-Service** attack, targeting to consume all resources on the host system.
- **Do not assume that containers are immutable!** They might contain outdated software, that must be periodically patched and upgraded.
- For more details, see http://bit.ly/2kEpV16.

Recap of Containers

- We covered the basic concepts about **Containers**, comparing them to Virtual Machines.
- We saw how to run a container out of a Docker image, list Docker images and extend them to create new images.
- We then saw how to simplify image building via Dockerfiles.
- We created a container serving web pages; we then connected containers to external file systems, to volumes and to `tmpfs` mounts. We also learned how to export and import containers.
- We studied also how to combine multiple containers in an application stack with `docker-compose`.
- We then reviewed some best practices in creating containers, and discussed some Docker limitations, in particular for what regards security.
- It is now your time to practice what was discussed through hands-on, using the test systems that are provided to you in SOSC22.