

# Containers, their composition and orchestration

Davide Salomoni

[davide@infn.it](mailto:davide@infn.it)

# Why this lecture?

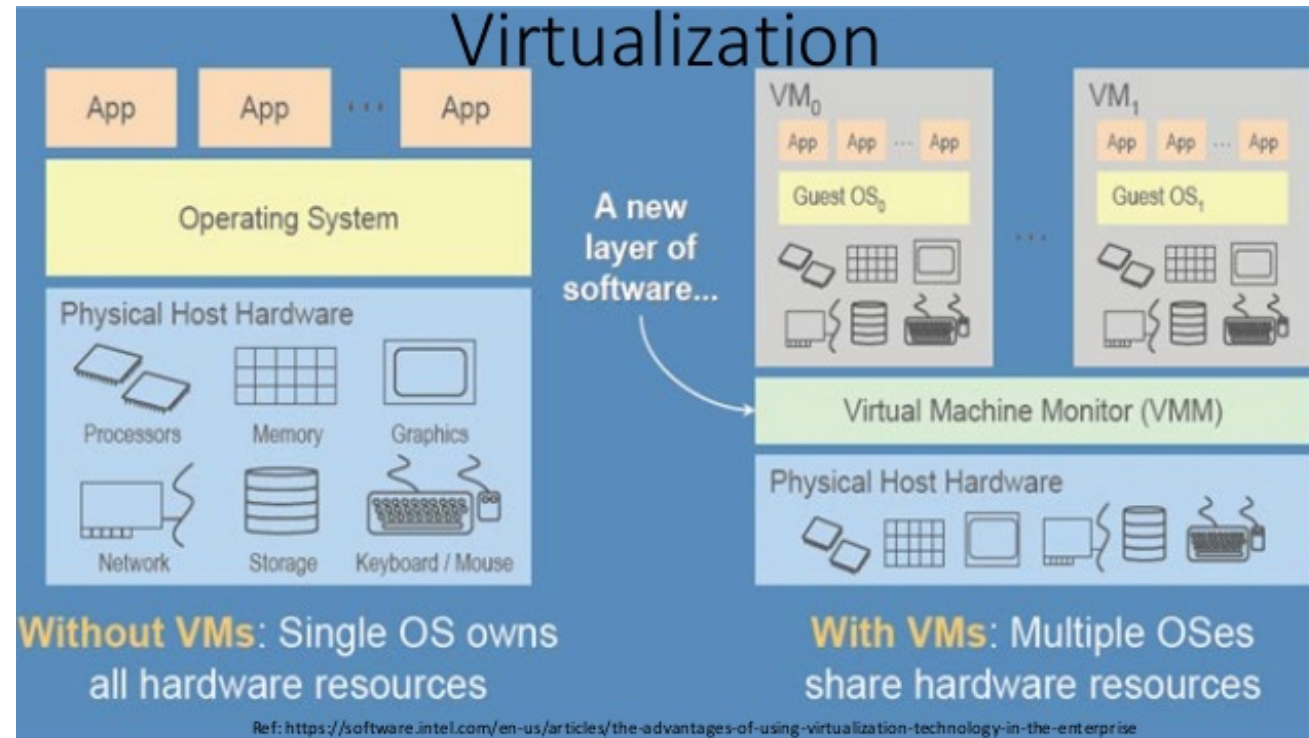
- In the first days of the school, we have seen that working with “intelligent systems” requires a significant number of skills in several topics (well beyond the “machine learning” area).
- We said that Clouds can help us to find and use computing and storage resources that we need for a variety of tasks, be they scientific or not.
- So far, we just *used* pre-packaged Cloud resources, in a variety of forms (e.g. web services, virtual machines, distributed storage). This is often OK in simple cases.
- In this and in the following lectures of today, we are going to see and explore ways to *customize* Cloud resources for more complex tasks.

# Agenda

- In this lecture we are going to cover:
  - Virtualization (we spoke several times already about “Virtual Machines”: what are them, really?)
  - Containers, their properties, their pros and cons.
  - How to embed your applications into containers and automatize the process of creating complex containers.
  - How to combine multiple containers into a so-called *application stacks*.
  - How to orchestrate containers across multiple nodes and scale their number up and down.

# Virtualization

- Informally, a Virtual Machine (VM) 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.

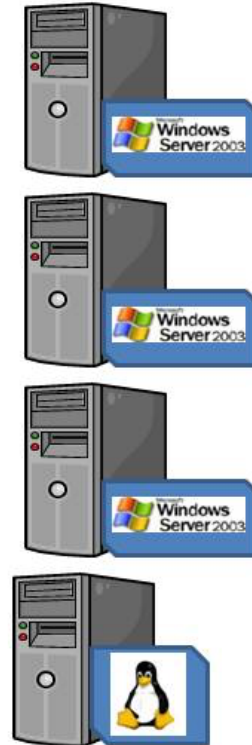




# Going virtual

Source: <http://bit.ly/2IVk6e5>

## Workloads without Virtualization



- Servers poorly utilized at average of 4% to 7% capacity
- Limited in failover capability
- Prone to hardware failure



## Workloads migrated To Virtual Machines Using Virtualization

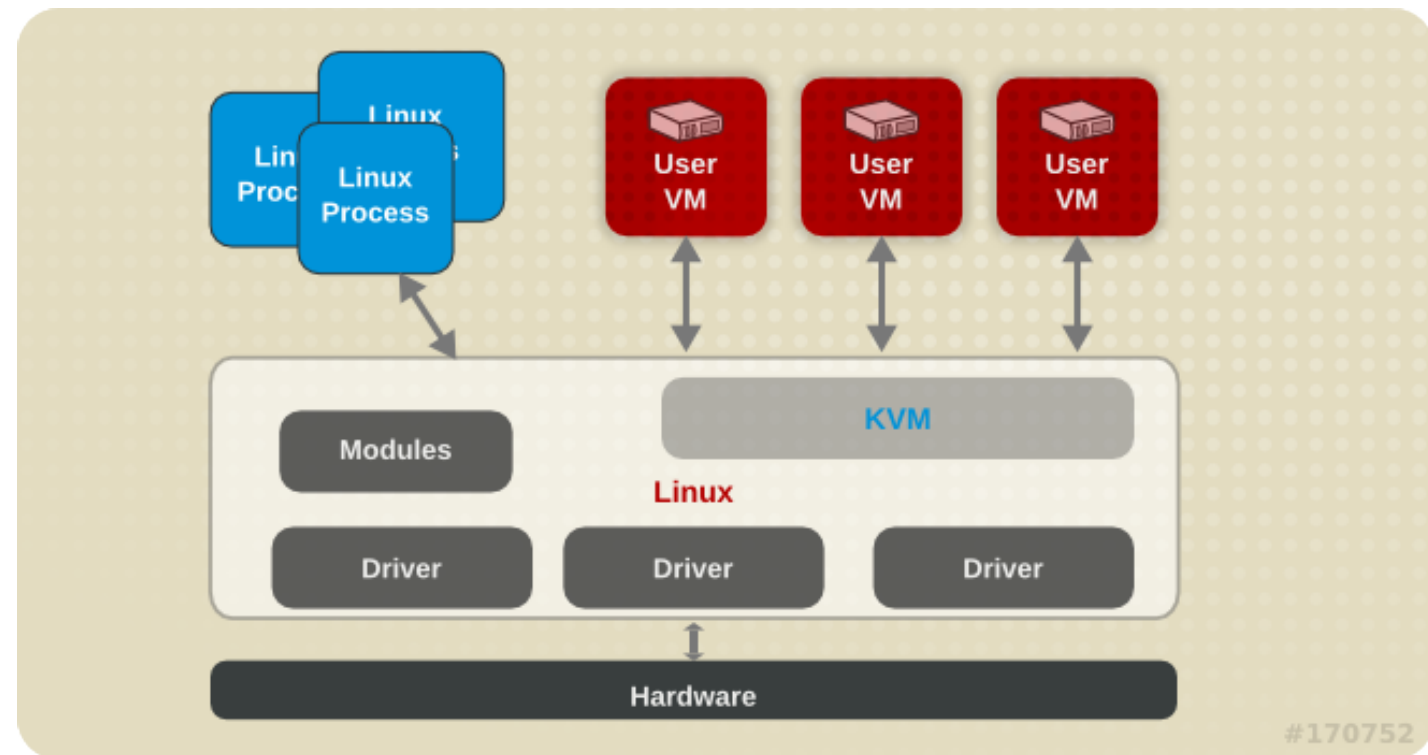


- Each workload is now encapsulated stacking its workload for better hardware utilization – around 80%
- Inherit virtualization capabilities include:
  - Dynamic resource pools
  - High availability without complicated clustering
  - Provision new servers in minutes
- Virtual Machines are hardware independent

# Virtualization with Linux KVM

- KVM is a *kernel module* to implement virtualization in Linux (there are other ways to handle virtualization in Linux, but we won't discuss them).

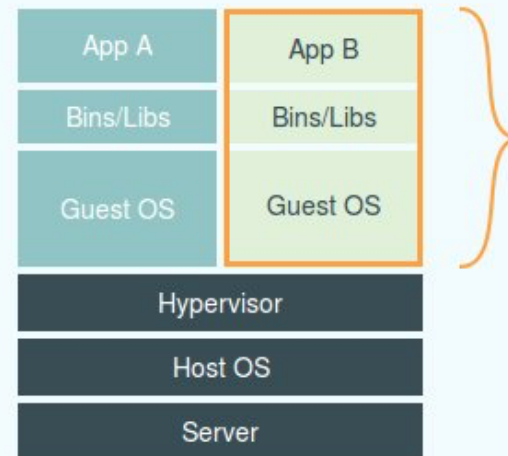
Source: <https://red.ht/2IUxJdr>



# Beyond Virtual Machines

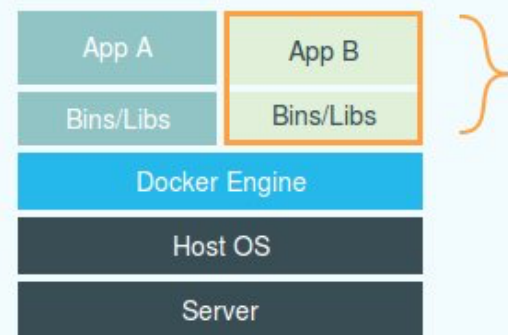
- Virtual Machines (VMs) carry a significant overhead with them → let's introduce **Docker Containers**.

Source: <http://bit.ly/2IVk6e5>



## Virtual Machines

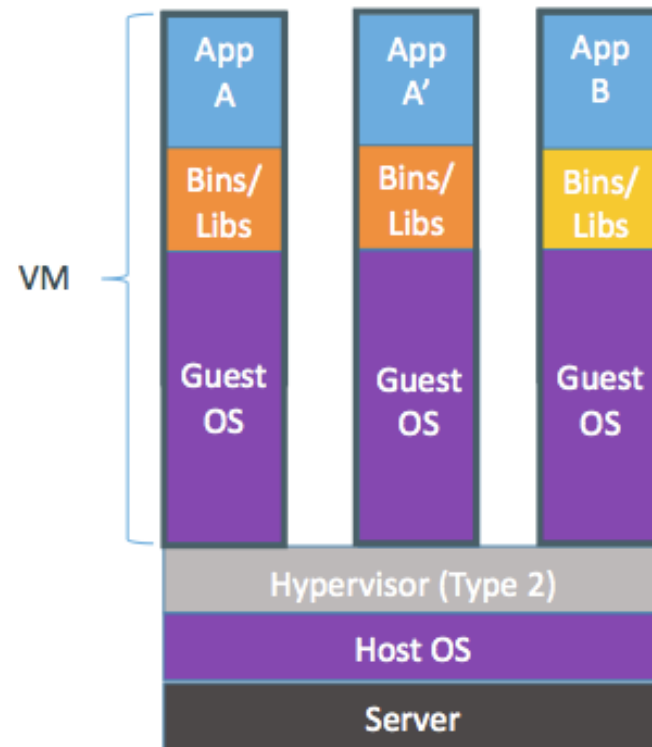
Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



## Docker

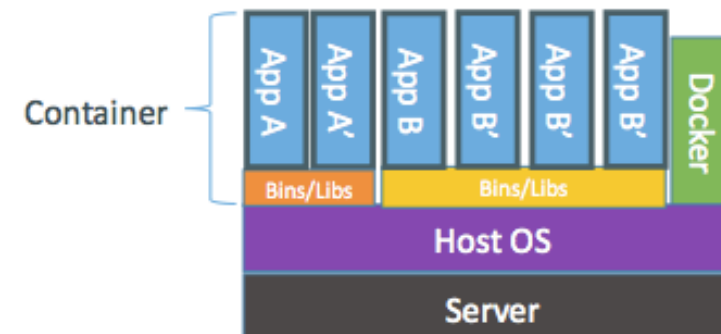
The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

# Containers are «lightweight VMs»



Containers are isolated,  
but share OS and, where  
appropriate, bins/libraries

...result is significantly faster deployment,  
much less overhead, easier migration,  
faster restart



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.

# Cargo Transport Pre-1960

Multiplicity of Goods



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

Multiplicity of  
methods for  
transporting/storing

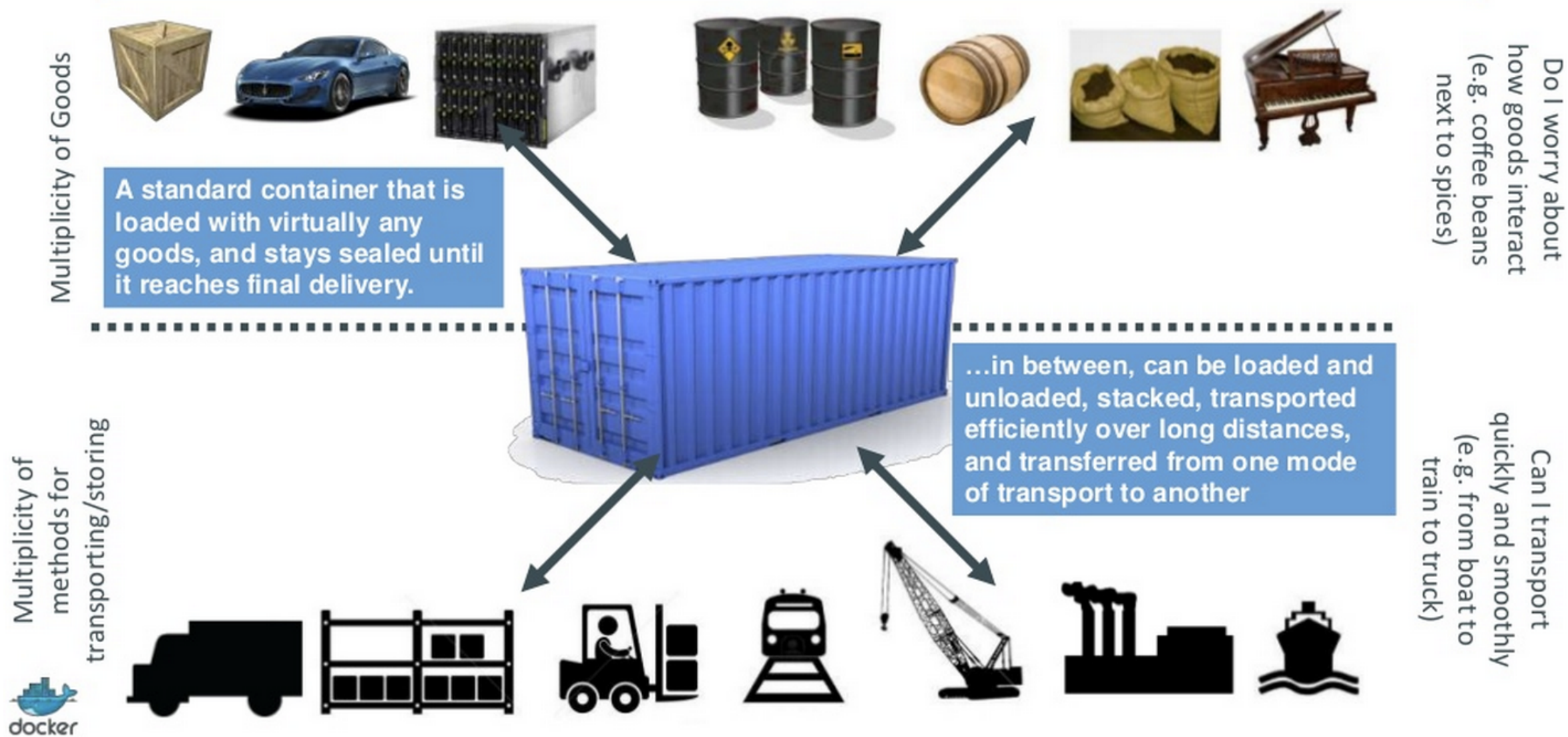


Can I transport quickly  
and smoothly  
(e.g. from boat to train  
to truck)





# 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

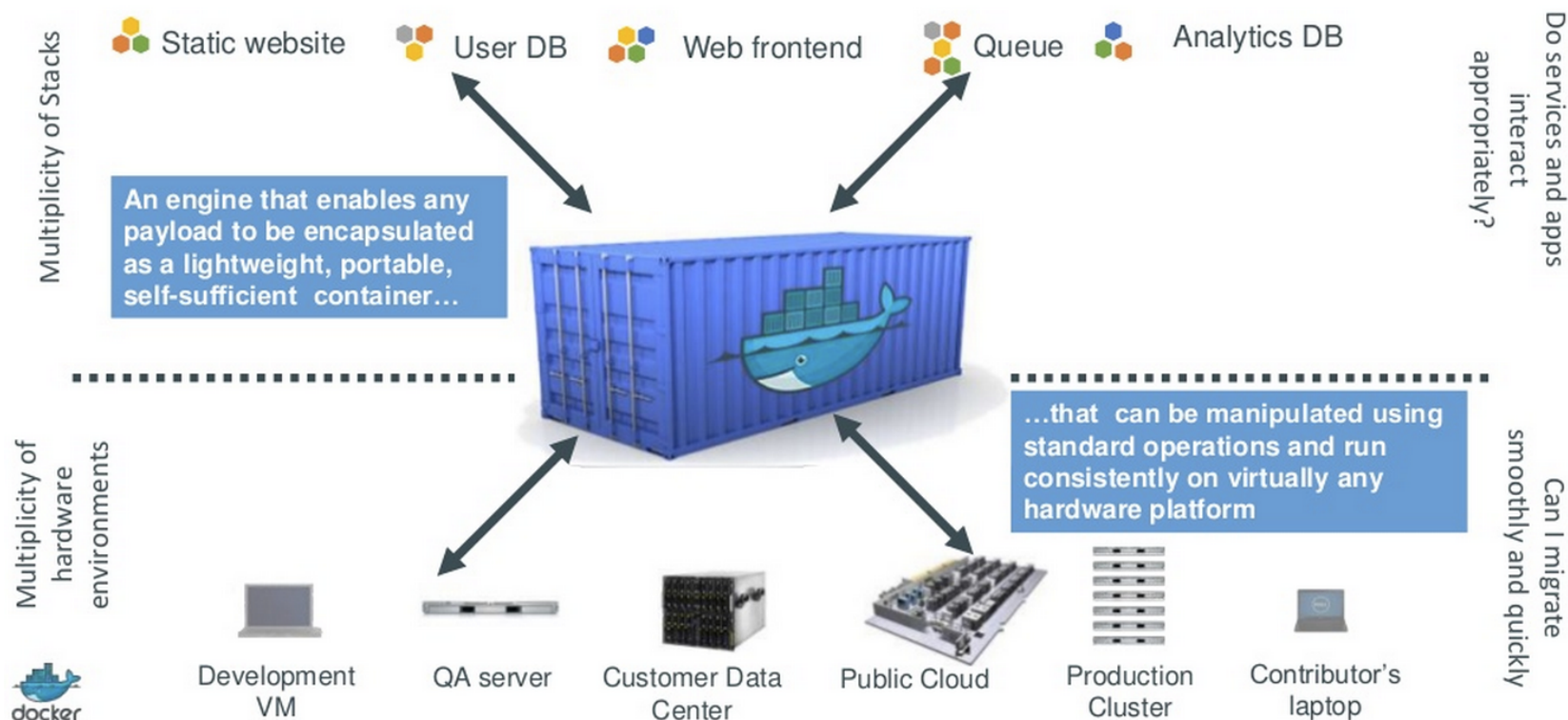


# OK, not everything always goes as planned...



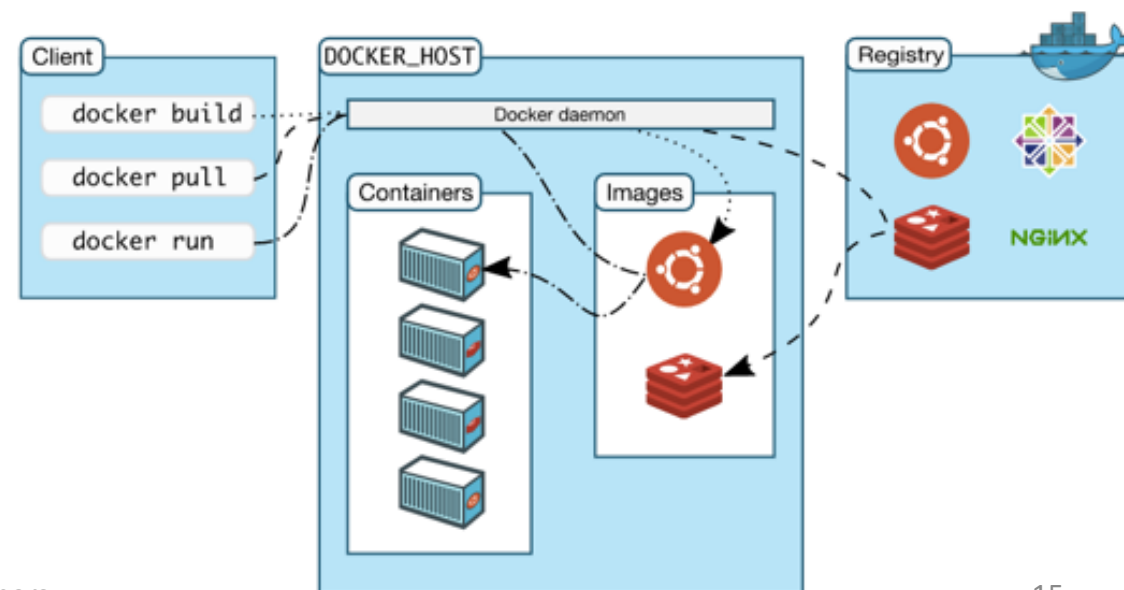
# Docker Containers

Docker is a shipping container system for code



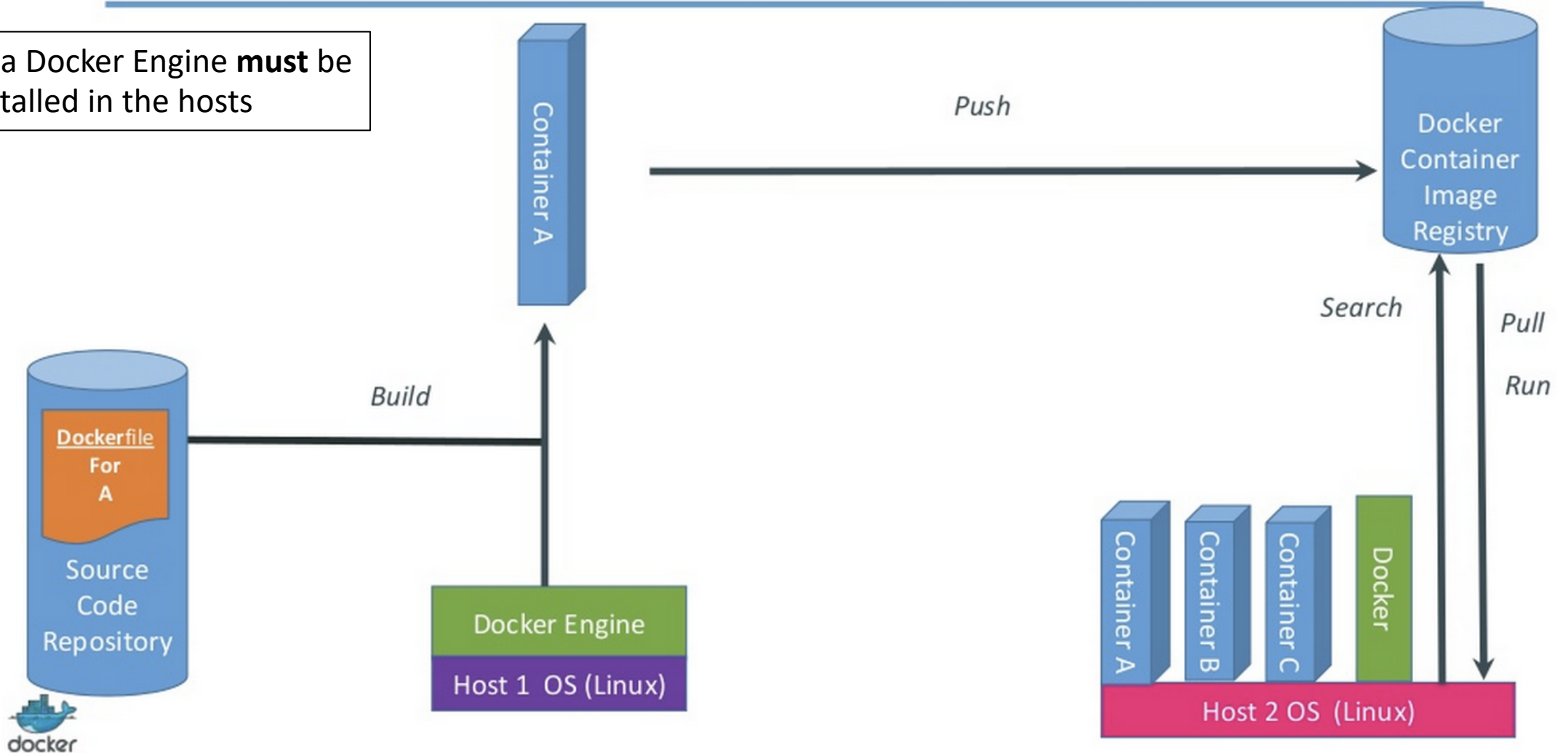
# Docker features

- Docker is an open source engine for the easy creation of lightweight, portable, self-sufficient containers from any application.
- The *same container* that a developer builds and tests on a laptop can run at scale, in production, on VMs, private, public clouds and more.
- Main features:
  - versioning (git-like)
  - component re-use
  - sharing (with public repositories)



# What are the basics of the Docker system?

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





# Hands-on preparation (1)

- In the hands-on exercises, we will be working **with 2 VMs**.
  - One of the two will have both a public and a private IP address. **We shall call it VM1.**
  - The second one will have a private IP address only (on the same subnet of the private IP address of the first VM). This IP address is of the form 172.16.x.y. This is the VM called `sosc19-XXp` (note the 'p' at the end). **We shall call it VM2.**
- Open a terminal shell with two tabs, or two terminal shells, and open an ssh connection *in each of them* **to the VM with the public IP address** (since it's the only one reachable from the Internet), i.e. "VM1".
  - Use the VM and the username that was assigned to you.
  - i.e. `ssh soscuserXX@<public IP address>`
- From one of the two shells, connect to the VM with the private IP address (i.e. "VM2").
  - i.e. `ssh soscuserXX@<private IP address>`
- **You should now be connected to both VM1 and VM2.**
  - Check with the `ifconfig` command that VM1 has 2 IP addresses and VM2 only 1 (plus the loopback address, which is always 127.0.0.1).

# Hands-on preparation (2)

- To make it more clear which VM we are working on, **we will change the Linux prompt.**
  - By default, the prompt is something like “`soscuserXX@sosc19-XX:~$`”. Let’s set it to “`soscuserXX@sosc-19-XX (VM1) :~$`”. To do this, you need to modify the file called `.bashrc` (note the dot at the beginning of the file) in your home directory.
  - On VM1, add the following string *at the end* of `.bashrc`, using your preferred editor (e.g. `vim` or `nano`):

```
PS1="\[\033[01;32m\]\u@\h (VM1) \[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ "
```

    - If you want to understand what this means in detail, see <https://www.ostechnix.com/hide-modify-usernamelocalhost-part-terminal/>
  - **Activate the change**, running `source .bashrc` (or logging out and then back in).
  - Do the same on VM2, changing VM1 to VM2 in the PS1 value above.

# Hands-on: the Docker *daemon*

- A **computer daemon** is a program that runs as a **background process** (versus, for example, a program that is run by an interactive user).
- In order to use docker on a machine, that machine must have the Docker engine (or the Docker daemon) installed and running. By default, this daemon is not normally installed.
  - Is it installed on VM1? How can you check?
- Install the Docker daemon on VM1 with the command  
`ubuntu@VM1:~$ sudo apt install docker.io`
- Check that Docker is properly installed with the command  
`ubuntu@VM1:~$ docker --version`  
It should return something like  
Docker version 18.09.5, build e8ff056

# Docker needs root access

- When we installed Docker, we had to write  
`sudo apt install docker.io`
- This is because, in order to install a program on a global basis on a system, we need root privileges. However, **we need to have root privileges also when we execute any docker command**, such as `docker info` (checking the docker version is an exception).
- If you don't use `sudo` before a `docker` command:

```
ubuntu@VM1:~$ docker info
Got permission denied while trying to connect to the Docker daemon
socket at unix:///var/run/docker.sock: Get
http://%2Fvar%2Frun%2Fdocker.sock/v1.39/info: dial unix
/var/run/docker.sock: connect: permission denied
```



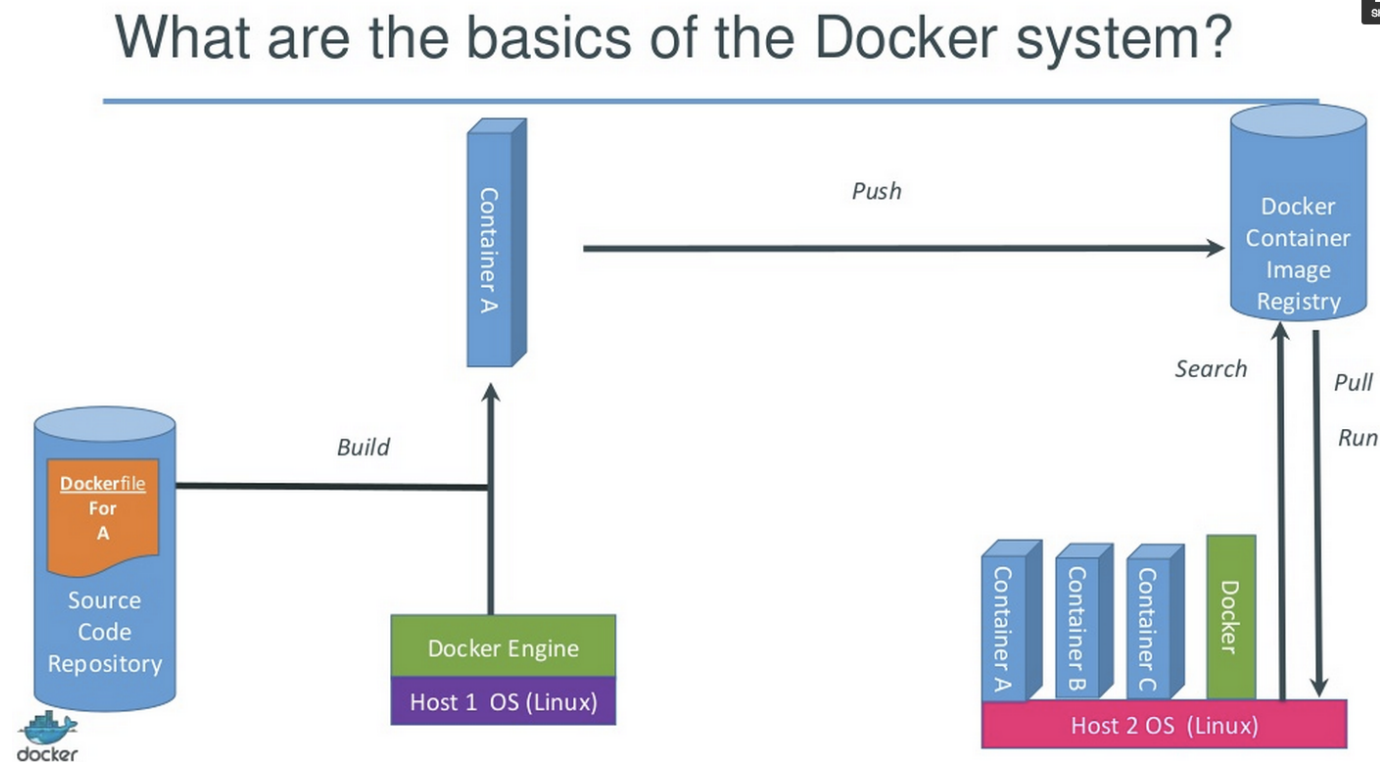
# docker commands without sudo

- To avoid specifying `sudo` before each docker command, we'll add our username to the docker Unix group:  
`ubuntu@VM1:~$ sudo usermod -aG docker ${USER}`
- Log out from and then log back in to VM1 in order to apply this. From now on, we can omit the `sudo` command before docker:

```
ubuntu@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 pull a container 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 VM1:**
  - **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 ubuntu echo "hello from the container"`  
hello from the container
    - Run a container in interactive mode:
      - `docker run -i -t ubuntu /bin/bash`
  - **Ship (push)** a Docker image to a Docker repository (by default, Docker Hub) – skip these commands for the time being, we’ll say more about this later on:
    - `docker login`
    - `docker push USER/my-image`

# How efficient is docker?

```
ubuntu@VM1:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	7698f282e524	7 days ago	<b>69.9MB</b>

→ 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.

```
ubuntu@VM1:~$ time docker run ubuntu echo "hello from the container"
hello from the container
```

```
real 0m0.988s
user 0m0.039s
sys 0m0.015s
```

→ 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 a second. How long would it take if we used a full VM?

# How to extend a docker container (1)

- Suppose you need to run a command inside a container, but that command 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 is not available:
  - `ubuntu@VM1:~$ 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`:
  - `ubuntu@VM1:~$ docker run ubuntu /bin/bash -c "apt update; apt -y install iputils-ping"`
- But it still doesn't work!
  - `ubuntu@VM1:~$ 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.
- Who can explain this? 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 the `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 is now working:

```
ubuntu@VM1:~$ 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:

```
ubuntu@VM1:~$ 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`:

```
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

- Besides containers, you can also remove images that 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: sha256:0dfbdc7dee936a74958b05bc62776d5310abb129cfde4302b7bcd0392561496
```

```
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

# Pushing images to Docker Hub (1)

- We have already seen the command `docker push <image>`. This writes an image **to Docker Hub**.
- In order to issue that command, you first need to have 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: everybody will be able to see the images you upload there!) and give it a name, for example *sosc19* (only lowercase is allowed), a description, and click on “Create”. This will create your public repository, called e.g. “sosc19”.

# Pushing images to Docker Hub (2)

- To push an image (for example the `ubuntu_with_ping` image we created earlier – create it again if you deleted it) 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 (username/repository) should be “dsalomoni/sosc19”. 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

```
ubuntu@VM1:~$ 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
```

```
ubuntu@VM1:~$ docker tag 7c45b9ad4de6 dsalomoni/bdp2:ubuntu_with_ping_1.0
```

Images after  
the new tag

```
ubuntu@VM1:~$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
ubuntu_with_ping    latest       7c45b9ad4de6     About an hour ago  97.2MB
dsalomoni/sosc19    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:

- **ubuntu@VM1:~\$** 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: dsalomoni

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

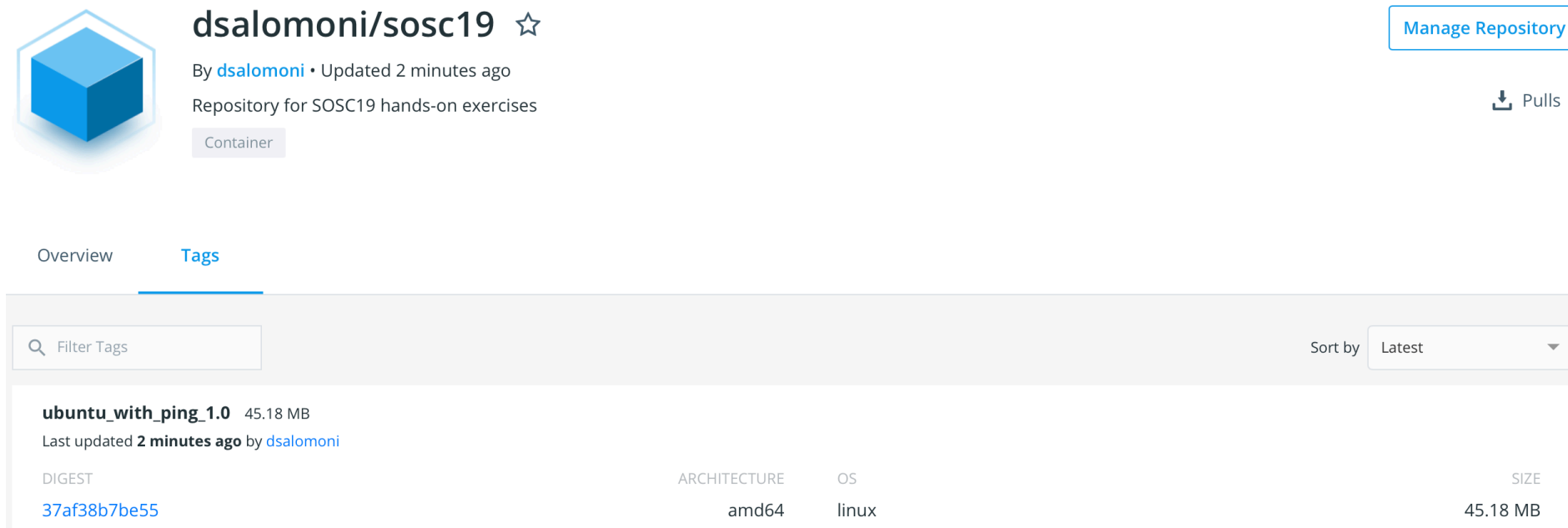
- Finally, we can push our image to Docker Hub:

- **ubuntu@VM1:~\$** docker push dsalomoni/sosc19:ubuntu\_with\_ping\_1.0

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

# Verifying our Docker Hub repository

- Go to Docker Hub (<https://hub.docker.com/>), login with your username, click on the “sosc19” repository, and then on “Public View” and on the tab “Tags”. You should see something like this:



The screenshot shows the Docker Hub repository page for **dsalomoni/sosc19**. The repository is marked as a "Container" and has a "Manage Repository" button. It shows 1 pull. The "Tags" tab is selected, displaying a list of tags. The first tag is **ubuntu\_with\_ping\_1.0**, which is 45.18 MB in size, last updated 2 minutes ago by dsalomoni. The digest is **37af38b7be55**. The architecture is **amd64** and the OS is **linux**.

DIGEST	ARCHITECTURE	OS	SIZE
<a href="#">37af38b7be55</a>	amd64	linux	45.18 MB

# 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).

# Dockerfiles

- Rather than modifying a container “by hand”, i.e. connecting interactively and then installing packages one by one 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
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 SOSC19.`  
`</html>`
- **Create** both the previous Dockerfile and the `index.html` file in your home directory on VM1.



# 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 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).
- If you forget the `-d` flag, you won't be able to interrupt your container interactively, and you will have to issue `docker stop <id>` from another shell.
- 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:

```
ubuntu@VM1:~$ docker ps
```

```
CONTAINER
```

ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORT
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 -d -p 8080:80 web_server` any time you want to instantiate a new web server.
- What happens if you type `docker run -d -p 8081:80 web_server` ?

# 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 -i -t 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 -i -t 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

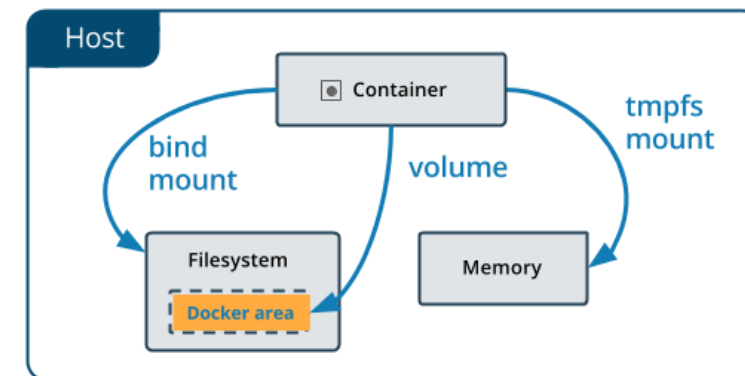
- So, what if we want to retain data within a container?
- We can map a directory that is available *on the host* (the system where we run the docker command, e.g. VM1), to a directory that is available *on the container*. This is done with the docker flag `-v`, like this:  

```
docker run -v /host/directory:/container/directory <other docker arguments>
```
- For example:
  - Create a directory in the scratch space `/scratch` and name it with your username with `mkdir /scratch/`whoami`` (**note the inverted ticks!**)
  - Create a file in that directory with `touch /scratch/`whoami`/a_newer_file` and check it is there with `ls /scratch/`whoami``
  - Now **map** that directory to the directory `/container_data` *on the container*:  

```
docker run -v /scratch/`whoami`:/container_data -i -t ubuntu /bin/bash
```
- Now, when you are *within the container*, if you write `ls /container_data` you should see the file you just created. **Do it now.**

# Connect a container to a 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 volumes are persistent, but are not tied to the specific filesystem of the host, and are completely managed by Docker itself.



We'll see what a tmpfs mount is later on

# Connect a container to a Docker volume (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 -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. Check what happened with `docker ps -a`.
- What do you see with the command `df -h` issued *within the 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
```

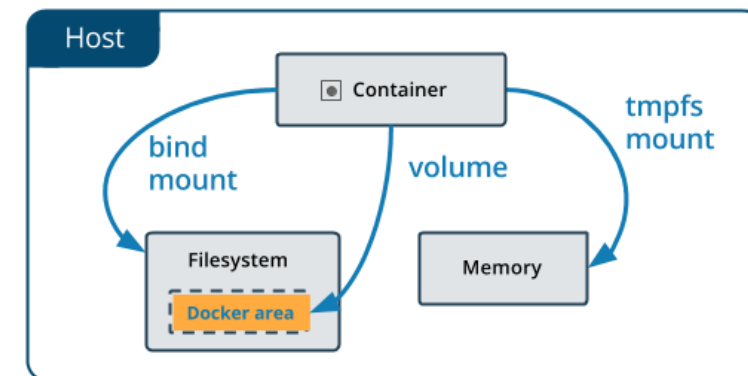
  - Note that the `docker system prune` command we showed previously **does not remove volumes!**



# tmpfs mounts

- If you are running Docker on Linux (this is the case for us here), 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 volume (be it filesystem-based or docker-based).
- 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 -it --name mytmp --tmpfs /app ubuntu /bin/bash
```



# Hands-on: create *your* image

- The assignment is to take one of Python programs you wrote here at SOSC and incapsulate it into a Docker image, using a Dockerfile to build it. **Note: any output should be written e.g. to a text file somewhere on VM1, i.e. not left on the container.**
- Verify that your image works as expected with `docker run`.
- You should then push this image to Docker Hub in your public repository.
- **This is something you should do on your own.**
- Hint: before building the final image, test it interactively to see how it goes.

# 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 pushed our images to Docker Hub, in a public repository. But what if we wanted to copy our images to another system, *without going through Docker Hub*?
- Docker allows us to export an image to a tar file specifying its name (you can later compress it, if you want to save some 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"
```

# Hands-on: copy your image to your laptop and run it

- You should now copy the image you created on VM1 in the previous assignment to your own laptop.
- You should then load and run it locally. There are a couple of cases here:
  1. **If you have Docker already installed on your laptop**, you can load and run the image immediately.
  2. **If you do not have Docker installed on your laptop**, install it.
    - **Windows:** <https://docs.docker.com/docker-for-windows/install/>
    - **Linux:** see the previous slides if you have a Ubuntu—like (e.g. Debian) Linux distribution. If you have RedHat, see <https://docs.docker.com/install/linux/docker-ce/centos/>
    - **MacOS:** <https://docs.docker.com/docker-for-mac/>

# Doing local development

- Now that you have Docker installed on your laptop, try out the commands you issued on VM1 (which is running on a Cloud) locally.
- In general, it is very handy to do local developments with docker *locally*, i.e. on your laptop (and in many cases this also applies to relatively complex environments, as we will see). Once we are happy with the results, we can move to Cloud resources for production or more scalable workloads.



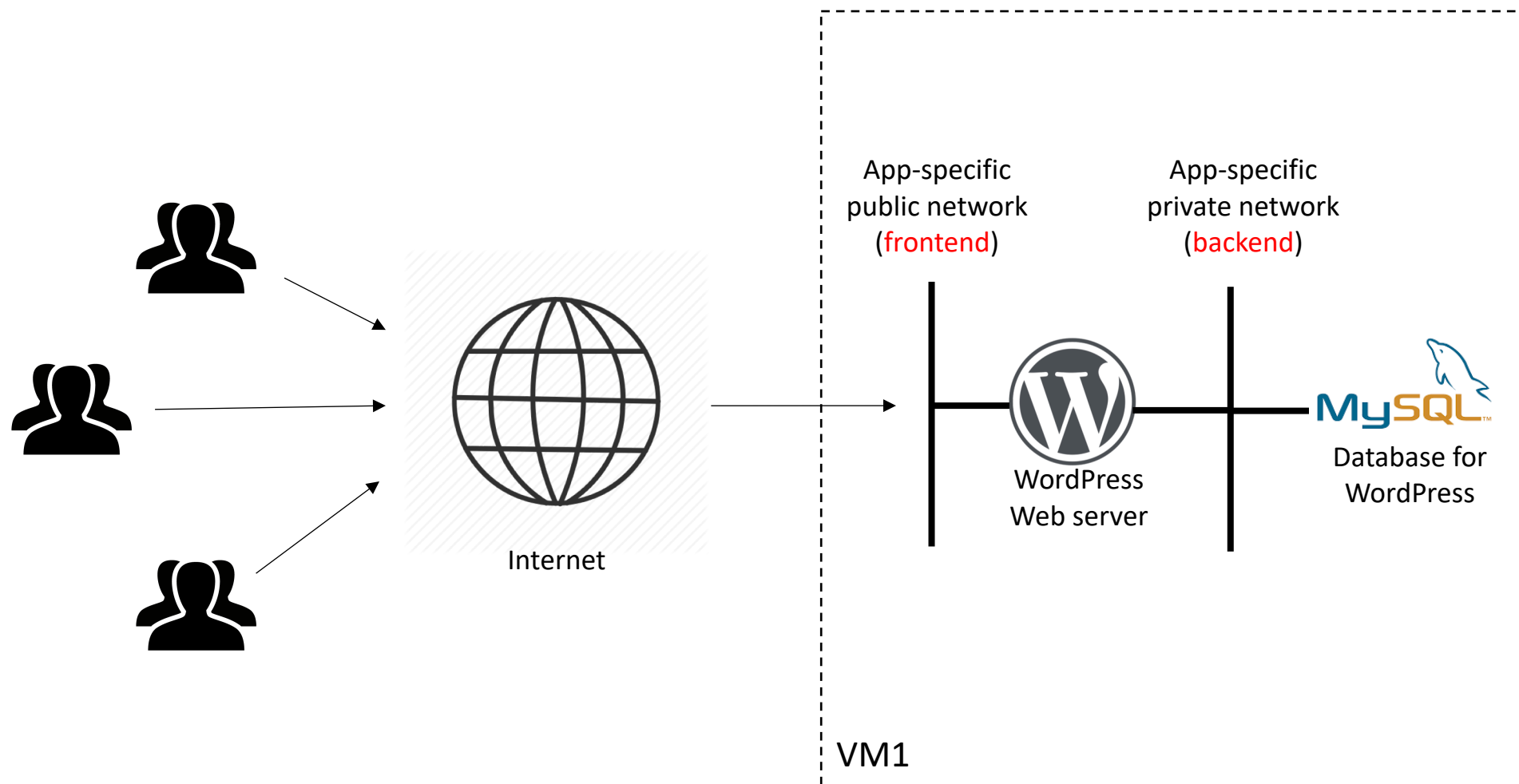
# Application stacks: docker-compose

- 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 VM1:
  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=testosc
- MYSQL\_DATABASE=wordpress
- MYSQL\_RANDOM\_ROOT\_PASSWORD=true

```
    networks:
```

- backend

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

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

```
  wordpress:
```

```
    image: wordpress:4.9.8
```

```
    depends_on:
```

- database

```
    environment:
```

- WORDPRESS\_DB\_HOST=database
- WORDPRESS\_DB\_USER=wordpress
- WORDPRESS\_DB\_PASSWORD=testosc

```
    ports:
```

- 8080:80

```
    networks:
```

- backend
- frontend

```
networks:
```

```
  backend:
```

```
  frontend:
```

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

Container image for WordPress  
(from Docker Hub)

Note that here we refer  
to the other container

Port 8080 on the host (VM1)  
is mapped to port 80 on the  
container

# Build & run the application stack

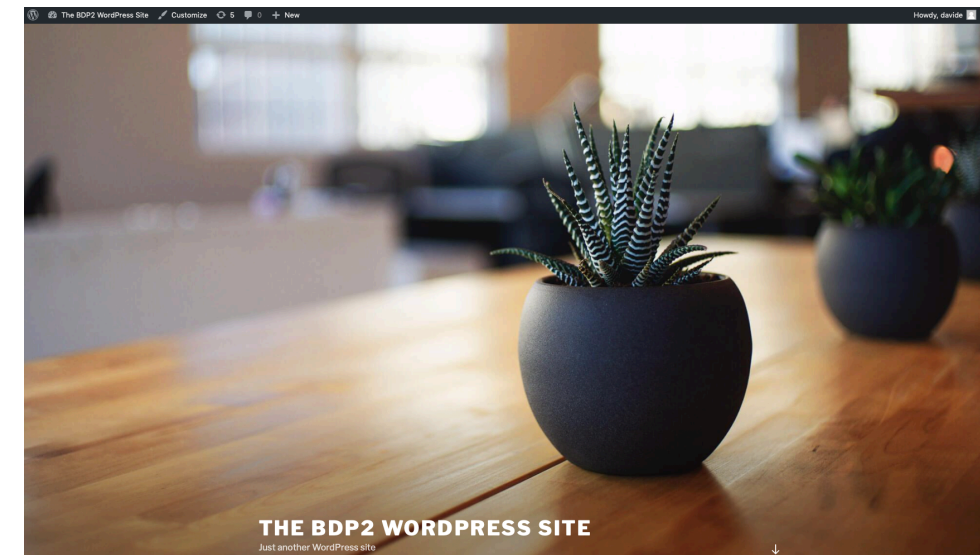
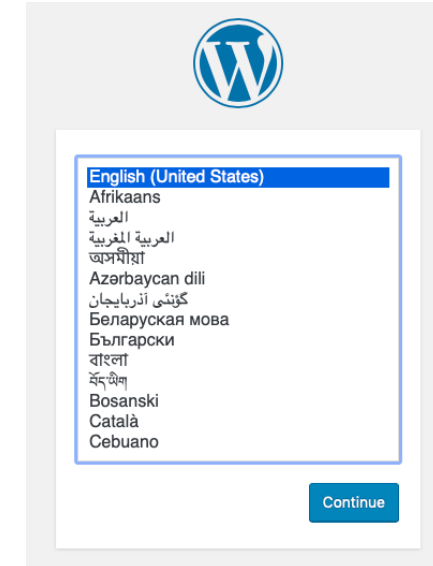
- Install docker-compose with  

```
# sudo apt install docker-compose
```
- 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.
  - **Remember to verify** that port 8080 is open in the inbound rules for VM1.
- 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.**



# 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

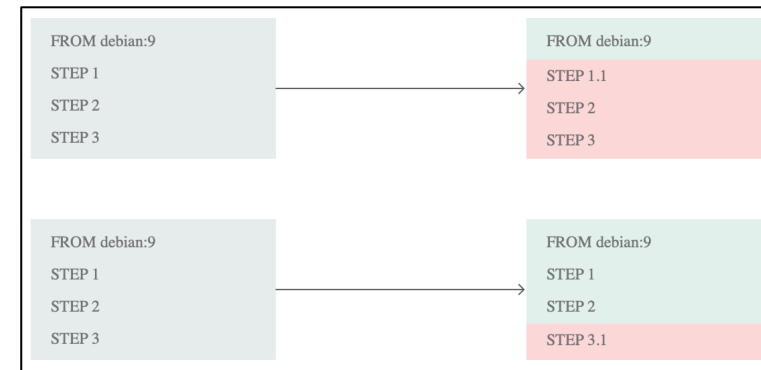
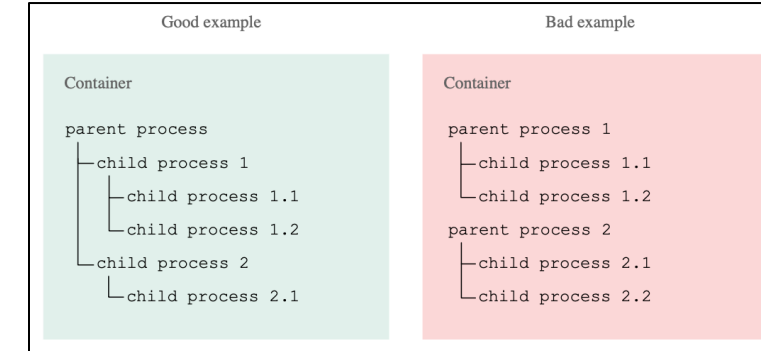
# Limitations of docker-compose

- As seen, `docker-compose` is very handy to create combinations of containers running on the same machine (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. We'll cover them later on.



# 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. 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/2Zr6Hyg>

# 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)!
- 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 e.g. 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 **host system is properly secured** (i.e. 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 should be immutable!** They might contain outdated software, that must be periodically patched and upgraded.
- For more details, see <http://bit.ly/2kEpV16>.

# A few problems with Docker

- There is no doubt that Docker containers are very handy and useful. However, in general, the adoption (i.e. installation) of Docker is quite slow in traditional clusters and in HPC centers.
  - What happens is that often Docker itself is **not installed** in a given set of computers. Therefore, one cannot run containers (unless one has got root privileges and can therefore install Docker autonomously).
  - This is often because the system administrators might believe that there could be security concerns with Docker, or because it is another service to maintain, or because it is too new... and so on.

# udocker

- In the INDIGO-DataCloud project, we developed **udocker**: it's a sort of “userland docker”, i.e. a tool which runs contents of Docker images without requiring any support from the kernel.
- There are no special dependencies, **aside from python 2.7 and libc**.
- In particular, `udocker` is intended to be run by unprivileged users.
- No special daemon is required. System-wide installation is possible but entirely optional (each user can “install it” individually).
- It is freely available at <https://github.com/indigo-dc/udocker>



# The udocker architecture

- It is a single-file python script.
- It fetches public images from Docker Hub by default.
  - It can also import image tarballs exported via `docker save`.
- It creates a container filesystem hierarchy in `$HOME/.udocker`
- It internally uses PRoot (see <https://proot-me.github.io>) for limited sandboxing.
  - Almost no CPU overhead.
  - Negligible data I/O overhead.
  - Sensible metadata I/O overhead.
- Other execution mechanisms than PRoot are available.

# udocker advantages

- Provides a docker-like command line interface.
- Supports a subset of docker commands: search, pull, import, export, load, create and run.
- Understands docker container metadata.
- Can be deployed by end users.
- Does not require privileges for installation.
- Does not require privileges for execution.
- Does not require compilation: just transfer the Python script and run.
- Encapsulates several execution methods.
- Includes the required tools already compiled to work across systems.
- Tested also with GPGPU and MPI applications.
- Runs on new and old Linux distributions, including CentOS 6, CentOS 7, Ubuntu 14, Ubuntu 16, Fedora, etc.



# udocker limitations

- Images cannot be created by `udocker`.
  - That is, you must use Docker on another system (where the Docker daemon is installed) to build images! You use `udocker` to **run** already built images.
- Privileged OS operations (such as, for example, `mount`) are not possible.
- Debugging inside containers does not work.
- **udocker is not a privilege boundary!** I.e., it does not enforce special security measures: the `udocker` process runs with the privilege of the current user.
- We don't have time to test `udocker` here. **However**, you are encouraged to download and test it with your existing containers (or with containers pulled e.g. from DockerHub).

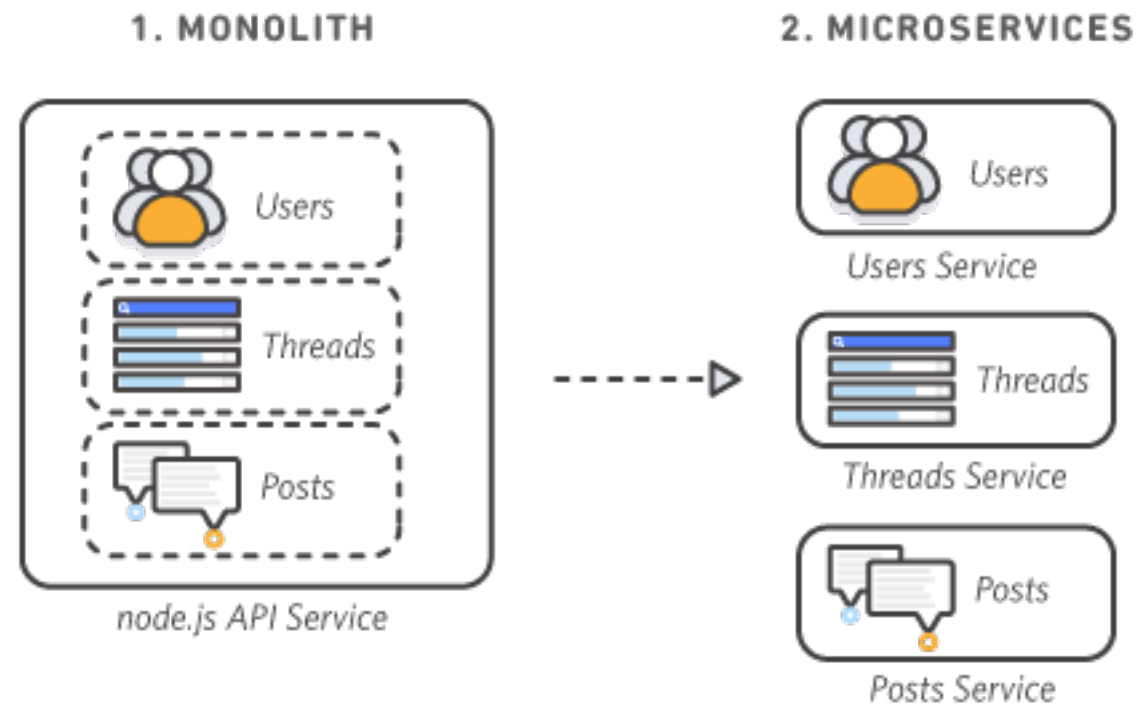
# Container Orchestration

# Microservices

- With Cloud-native applications, an analogy is often made about seeing traditional apps as **pets** (each one is unique and irreplaceable) vs. cloud apps as **cows** (there are many identical instances of a functionally equivalent “item”).
- **Microservices** are a way to build applications as a **collection of (potentially many) small autonomous services** vs. creating a big service (or anyway a few *fat ones*), called sometimes *monolith*.
- At high level, microservices reflect at the architectural level a **culture of autonomy and responsibility** in an organization: the single microservice can be developed and managed independently by different teams.
- In microservices architectures, the multiple, independent processes communicate with each other through the network.



# Application architectures



# Monoliths vs. microservices

## Monolithic Applications



- Do everything
- Single application
- You have to distribute the entire application
- Single database
- Keep state in each application instance
- Single stack with a single technology

## Microservices



- Each has a dedicated task
- Minimal services for each function
- Can be distributed individually
- Each has its own database
- State is external
- Each microservice can adopt its own preferred technology

Adapted from AWS

# Container orchestration

- We saw how **containers** help us to easily create applications that are – as the name says – self-contained.
- On the other hand, we also said that **microservice architectures** are based on the composition of many independent (but communicating) services.
- **Let's combine these two points:** containers can greatly help with the creation of a microservice architecture. Actually, through `docker-compose` we already learned how to create multiple containers linked together in **Application Stacks**.
- However, `docker-compose` is limited to the composition of containers within a single host. On the other hand, in general microservices are deployed across multiple hosts.
- We therefore need to explore how to effectively *orchestrate* many containers across several hosts. This is what we call **container orchestration**.

# Docker Swarm (1)

- Docker Swarm is the traditional way of orchestrating containers with Docker. Compared to other methods such as Kubernetes, it is relatively simple to use. Its main features are:
  - **Cluster management is integrated with the Docker Engine:** no other software than docker is needed.
  - **Design is decentralized:** this means that any node in a Docker Swarm can assume any role at runtime.
  - **Scaling:** the Swarm manager can automatically scale up and down services, adding or removing tasks.
  - **Desired state reconciliation:** if something happens to a Swarm cluster (e.g. some containers crash), the Swarm manager will try to reconcile the state of the cluster to its desired state (e.g. bringing up some more containers).

# Docker Swarm (2)

- Docker Swarm features, continued:
  - **Multi-host networking**: the Swarm manager can handle an overlay network spanning your services.
  - **Service discovery**: there is a DNS server embedded in each Swarm. The Swarm manager discovers services and assigns to each of them a unique DNS name.
  - **Load balancing**: you can specify how to distribute services among nodes.
  - **Secure by default**: the communication among all nodes in a Swarm cluster is protected by the cryptographic protocol called TLS (Transport Layer Security).
  - **Rolling updates**: if anything goes wrong, you can roll-back a task to a previous version of the service.



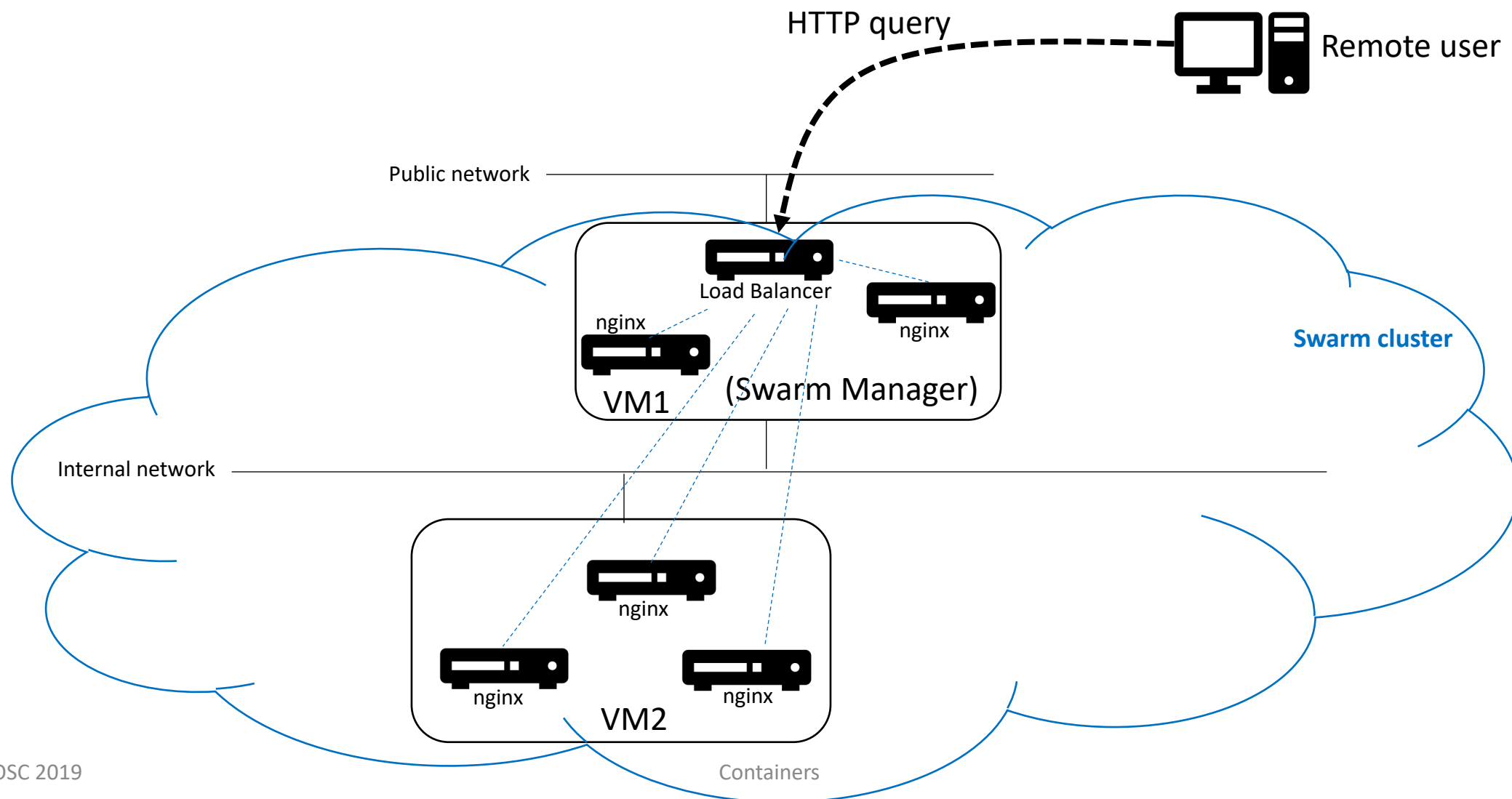
# Hands-on with Docker Swarm

- We'll loosely follow <https://docs.docker.com/engine/swarm/swarm-tutorial/>.
- For this hands-on, we need **two VMs with Docker installed**.
  - One of these machines (VM1) will be the manager of the Swarm cluster, the other (VM2) will be called a worker.
  - **Important: make sure that Docker is installed on both VM1 and VM2. Check now and take action if this is not the case.**
- We need the IP addresses of the 2 machines involved, as well as the following open ports for all of them, to allow communication among the nodes. This is not an issue in our setup (where these ports are already open), but it may be in other environments:
  - **TCP port 2377** for cluster management communications.
  - **TCP and UDP port 7946** for communication among nodes.
  - **UDP port 4789** for overlay network traffic.

# Docker Swarm hands-on: our use case

- To keep things nice and simple, we'll use a Docker Hub container called "nginx" (*do you remember how to find it?*). Nginx is a commonly used web server (see <https://nginx.org/en/>), like Apache.
- We'll create a **Swarm service** based on the nginx container and deploy it in 5 instances, distributed across 2 VMs (VM1 and VM2). VM2 will *not* be directly accessible from the Internet. So, in the end we'll have instantiated 5 web servers.
- We'll then **deploy a load balancer** on VM1 (the manager). The load balancer will be reachable via a public IP address. When people hit this IP address, the load balancer will route our requests to one of the nginx containers on VM1 *or* VM2.

# Docker Swarm: our architecture



# Create a Swarm cluster

- Login to the VM that should become the “Swarm manager”. This is VM1.
- On the manager, issue the command
  - `docker swarm init --advertise-addr <MANAGER-PRIVATE-IP>`
  - This initializes a Swarm cluster and tells the workers about the IP address of the Swarm manager (which by default also becomes a worker). Note that this should be **the manager’s private IP address**, not the public one.
  - Docker answers confirming that the current node is now manager and gives us the command to add a worker to the Swarm cluster. **Note it down.**
- Now log in to VM2, and issue the command reported above by the manager.
  - It should be something like `docker swarm join -token <token> <ip_addr>:2377`
- On the manager, issue the command `docker node ls` to view the **current state of the Swarm cluster**. It should show the manager and two workers, all in the “active” state. There are no running services in the cluster yet.

# Create a Swarm service

- We will now **create a “service”**. We have to define:
  - How to name it – we’ll call it “**web\_swarm**”.
  - The container image it is based on (nginx, found on DockerHub).
  - The port that can be used to contact the service.
  - How many replicas of the service we want to deploy.
- This is the command we have to issue on the manager:

```
docker service create --replicas 5 -p 8082:80 --name web_swarm nginx
```

  - With this command, we create 5 docker containers, each one based on the nginx image. These containers will be automatically distributed across our Swarm cluster. Each container will expose port 80, which will be mapped to port 8082 on a VM host (VM1 or VM2).

# Check the status of the Swarm service

- The **status of our service** can be checked on the manager with  
`docker service ls`
  - It may take some time before the service is shown as replicated 5 times, as requested – just repeat the command until it shows 5/5 replicas.
- In order to see **where (i.e. on which nodes) the service was distributed by Swarm**, issue this command on the manager:  
`docker service ps web_swarm`
- Once you have the 5 `web_swarm` replicas running, log in to either VM1 or VM2 and issue this command there:  
`docker ps`
  - You should see that one or more nginx containers are running on the node.

# How to access the `web_swarm` service

- Remember that, so far, the nodes of the Swarm cluster are only reachable via their private IP address. Therefore, we cannot directly use a browser to reach the web servers.
- But internally they can be reached (look back at the architectural diagram). So, log in e.g. to the manager and issue the command  

```
curl http://<private_ip_address_of_VM1>:8082/ (or VM2)
```

  - You should get an answer, with the default html page shown by a fresh nginx installation.
  - Note that you will get an answer even if there is no `web_swarm` container running on VM1 (or VM2). **How can you prove that?**

# Scaling up or down and draining

- When we created our service, we specified `--replicas 5`. If you want to **scale the service** to a different number of replicas, just issue this command on the manager:

```
docker service scale web_swarm=7
```

- What is happening? On the manager, check with

```
docker service ls
```

```
docker service ps web_swarm
```

- Now suppose that you want to remove the service `web_swarm` from e.g. VM2 (because, for example, you want to shut it down for any reason). This is called draining a node. Try this:

```
docker node update --availability drain <VM2>
```

- What is happening? Check with `docker service ps web_swarm`.



# Load balancing the web servers

- We now want to create a **load balancer** on the manager node. Its purpose is to expose a public IP address which will be reachable from the Internet, and balance the queries to that IP address to the `web_swarm` services that are deployed in the Swarm cluster.
- The same nginx container that we previously used to create web servers can also be configured to act as load balancer. We just need to have a suitable nginx configuration file.
  - In this configuration file, we need to list the IP address (the private IP addresses, in our use case!) of all the hosts participating to the Swarm cluster.
  - That is, the **private IP addresses** of VM1 and VM2.

# The nginx configuration for load balancing



- On the manager, create a directory called swarm (or whatever) and create this file into it, calling it nginx.conf:

```
worker_processes 1;
events { worker_connections 1024; }
http {
    sendfile on;
    upstream swarm_cluster {
        server <manager_ip_addr>:8082;
        server <VM1_ip_addr>:8082;
        server <VM2_ip_addr>:8082;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://swarm_cluster;
        }
    }
}

log_format upstreamlog '[$time_local] from $remote_addr to $upstream_addr';
access_log /var/log/nginx/access.log upstreamlog;
```

# Create and run the load balancer

- **On the manager**, create the following Dockerfile in the same directory where you have put `nginx.conf` (e.g. `swarm`):

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

- We can now build and then run our container with the load balancer configuration with commands we already know:

```
docker build -t load_balancer .
docker run -p 8080:80 -d load_balancer
```

- If we now open `http://<manager_public_ip>:8080/`, we should get a web page displayed. **Try it out now.**
  - From which `web_swarm` node is the answer coming? In the `nginx.conf` file we told the web server to log some information. Look at this information with the following command:

```
docker logs -f <load_balancer container>
```

# A few notes

- Docker Swarm services are persistent. If you shut down both nodes and then start only the manager (i.e. VM1), you will see that the manager brings up all replicas automatically on itself.
  - The load balancer configuration, on the other hand, is a stand-alone container and does not automatically restart.
- Remove a Swarm service with:

```
docker service rm <service_name>
```
- An interesting point is to combine Docker Swarm with custom Docker images or with Docker Compose. This is left as an exercise.



# Recap of our journey

- We covered basic concepts about **Containers**, comparing them to Virtual Machines.
- We installed Docker first on VM1 and saw how to run a container, list docker images and extend them to create new images.
- We then saw how to push images to repositories on Docker Hub and simplified the building of images via Dockerfiles.
- We created an image 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 discussed some Docker limitations, in particular with regard to security and privileges, and introduced `udocker` as a way to work around them.
- We then moved on to discuss microservices and container orchestration, using Docker Swarm as an example. As hands-on, we created a Swarm cluster load balancing multiple web servers, distributed across multiple nodes.
- **Our next step** will be to consider **further orchestration technologies and solutions**, as well as ways to **automate and simplify the usage of distributed resources**.