Lecture 2 – Virtualization and Containerization

Gianluca Quercini

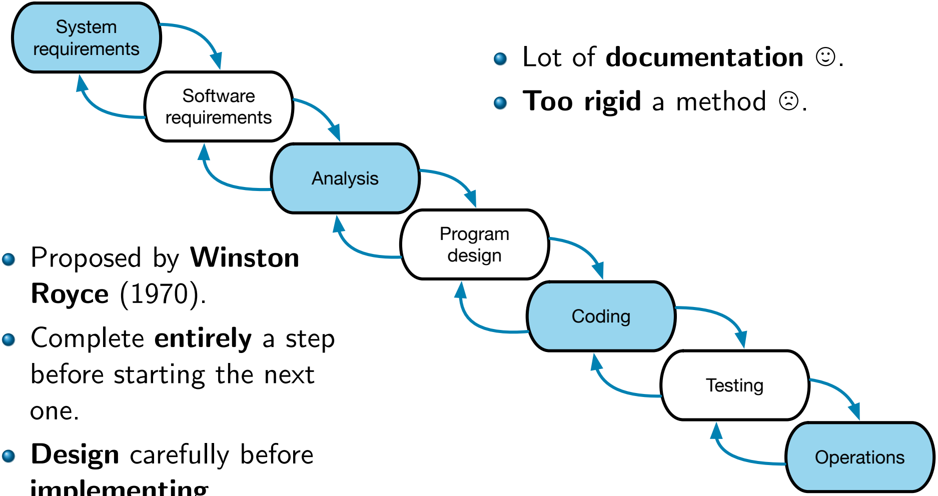
CentraleSupélec

SG8, 2019 – 2020

A bit of history

- Evolution of **software development methodologies**.
  - Waterfall (1970).
  - Agile (early 2000).
  - DevOps (2008-2009).
- Evolution of **software architectural styles**.
  - Monolithic.
  - Microservices.

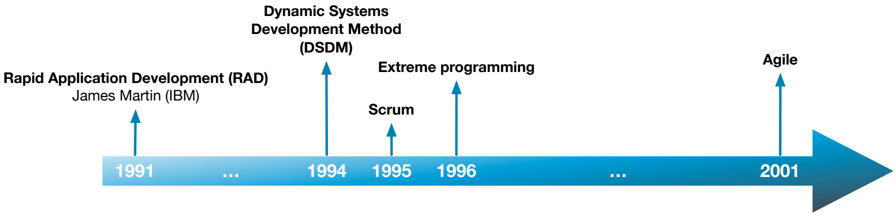
Waterfall method



- Lot of **documentation** ☺.
- **Too rigid** a method ☹.

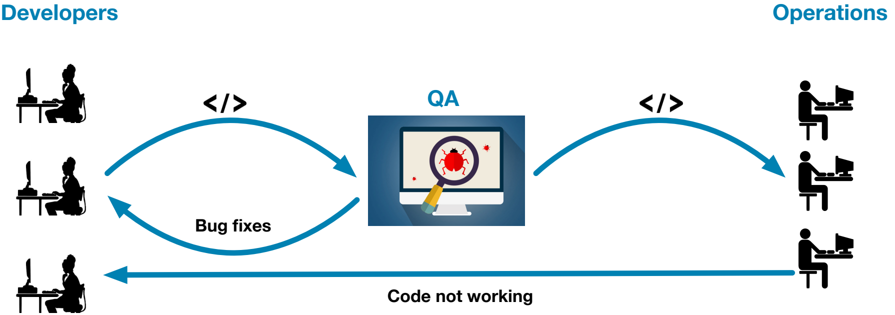
- Proposed by **Winston Royce** (1970).
- Complete **entirely** a step before starting the next one.
- **Design** carefully before **implementing**.

Towards Agile



- **RAD**: prototyping.
- **DSDM**: timeboxing, MoSCoW.
- **Scrum**: sprints, daily standups.
- **Extreme programming**: test-driven development.
- **Agile**: frequent software releases, feedback from the customers, embrace changes.

Towards DevOps



- **Developers** write some code.
- The **Quality Assessment (QA)** team tests the code.
  - They might send the code back to the developers for bug fixing.
- The **Operations** team receives the code ready for production.
  - They might send the code back to the developers if it doesn't work.

Towards DevOps

Even in companies that embraced Agile, teams worked in **silos**.

- **Developers** don't know the **production environment**.
- **Operations** don't know the **development environment**.
- When something doesn't work, they **blame each other**.

Developers and operations have **conflicting priorities**.

- **Developers** are evaluated on the number of **new features**.
- **Operations** are evaluated on the **system stability**.
- But changes threaten stability.

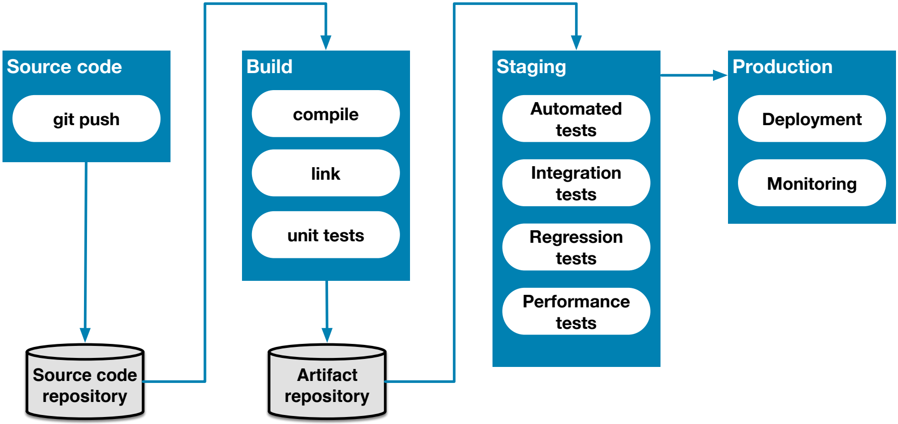
Image source

DevOps

Cultural movement to address the Developers (**Dev**) - Operations (**Ops**) divide.

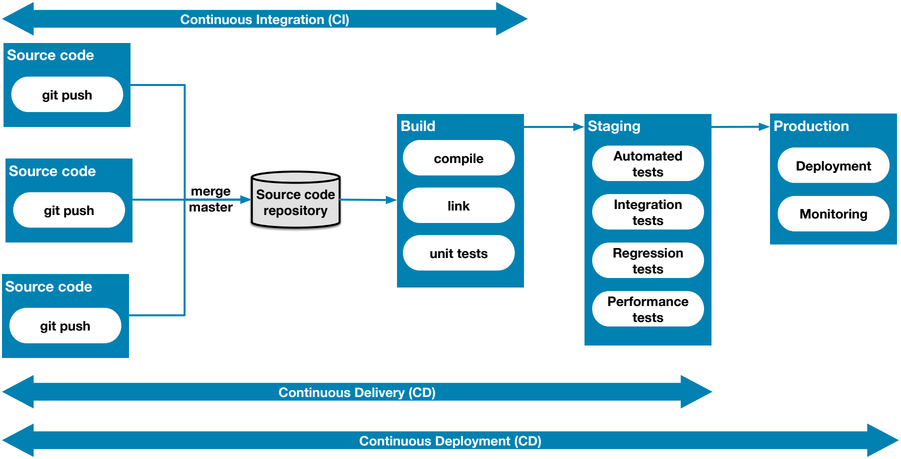
- **Collaborative learning** among the teams.
- **Automating** the software delivery pipeline.
- **Continuous feedback** in the teams.
  - Solve the problems as early as possible.
- **Continuous experimentation**, to learn from failures.
- **Collaboration** towards common goals.

DevOps — Software delivery pipeline



▸ Based on this description

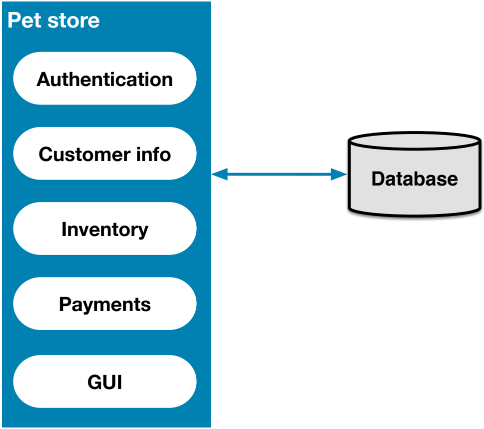
DevOps — CI/CD



The key to DevOps practices is **automation**.

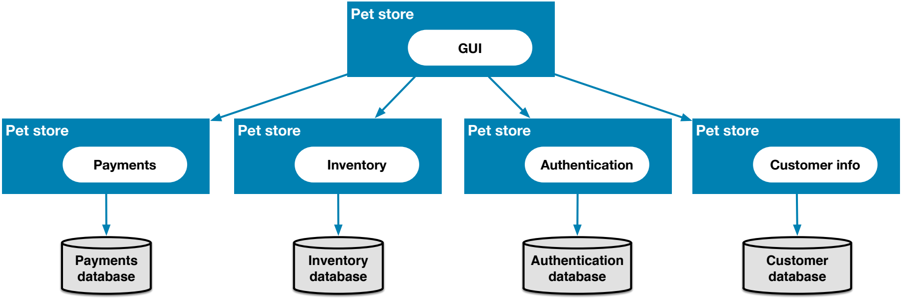
Monolithic architecture

- **Traditional applications** have a **monolithic architecture**.
- All features and services are coded into a single application.
- **Example.** Pet store application. ▸ Credit



Microservices architecture

- Application composed of many **loosely coupled** and **independently deployable** smaller components, called **microservices**. ▸ Ref
- Each microservice implements a specific feature of the application.
- Microservices interact by using APIs.



Microservices — Advantages

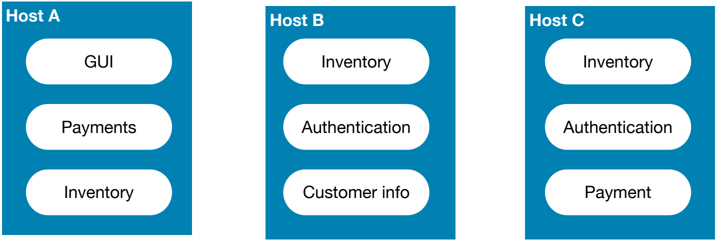
- **DevOps principles**.
  - Modularity.
  - Continuous delivery and deployment.
  - Failure isolation.
- Different microservices, **different technologies**.
- Microservices **scaled independently** of one another.
  - With a monolithic architecture, the whole application needs to be scaled even if only one module experiences an increased load.
- **Decentralized databases**.
  - Custom database technology for each microservice.
- Good architectural choice for **cloud-native applications**.

Microservices: package and deploy

How are microservices **packaged** and **deployed**? ▸ Ref

- Multiple service instances per host.
- Single service instance per host.
- Service instance per **virtual machine**.
- Service instance per **container**.
- **Serverless** deployment.

Multiple service instances per host



- Each host runs multiple service instances.
- **Efficient** use of resources ☺.
- **Limited isolation** of the service instances ☹.
- **Difficult to limit** the resources that a service uses ☹.
- **Difficult deployment** ☹.
  - Services are written with different languages and frameworks.

Single service instance per host

- Each single service instance is deployed on one host.
- **Isolation** between service instances ☺.
- **Easy monitoring** of the resources used by a service ☺.
- **Easy deployment** ☺.
- **Inefficient use of the resources** ☹.

Service instance per virtual machine (VM)

- Each microservice is packaged as a VM.
- Solution adopted by Netflix.

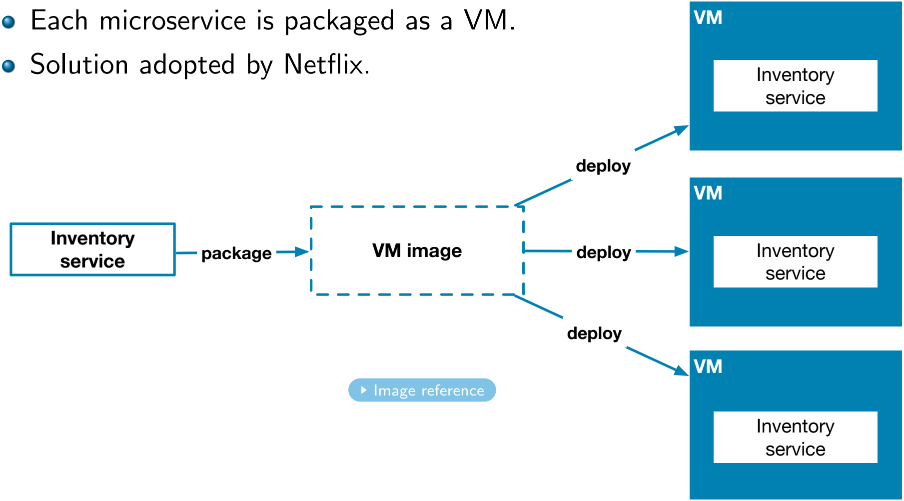


Image reference

Service instance per virtual machine (VM)

- **Complete isolation** ☺.
- **Fixed amount of resources** (e.g., CPU and memory) per VM ☺.
- **Encapsulation.** Technological details hidden in the VM ☺.
  - All services are started and stopped in the same way.
- Mature **cloud infrastructure**: load balancing and autoscaling ☺.
- VMs images are **slow to build** and instantiate ☹.
  - A VM typically contains a full-blown operating system.
  - Solutions exist to generate lightweight images.

Service instance per container

- Microservice packaged as a **container image**.
  - **Container**: lightweight VM.
- Containers can run on a VM or a physical server (bare metal).

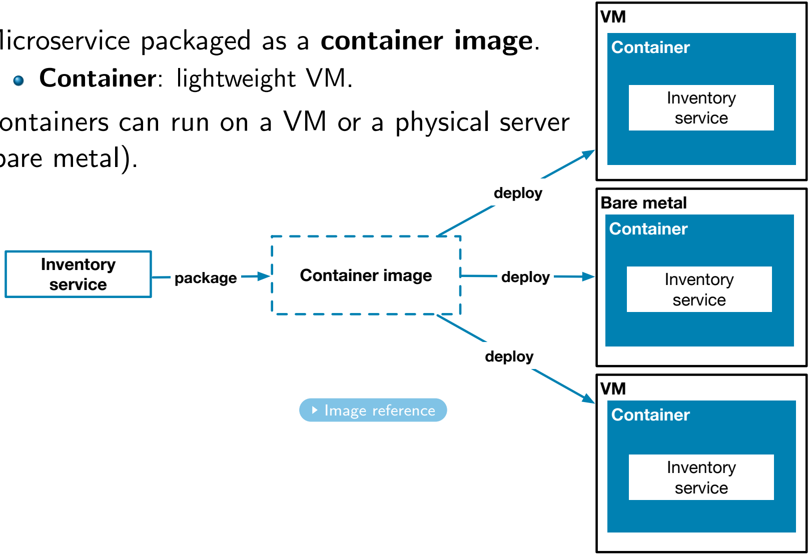
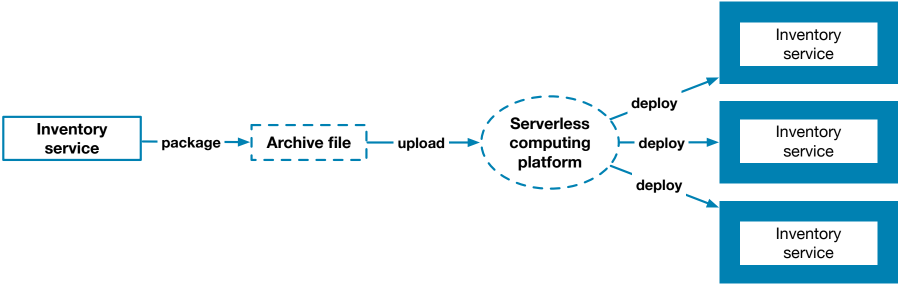


Image reference

Service instance per container

- All advantages of VMs and
- Container images are **fast to build** ☺.
  - Containers don't ecapsulate an entire operating system.
  - Containers only include the service and only what it takes to run the service.
- **Less mature** cloud infrastracture than for VMs ☹.

Serverless deployment



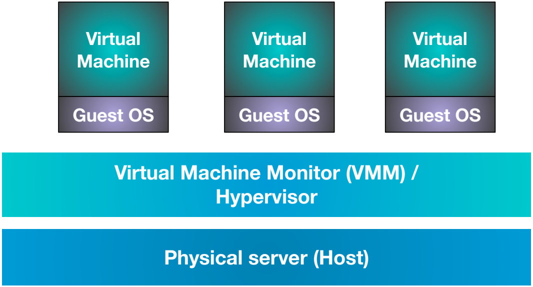
- Microservice packaged as an archive (e.g., zip, tar).
- The **serverless computing platform** runs as many instances of the service as needed.
- No visibility into the underlying infrastructure.

Serverless deployment

- **No need to configure** the underlying infrastructure ☺.
- **Automatically scale** the service to handle the load ☺.
- **Pay per request**, instead of paying for the infrastructure ☺.
- **Limited** language support ☹.
- **Unfit** for long-running services ☹.
- **No control** over the performances ☹.

Virtualization

- **Abstraction** of some physical resource into a logical object.
- Gerald Popek, Robert Goldberg (1974). First virtualization framework.
  - **Fidelity**. Virtual environment = host environment.
  - **Isolation**. Hypervisor has a complete control over system resources.
  - **Performance**. VM performances ≈ Host performances.



Reference

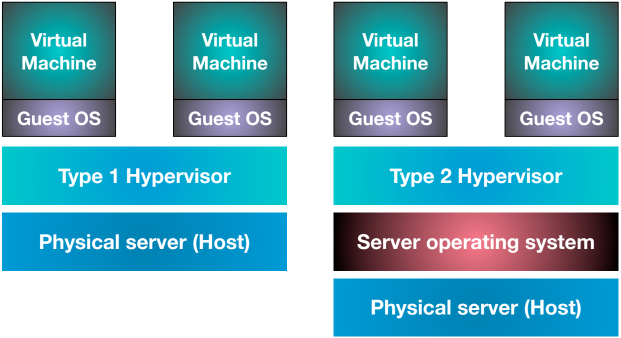
Virtualization

- Many factors pushed the adoption of the **one server, one application** policy.
  - Operating system (Microsoft Windows, as a **single-user system**).
  - Need to isolate applications.
  - Low hardware costs.
- Data centers grew larger and larger.
  - Difficult maintenance, energy consumption...
- **Inefficient use** of computing resources.
  - Many servers were idle 95% of the time.
- **Virtualization** allowed to stuff **several virtual servers** into a single physical server.
  - This is called **server consolidation**.
  - Better resource use, cost savings, less energy consumption.

Reference

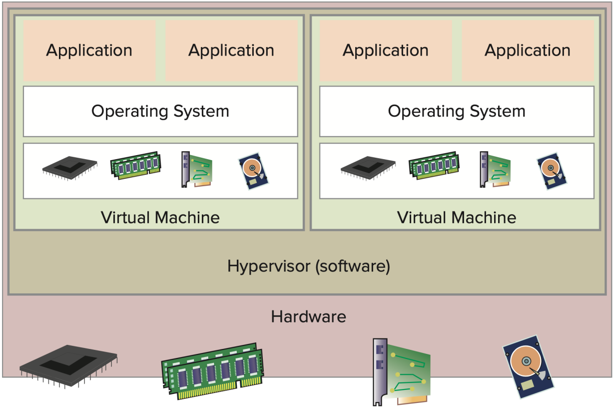
Hypervisor

- **Software component** that manages the interactions between the virtual machines and the underlying hardware.
- Type 1 (*bare metal*) hypervisors: VMware ESX, Microsoft Hyper-V.
- Type 2 hypervisors: VirtualBox.



Role of the Hypervisor

- Present an **abstract view** of the **physical resources** to the guests.
- Allocate the resources when they are requested from the guests.



Reference

Virtual Machines

A **virtual machine** (VM) consists of the following components:

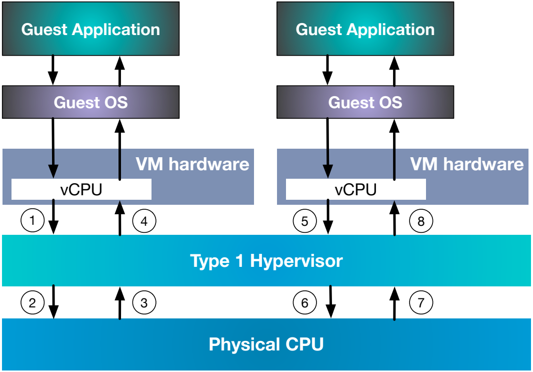
- One or more **applications**.
- An **operating system**.
- A set of **virtual hardware devices**.

On disk, a **virtual machine** is a **collection of files**:

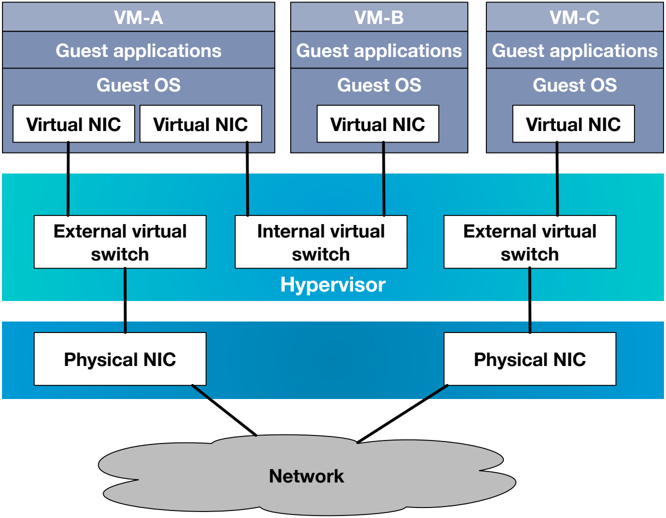
- Files that compose the virtual machine itself.
- A **configuration file**.
  - List of hardware devices: CPU, memory, storage.
- Virtual **disk files**.
  - Content of a virtual hard drive.

CPU in virtual machines

- The hypervisor assigns **time slices** on the physical CPU to execute instructions.
- No one-to-one mapping between virtual and physical CPUs.



Networks in virtual machines



Reference

Networks in virtual machines

- **Virtual switches** play a similar role to physical switches.
  - Connect several devices to each other.
- Combination of **external** and **interal** virtual switches.
- **Secure** applications and servers.
  - VM-B cannot be accessed through the physical network.
  - Traffic between VM-A and VM-B is transparent to the physical network.
- **Fast communication** between virtual servers.
  - VM-A and VM-B communication happens **in memory**.
- Possibility of **migrating** a VM from a physical server to another.
  - Without interrupting the guest applications.

Advantages of virtual machines

- **Multiple virtual servers** in a physical server.
  - **Better** resource use.
  - **Lower** costs.
  - **Simplified** data center management.
- **Easy creation** of virtual machines from templates or clones.
  - A virtual server can be up and running in hours.
  - A physical server may need weeks to be up and running.
- **Easy migration** of virtual machines.
  - Reduced or minimized **downtime**.

Why containerization?

- Virtual machines: **good solution** to package a microservice.
  - The microservice is **embedded** into the VM.
  - **Easy deployment** on any machine.
- However, virtual machines are **heavy**.
  - They contain a **full-blown operating system** (e.g., Linux, Windows).
  - Starting or migrating a VM can be **time-consuming**.

> Using a VM to package a single application is tantamount to use a "gigantic ship to transport a truck load of bananas".
>
> — Gabriel N. Schenker, Reference

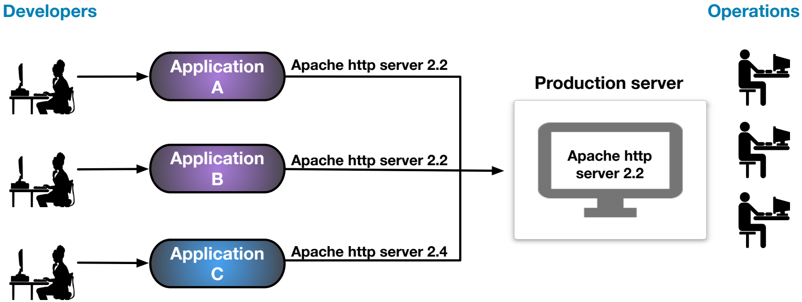
Why containerization?

Back in the old days, loading and unloading ships was difficult as each product came with a **different package**. ▸ Image source

Then, there came **containers**, simple boxes with **standard dimensions**. ▸ Image source

Why containerization?

- Each application came with a **"different package"**.
- Deployment was **application-dependent**.
- Difficult **dependency management** in production.



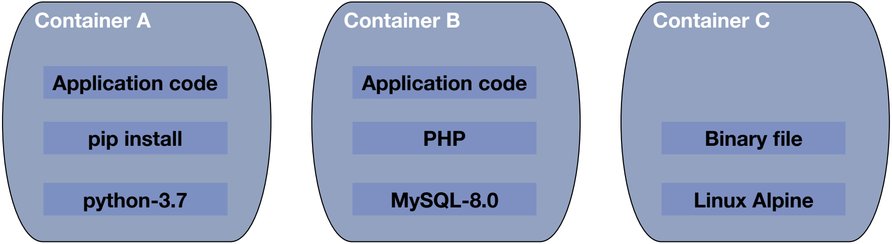
- **Containers** standardize application **development** and **deployment**.
- Great solution in a **cloud environment**.

What are containers?

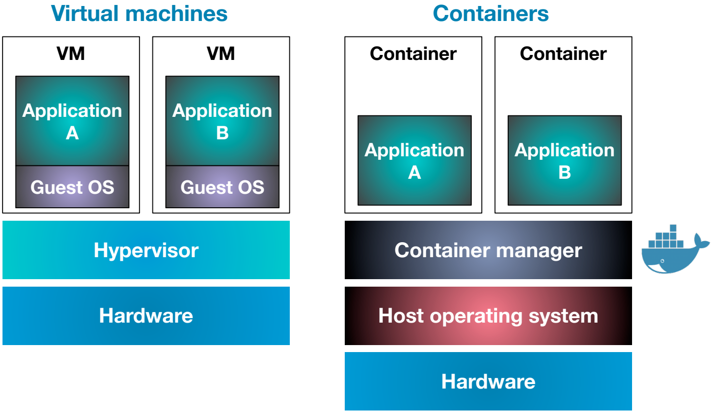
Definition (Container)

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

- A **container** packages a piece of software with **all its dependencies**.
- Alternative (lighter) solution to virtual machines ▸ Reference.



Virtual machines and containers: differences

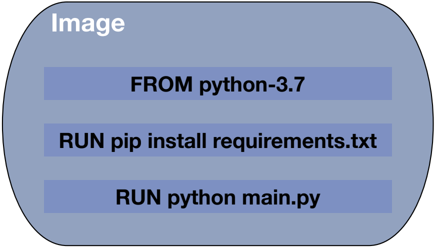


The container manager that we study in this course is **Docker**.

What is Docker?

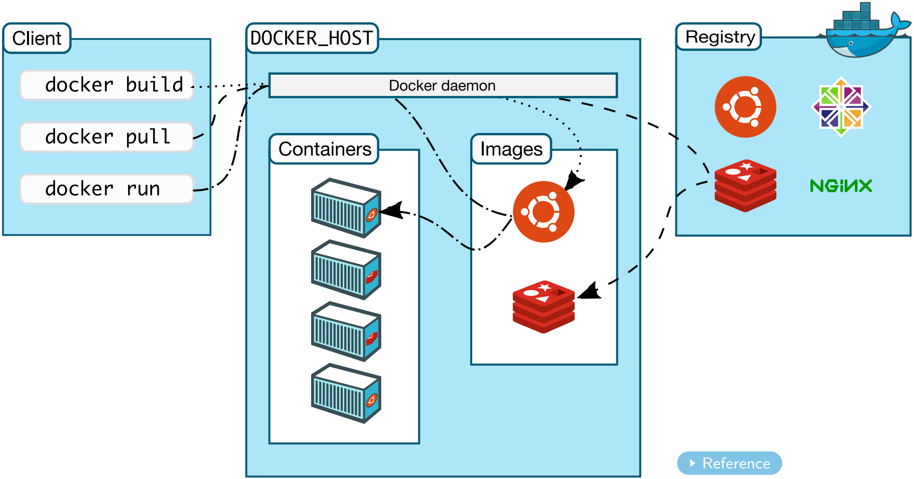
Definition (Docker)

Docker is an **open platform** for **developing**, **shipping**, and **running** applications. It provides the ability to package and run an application in a loosely isolated environment called a **container** Reference.



- Docker creates and runs a container from an **image**.
- **Image**: **read-only template** with instructions for creating and running a container.
- An image contains all files necessary to run an application in a container.

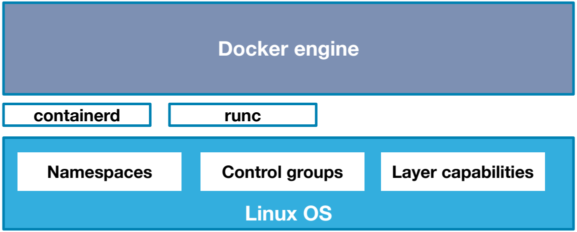
How does Docker work?



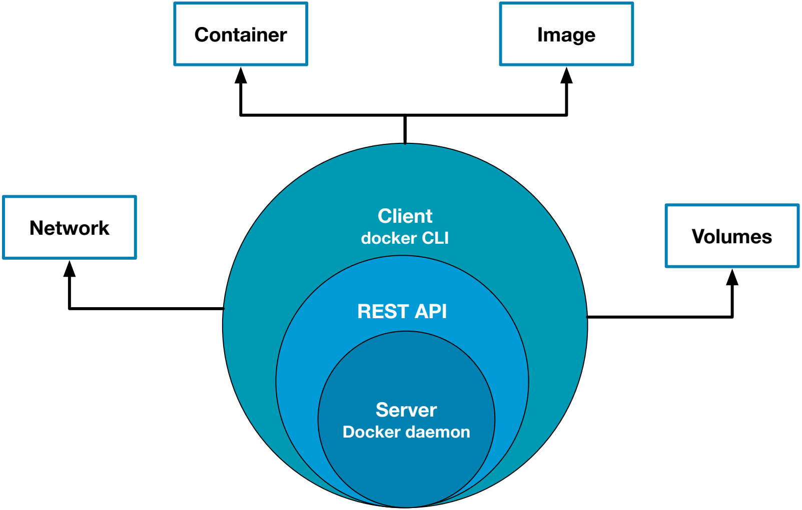
▸ Reference

High-level Docker architecture

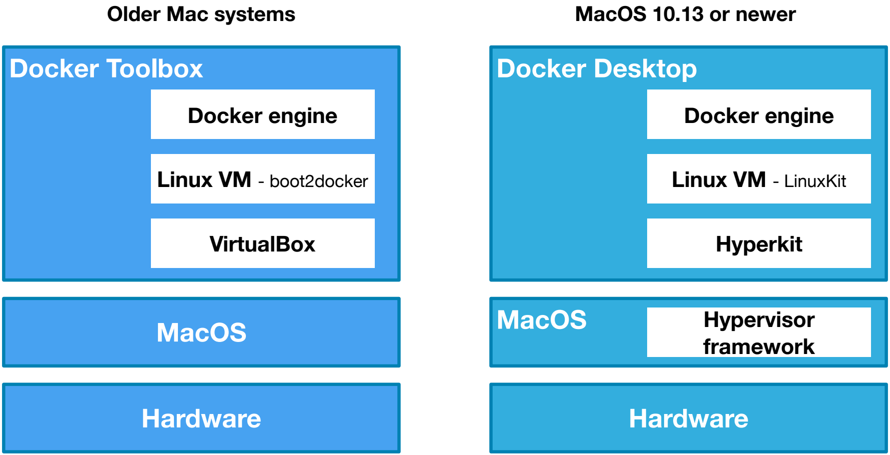
- **Docker engine**. Client-server application for building and containerizing applications.
- **containerd**. It manages the complete container lifecycle (image pull, container execution, low-level storage and network)
- **runc**. Command-line tool to run containers. Invoked by **containerd** with the proper parameters.
- Docker uses several features of the **Linux kernel**: **namespaces** (isolated workspaces), **control groups** (resource limitation), **union file systems** (build images).



Docker engine

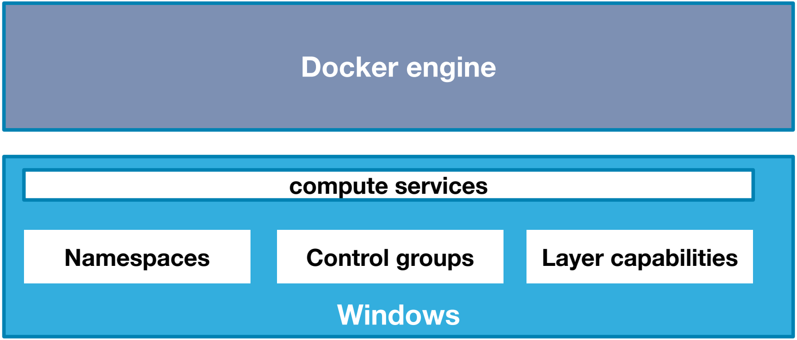


Docker on MacOS

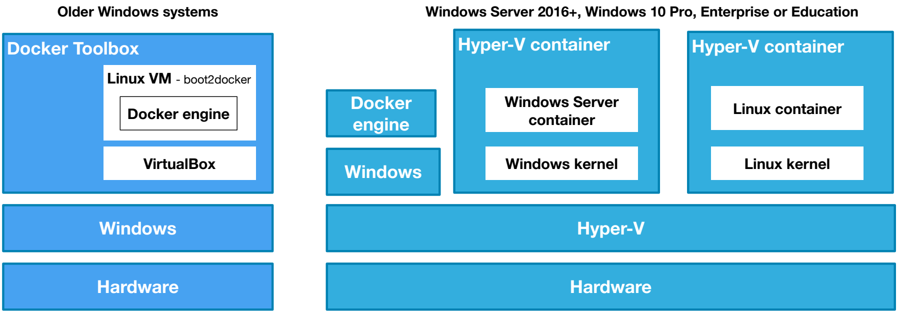


Docker on Windows

- Docker Engine **natively** supported in **Windows Server** (since 2016).
- The Windows Server kernel includes the technology to run containers.
- Definition of **Windows containers**.



Docker on Windows



Running containers

docker run [options] image-name [command] [arg]

- **options**. List of options.
- **image-name**. Fully qualified name of the image used to run the container.
- **command**. The command to be executed in the container.
- **arg**. The arguments taken by the command executed in the container.
- The arguments between square brackets are **optional**.
  - The only **mandatory argument** is the **image name**.

Fully-qualified image names

registry.redhat.io/rhel8/mysql-80:1-69

- Name of the registry providing the image.
  - **Default**: registry.hub.docker.com
- Name of the user or organization owning the image.
  - **Default**: library
- Name of the image.
  - **Mandatory**.
- Tag (or, version) of the image.
  - **Default**: latest

Image names

Ubuntu image on DockerHub

- registry.hub.docker.com/library/ubuntu:latest
- registry.hub.docker.com/library/ubuntu:18.04
- ubuntu:bionic
- ubuntu

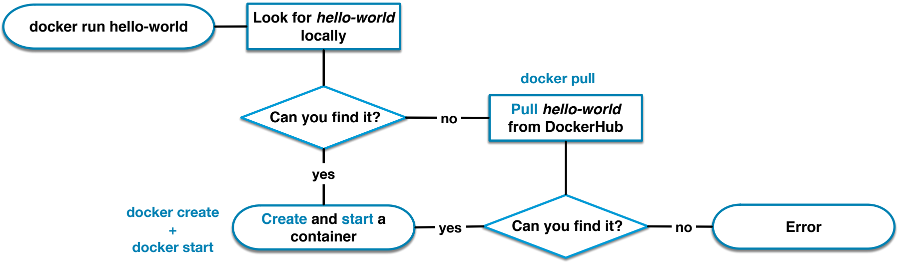
Supported tags and respective `Dockerfile` links

- `18.04`, `bionic-20200311`, `bionic`, `latest`
- `19.10`, `eoan-20200313`, `eoan`, `rolling`
- `20.04`, `focal-20200319`, `focal`, `devel`
- `14.04`, `trusty-20191217`, `trusty`
- `16.04`, `xenial-20200212`, `xenial`

Reference

What happens when we run a container?

docker run hello-world



- The **containerized application** (the one inside the container) is run.
- When the containerized application terminates, the container is **stopped**.

How to list containers?

To list **running containers**:

```
docker container ls
```

To list **all containers** (running and stopped):

```
docker container ls -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
a997a41e815c	hello-world	"/hello"	6 seconds ago	Exited (0) 5 seconds ago		peaceful_roentgen

- The command executed when we run the container is *hello*.
- This is a **binary file** stored in the container.

Useful options of *docker run*

Give a container a name:

```
docker run –name my-hello-container hello-world
```

Remove a container when it stops:

```
docker run –rm hello-world
```

Interact with a container:

```
docker run -it ubuntu
```

- The last command opens a shell into the container.
- The shell is used to navigate the file system of the container.

Execute a command inside the container

Run **docker container ls -a** and look at the container created from the image *ubuntu*:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c03377554071	ubuntu	"/bin/bash"	11 minutes ago	Exited (0)		peaceful_fervent_cori

- When the container starts, the command */bin/bash* is executed.
  - Bash is a Unix shell (command-line terminal).
- We can pass that shell any Linux command to execute.

```
docker run –rm ubuntu ls
```

What will this command output be?

Execute a command with arguments inside a container

- We pass any command executed inside a container some **arguments**.

The command *ping*

The command *ping* is used to test the reachability of a host on a IP network.

We run a container from the image *alpine* (yet another Linux distribution):

```
docker run –rm alpine ping www.centralesupelec.fr
```

- **ping** is the **command**.
- **www.centralesupelec.fr** is the **argument**.

Create, start and run: let's clarify

- The command **docker run** wraps two Docker commands:
  - **docker create** that creates a container.
  - **docker start** that starts the container.

Example

docker run –rm alpine ping www.centralesupelec.fr

is equivalent to :

docker create –rm –name my-ping alpine ping www.centralesupelec.fr

docker start -a my-ping

- The option **-a** attaches the **standard output** of the container.
- The container starts with the parameters set by **docker create**.
  - The command **ping www.centralesupelec.fr** is executed.
  - When the container is **stopped**, it gets **removed** (option **-rm**).

Create, start and run: an example

What do you obtain from the following command?

```
docker run -it –name my-ubuntu ubuntu
```

If you stop the container *my-ubuntu*, how would you start it again?

What is an image?

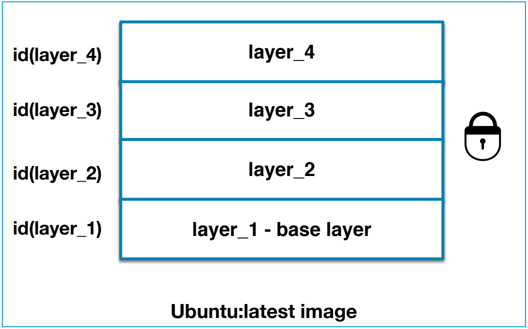
Definition (Image)

An **image** is a read-only template with instructions for creating a Docker container. ▸ Reference

- An image contains a collection of **files** and **folders** necessary to run a containerized application.
- The files and folders are organized into a **layered filesystem**.

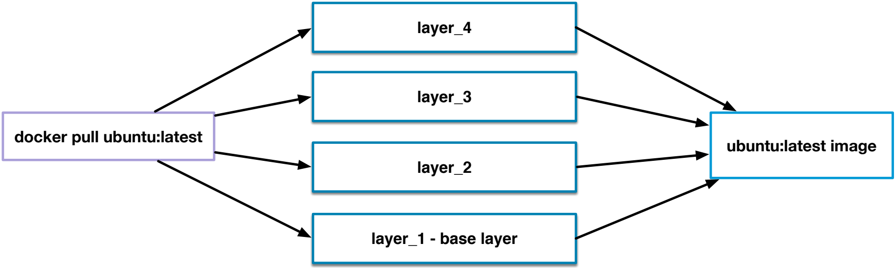
Anatomy of an image

- An image is a collection of **layers**.
- Each layer contains files and folders.
- Each layer has an **identifier**.
- Importantly, layers are **read-only**.



Pulling the layers

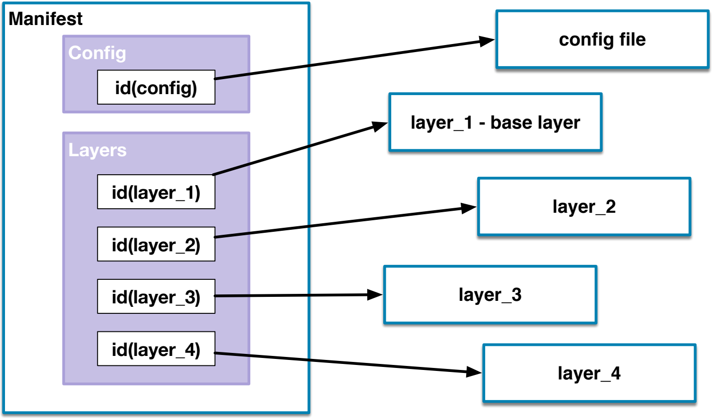
- Layers are **independent** of any image.
- To **pull an image** from the registry, Docker:
  - Pulls each layer **separately**.
  - Assembles the layers into a **layered filesystem**.



- How does Docker retrieve the layers? The **manifest file**.

The manifest file

- **ubuntu:latest** is associated to a file called the **manifest**.
  - Pointer to a **configuration file**: info to run a container from the image.
  - Pointer to the files containing the layers.



Manifest file of *ubuntu:latest*

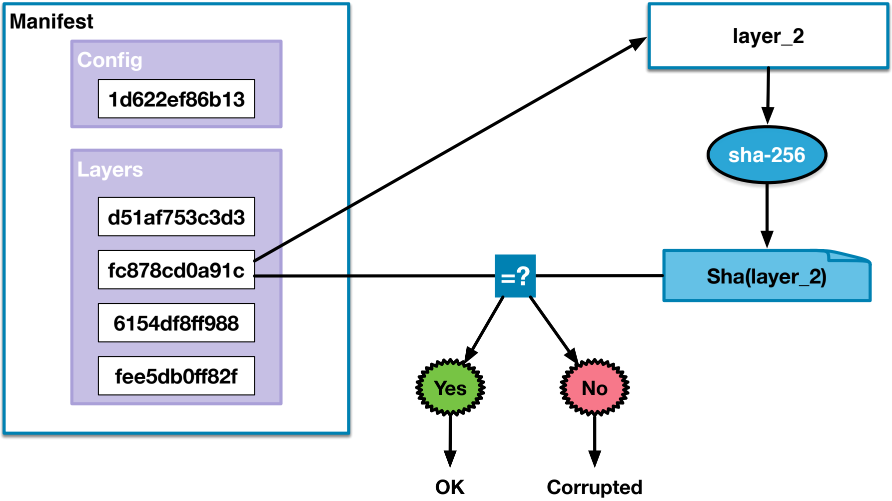

```
{
      "schemaVersion": 2,
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "config": {
        "mediaType": "application/vnd.docker.container.image.v1+json",
        "size": 3408,
        "digest": "sha256:1d622ef86b138c7e96d4f797bf5e4baca3249f030c575b9337638594f2b63f01"
      },
      "layers": [
        {
          "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
          "size": 28556247,
          "digest": "sha256:d51af753c3d3a984351448ec0f85ddafc580680fd6dfce9f4b09fdb367ee1e3e"
        },
        {
          "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
          "size": 32304,
          "digest": "sha256:fc878cd0a91c7bece56f668b2c79a19d94dd5471dae41fe5a7e14b4ae65251f6"
        },
        {
          "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
          "size": 847,
          "digest": "sha256:6154df8ff9882934dc5bf265b8b85a3aeadba06387447ffa440f7af7f32b0e1d"
        },
        {
          "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
          "size": 163,
          "digest": "sha256:fee5db0ff82f7aa5ace63497df4802bbadf8f2779ed3e1858605b791dc449425"
        }
      ]
    }
  }
```

Layer identifiers

- **Identifier** of a layer: **SHA256 digest** of its content.
- **Identifier** of the configuration file: a **SHA256 digest** of its content.
- **Digest** of a file: cryptographic representation of the file **content**.
    - Computed with the **SHA-256 algorithm**.
- Important to verify the **integrity** of a layer.
- Two files have the **same digest** iff they have the **same content**.

$$Sha(x) = Sha(y) \iff x = y$$

Layer integrity



Sharing layers

- Two images might reference the same layers.
- Layers are only downloaded once.

Example: pulling *ubuntu:latest*

```
docker pull ubuntu
```

```
Using default tag: latest

latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
```

Pulling each layer independently

```
Digest:
sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7
```

SHA-256 digest of the manifest file of the image

Example: pulling quercinigia/ubuntu-ping:latest

```
docker pull quercinigia/ubuntu-ping
```

```
Using default tag: latest

latest: Pulling from quercinigia/ubuntu-ping:latest
d51af753c3d3: Already exists
fc878cd0a91c: Already exists
6154df8ff988: Already exists
fee5db0ff82f: Already exists
2d2158c9757f: Pull complete

Digest:
sha256:0ae0d96e80f456f7d078b2cff5983ee26a0767260fc5c4df11141b224c74ff0a
```

Layers from ubuntu:latest

Image identifier

Command to list local images:

```
docker images
```

Or:

```
docker image ls
```

The output is:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	$\boxed{1d622ef86b13}$	3 days ago	73.9MB

SHA-256 digest of the configuration file of the image

How to build an image

- **Working example**: sentiment analysis of sentences in Python.

Source main.py

```
import sys
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

if __name__=='__main__':
    nltk.download('vader_lexicon')
    sid = SentimentIntensityAnalyzer()
    sentence = sys.argv[1]
    print("\nSentiment analysis on the following sentence ", sentence)
    print(sid.polarity_scores(sentence))
```

- The image must include:
  - The source *main.py*
  - All the dependencies (e.g., *nltk*).
  - The Python interpreter.
  - Instructions on how to run the code.
- Most common way to build an image: **Dockerfile**.

Dockerfile

- **Each line** of the Dockerfile corresponds to a **layer** in the image.
- **FROM**. Specifies the **base image**.
- **RUN**. Used to run any valid Linux command.
- **WORKDIR**. Sets the working directory in the image.
- **ENTRYPOINT**. Command to execute when a container is run.
- **CMD**. Any argument passed to the entrypoint command.

Build instructions	FROM python:3.7-slim	build from base image
	RUN mkdir -p /app	create folder /app in the image
	WORKDIR /app	set working directory to /app
	COPY ./main.py ./requirements.txt /app/	copy files to /app
	RUN pip install -r requirements.txt	install the Python dependencies
Run instructions	ENTRYPOINT ["python", "main.py"]	run python main.py...
	CMD ["A default sentence"]	....with an argument

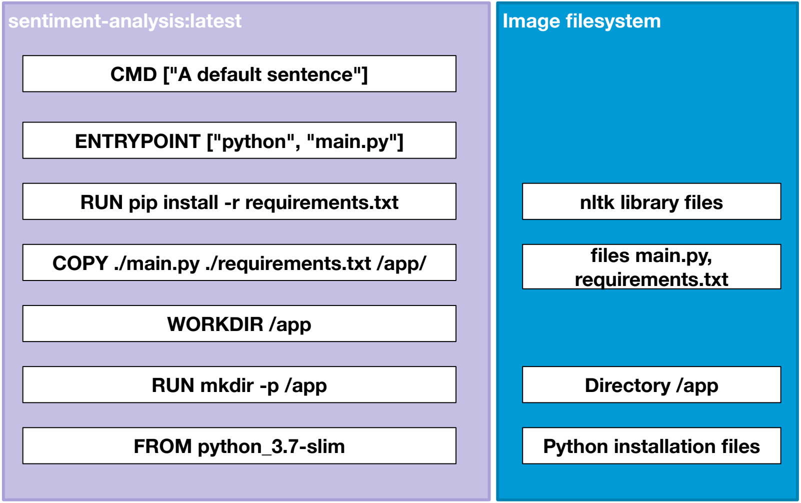
Build the image

```
docker build -t sentiment-analysis:latest .
```

- The Docker engine looks for a file named **Dockerfile** in the current directory.
- Executes the instructions in the Dockerfile and creates an image tagged *sentiment-analysis:latest*.
- If the Dockerfile has another name (e.g., *Dockerfile-sentiment*), the command to build the image is as follows:

```
docker build -t sentiment-analysis:latest -f Dockerfile-sentiment .
```

The resulting image



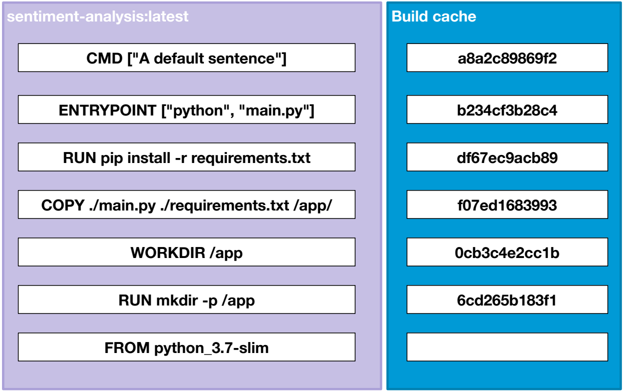
A closer look at the image history

```
docker history sentiment-analysis
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
a8a2c89869f2	20 seconds ago	/bin/sh -c #(nop) CMD ["A default sentence"]	0B	
b234cf3b28c4	20 seconds ago	/bin/sh -c #(nop) ENTRYPOINT ["python" "mai…	0B	
df67ec9acb89	21 seconds ago	/bin/sh -c pip install -r requirements.txt	18.6MB	
f07ed1683993	26 seconds ago	/bin/sh -c #(nop) COPY multi:4e5c15fb4242a23…	334B	
0cb3c4e2cc1b	26 seconds ago	/bin/sh -c #(nop) WORKDIR /app	0B	
6cd265b183f1	26 seconds ago	/bin/sh -c mkdir -p /app	0B	
59ade49ea505	3 days ago	/bin/sh -c #(nop) CMD ["python3"]	0B	
<missing>	3 days ago	/bin/sh -c set -ex; savedAptMark="$(apt-ma…	7.51MB	
<missing>	3 days ago	/bin/sh -c #(nop) ENV PYTHON_GET_PIP_SHA256…	0B	
<missing>	3 days ago	/bin/sh -c #(nop) ENV PYTHON_GET_PIP_URL=ht…	0B	
<missing>	3 days ago	/bin/sh -c #(nop) ENV PYTHON_PIP_VERSION=20…	0B	
<missing>	3 days ago	/bin/sh -c cd /usr/local/bin && ln -s idle3…	32B	
<missing>	3 days ago	/bin/sh -c set -ex && savedAptMark="$(apt-…	95.1MB	
<missing>	4 days ago	/bin/sh -c #(nop) ENV PYTHON_VERSION=3.7.7	0B	
<missing>	4 days ago	/bin/sh -c #(nop) ENV GPG_KEY=0D96DF4D4110E…	0B	
<missing>	4 days ago	/bin/sh -c apt-get update && apt-get install…	7.03MB	
<missing>	4 days ago	/bin/sh -c #(nop) ENV LANG=C.UTF-8	0B	
<missing>	4 days ago	/bin/sh -c #(nop) ENV PATH=/usr/local/bin:/…	0B	
<missing>	4 days ago	/bin/sh -c #(nop) CMD ["bash"]	0B	
<missing>	4 days ago	/bin/sh -c #(nop) ADD file:9b8be2b52ee0fa31d…	69.2MB	

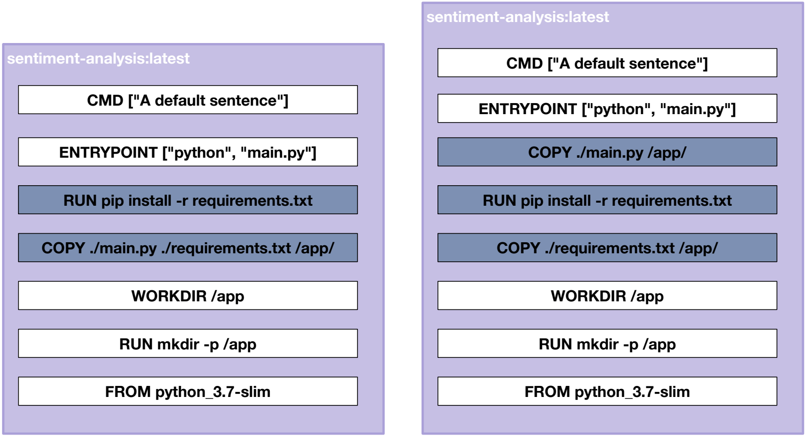
Build cache

- For each layer, the Docker engine creates a **cache image**.
- When building again, only the modified layer and those above are built again (**build cache**).



Build cache

- For efficient builds, order matters.
- When we modify the source code, the dependencies are installed again.



Dockerfile best practices

- **Small number of layers**.
  - Example: merge two consecutive RUN commands.
- **Order layers correctly**.
  - Efficient use of the build cache.
- **Small size images**.
  - Put only necessary files into images.

Running our container

What's the output of the following command?

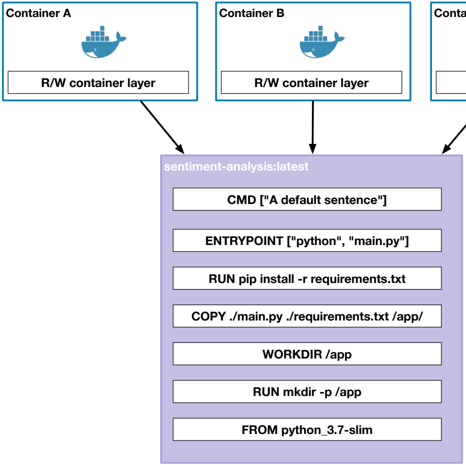
```
docker run –rm sentiment-analysis
```

What's the output of the following command?

```
docker run –rm sentiment-analysis "This is my positive sentence"
```

The container layer

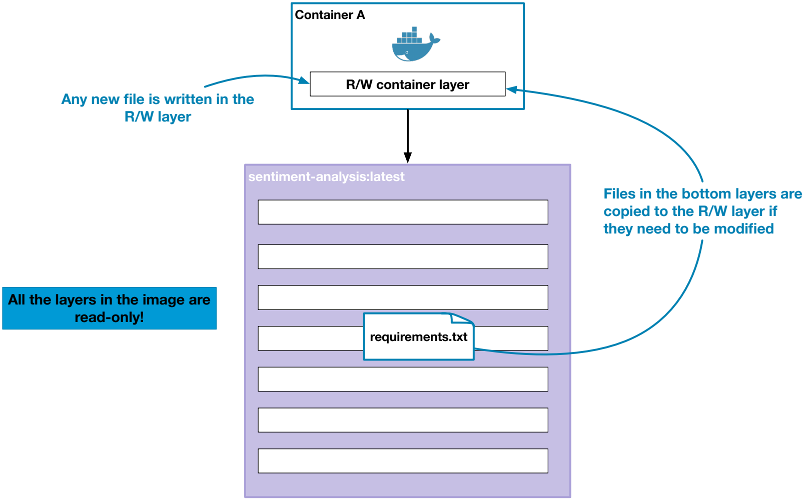
- A container is a **runnable instance** of an image.
- It includes all the layers of the image, and a **writable container layer**.



All files created by the containerized app are stored in the **writable container layer**.

When the container is removed, **all files** in the writable container layer are **destroyed**.

Running a container: copy-on-write (COW)



Highlights

- An image is composed of a **layered filesystem**.
- Layers are **independent** of the image.
  - Layers can be shared by images.
- Layers are **identified** by their **content digest**.
  - Possibility of **checking their integrity**.
- Layers are **immutable**.
  - Images, being a collection of immutable layers, are immutable too.
  - An image always behaves the same.
- A container is a **runnable instance** of an image with a **R/W top layer**.
- New files are written to the R/W container layer.
- Files in the bottom layers are copied to the R/W top layer before any modification.

Volumes

- All data created by a containerized application is written to the R/W container layer.
- The R/W layer is **destroyed** when the container is removed.

How can we persist data beyond a container's lifecycle?

- The answer is: **volumes**.

Volumes

Definition (Volume)

A **volume** is a directory that lives outside of the image filesystem and corresponds to a directory in the host computer.

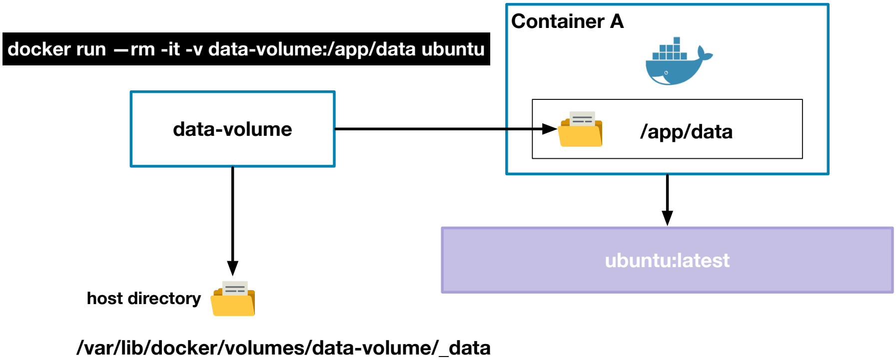
- We can think of a **volume** as an external storage drive for a container.
- Before using a volume, we need to declare it.

```
docker volume create data-volume
```

- *data-volume* is attached to a directory in the **host computer**.
  - Docker for Linux. Directory accessible under */var/lib/docker/volumes*.
  - Docker for Mac. Directory accessible in the Linux VM.
  - Docker for Windows. Directory accessible in the Hyper-V container.

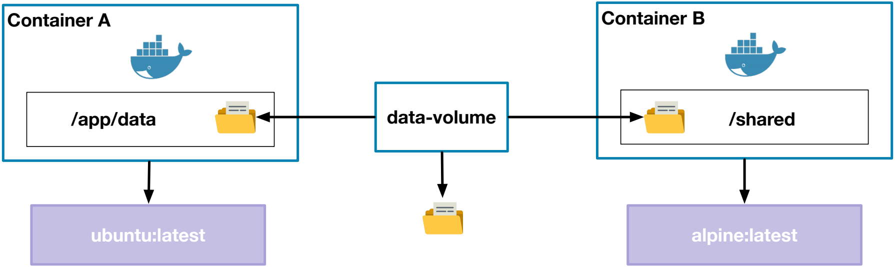
Mounting a volume

- All files written in the directory */app/data* are written to the volume *data-volume*.



Data sharing

- Volumes are a great way to **share data** between containers.



- The two containers don't need to be run from the same image.
- The two containers can mount the volume at any directory.

Networks

- Containers **isolate** the applications from their environment.
- Great feature in a micro-service architecture.

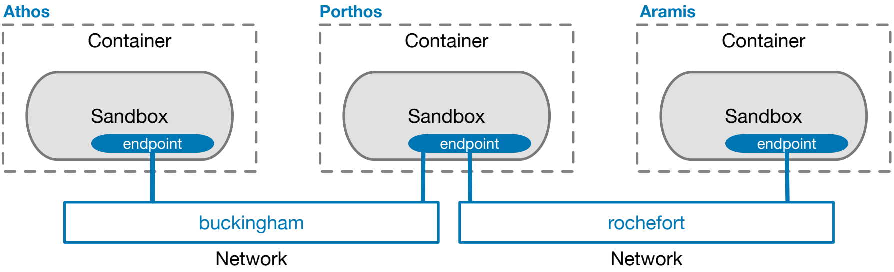
How can two containers running on the same host **communicate**?

- Docker provides an object called **network**.

Container Network Model

Docker defines a **container network model** consisting of three elements:

- **Sandbox**. Contains the **network configuration** (IP and MAC addresses, routing tables, DNS records).
- **Endpoint**. Connection between the sandbox and a network.
- **Network**. Provides the functions to connect two or more endpoints.
  - A network implementation is called a **driver**.



Driver

Running the command **docker network ls** gives the following output:

NETWORK ID	NAME	DRIVER	SCOPE
7ca6f4877494	bridge	bridge	local
4140d2566476	host	host	local
b35755c21cd2	none	null	local

- **bridge**. Network based on Linux bridges.
- **host**. Network of the host.
- **null**. Used to disconnect containers from any network.

All these drivers allow **single-host networking**.

Creating networks

- Command to create a network:

```
docker network create buckingham
```

- Run a container and attach it to a network:

```
docker run –rm -it –name c1 –network buckingham alpine
docker run –rm -it –name c2 –network buckingham alpine
```

- If we need to attach a container to another network:

```
docker network connect bridge c1
```

Inspecting a network

docker inspect buckingham


```
....
"IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
           {
              "Subnet": "172.18.0.0/16",
              "Gateway": "172.18.0.1"
           }
        ]
     },
....
```

```
....
"Containers": {
        "4ba0f09c5f7b": {
           "Name": "c2",
           "EndpointID": "5557c6fe08f2",
           "MacAddress": "02:42:ac:12:00:03",
           "IPv4Address": "172.18.0.3/16",
           "IPv6Address": ""
        },
        "b971823f57c0": {
           "Name": "c1",
           "EndpointID": "5fbaccf81b42",
           "MacAddress": "02:42:ac:12:00:02",
           "IPv4Address": "172.18.0.2/16",
           "IPv6Address": ""
        }
     },
....
```

Inspecting a network

- Docker daemon acts as a **DHCP server**.
- The **hostname** of a container is **its identifier**.
- But one can also use its name.

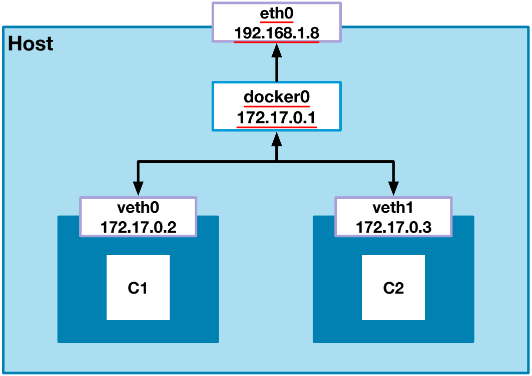
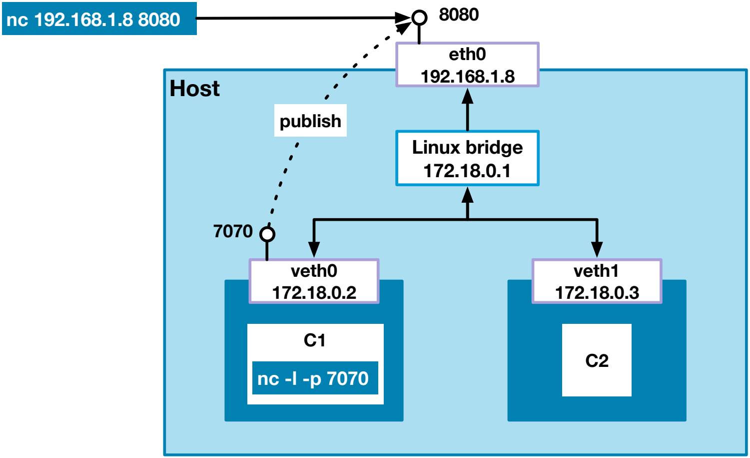


Image source

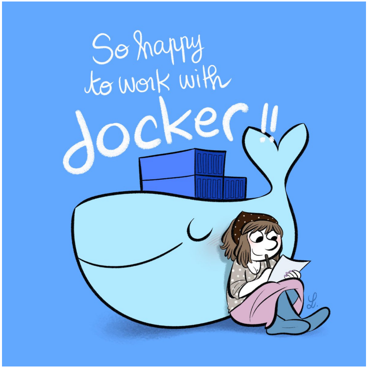
Publishing ports



docker run --rm -it --name c1 --network buckingham -p 8080:7070 alpine

Resources and implementation

- A docker primer ▸ Click here.
- Tutorial: Getting started with Docker ▸ Click here.



▸ Image source