- Goals and topics

- Goals
  - introduce Docker
  - show you how to run a Spring Boot application in Docker
- Subjects
  - Docker
  - Docker in practice
  - how Docker works
  - a containerized application
  - discussion

* Docker

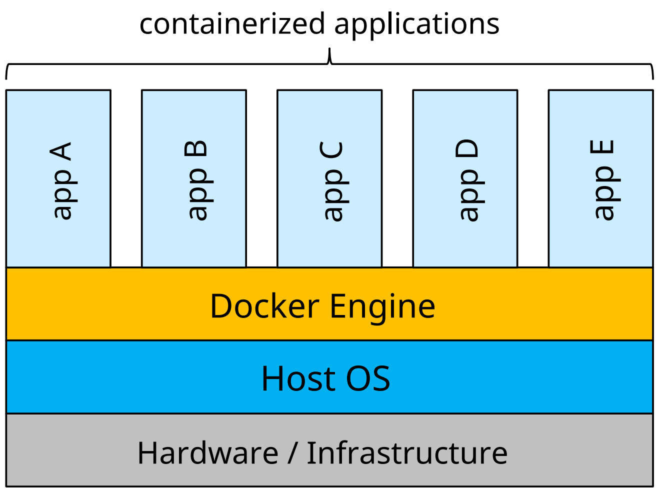
- *Docker* (www.docker.com) is a container platform to build, release and run distributed applications - in a simple, fast, scalable and portable way
  - a *Docker container* is a standardized software unit, which packages a software service, along with its configurations and dependencies
    - a container contains everything needed to run that software service - executable code, configurations, libraries, and system tools
  - Docker containers are lightweight (they use few resources and boot quickly), standardized and open (and therefore portable: they can run with major Linux distributions and with Windows and Mac OS, and even in the cloud) and secure

History

- Docker platform (2013) was initially built on top of LXC containers (2008)
  - LXC offers a set of kernel features for container management - which are low-level and often difficult to use directly.
  - Docker built on this foundation to provide a more powerful and easier-to-use set of high-level tools and features
  - Docker today is based on libraries *containerd* And *runc* (2014, 2015) - as well as on *cgroup* And *namespace*
  - Docker was an instant hit and is used in production by many companies - few technologies have seen such an adoption rate

Docker

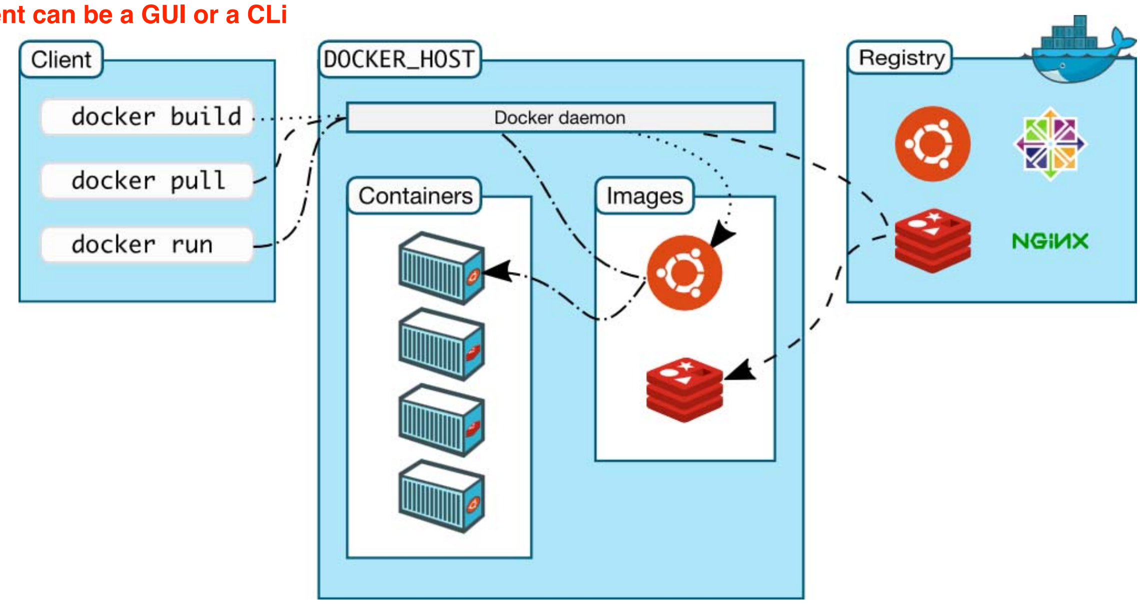
- The platform *Docker* allows for a separation between applications and execution infrastructure
  - to simplify the release of applications
  - to ensure the portability of services implemented through containers - both on premises and in the cloud



Docker Engine

- The fundamental core of the Docker platform is *Docker Engine*

the client can be a GUI or a CLi



Docker Engine

- Docker Engine is based on a client-server architecture
  - the *server* is a host capable of running and managing Docker containers
    - runs the Docker daemon process (dockerd)
    - manages a set of Docker objects - containers, images, networks and volumes
  - the *client* (docker) accepts commands from the user via a CLI interface and communicates with the Docker daemon on the host
    - communication takes place via a REST API
  - the *registry* contains a set of images
    - Docker's public registry is Docker Hub

Containers and images

- Two basic types of Docker objects
  - a *container* it is, in fact, a container instance, which contains an application or a service - along with everything needed to run it
    - it's a concept dynamic, runtime
    - can be run on a host
  - a *image* is a template for creating containers
    - it's a concept static
    - it cannot be done directly
  - relationship between container and images
    - each container is created from an image
    - from one image it is possible to create many containers

Images

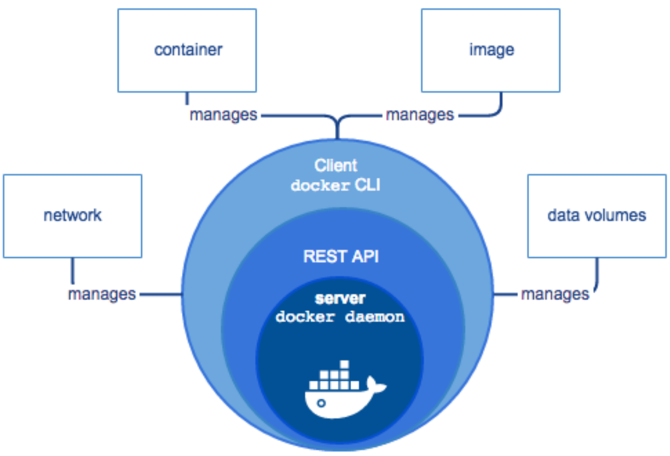
- In practice, a '*image* is a set of files - representing a container's file system snapshot
  - eg, an image with an Ubuntu OS, Open JDK, and a specific Java application of interest
  - another image could be specific to NGINX or to Apache Kafka
  - an image is a concept static, inert
    - it is not done directly
    - it has no state of its own
    - it is immutable

Container

- A *container* is an executable instance of container, created from a Docker image
  - an "application container" - which contains an application or service
  - for example, a distributed software system might include
    - N containers that are all replicas of a web application of interest (based on the same image)
    - an additional container to distribute client requests among the N replicas of the web application of interest (based on an image for NGINX)
  - a container is a concept dynamic, runtime
    - can be run on a host
    - has its own state - which can change during execution
      - e.g., the contents of the file system (in the disk) or the state of the sessions (in the main memory)

The Docker server

- To summarize, the Docker server
  - runs the Docker daemon process
  - manages a set of Docker objects - mostly containers and images
  - allows access to its clients, local and remote, through CLI and REST



Registry of images

- A *registry* is a service (public or private) that contains a collection of container images
  - *Docker Hub* (https://hub.docker.com) is Docker's public registry - but private registries are also possible
  - a *repository* is a portion of a registry that contains a set of container images - usually they are variants or different versions of the same image
- A public registry typically contains *basic images* - which contain only an OS, but in some cases also basic software - but not application software
  - eg, base images are ubuntu, postgres, wurstmeister / kafka And openjdk

Functionality and use

- Here are the main features offered by the Docker platform
  - create a container (a container instance) from a container image
  - start, monitor, inspect, stop and destroy containers
  - create and manage container images
  - manage related groups of containers - in which to run multi-container distributed applications

* Docker in practice

- Interaction with a Docker host is done through an interface (CLI or remote, the remote interface is based on a REST API)
  - this API is command based docker - with numerous options / commands / operations for managing images and containers (and other Docker objects) and their life cycle
    - the commands docker image for image management
    - the commands docker container for container management
  - some commonly used commands exist in two versions, one extended and one short
    - e.g., docker container run And docker run
    - e.g., docker image ls And docker images

Docker in practice

- Some basic Docker commands
  - docker image build (or docker build) allows you to build a (custom) image
    - docker build -t *image-name context*
  - docker container create (or docker create) allows you to create a new container from an image
    - docker create --name =*container-name image-name*
  - docker container start (or docker start) allows you to run a container (already created)
    - docker start *container-name*
  - docker container run (or docker run) creates and executes a new container (possibly anonymous), using a single command
    - docker run [--name =*container-name*] *image-name*

Creating and running containers

- A first minimal example - based on the image hello-world available at the Docker Hub
  - docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with: $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID: https://hub.docker.com/

For more examples and ideas, visit: https://docs.docker.com/engine/userguide/

Creating and running a container

- Another example, based on another default image
  - docker run docker / whalesay cowsay Hello, world!

```
 ______________
<Hello, world! >
 - - - - - - - - - - - - - - -
    \
     \
      \
                    # #         .
              ######          ==
           ########          ===
      / "" "" "" "" "" "" "" "" ___ / ===
 ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ / === - ~~~
      \______ or __/
       \ \           __ /
         \ ____ \ ______ /
```

Image construction

- For building a custom image, Docker uses a type approach *infrastructure-as-code* - based on a special text file named **Dockerfile**
  - the Dockerfile contains all the commands to execute to build a custom image
  - the command docker build -t *image-name context* allows you to automatically build an image (named *image-name*) starting from a context *context*
    - the context can be a local folder - in particular, . - or a location on a Git repository
    - the context must contain the Dockerfile, along with any other files of interest (e.g., binary files, scripts and templates)

Dockerfile - FROM and ENTRYPOINT

- A **Dockerfile** it is made up of a sequence of instructions

```
# Hello world
FROM busybox: latest
ENTRYPOINT ["echo", "Hello, world!"]
```

  - education **FROM** specify the base image from which to build the custom image (and possibly its version)
    - e.g., busybox (is a minimal Linux distribution) or ubuntu: 18.04
  - **ENTRYPOINT ["*executable*","*param1*","*param2*", ...]** is a statement that specifies the executable or command to be executed by the containers that will be created from this image
  - a Dockerfile it must begin with an education FROM and usually ends with a single statement ENTRYPOINT

Creation of the image and container

- Building an image
  - docker build -t myhello. - from the folder that contains the Dockerfile seen above
    - create a new picture named myhello
- Creating a container
  - docker create --name = myhello myhello
    - create a new container named myhello starting from the image myhello
- Running a container
  - docker start -i myhello
    - start the container myhello (interactively)
    - in this case, display Hello, world! and then it ends

```
Hello, world!
```


Dockerfile - CMD

- Education **CMD** allows you to specify arguments for the statement ENTRYPOINT - with the remark that these arguments can be overridden when the container starts

```
# Hello world
FROM bosybox: latest
ENTRYPOINT ["echo"]
CMD ["Hello, world!"]
```

- docker build -t myhello2.
- docker run myhello2

```
Hello, world!
```

- docker run myhello2 Hello, world!

```
Hello World!
```

Example: Apache HTTP Server

- In the Dockerfile other instructions can also be used
  - e.g., the Dockerfile for an Apache HTTP server

```
# Dockerfile for Apache HTTP Server

FROM ubuntu: 18.04

# Install apache2 package
RUN apt-get update && \
    apt-get install -y apache2

# Other instructions
ENV APACHE_LOG_DIR / var / log / apache2
VOLUME / var / www / html
EXPOSE 80

# Launch apache2 server in the foreground
ENTRYPOINT ["/ usr / sbin / apache2ctl", "-D", "FOREGROUND"]
```

  - now we explain the new instructions

The RUN instruction

- Education **RUN** specifies a command to run when building an image
  - eg, to request execution of a command or script during the provisioning of the container image - and not during the execution of the container
  - a Dockerfile it can contain multiple instructions RUN - which are performed sequentially
- The main difference between education ENTRYPOINT and instructions RUN it is the moment of their execution
  - the instructions specified by RUN are performed during the construction of an image
  - the instruction specified by ENTRYPOINT it will be executed by the containers created from the image

The RUN instruction

- It is usually preferable to have in one Dockerfile one instruction RUN (or a few) - which specify a sequence of commands separated by && \ - instead of lots of instructions RUN
  - eg

    # Install apache2 package (better!) RUN
    apt-get update && \
        apt-get install -y apache2

  - should be preferred to

    # Install apache2 package (worst!) RUN
    apt-get update
    RUN apt-get install -y apache2

  - the explanation for this advice is given below

Other instructions

- Other instructions for the Dockerfile
  - education **COPY *src dest*** copy a set of files or folders from the source ***src*** (which must be relative to the context of the construction of the image) to the destination ***dest*** (in the container)
  - education **ADD *src dest*** it is similar - but allows you to copy remote files (i.e. external to context) into the container
  - education **ENV *key value*** set an environment variable in the container

The VOLUME statement

- Other instructions for the Dockerfile
  - education **VOLUME *path*** defines an external mount point - for data on the host system or in another container
    - education VOLUME must be used in conjunction with other command options docker create And docker run
      - the option -v *host-src: container-dest* to mount a host system folder in the container - it is a shared folder between the host and the container
      - the option -volumes-from =*container-name* to mount a volume managed by another container into the container

The EXPOSE statement

- Other instructions for the Dockerfile
  - education **EXPOSE *port*** specifies that the container listens at runtime on the port ***port***
    - this statement is usually used in conjunction with other command options docker create And docker run to publish (this is Docker's term for port forwarding) some ports of a container on its host
      - the option -p *host-port: container-port* to publish a specific port exposed by the container to a specific port on the host
      - the option -P. to publish all ports exposed by the container to random host ports
    - it should be noted that containers can still communicate with each other even on ports that are not exposed or not published on the host

Example: Apache HTTP Server

- Dockerfile for an Apache HTTP server

```
# Dockerfile for Apache HTTP Server

FROM ubuntu: 18.04

# Install apache2 package
RUN apt-get update && \
    apt-get install -y apache2

# Other instructions
ENV APACHE_LOG_DIR / var / log / apache2
VOLUME / var / www / html
EXPOSE 80

# Launch apache2 server in the foreground
ENTRYPOINT ["/ usr / sbin / apache2ctl", "-D", "FOREGROUND"]
```


Example: Apache HTTP Server

- Image construction
  - docker build -t myapache. - from the folder that contains the Dockerfile
- Container creation
  - docker create
    - v ~ / projects / www: / var / www / html -p 8080: 80
    - - name = myapache myapache
    - the pages served by the HTTP server are those in the host folder ~ /projects / www
    - HTTP server is redirected to host port 8080
- Container execution
  - docker start myapache - start the container myapache
    - then you can access the HTTP server from the host on http: // localhost: 8080

Other Docker commands

- Other useful Docker commands
  - to list containers running (or even stopped)
    - docker container ls - or docker ps [-a]
  - to inspect the ports used by a container - especially useful when using the option -P.
    - docker container port *container-name* - or docker port
    - the result is of form 80 / tcp -> 0.0.0.0:8080
  - to inspect a container or image
    - docker container inspect *container-name* - or docker inspect
    - returns container or image information (in JSON format) - e.g., network configuration (including port publishing) and volume sharing

Other Docker commands

- Other useful Docker commands
  - to view the logs generated in a container
    - docker container logs *container-name* - or docker logs
  - to stop a running container
    - docker container stop *container-name* - or docker stop
  - to remove a container
    - docker container rm *container-name* - or docker rm
  - to stop all running containers (use with caution!)
    - docker stop $ (docker ps -a -q)
  - to remove all containers (use with caution!)
    - docker rm $ (docker ps -a -q)

Other Docker commands

- Other useful Docker commands
  - to list images in the local cache
    - docker image ls - or docker images
  - to remove an image from the local cache
    - docker image rm *image-name* - or docker rmi
  - to remove all images from the local cache (use with caution!)
    - docker rmi -f $ (docker images -q)

Other Docker commands

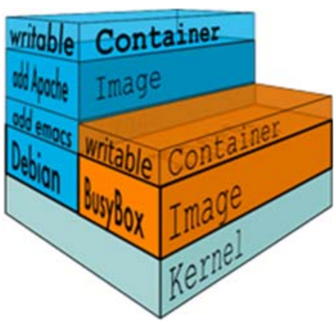
- Other useful Docker commands
  - the Docker client can also be used to specify commands to run on a remote Docker host *docker-host*
    - docker -H = tcp: //*docker-host*: 2375 *command*
    - docker -H = tcp: //*docker-host*: 2376 *command*
    - port 2376, unlike 2375, supports secure access over TLS
    - the Docker host must be enabled for remote access
  - alternatively, you can specify the remote Docker host using the environment variable DOCKER_HOST
    - export DOCKER_HOST = tcp: //*docker-host*: 2375
    - docker *command* - the command is executed on *dockerhost* instead of locally

* How Docker works

- We still discuss how Docker works - specifically, the following aspects
  - format of images and containers
  - construction of images
  - creation of containers
  - running container
  - data sharing (volumes)
  - networks
  - registry

- Format of images (and containers)

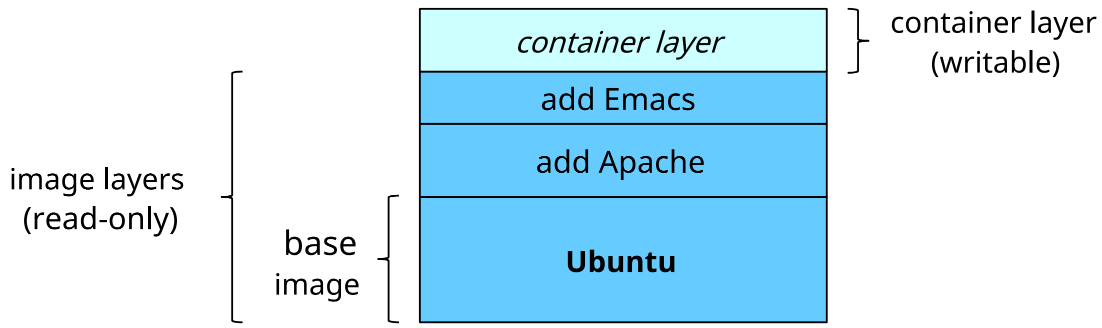
- A key element of Docker is the format used for the image and container file system
  - the file system of an image (or container) is made up of a sequence of layers - each layer is a set of files



  - these layers are combined into a single consistent file system using one *Union File System* (*UFS*)
    - a file is read in the topmost layer in which it is located
    - in a container, the only layer that can be written at runtime is the topmost layer

Format of images (and containers)

- A key element of Docker is the format used for the image and container file system
  - the file system of an image (or container) is made up of a sequence of layers - each layer is a set of files

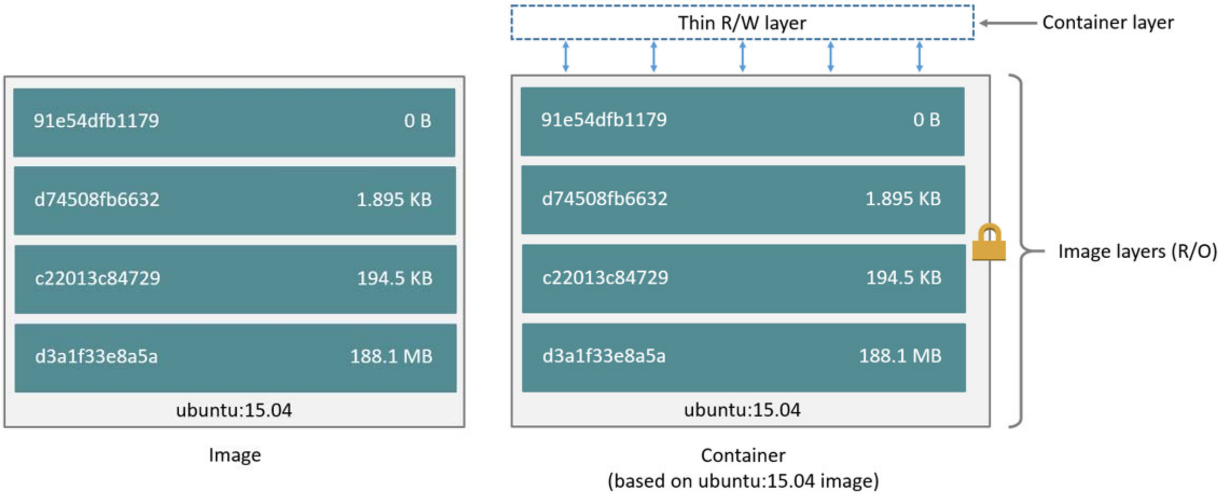


Format of images (and containers)

- In the file system of each image (or container), the base is always a base image - usually it contains an OS and its libraries
  - each subsequent layer usually corresponds to the installation of a package, middleware or application
  - in addition to these layers, each container (but not the images) has a final additional layer, which is the only editable part of the container's file system
    - all writes, modifications and deletions performed in the container operate on the latter additional layer
  - this "light" format
    - allows you to share layers between images and between containers
    - makes it easy to update images (e.g., to update an application to a new version) - which can be done by updating or adding layers, rather than completely rebuilding the images

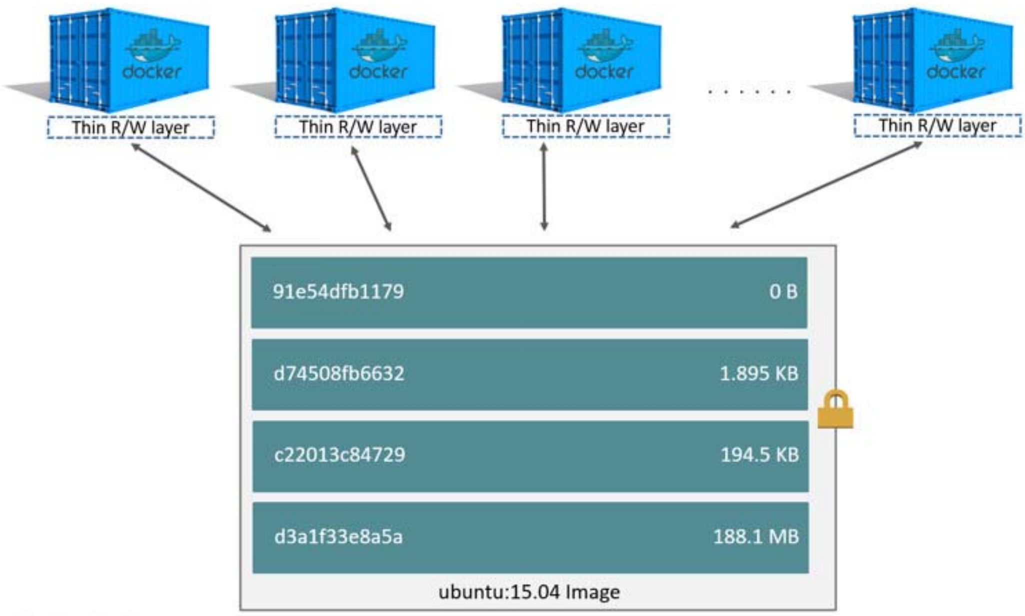
Images and containers

- An image
- A container (or rather, its file system)



Images and containers

- An image shared by multiple containers



- Image building

- Building a custom image is based on running a Dockerfile, and happens as follows
  - first, the base image specified by FROM it is downloaded from the registry into an image cache of the host (if it is not already present in the cache)
    - this base image (made up of one or more layers) is used as the base layer (s) of the new custom image
  - then, repeatedly, each instruction of the Dockerfile (particularly, RUN) is done using a new writable layer on top of the current image
    - the result of executing a del statement Dockerfile is then saved (before executing the next instruction)
    - Docker recommends minimizing the number of layers in images and containers - and thus minimizing the number of instructions RUN of a Dockerfile

- Creation of containers

- The creation of a container always takes place from an image
  - a container consists of a file system and meta-data
    - the (layered) file system of the container is obtained from the initial image, to which a new writable layer, specific for the container is added on top - this layer is allocated in the host's file system
  - the images are instead immutable and can be shared by multiple containers

- Running containers

- Running a container
  - when a container is requested to run, the container engine allocates runtime resources for the container
    - eg, allocate (in the host kernel) a set of namespaces and configure the network for the container
  - then start the container, starting from its file system
  - finally, the container executes the command specified by ENTRYPOINT (with the arguments specified by CMD or from the command line)

- Volumes and data sharing

- Container storage itself is ephemeral - when a container is destroyed, all of its data is lost
  - how can persistent data be managed?
- A *volume* is a directory outside the container UFS
  - a volume can be accessed, shared and reused by multiple containers
  - a volume allows you to manage persistent data, independently of the life cycle of the individual containers that can access it

Volumes and data sharing

- A first possibility is to mount a host folder as a volume in a container running on the host, using the option
  - v from docker create you hate docker run
    - docker create -v ~ / projects / www: / var / www / html...
    - in this case, the data resides on the host (i.e., in a predefined absolute location of the host's file system) - and not in the container
      - therefore, changes to this data are made on the host, persistently
      - an example of use is to redirect the log files of a server running in a container to the host

Volumes and data sharing

- Another possibility is to have volumes shared between containers - but not tied to a specific folder on the host (in an absolute default location of its file system) - in this case the option must also be used -volumes-from
  - a container must first be created, using the option -v to indicate a shared folder of the container - but without binding this volume to any folder on the host
    - the volume will reside in this container
    - in practice, this volume resides on the host, but not in an absolute default location
  - then you can create other containers that access that shared volume, with the option -volumes-from *container-name*
  - if the container in which a volume resides is deleted, the volume is still kept (unless an explicit deletion is requested)

- Docker allows you to manage network communication between containers as well as with the host
  - during installation, Docker automatically creates three networks, bridge, host And it is not - but it is also possible to create others
  - the network bridge (in "bridge" mode) is associated with the virtual interface docker0 on the host and a private network 172.17.0.1/16
  - when a container is run, Docker associates it with a free IP address on the network bridge
    - it is possible to connect a container to a different network using the option -network =*network*
  - containers can communicate with each other knowing the absolute position (IP address and port) of the various services present on the network
  - the network host instead, it adds a container to the host's network

- Learn more about networks
  - docker inspect allows you to find the information you need to communicate with a container over the network
    - eg, the Apache HTTP server may be exposed to the address 172.17.0.2:80 (of the private network)
  - it is also possible to make these services accessible to the host and outside the host via port mapping (port forwarding) - via options -p And -P. from docker create And docker run
    - Docker manages these options by automatically configuring the NAT rules on the host iptables
    - the option -ip it also allows you to associate a specific IP address (valid for the host) to a container

- Learn more about networks
  - using a *user-defined network* (instead of the network bridge) containers can communicate with each other also through their own "logical" name - as well as by their IP address
    - the container engine operates from DNS for its containers
  - creation of a user-defined network
    - docker network create -d *network-driver network-name*
    - e.g., docker network create -d bridge my-net
  - connecting a container to a network
    - docker run --network = my-net --name = container1 -it busybox
    - other containers connected to this network can see this container by its "logical" name container1
    - a container can also be connected to multiple networks

- Registry

- A registry is a service for managing a set of container images
  - main operations of a registry
    - docker pull *image-name* - download an image from the registry to the host's local cache - otherwise, docker build it does this automatically
    - docker push *image-name* - upload an image to the registry
    - registry query
  - Docker's public registry is *Docker Hub* - some of the images it manages are "official"
    - alternatively, *Docker Registry* is a tool for managing your own private registry
    - in the spirit of Docker, Docker Registry can run as a container

Using Docker Hub

- Using Docker Hub - you must first create your own account, e.g., aswroma3
  - login
    - docker login [-u aswroma3] [-p *password*] [*server*]
  - creation and “tagging” of an image
    - docker build -t aswroma3 / myhello. or
    - docker build -t myhello. followed by
      docker tag myhello aswroma3 / myhello
  - saving an image on the registry (it must be "tagged")
    - docker push aswroma3 / myhello
  - loading an image from the registry (optional)
    - docker pull aswroma3 / myhello
  - creating and running a container from the image
    - docker run aswroma3 / myhello

Docker Registry

- Managing a (private) Docker Registry - in the spirit of Docker, it can be run as a container
  - start the registry (the -d runs the container in the background) - suppose on the node myregistry
    - docker run -d -p 5000: 5000 --restart = always --name registry
      - v / var / local / docker / registry: / var / lib / registry registry: 2
  - creation and “tagging” of an image
    - docker build -t myhello.
    - docker tag myhello myregistry: 5000 / myhello
  - saving an image on the registry (it must be "tagged")
    - docker push myregistry: 5000 / myhello
  - loading an image from the registry
    - docker pull myregistry: 5000 / myhello
  - creating and running a container from the image
    - docker run myregistry: 5000 / myhello

- General recommendations

- Some recommendations on containers - and related images
  - only one process per container
    - supports the reuse of images and containers
    - supports horizontal scaling
  - "ephemeral" containers (*ephemeral*, i.e. temporary, passenger, and stateless) - as far as possible
    - so that a container can be stopped and destroyed and then replaced by another container as quickly as possible
    - supports availability and scalability
  - minimal containers
    - use the smallest possible base image, avoid installing unnecessary packages, and minimize the number of layers
    - supports availability

* A content-based application

- Before concluding, here is an example of running a simple web application (Spring Boot) in a Docker container
  - the application lucky-word - see the handout on Spring Boot
- here is the Dockerfile (in the root folder of the Spring Boot project) which uses an image with Open JDK

```
# Dockerfile for the lucky-word application
FROM openjdk: 11-jdk
# Install the application binary
ADD build / libs / lucky-word.jar lucky-word.jar
EXPOSE 8080
# Launch the Java application
ENTRYPOINT ["java", "-Xmx128m", "-Xms128m"]
CMD ["-jar", "lucky-word.jar"]
```

A containerized application

- Here's how to build and run this application
  - first of all, you need to build the Spring Boot application lucky-word (in the development environment)

gradle build

  - after that, we need to build a container image for the application (in the environment for Docker)

build container image docker
build --rm -t lucky-word.

  - finally, you have to create and start the container (always in the environment for Docker)

run application with default docker run -p 8080:
8080 lucky-word profile

or, to run the application with the Italian profile docker run -p
8080: 8080 lucky-word
- jar -Dspring.profiles.active = italian lucky-word.jar

A common mistake

- Be careful to avoid the following common mistakes (which everyone makes sooner or later)
  - after modifying (the source code of) an application, remember (always!) to do the following
    - build (or repeat) the (Java) build of the application
    - build (or repeat) the (Docker) build of the Docker image
    - push (or repeat) the push to Docker Hub of the Docker image (if necessary)
    - Sometimes it may also be necessary to remove the older version of the Docker image from the local cache

* Discussion

- The Docker platform has quickly established itself as the reference technology for containers
  - many companies (including large companies like Google!) use Docker not only for development and testing, but also as a production environment for applications with critical requirements for availability, scalability and elasticity
  - Docker is supported both on premises and in the cloud
  - thanks to Docker, containers have become an alternative and complementary application delivery technology to virtualization
  - the rationale for using Docker will be more evident after discussing the composition and orchestration of Docker containers - which is the topic of subsequent lecture notes
  - the advice of Sam Newman (author of Building Microservices) And
    - "I strongly suggest you give Docker a look "