

Introduction to Docker

Giovanni Maria Cristiano

University of Naples «Parthenope»

giovannimaria.cristiano001@studenti.uniparthenope.it



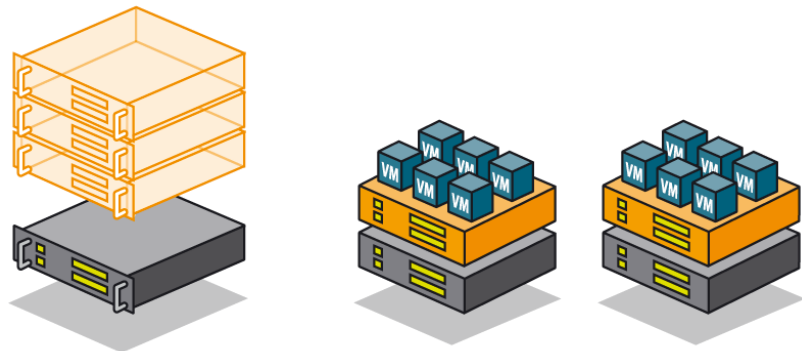
Parte 1

Roadmap

- **Virtualization**
- **VMs vs Containers**
- **What is Docker?**
- **Docker Components**
- **Docker Desktop**
- **Exercises**

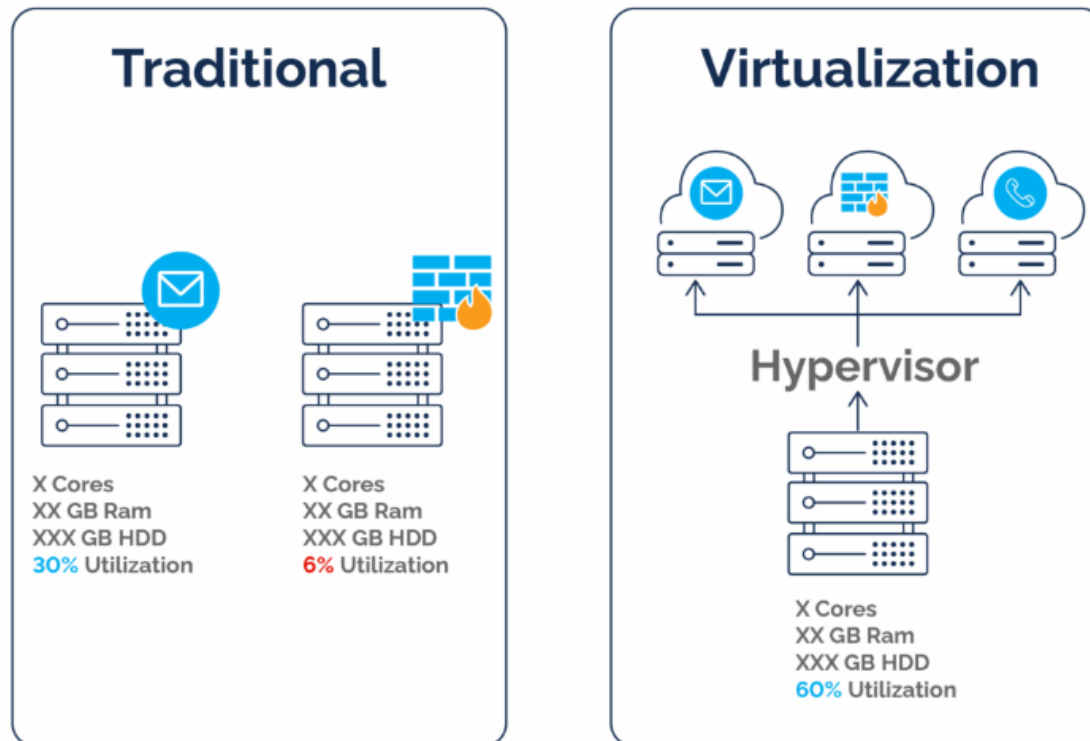
What is Virtualization?

- Virtualization is a process that allows for more efficient utilization of physical computer hardware and is the foundation of cloud computing.
- Virtualization allows the partitioning of the hardware components of a single computer (processors, memory, storage and more) into multiple virtual machines.
- This approach is particularly valuable in data centers and IT infrastructures, where flexibility, efficiency, and resource optimization are priorities.



What is Virtualization?

- In a traditional approach, a single operating system runs on dedicated hardware with exclusive resources. Virtualization, on the other hand, abstracts and divides physical resources among virtual machines (VMs), enabling dynamic sharing, greater flexibility, and advanced isolation. This allows running multiple operating systems simultaneously on a single machine, enhancing resource management and overall system security.



History of Virtualization

- The initial computing systems were large and expensive. The demand for usage led to the evolution of time-sharing systems to enable multiple application to run simultaneously.
- Despite the progress, there were issues with reliability. The idea of virtual machines emerged, allowing the simulation of one computer system on another.
- Virtualization gained renewed attention in the 1990s, notably through a research project at Stanford University that led to the founding of VMware Inc. This marked the resurgence of virtualization technology for commodity hardware.
- In the present day, virtualization is a prominent trend used to reduce management costs.

History of Virtualization

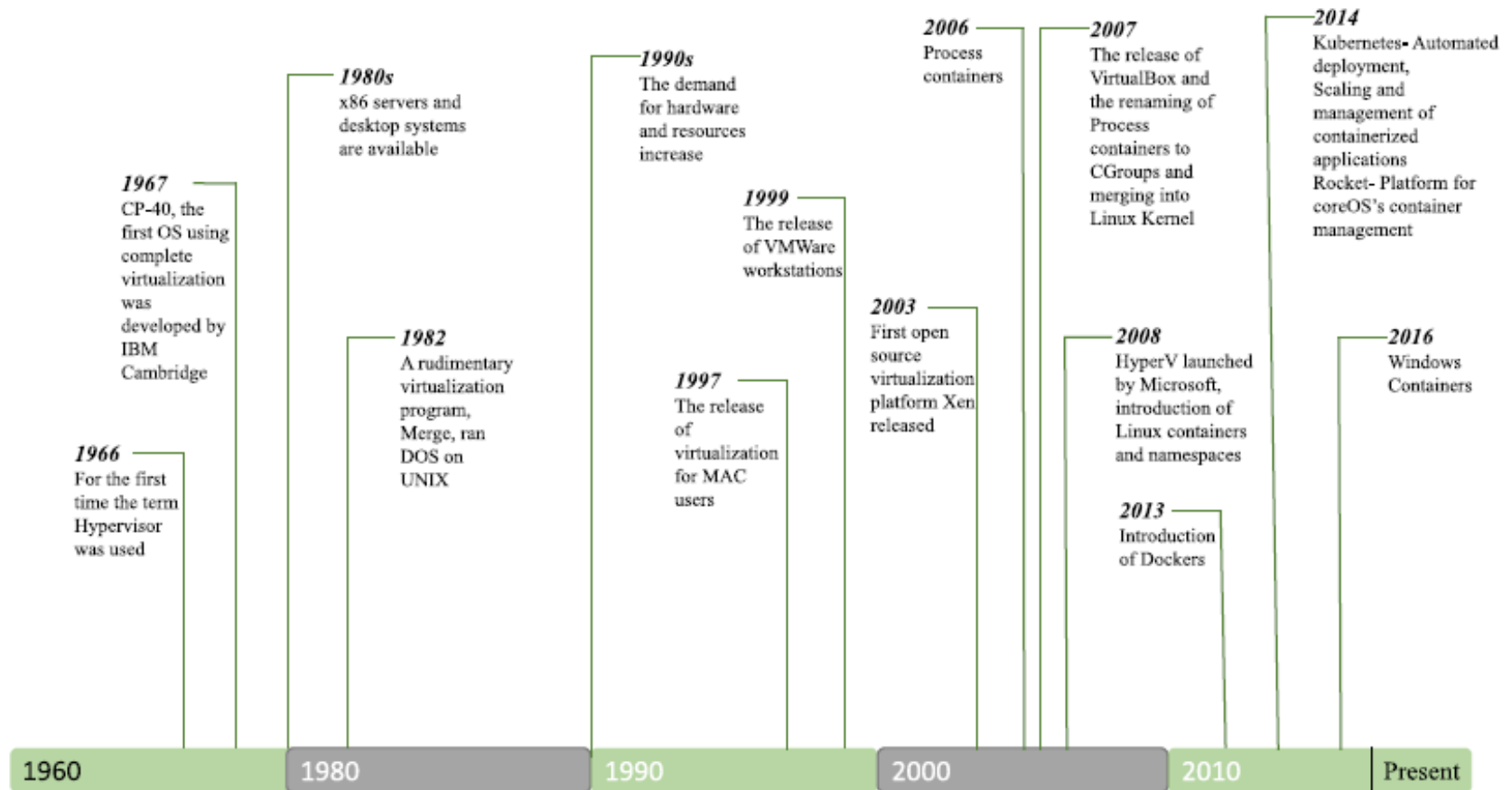


Fig. 4. Looking back at 60 years of virtualization history.

Introducing the new paradigm of Social Dispersed Computing: Applications, Technologies and Challenges

Advantages of Virtualization

- **Resource efficiency** : Virtualization allows multiple virtual machines to run on a single physical server, maximizing the use of hardware resources. This leads to higher performance and cost savings on hardware.
- **Isolation and Security**: Virtual machines are isolated from each other, meaning that issues in one virtual machine do not affect others. This improves overall system security and stability.
- **Agility and Flexibility**: Virtualization enables rapid creation, cloning, and deployment of virtual machines.
 - Disaster recovery
 - You can create your own infrastructure dynamically, with a few lines of codes (or mouse clicks)

Advantages of Virtualization

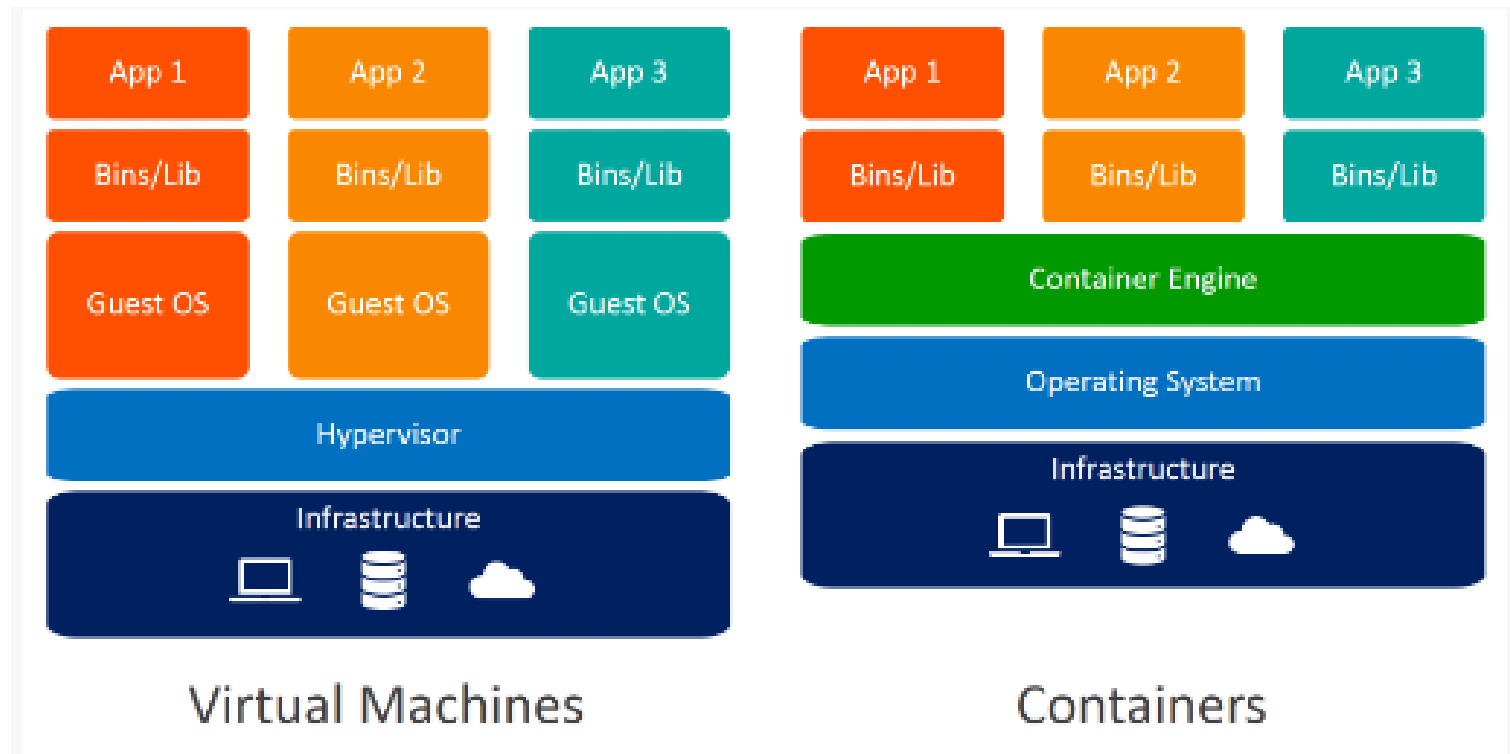
- **High availability:** virtualization provides an extremely reliable system without any points of failure in hardware or software.
- **Scalability:** facilitates horizontal scalability, allowing the easy addition or removal of virtual resources based on needs.
- **Simplified Testing and Development:** Virtualization provides isolated test and development environments, allowing developers to experiment without impacting production systems.

Limitations of Virtualization

- **Performance overhead:** the use of hypervisor for managing virtual machines can result in some performance overhead, as resources need to be shared and managed.
- **Initial costs:** The initial implementation of virtualization can involve significant costs, including specialized hardware and virtualization software.
- **Resource sharing:** In some cases, resource sharing may lead to situations where the performance of one VM can be affected by the activities of other VMs on the same physical machine.

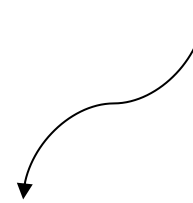
It's important to say that most of limitation can be solved thanks to correct management and implementation.

Approaches/Solutions



Virtual Machine

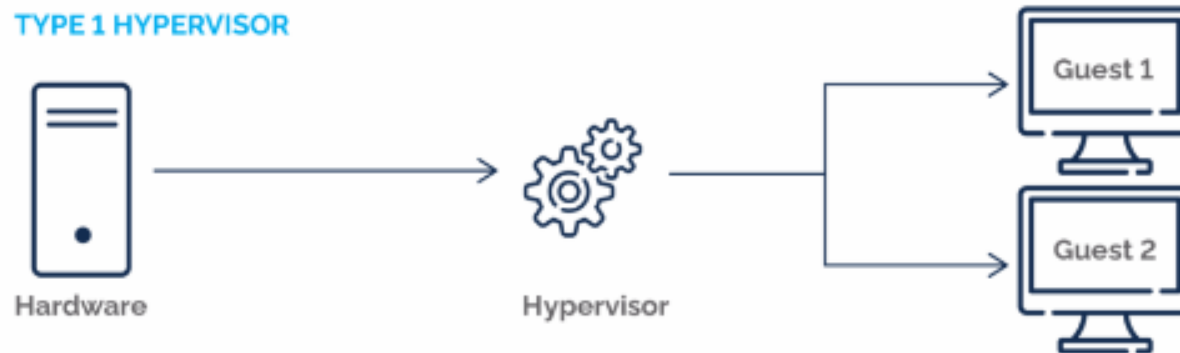
- **A Virtual Machine (VM)** is a compute resource that uses software instead of a physical computer to run programs and deploy apps.
- One or more virtual “guest” machines run on a physical “host” machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host.
- One of the most important components is called a **hypervisor**:
 - Type 1 (Bare-Metal);
 - Type 2 (Hosted).



<https://www.ibm.com/it-it/topics/hypervisors>

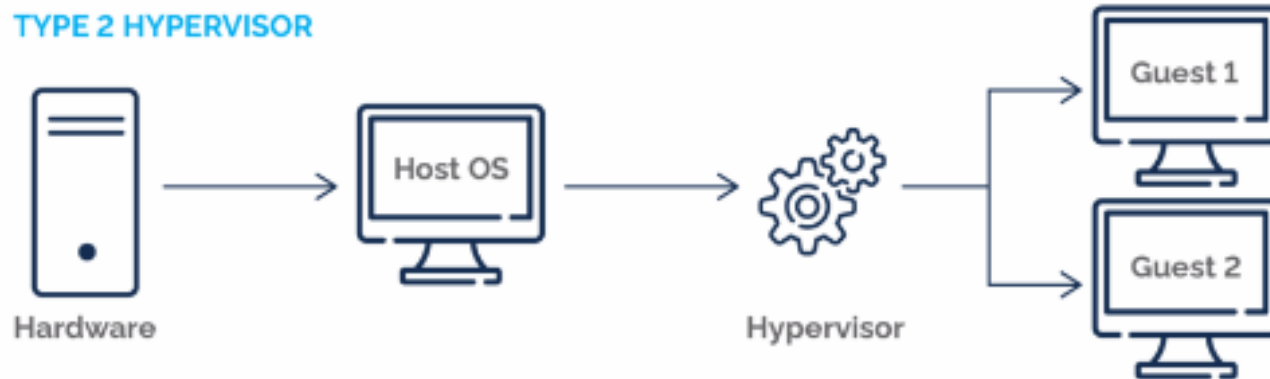
Virtual Machine

- **Type 1 (Bare-Metal):** runs directly on the underlying physical hardware of the computer, interacting directly with the CPU, memory, and physical storage. For this reason, Type 1 hypervisors are also referred to as bare-metal hypervisors. It replaces the host operating system.



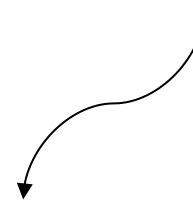
Virtual Machine

- **Type 2 (Hosted):** it doesn't run directly on the underlying hardware. Instead, it runs as an application within an operating system. They are more suitable for individual PC users who need to run multiple operating systems.



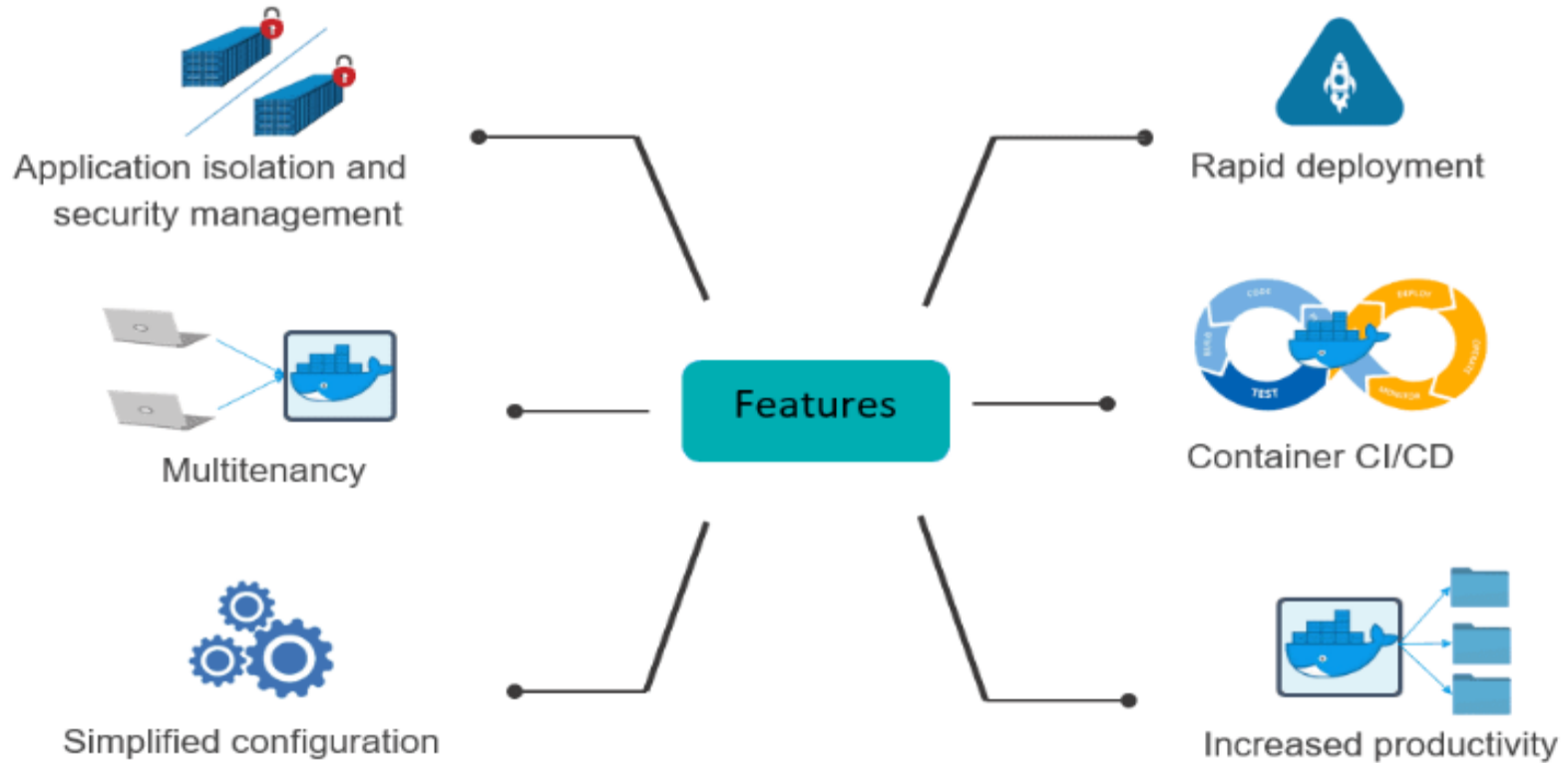
Containers

- Containers are a form of operating system virtualization.
- A single container might be used to run anything from a small microservice or software process to a larger application.
- Inside a container are all the necessary executables, binary code, libraries, and configuration files.
- Compared to server or machine virtualization approaches, however, containers do not contain operating system images. This makes them more lightweight and portable.



<https://www.redhat.com/en/topics/containers>

Containers



Containers

Let's see the main advantages of containers:

- **Lightweight:** Containers share the machine OS kernel and therefore they don't need a full OS instance per application. This makes the container files smaller.
- **Portable:** Containers are a package having all their dependencies with them, this means that we have to write the software once and the same software can be run across different laptops, without the need of configuring the whole software again.
- **Fast Startup:** Containers start applications more quickly than virtual machines since there is no need to emulate an entire operating system. Containers can be launched almost instantly.
- **Consistent Environment:** Due to their portability, containers ensure greater consistency between development and production environments, reducing issues related to differences between environments.

Benefits from Containers

- **Solving the «Works on my machine» problem:**
 - Without containers, an app may work during development but fail in production due to environment differences. Containers solve this by packaging all dependencies, ensuring consistency across different systems.
- **Isolated Environments:**
 - Containers allow applications with different dependencies to run on the same machine without conflicts. For example, multiple Python versions can coexist without issues, avoiding compatibility problems caused by system updates.

Benefits from Containers

➤ **Simplified Development:**

- Applications often rely on multiple services (e.g., databases, caching). Manually installing and managing them can be complex. With containers, each service can be started with a single command, making development easier.

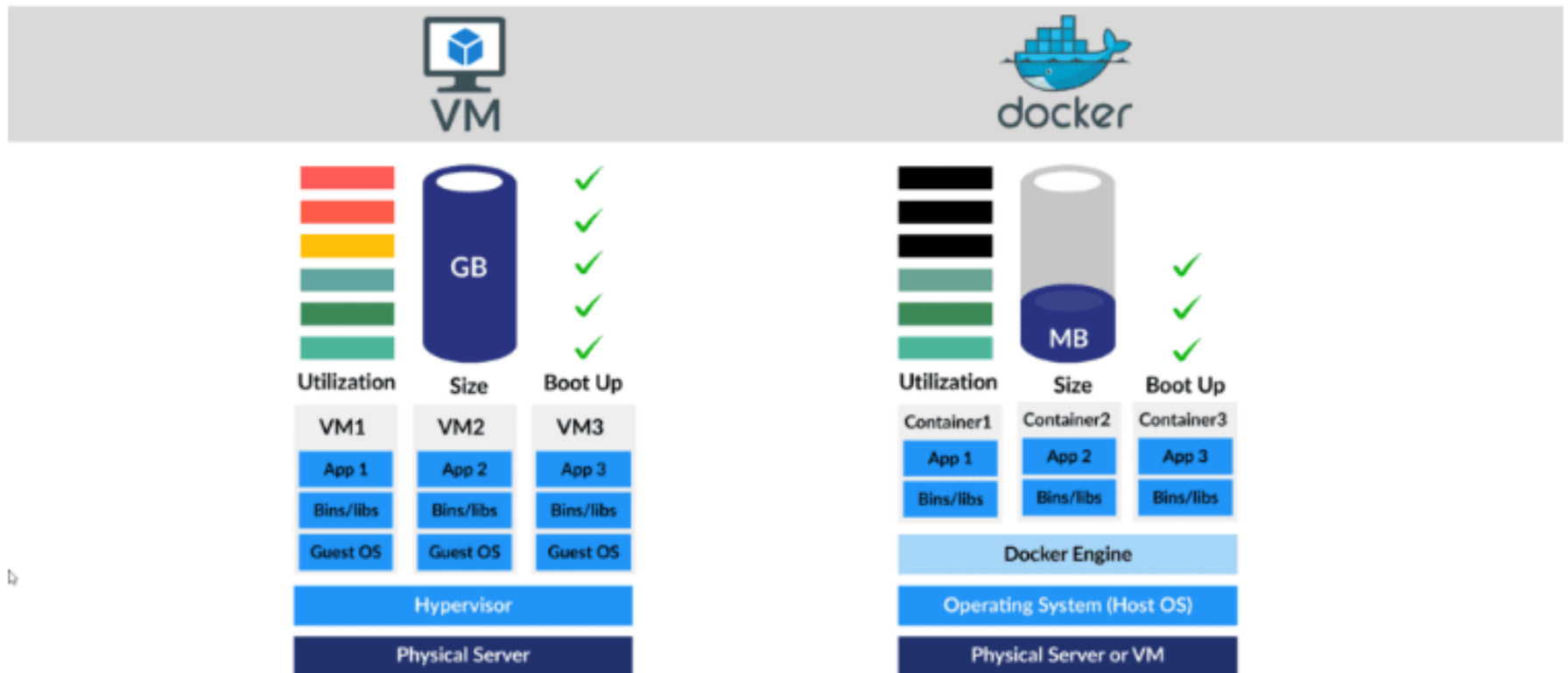
➤ **Scalability:**

- Containers can be started and stopped quickly with minimal resource overhead. Using orchestration tools, multiple instances can be managed efficiently, distributing traffic and ensuring service availability even in case of failures

VM vs Containers

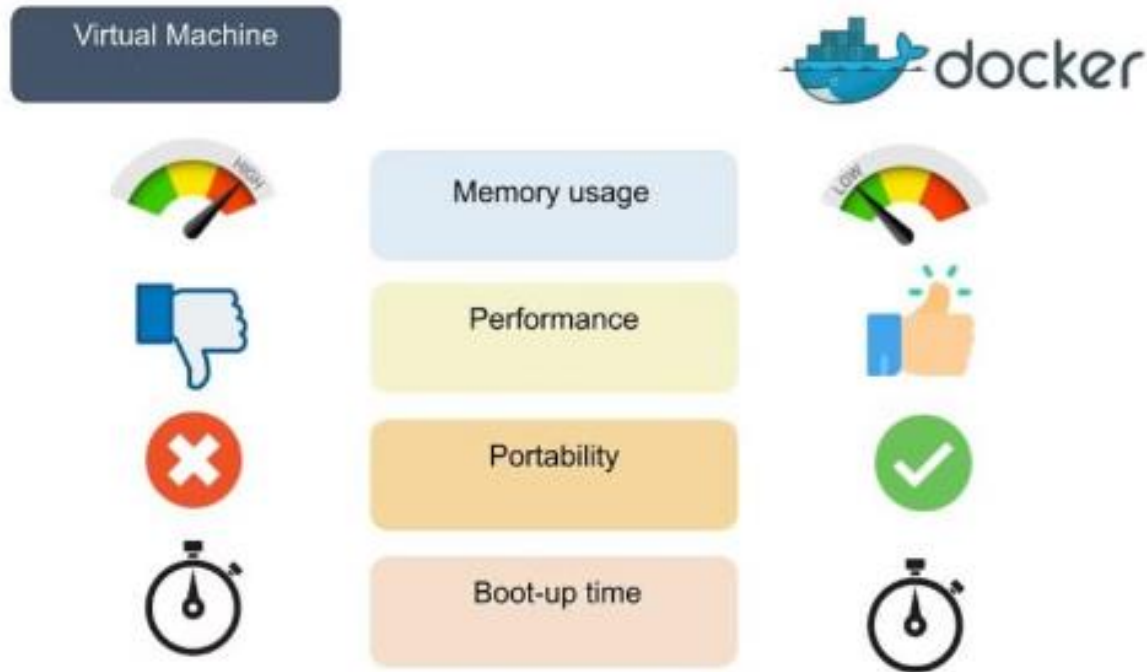
- Virtual machines access the hardware of a physical machine through a hypervisor.
 - The hypervisor creates an abstraction layer allowing the VM to access CPU, memory, and storage.
 - Containers, on the other hand, represent a package that includes an executable with the dependencies it needs to run.
- Virtual machines are typically more resource-intensive than containers.
 - However, virtual machines also provide a high level of isolation, which can be important for security reasons.
 - Containers are more lightweight and portable than virtual machines. This makes them a good choice for applications that need to be deployed quickly and easily, where compute must be optimized.

VM vs Containers



VM vs Containers

Major differences are:



VM vs Containers

	Container	Virtual Machines (VMs)
Boot-Time	Boots in a few seconds.	It takes a few minutes for VMs to boot.
Runs on	Dockers make use of the execution engine.	VMs make use of the hypervisor.
Memory Efficiency	No space is needed to virtualize, hence less memory.	Requires entire OS to be loaded before starting the surface, so less efficient.
Isolation	Prone to adversities as no provisions for isolation systems.	Interference possibility is minimum because of the efficient isolation mechanism.
Deployment	Deploying is easy as only a single image, containerized can be used across all platforms.	Deployment is comparatively lengthy as separate instances are responsible for execution.
Performance	Limited performance	Native performance



What is Docker?

- **Docker** is an open platform for developing, shipping, and running applications.
- Docker enables you to separate your applications from your infrastructure to deliver software quickly.
- Docker offers the capability to encapsulate and execute an application within a flexibly isolated space known as a **container**. This isolation and security feature enables the concurrent execution of numerous containers on a designated host.



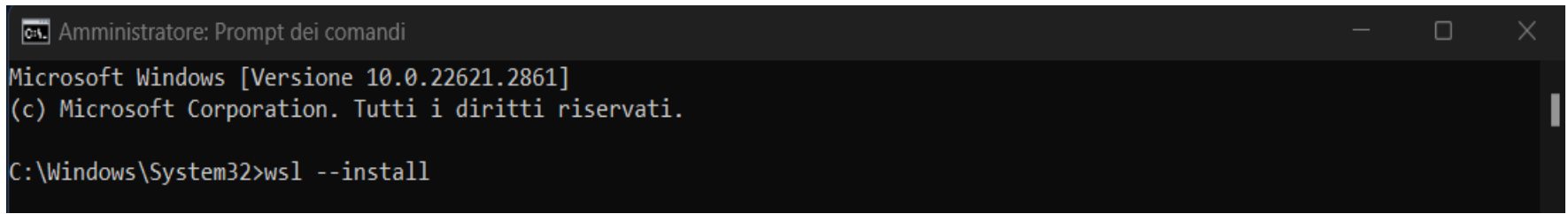
How to install Docker Desktop?

➤ First, we need a WSL (Windows Subsystem for Linux) and this requirements.

- WSL version 1.1.3.0 or later.
- Windows 11 64-bit: Home or Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.
- Windows 10 64-bit:
 - We recommend Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher.
 - Minimum required is Home or Pro 21H2 (build 19044) or higher, or Enterprise or Education 21H2 (build 19044) or higher.
- Turn on the WSL 2 feature on Windows. For detailed instructions, refer to the [Microsoft documentation](#) .
- The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11:
 - 64-bit processor with [Second Level Address Translation \(SLAT\)](#) .
 - 4GB system RAM
 - Enable hardware virtualization in BIOS. For more information, see [Virtualization](#).

How to install Docker Desktop?

- First, we need a WSL (Windows Subsystem for Linux) and this requirements.



```
Amministratore: Prompt dei comandi
Microsoft Windows [Versione 10.0.22621.2861]
(c) Microsoft Corporation. Tutti i diritti riservati.

C:\Windows\System32>wsl --install
```

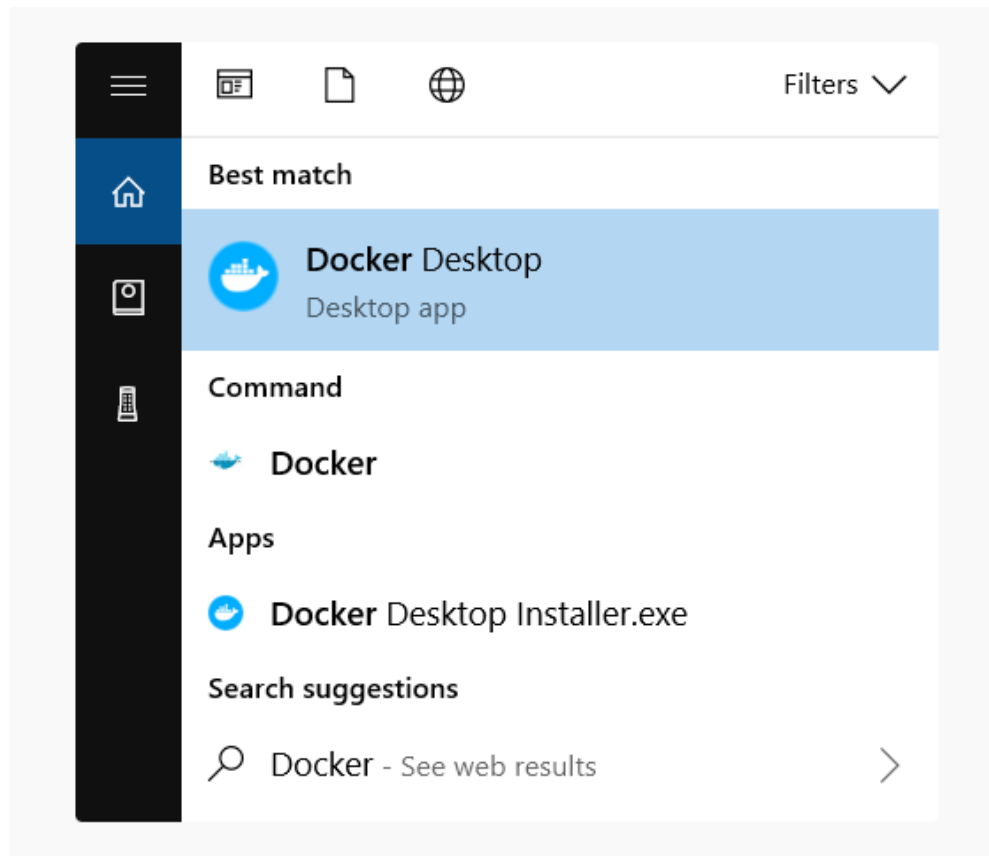
How to install Docker Desktop?

- After, click on this link <https://docs.docker.com/desktop/install/windows-install/>, and install Docker Desktop, following the instruction.

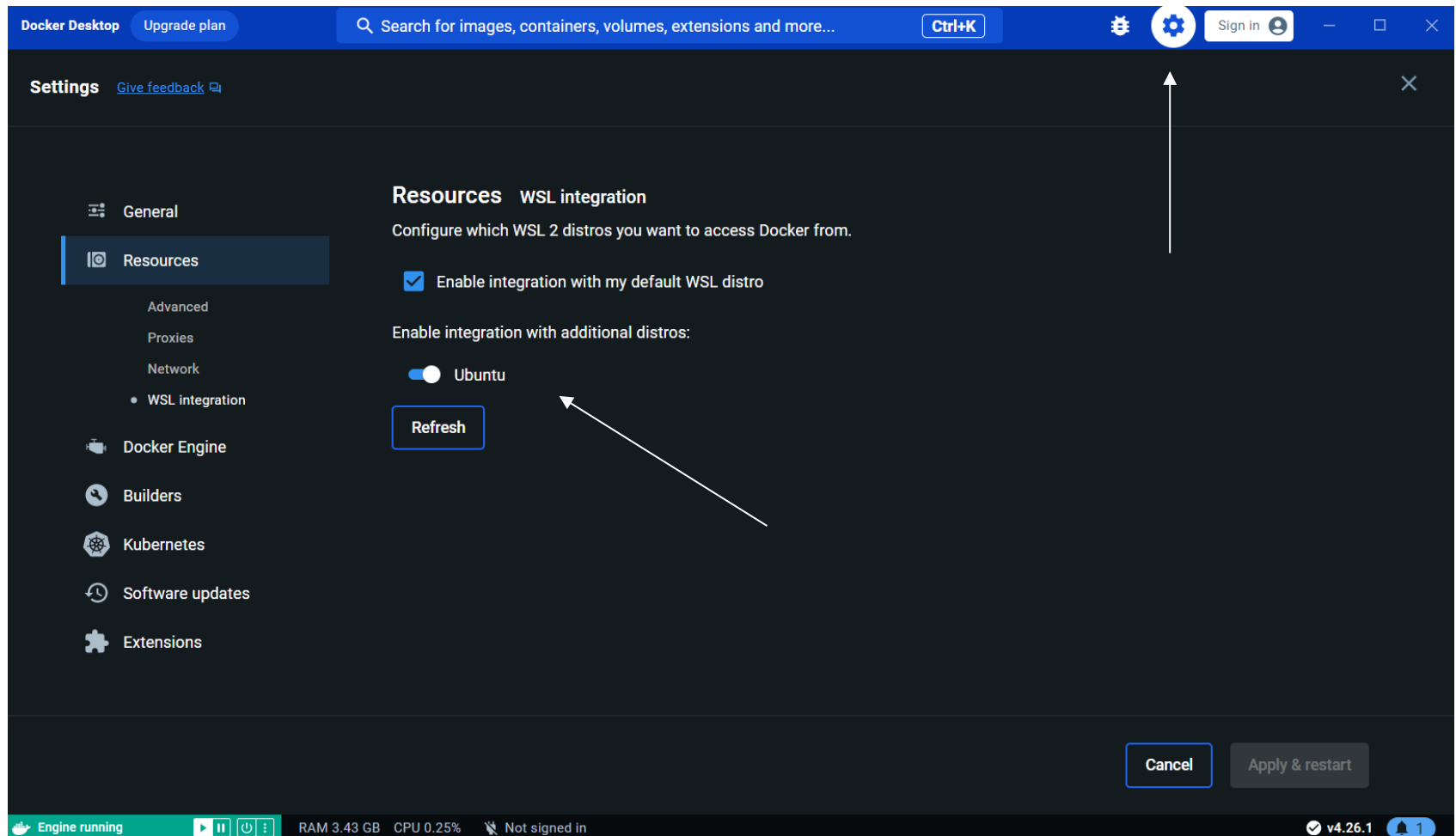
1. Double-click **Docker Desktop Installer.exe** to run the installer.
2. When prompted, ensure the **Use WSL 2 instead of Hyper-V** option on the Configuration page is selected or not depending on your choice of backend.
If your system only supports one of the two options, you will not be able to select which backend to use.
3. Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
4. When the installation is successful, select **Close** to complete the installation process.
5. If your admin account is different to your user account, you must add the user to the **docker-users** group. Run **Computer Management** as an **administrator** and navigate to **Local Users and Groups > Groups > docker-users**. Right-click to add the user to the group. Sign out and sign back in for the changes to take effect.

How to install Docker Desktop?

- Docker Desktop does not start automatically after installation. To start Docker Desktop, search for Docker and select Docker Desktop:

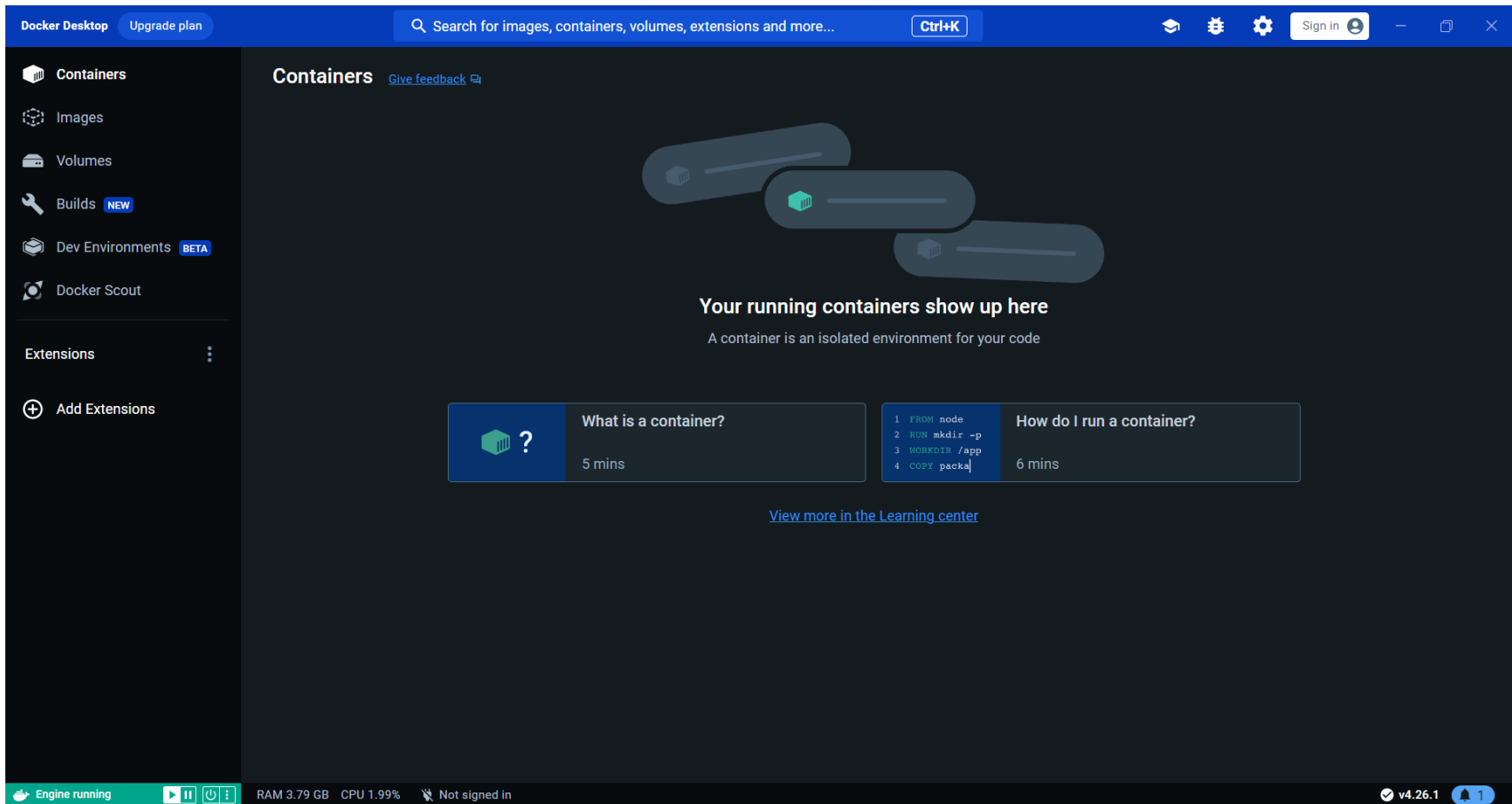


How to install Docker Desktop?



Docker Desktop

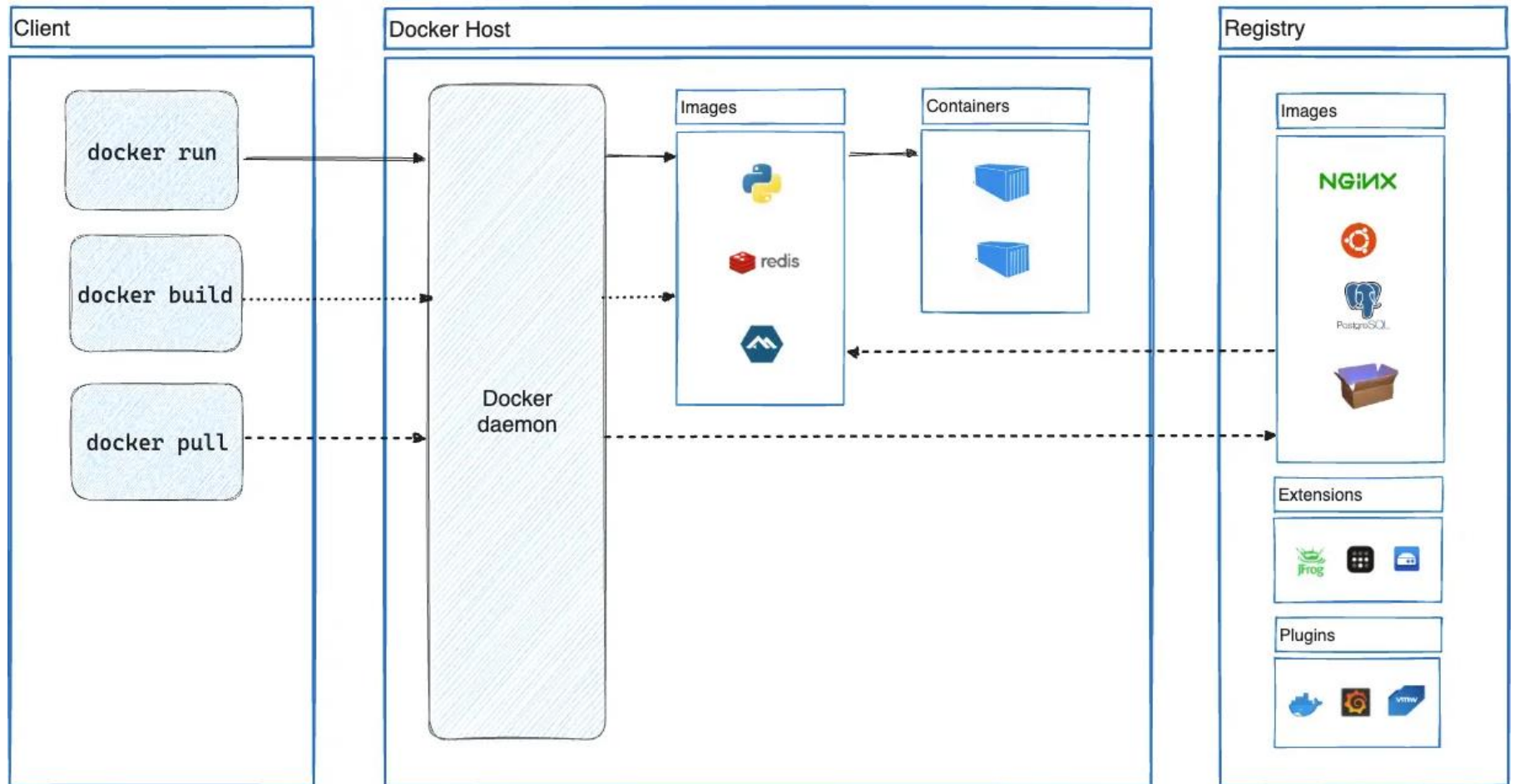
➤ If you open Docker Desktop, it appears like:



Docker Desktop

- The **Containers** view provides a runtime view of all your containers and applications. This view also provides an intuitive interface to perform common actions to inspect, interact with, and manage your containers.
- The **Images** view displays a list of your Docker images. It contains clean-up options to remove unwanted images from the disk to reclaim space.
- The **Volumes** view displays a list of volumes and allows you to easily create and delete volumes and see which ones are being used.

Docker



<https://docs.docker.com/get-started/overview/#docker-architecture>

Docker

- As seen in previous slide, Docker uses a client-server architecture. Let's see how it works.
- Communication between Client and Daemon:
 - The **Docker client** is the user interface through which users interact with Docker. It can send commands to the Docker daemon to perform actions such as creating or running containers.
 - The **Docker daemon** is the primary process that handles containers on the operating system. It takes care of tasks like creating, running, and distributing Docker containers.

The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and Daemon, communicate using a REST API.

Docker

➤ **Docker Pull** is used for downloading a Docker image from a registry. How?

- Docker client send a request to the Docker Daemon;
- The Docker Daemon checks if the image is present locally.
- If the image is not present, the Daemon downloads it from the specified Docker registry (e.g., Docker Hub) and stores it locally.
- If the image is already present, Docker checks if a newer version exists in the registry and, if necessary, downloads and updates it locally.

A terminal window with a dark background. The title bar shows 'giammino@Giammino: ~' and window control buttons. The prompt is 'giammino@Giammino:~\$' and the command entered is 'docker pull image_name'.

```
giammino@Giammino:~$ docker pull image_name
```

Docker

➤ **Docker Pull** is used for downloading a Docker image from a registry. How?

- Docker client send a request to the Docker Daemon;
- The Docker Daemon checks if the image is present locally.
- If the image is not present, the Daemon downloads it from the specified Docker registry (e.g., Docker Hub) and stores it locally.
- If the image is already present, Docker checks if a newer version exists in the registry and, if necessary, downloads and updates it locally.

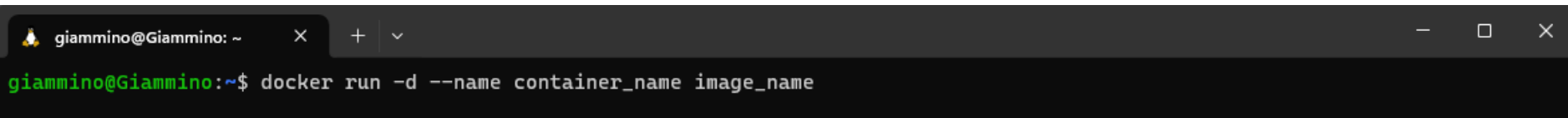
```
giammino@Giammino: ~  
giammino@Giammino:~$ docker pull ubuntu  
Using default tag: latest  
latest: Pulling from library/ubuntu  
a48641193673: Pull complete  
Digest: sha256:6042500cf4b44023ea1894effe7890666b0c5c7871ed83a97c36c76ae560bb9b  
Status: Downloaded newer image for ubuntu:latest  
docker.io/library/ubuntu:latest
```

Docker

- **Docker Build** is used for create a Docker image from a Dockerfile.
 - The user executes 'docker build' command in the directory containing the Dockerfile.
 - The Daemon reads the Dockerfile and follows the instructions to create an image.
 - The resulting image includes the application code, dependencies, and configurations specified in the Dockerfile.
 - The image is tagged with the specified name and optionally a version.

Docker

- **Docker Run** is used for running a container based on crated image.
- The user runs the 'docker run' command specifying the image to use.
 - The Docker Daemon creates an isolated instance of the container based on the specified image.
 - The container runs in an isolated state, starting the main process defined in the image.

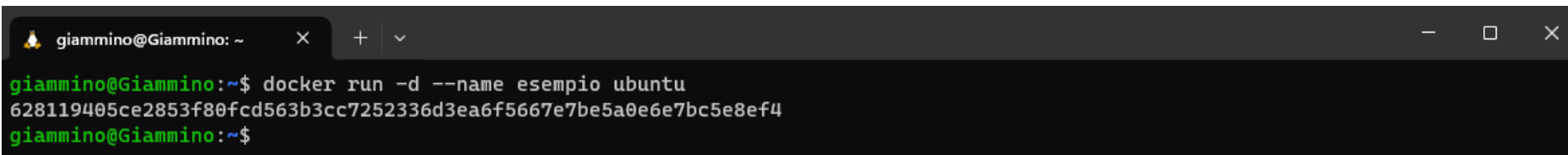
A terminal window with a dark background. The title bar shows 'giammino@Giammino: ~' and standard window controls. The command prompt is 'giammino@Giammino:~\$' and the command entered is 'docker run -d --name container_name image_name'.

```
giammino@Giammino:~$ docker run -d --name container_name image_name
```

Using `-d`, the container runs in the background

Docker

- **Docker Run** is used for running a container based on crated image.
- The user runs the 'docker run' command specifying the image to use.
 - The Docker Daemon creates an isolated instance of the container based on the specified image.
 - The container runs in an isolated state, starting the main process defined in the image.

A terminal window with a dark background. The title bar shows 'giammino@Giammino: ~' and window control buttons. The terminal text shows a user running 'docker run -d --name esempio ubuntu' and receiving a long alphanumeric ID as output.

```
giammino@Giammino: ~  
giammino@Giammino:~$ docker run -d --name esempio ubuntu  
628119405ce2853f80fcd563b3cc7252336d3ea6f5667e7be5a0e6e7bc5e8ef4  
giammino@Giammino:~$
```

Docker Components

Docker consists of several components that work together to enable the creation, distribution, and execution of applications in containers. The main components are:

- **Docker Engine:** is the core component of Docker, responsible for the execution and management of containers. It is the heart of the Docker system and performs various key functions to enable the creation, execution, and management of containers.
- **Docker CLI (Client Line Interface):** it is the command-line interface that allows users to interact with Docker.
- **Docker Hub:** It is the official repository of Docker images. Developers can upload their images to Docker Hub for sharing and reuse by other users

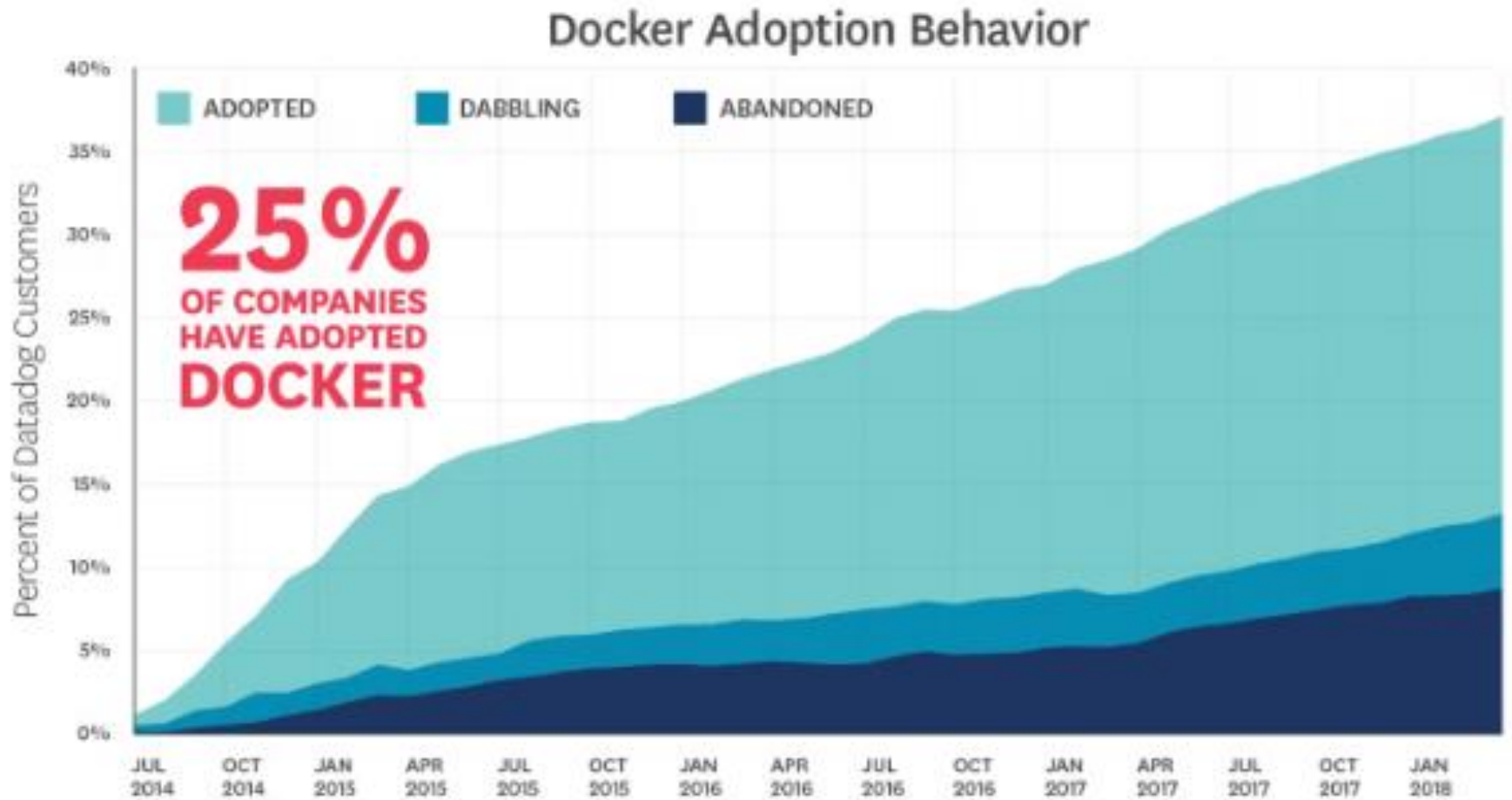
Docker Components

- **Docker Image:** images are used as a base for creating containers, and contains code, libraries, dependencies and configurations.
- **Docker Container:** it is an executable instance of a Docker image. A container is isolated from other containers and the host system, ensuring portability
- **Dockerfile:** it is a configuration file that contains the instructions needed to create a Docker image.
- **Docker Network:** Docker provides a networking system that allows containers to communicate with each other. You can define custom networks to isolate containers or connect containers to existing networks.
- **Docker Volume:** volumes provide a way to store data persistently outside the lifecycle of the container. In other words, data in volumes is not deleted when the container is stopped or removed.

Why Docker?



Why Docker?



Not only Docker



Let's Practice

- Let's practice with Docker and containers. To do this, let's start with a simple example where the output on the terminal displays the message "Hello from Docker!"
- Additionally, it is explained how that message was generated.

```
giammino@Giammino: ~  
giammino@Giammino:~$ docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
c1ec31eb5944: Pull complete  
Digest: sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493  
Status: Downloaded newer image for hello-world:latest  
  
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/get-started/  
giammino@Giammino:~$
```

Let's Practice

- If you want to list the image, you can do this:

```
giammino@Giammino:~$ docker images
REPOSITORY    TAG       IMAGE ID      CREATED        SIZE
hello-world    latest    d2c94e258dcb  7 months ago  13.3kB
giammino@Giammino:~$
```

- As said before, we can build an image using a Dockerfile, that looks something like this:

```
FROM <image>:<tag>

RUN <install some dependencies>

CMD <command that is executed on `docker container run`>
```

Let's Practice

- With this commands, it's possible to see the running containers:

```
giammino@Giammino:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
33ceb4c9c4b1	hello-world	"/hello"	10 minutes ago	Exited (0)		youthful_galois

- If you want to remove the container and image:

```
giammino@Giammino: ~
```

```
giammino@Giammino:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
33ceb4c9c4b1	hello-world	"/hello"	13 minutes ago	Exited (0)		youthful_galois

```
giammino@Giammino:~$ docker rm 33ce
33ce
giammino@Giammino:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

```
giammino@Giammino:~$ docker rmi hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493
Deleted: sha256:d2c94e258dcb3c5ac2798d32e1249e42ef01cba4841c2234249495f87264ac5a
Deleted: sha256:ac2880ec8bb38d5c35b49d45a6ac4777544941199075dff8c4eb63e093aa81e
giammino@Giammino:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
giammino@Giammino:~$
```

Most used commands

command	explain	shorthand
<code>docker image ls</code>	Lists all images	<code>docker images</code>
<code>docker image rm <image></code>	Removes an image	<code>docker rmi</code>
<code>docker image pull <image></code>	Pulls image from a docker registry	<code>docker pull</code>
<code>docker container ls -a</code>	Lists all containers	<code>docker ps -a</code>
<code>docker container run <image></code>	Runs a container from an image	<code>docker run</code>
<code>docker container rm <container></code>	Removes a container	<code>docker rm</code>
<code>docker container stop <container></code>	Stops a container	<code>docker stop</code>
<code>docker container exec <container></code>	Executes a command inside the container	<code>docker exec</code>

Exercises

- **Exercise 1.1:** Start three containers. Stop two containers and leave one container running. Submit the output `docker ps -a` which shows 2 stopped containers and one running.
- **Exercise 1.2:** Now we have containers and images that are no longer in use. Running the correct commands will confirm this. So, clean docker by removing all images and containers.

- Now we can do something more. Let's use an Ubuntu image.

```
root@dad4b2e1092f: /  
giammino@Giammino:~$ docker run -it ubuntu  
Unable to find image 'ubuntu:latest' locally  
latest: Pulling from library/ubuntu  
a48641193673: Pull complete  
Digest: sha256:6042500cf4b44023ea1894effe7890666b0c5c7871ed83a97c36c76ae560bb9b  
Status: Downloaded newer image for ubuntu:latest  
root@dad4b2e1092f:/# ls  
bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var  
root@dad4b2e1092f:/#
```

–it, specifies the interactive option, allowing user interaction with the container through the terminal.

You can also specify the version of ubuntu with a tag, e.g. 'docker run –it ubuntu:version'

Build an image

- As we know, images are the basic building blocks for containers and other images. By learning what images are and how to create them, you are ready to start using containers.
- When running a command such as `docker run hello-world`, docker will automatically search for the image.
- We can search for images in the Docker Hub with `docker search`. For example, you can search the image `hello-world` by using the following command: `docker search hello-world`.

```
$ docker search hello-world
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
hello-world	Hello World!...	1988	[OK]	
kitematic/hello-world-nginx	A light-weig...	153		
tutum/hello-world	Image to tes...	90		[OK]
...				

Build an image

- We can build an image using a Dockerfile.
- Dockerfile is simply a file containing an image's build instructions. You define what should be included in the image with different instructions.
- Let's take a most simple application and containerize it first. Create a script called `hello.sh`

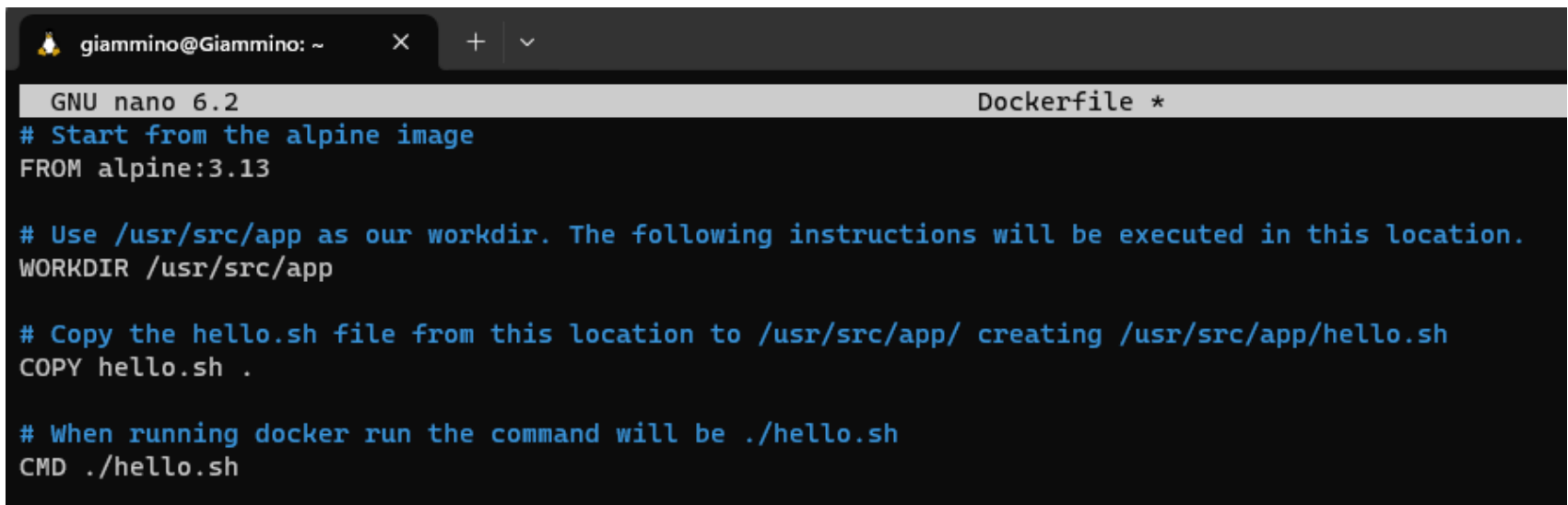
hello.sh

```
#!/bin/sh  
  
echo "Hello, docker!"
```

- And now let's create an image from it.

Build an image

- We have to create a Dockerfile that declares all of the required dependencies. At least it depends on something that can run shell scripts. We will choose Alpine, a small Linux distribution that is often used to create small images.



```
giammino@Giammino: ~  
GNU nano 6.2 Dockerfile *  
# Start from the alpine image  
FROM alpine:3.13  
  
# Use /usr/src/app as our workdir. The following instructions will be executed in this location.  
WORKDIR /usr/src/app  
  
# Copy the hello.sh file from this location to /usr/src/app/ creating /usr/src/app/hello.sh  
COPY hello.sh .  
  
# When running docker run the command will be ./hello.sh  
CMD ./hello.sh
```

Build an image

- By default, if we use build, docker will look a file named Dockerfile. So, we can build our image as:

```
giammino@Giammino: ~  
giammino@Giammino:~$ docker build . -t test  
[+] Building 2.8s (8/8) FINISHED  
=> [internal] load .dockerignore  
=> => transferring context: 2B  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 388B  
=> [internal] load metadata for docker.io/library/alpine:3.13  
=> [internal] load build context  
=> => transferring context: 67B  
=> [1/3] FROM docker.io/library/alpine:3.13@sha256:469b6e04ee185740477efa44ed5bdd64a07bbdd6c7e5f5d169e540889597b911  
=> CACHED [2/3] WORKDIR /usr/src/app  
=> CACHED [3/3] COPY hello.sh .  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:c35d7a69d8a3c71cc1c93fff8b283c2367e1dbeaa372e29a0186c1e0e9c00cc2  
=> => naming to docker.io/library/test  
  
What's Next?  
View a summary of image vulnerabilities and recommendations → docker scout quickview  
giammino@Giammino:~$ docker images  
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE  
test          latest   c35d7a69d8a3   5 minutes ago  5.62MB  
giammino@Giammino:~$
```

Build an image

- By default, if we use build, docker will look a file named Dockerfile. So, we can build our image as:
- After that, we can run the container from image

```
giammino@Giammino: ~  
giammino@Giammino:~$ docker run test  
Hello docker!  
giammino@Giammino:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
59a939eb4e8c	test	"/bin/sh -c ./hello...."	9 seconds ago	Exited (0) 8 seconds ago		jovial_bhaskara

```
giammino@Giammino:~$
```

Exercise

- It is also possible to manually create new layers on top of an image.
- Create a new file called `additional.txt` and copy it inside a container.

Exercise

- It is also possible to manually create new layers on top of an image.
- Create a new file called additional.txt and copy it inside a container.

```
# do this in terminal 1
$ docker run -it hello-docker sh
/usr/src/app #
```

```
# do this in terminal 2
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
9c06b95e3e85	hello-docker	"sh"	4 minutes ago	Up 4 minutes		zen_rosalind

```
$ touch additional.txt
$ docker cp ./additional.txt zen_rosalind:/usr/src/app/
```

Exercise

➤ If you want to save the changes:

```
# do this in terminal 2
$ docker commit zen_rosalind hello-docker-additional
sha256:2f63baa355ce5976bf89fe6000b92717f25dd91172aed716208e784315bfc4fd
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-docker-additional	latest	2f63baa355ce	3 seconds ago	7.73MB
hello-docker	latest	444f21cf7bd5	31 minutes ago	7.73MB

What is a **Dockerfile**?

- Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
- This file specifies the necessary steps to configure an environment inside a container, defining which packages to install, which files to copy, and how to perform specific actions within the container.

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

What is a Dockerfile?

- In the previous example, each instruction creates one layer:
 - **FROM**: creates a layer from the ubuntu:22.04 Docker image.
 - **COPY**: adds files from your Docker client's current directory.
 - **RUN**: build your application with make.
 - **CMD**: specifies what command to run within the containers.

A Dockerfile must begin with a **FROM** instruction.

What is a Dockerfile?

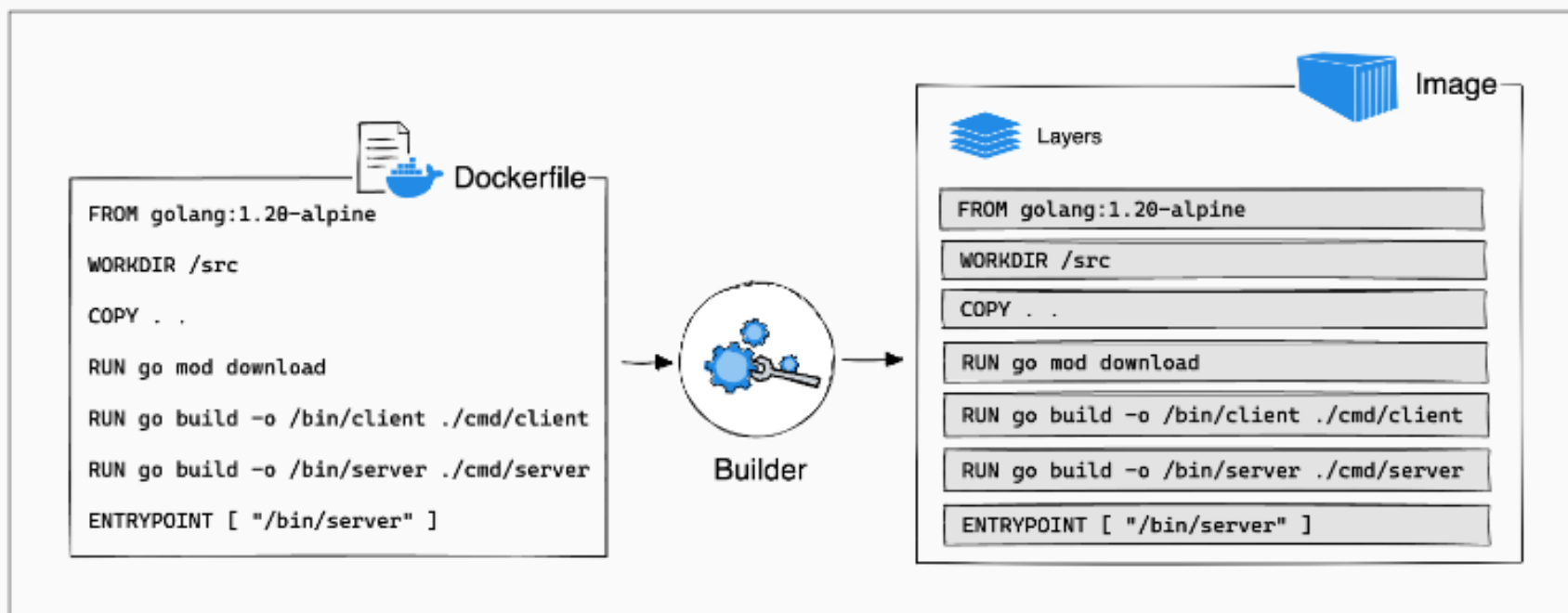
➤ The Dockerfile supports the following instructions:

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.
MAINTAINER	Specify the author of an image.
ONBUILD	Specify instructions for when the image is used in a build.
RUN	Execute build commands.
SHELL	Set the default shell of an image.
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts.
WORKDIR	Change working directory.

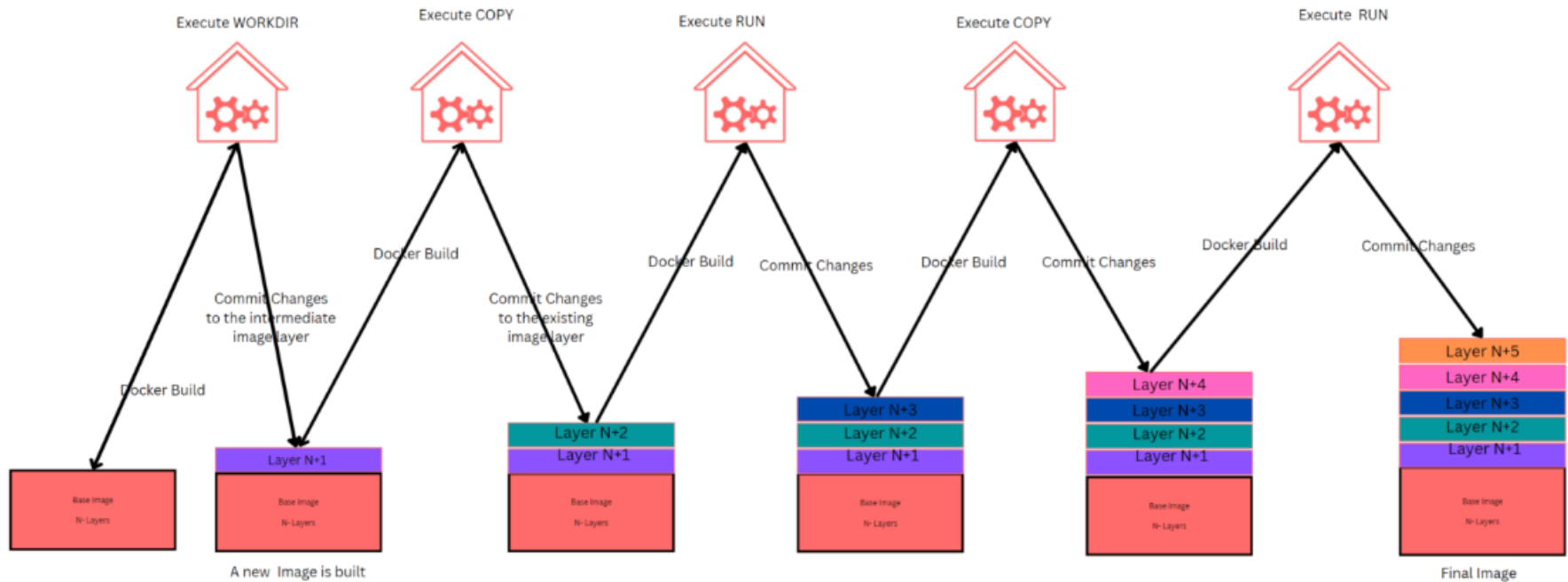
<https://docs.docker.com/engine/reference/builder/>

Layers

- The order of Dockerfile instructions matters. A Docker build consists of a series of ordered build instructions.
- Each instructions in a Dockerfile translates to an image layer.



Layers



Docker Image

- A Docker image is composed of multiple layers stacked on top of each other.
- Each layer represents a specific modification, such as adding a new file or modifying an existing one.
- Once a layer is created, it becomes immutable. It can't be changed.
- The various parts or layers of a Docker image are stored in the Docker engine's cache. This caching system contributes to making the process of creating Docker images more efficient.
- To understand better, let's take the previous example.

Docker Image

- The time needed to create the image is 2.8s.

```
giammino@Giammino: ~  
[+] Building 2.8s (8/8) FINISHED  
=> [internal] load .dockerignore  
=> => transferring context: 2B  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 388B  
=> [internal] load metadata for docker.io/library/alpine:3.13  
=> [internal] load build context  
=> => transferring context: 67B  
=> [1/3] FROM docker.io/library/alpine:3.13@sha256:469b6e04ee185740477efa44ed5bdd64a07bbdd6c7e5f5d169e540889597b911  
=> CACHED [2/3] WORKDIR /usr/src/app  
=> CACHED [3/3] COPY hello.sh .  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:c35d7a69d8a3c71cc1c93fff8b283c2367e1dbeaa372e29a0186c1e0e9c00cc2  
=> => naming to docker.io/library/test  
  
What's Next?  
View a summary of image vulnerabilities and recommendations → docker scout quickview  
giammino@Giammino:~$ docker images  
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE  
test          latest   c35d7a69d8a3   5 minutes ago  5.62MB  
giammino@Giammino:~$
```

Docker Image

- If we repeat the same operation, changing only the name of the image we see that the time decreases. Why? Thanks to Docker cache

```
giammino@Giammino:~$ docker build . -t test:v2
[+] Building 0.8s (8/8) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 388B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/alpine:3.13 0.6s
=> [1/3] FROM docker.io/library/alpine:3.13@sha256:469b6e04ee185740477efa44ed5bdd64a07bbdd6c7e5f5d169e540889597b 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 29B 0.0s
=> CACHED [2/3] WORKDIR /usr/src/app 0.0s
=> CACHED [3/3] COPY hello.sh . 0.0s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:c35d7a69d8a3c71cc1c93fff8b283c2367e1dbeaa372e29a0186c1e0e9c00cc2 0.0s
=> => naming to docker.io/library/test:v2 0.0s
```

What's Next?

View a summary of image vulnerabilities and recommendations → `docker scout quickview`
giammino@Giammino:~\$

Docker Image

- If we want to see the presence of layers within our image:

```
giammino@Giammino:~$ docker history test
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
c35d7a69d8a3	17 hours ago	CMD ["/bin/sh" "-c" "./hello.sh"]	0B	buildkit.dockerfile.v0
<missing>	17 hours ago	COPY hello.sh . # buildkit	32B	buildkit.dockerfile.v0
<missing>	17 hours ago	WORKDIR /usr/src/app	0B	buildkit.dockerfile.v0
<missing>	16 months ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B	
<missing>	16 months ago	/bin/sh -c #(nop) ADD file:7fd90c097e2c4587d...	5.62MB	

Why do some layers have a size of 0? Some instructions, may not have a direct impact on the image's file system.

Docker Volumes

- Volumes in Docker are used to retain data even when a container is stopped or removed. Unlike a container's internal filesystem, volumes persist and can be shared between multiple containers.
- It is possible to create a link between a directory or file on the **host machine** and a directory inside a **Docker container**
 - Unlike Docker-managed volumes, bind mounts provide more **flexibility** but also require careful handling because they directly interact with the host filesystem.

Feature	Docker Volumes	Bind Mounts
Management	Managed by Docker (<code>/var/lib/docker/volumes/</code>)	Uses existing host directories
Portability	Works across different environments	Tightly coupled with host system paths
Security	Isolated from the host system	Can modify host files directly
Performance	Optimized for container storage	Depends on host filesystem

Docker Volumes

➤ Some useful commands:

Command	Description
<code>docker volume create</code>	Create a volume
<code>docker volume inspect</code>	Display detailed information on one or more volumes
<code>docker volume ls</code>	List volumes
<code>docker volume prune</code>	Remove unused local volumes
<code>docker volume rm</code>	Remove one or more volumes
<code>docker volume update</code>	Update a volume (cluster volumes only)

Docker Volumes and..

- Docker containers often run services that need to be accessible from the outside, such as a web server or a database. Docker allows exposing and publishing ports to handle external access.
- Docker provides two different ways to handle ports:
 - **EXPOSE**: is used inside a Dockerfile to document that a container will use a specific port. However, it does not make the port accessible from the host.

```
1 FROM nginx
2 EXPOSE 80
```

Docker Volumes and..

- Docker containers often run services that need to be accessible from the outside, such as a web server or a database. Docker allows exposing and publishing ports to handle external access.
- Docker provides two different ways to handle ports:
 - **EXPOSE**: is used inside a Dockerfile to document that a container will use a specific port. However, it does not make the port accessible from the host.
 - **-p**: is used when running a container to map a port from the container to the host, making it accessible externally.

```
gmcric@Giammino:~$ docker run -d -p 8080:80 nginx
```

Let's Practice - 1

- **Esercizio 1 (Web Server Flask con Volumi):**
 - Crea un'applicazione Python che permetta di scrivere e leggere file di testo all'interno di un container Docker. Il contenuto scritto deve essere persistente solo se viene utilizzato un volume.
- Quindi, l'applicazione deve:
 - Consentire di scrivere un messaggio in un file di testo;
 - Leggere e mostrare il contenuto del file, se esiste;
- Avviare il container senza volume e osserva cosa succede dopo il riavvio. Poi avvia il container con un volume e verifica la differenza.

Let's Practice - 2

- **Esercizio 2 (Generatore di Log con Persistenza Usando Volumi):**
 - Crea un'applicazione Python che genera periodicamente dei log in un file di testo. L'applicazione deve essere eseguita in un container Docker e salvare i log in una directory specifica.
- Eseguire il container senza volume → I log vengono persi dopo la rimozione del container.
- Eseguire il container con un volume → I log rimangono disponibili anche dopo la rimozione e il riavvio del container.

Introduction to Docker

Giovanni Maria Cristiano

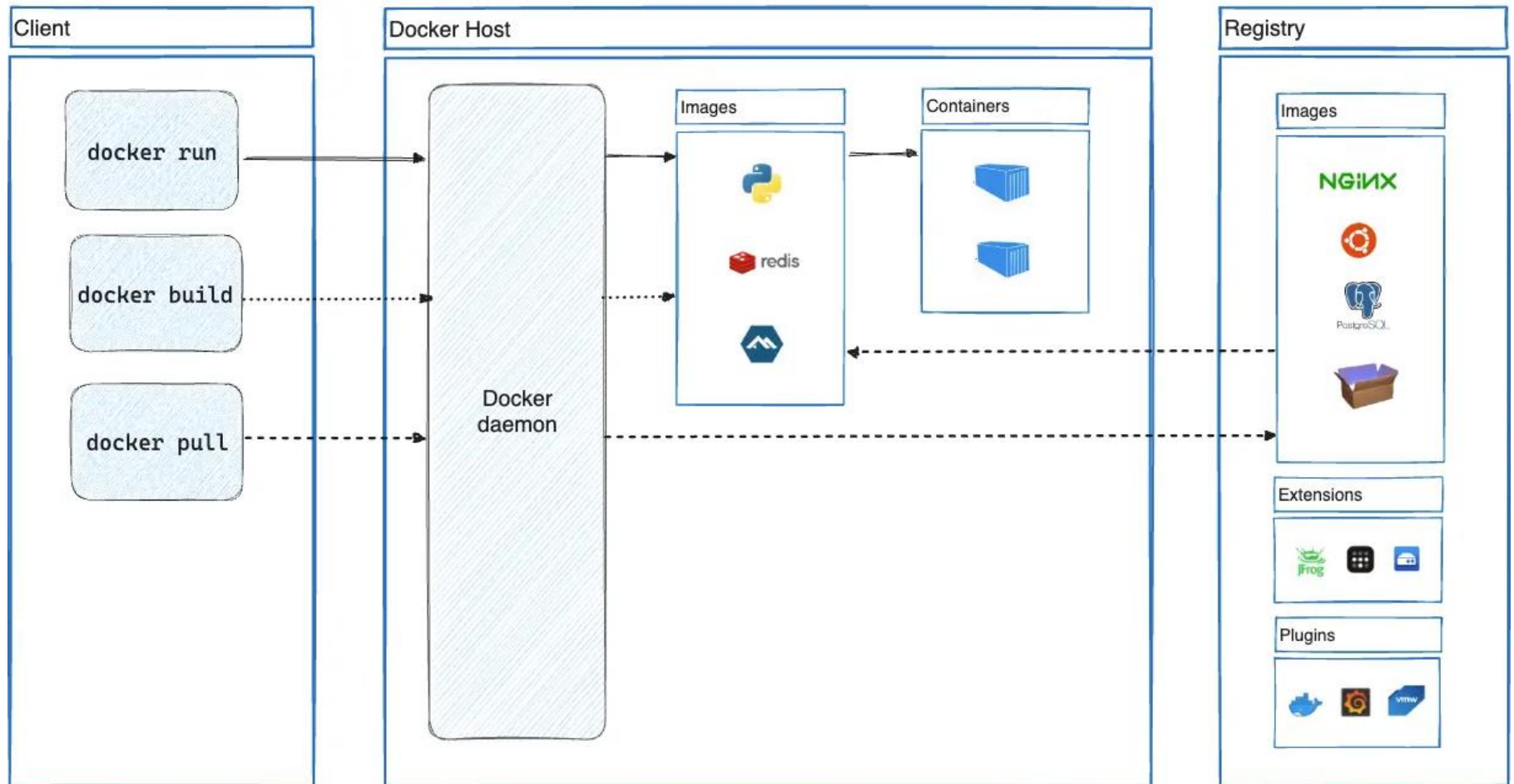
University of Naples «Parthenope»

giovannimaria.cristiano001@studenti.uniparthenope.it



Parte 2

Docker



<https://docs.docker.com/get-started/overview/#docker-architecture>

Roadmap

- **Docker Compose**
- **Docker Networking**
- **Exercises**

Docker Compose

- **Docker Compose** is a tool for defining and running applications that use multiple containers. It allows you to manage services, networks, and volumes using a **YAML** configuration file, making the process more structured and efficient.

gestire più container con un solo file di configurazione

- With a single command, you can start and manage all the services defined in the **configuration file**. **Docker Compose** is useful in various environments, including development, testing, staging, and production. It provides commands to handle the application's lifecycle, such as:

- Starting, stopping, and rebuilding services;
- Checking the status of running containers;
- Viewing real-time logs;
- Running one-time commands on a service.

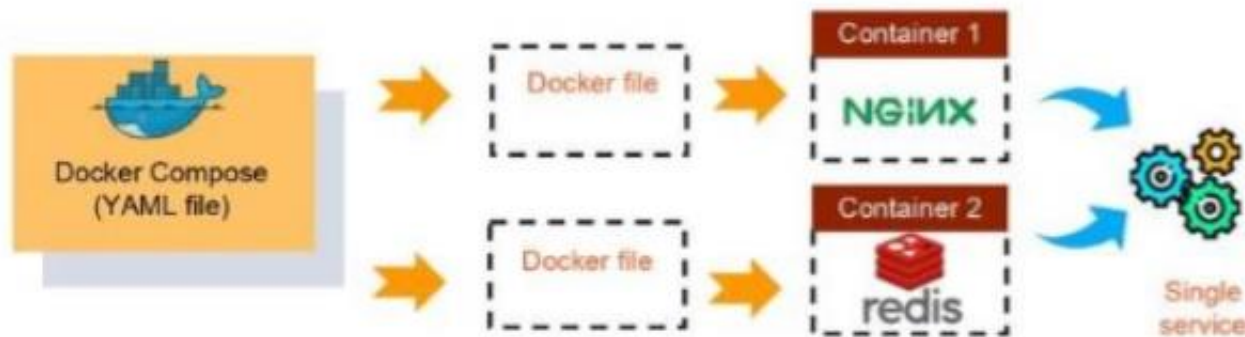


Why use Docker Compose?

- Using Docker Compose offers several benefits that streamline the development, deployment, and management of containerized applications:
 - Docker Compose simplifies managing multi-container applications by defining everything in a single YAML file. This makes it easier to coordinate services and replicate environments.
 - It also improves collaboration by allowing teams to share configuration files, leading to smoother workflows and quicker issue resolution.
 - Compose optimizes development by caching configurations, so unchanged services restart quickly without rebuilding containers.
 - With support for variables, it allows customization for different environments or users, making it more flexible.

Why use Docker Compose?

- For example:
- If you have an application that requires NGINX server and Redis database, you can create a Docker Compose file that can run both the containers as a service without the need to start each one separately.



all'interno del docker compose ci sono dei docker file per settare per esempio i server in un determinato modo

Basic Fields of Docker Compose

- **version**: specifies the Docker Compose file format version (e.g., "3.8")
- **services**: defines the application's services (containers).
 - In each service, you can use some other fields such as image, build, container name, ports, etc..
- **volumes**: defines named volumes that can be shared between containers.
- **networks**: specifies custom networks for communication between containers.

Flow

- Create a Docker Compose file:
 - Create a new file named **'docker-compose.yml'** in your project directory.
- Define Services:
 - Inside **'docker-compose.yml'**, define the services required for your application.
- Configure Options:
 - Add configuration options such as volumes, networks, environment variables, etc., based on your project's needs.
- Run Services:
 - Use the ***docker compose up*** commands to start the services defined in the file.

`docker - compose up`

Example of Docker Compose

```
1  version: '3.3'
2
3  services:
4    db:
5      image: mysql:5.7
6      volumes:
7        - db_data:C:\Users\User\Desktop\dcompose
8      restart: always
9      environment:
10        MYSQL_ROOT_PASSWORD: rootwordpress
11        MYSQL_DATABASE: wordpress
12        MYSQL_USER: wordpress
13        MYSQL_PASSWORD: wordpress
14
15    wordpress:
16      depends_on:
17        - db
18      image: wordpress:latest
19      ports:
20        - "8000:80"
21      restart: always
22      environment:
23        WORDPRESS_DB_HOST: db:3306
24        WORDPRESS_DB_USER: wordpress
25        WORDPRESS_DB_PASSWORD: wordpress
26  volumes:
27    db_data:
```


Docker Compose Commands

```
docker compose version  
docker compose config
```

```
docker compose start  
docker compose stop  
docker compose restart  
docker compose run
```

```
docker compose create  
docker compose attach  
docker compose pause  
docker compose unpause
```

```
docker compose wait  
docker compose up  
docker compose down
```

```
docker compose ps  
docker compose top  
docker compose events  
docker compose logs
```

```
docker compose images  
docker compose build  
docker compose push  
docker compose cp  
docker compose exec
```

Docker Cheatsheet

Docker base controls

List of containers started

`docker ps`

List of all containers

`docker ps -a`

Recovers the docker configuration

`docker info`

Recover docker version

`docker version`

Builds a Docker image

from a "Dockerfile" or a container.

`docker build -t <Image>: <Tag>`

Connects to a remote repository

`docker login < repository>`

Push image to remote repository

`docker push < Image_Name>: <Tag>`

Extracts the image from

the remote repository

`docker pull < Image_Name >`

Network commands

List of networks

`docker network ls`

Control network information

`docker network inspector < Network >`

Create a network

`docker network create < Network >`

Removes a network

`docker network rm < Network >`

Connect a container to the network

`docker network connect`

`< Network > < Container >`

Specifies the IP address of the container interface

`docker network connect --ip`

`< IP > < Network > < Container >`

Disconnect the network container

`docker network disconnect`

`< Network_Name > <Container>`

Image Control

List locally available images

`docker images`

Execute the image

`docker run < Image >`

Create an image

`docker create < Image>: <Tag>`

Delete image

`docker rmi < Image >`

Saving images in a tar archive

`docker save < Image >`

Search for Docker images

`docker search < image >`

Recover a docker image

`docker pull < image >`

Builds an image from a "dockerfile"

`docker build -t <image-name> .`

Removes all unnecessary docking images

`docker image prune`

Volume controls

Image controls

`docker volume ls`

Volume control

`docker volume inspector < Volume >`

Create a volume

`docker volume create < Volume >`

Remove volume

`docker volume rm < Volume >`

Removes unused volumes

`docker volume prune`

Docker-Compose

Start all services defined in the docker-compose.yml file.

`docker-compose up`

Stop and remove all containers, networks, and volumes associated with the services defined in the docker-compose.yml file.

`docker-compose down`

List all containers started by the docker-compose up command, along with their status.

`docker-compose ps`

View the logs of all containers or a specific service.

`docker-compose logs`

Build or rebuild all Docker images defined in the docker-compose.yml file.

`docker-compose build`

Stop all containers started by docker-compose up command without removing them.

`docker-compose stop`

Restart all containers or a specific service.

`docker-compose restart`

Run a command in a running container.

`docker-compose exec`

Pull updated images for services from their respective repositories.

`docker-compose pull`

Scale the number of containers for a specific service.

`docker-compose up --scale`

`<service>=<n>`

Example 1

- To understand how docker compose works, let's take a simple example.
- Previously, we said that docker compose is useful for multi-container applications. To understand how it works, let's start with a single container.
- We're going to do the same thing twice.
 - First with the classic commands to start a container.
 - Next, we will use docker compose
- Our goal is to start an **NGINX** server on port 8080.

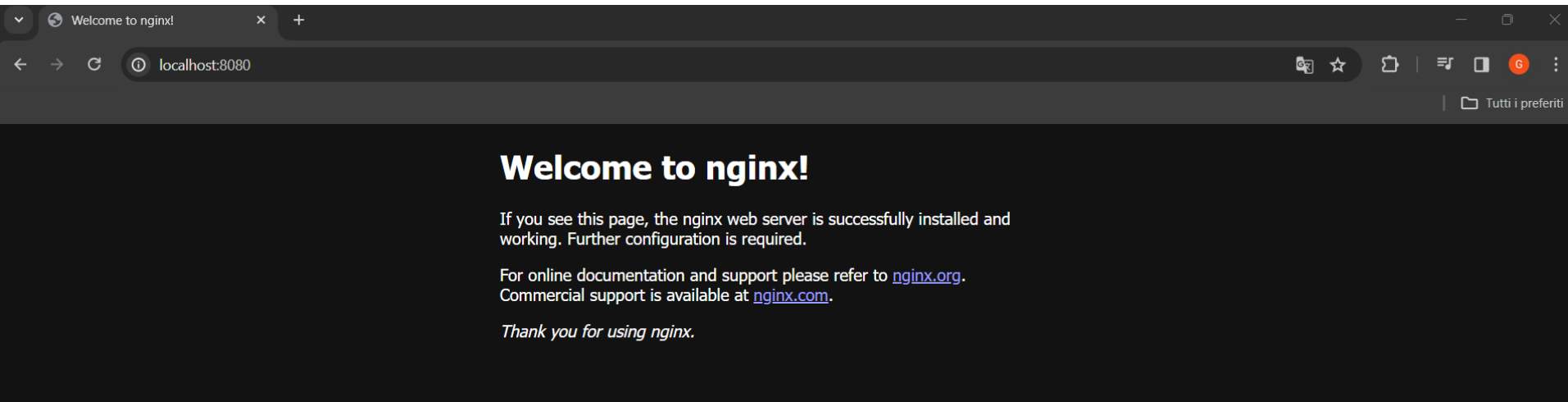
Example 1

```
giammino@Giammino: ~  
giammino@Giammino:~$ ls  
Cortina_challenge Cortina_challenge1 Dockerfile esempio.py hello.sh snap  
giammino@Giammino:~$ docker run --name esempio -it -p 8080:80 nginx  
Unable to find image 'nginx:latest' locally  
latest: Pulling from library/nginx  
af107e978371: Pull complete  
336ba1f05c3e: Pull complete  
8c37d2ff6efa: Pull complete  
51d6357098de: Pull complete  
782f1ecce57d: Pull complete  
5e99d351b073: Pull complete  
7b73345df136: Pull complete  
Digest: sha256:2bdc49f2f8ae8d8dc50ed00f2ee56d00385c6f8bc8a8b320d0a294d9e3b49026
```

- What do these commands do?
- docker run starts a new container;
 - --name esempio is the name of the container
 - -p 8080:80 maps port 8080 of the host to port 80 of the container. So, if you access the host on port 8080, the traffic is routed to the container on port 80.
 - nginx specifies the Docker image to use.

Example 1

- If we go to our browser and write localhost:8080, we can see this.



- Now our goal is to use docker compose.

Example 1

- What are the steps to follow?
 - Create a folder and move in.
 - Create a file “docker-compose.yaml”

```
giammino@Giammino: ~/test  ×  +  ▾  
giammino@Giammino:~$ mkdir test  
giammino@Giammino:~$ cd test  
giammino@Giammino:~/test$ nano docker-compose.yaml  
giammino@Giammino:~/test$
```

- Write these instructions.

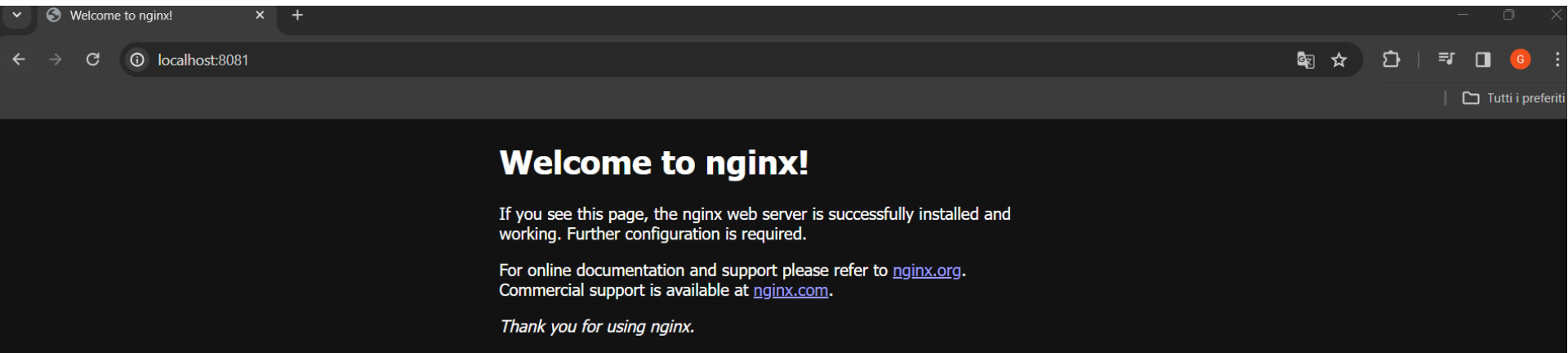
```
giammino@Giammino: ~/test  ×  +  ▾  
GNU nano 6.2 docker-compose.yaml *  
version: "3"  
services:  
  website:  
    image: nginx  
    ports:  
      - "8081:80"  
    restart: always
```

Example 1

```
giammino@Giammino:~/test$ docker-compose up
[+] Running 2/1
 ✓ Network test_default      Created          0.1s
 ✓ Container test-website-1  Created          0.1s
Attaching to website-1
website-1 | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
website-1 | /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
website-1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
website-1 | 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
website-1 | 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
website-1 | /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
website-1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
website-1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
website-1 | /docker-entrypoint.sh: Configuration complete; ready for start up
website-1 | 2024/01/02 09:06:25 [notice] 1#1: using the "epoll" event method
website-1 | 2024/01/02 09:06:25 [notice] 1#1: nginx/1.25.3
website-1 | 2024/01/02 09:06:25 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
website-1 | 2024/01/02 09:06:25 [notice] 1#1: OS: Linux 5.15.133.1-microsoft-standard-WSL2
website-1 | 2024/01/02 09:06:25 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker processes
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 29
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 30
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 31
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 32
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 33
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 34
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 35
website-1 | 2024/01/02 09:06:25 [notice] 1#1: start worker process 36
```

Example 1

- If we go to our browser and write localhost:8081, we can see this.



Example 2

Using the Flask framework, the application features a hit counter in Redis, providing a practical example of how Docker Compose can be applied in web development scenarios.

Create a directory for the project:

```
$ mkdir composetest  
$ cd composetest
```

Example 2

Create a file called `app.py` in your project directory and paste the following code in:

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return f'Hello World! I have been seen {count} times.\n'
```

Example 2

Create another file called `requirements.txt` in your project directory

```
flask  
redis
```

Example 2

Create a `Dockerfile` and paste the following code in:

```
# syntax=docker/dockerfile:1
FROM python:3.10-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run", "--debug"]
```

Example 2

Create a file called `compose.yaml` in your project directory and paste the following:

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

Example 2

- From your project directory, start up your application by running docker compose up.
- Enter localhost:8080 in a browser to see the application running.
- You should see a message in your browser saying “Hello World”
- **Now, build something on your own.**

Docker Networking

- This was an easy example to understand how Docker Compose works. But if, for instance, we have a server and a database that need to communicate with each other, how do we proceed?
- Firstly, we need to modify the Docker Compose file. Subsequently, we have to introduce the concept of Docker networking.

➤ **What is Docker Networking?**

Docker Networking

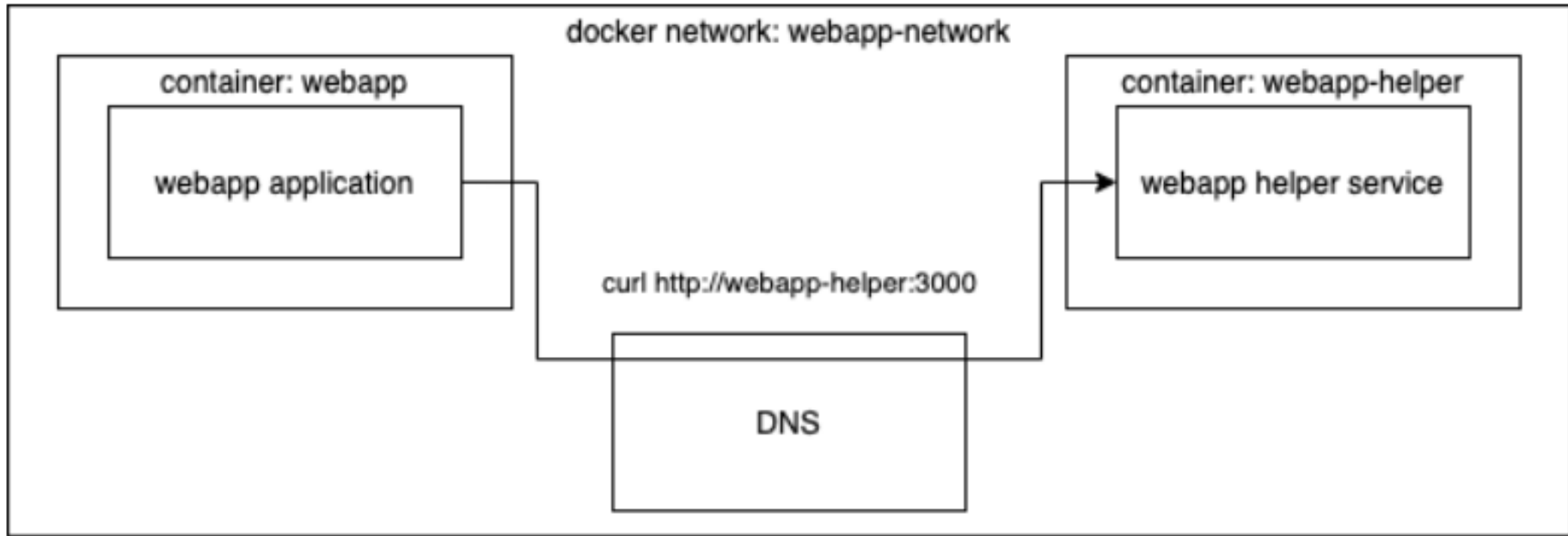
- **Container networking** refers to the ability of containers to connect to and communicate with each other.
- Containers have networking enabled by default, and they can make outgoing connections. A container has no information about what kind of network it's attached to, or whether its peers are also Docker workloads.
- A container only sees a network interface with an **IP address**, a **gateway**, a **routing table**, **DNS services**, and other networking details.

Docker Networking

- **Docker networking** refers to the set of technologies and tools provided by Docker to enable communication and connectivity between Docker containers and between containers and the outside world, including other systems or the internet. Docker networking allows containers to interact with each other and with external service.
- Docker provides a default bridge network (*docker0*) for communication between containers by default.
- *docker0* is a bridge-type network. A bridge network allows containers to communicate with each other using an internal bridge within the host.
- *docker0* is also involved in port mapping when exposing container services externally.

Docker Networking

- A typical example is the following:



- Two services, webapp and webapp-helper, exist within a shared network. The webapp-helper, with a server on port 3000, is sought by webapp. As they are defined in the same docker-compose.yml file, Docker Compose automatically establishes a network. Consequently, webapp can effortlessly make a request to webapp-helper:3000.

Network Drivers

- **bridge:** The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are commonly used when your application runs in a container that needs to communicate with other containers on the same host.
- **host:** Remove network isolation between the container and the Docker host, and use the host's networking directly.
- **overlay:** It allows communication between containers on different hosts, which is useful for Docker Swarm. It creates a distributed network across multiple machines.

Network Drivers

- **ipvlan:** IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.
- **macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.
- **none:** Completely isolate a container from the host and other containers.

Network Drivers

➤ In summary..

Driver	Description
bridge	The default network driver.
host	Remove network isolation between the container and the Docker host.
none	Completely isolate a container from the host and other containers.
overlay	Overlay networks connect multiple Docker daemons together.
ipvlan	IPvlan networks provide full control over both IPv4 and IPv6 addressing.
macvlan	Assign a MAC address to a container.

Example

- It is also possible to define the network manually in a Docker Compose file. A major benefit of a manual network definition is that it makes it easy to set up a configuration where containers defined in two different Docker Compose files share a network, and can easily interact with each other.
- Let us now have a look how a network is defined in docker-compose.yml:

Example

```
1  version: '3.9'
2
3  services:
4
5      #KAFKA CLUSTER
6      zookeeper:
7          # https://hub.docker.com/r/confluentinc/cp-zookeeper
8          image: confluentinc/cp-zookeeper:${ZOOKEEPER_VERSION}
9          container_name: zookeeper
10         hostname: zookeeper
11         restart: always
12         environment:
13             ZOOKEEPER_CLIENT_PORT: ${ZOOKEEPER_PORT}
14         volumes:
15             - siem-zookeeper-data:/var/lib/zookeeper/data
16             - siem-zookeeper-log-data:/var/lib/zookeeper/log
17         networks:
18             - soar-network
19         ports:
20             - ${ZOOKEEPER_PORT}:${ZOOKEEPER_EXPOSED_PORT}
```

```
185     networks:
186         soar-network:
187             name: soar-network
188             external: true
189             driver: bridge
```

Example

- If you want to see the network parameters, you can use *docker network inspect name_of_the_network*

```
"Name": "certify-network",
"Id": "09f1a03d2eee6b4d40324ac0760b3cf56924eb5801d14da693c1762735bcf60b",
"Created": "2024-12-13T12:40:57.169805254+01:00",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
  "Driver": "default",
  "Options": {},
  "Config": [
    {
      "Subnet": "172.19.0.0/16",
      "Gateway": "172.19.0.1"
    }
  ]
},
"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
  "Network": ""
},
"ConfigOnly": false,
"Containers": {
  "3a434ed5204999a9809c3041a21ac9638324d89d4f4a05eeab343ff44aa1752e": {
    "Name": "zookeeper",
    "EndpointID": "9a617e121488091904137d4c2c3931c611ec6d933ef07e892b97d4fe4599e304",
    "MacAddress": "02:42:ac:13:00:03",
    "IPv4Address": "172.19.0.3/16",
    "IPv6Address": ""
  },
  "c073305c3af413895745f34e9e04f7870836dd342c09eedf4d1ec458144f13b": {
    "Name": "probes-converter-1",
    "EndpointID": "4227d34cfc8f2e54cee69340f831f62a105fd5e5ee81d62292cf8ecf14746f97",
    "MacAddress": "02:42:ac:13:00:02",
    "IPv4Address": "172.19.0.2/16",
    "IPv6Address": ""
  },
  "faa67ba5f607dc7a80d4a9c3c447c4feef90123f22382b694ce108ebaefec83d": {
    "Name": "kafka",
    "EndpointID": "70fe635186da35c822b797af48eda81114cfc96af3b87895081206d083a7363c",
    "MacAddress": "02:42:ac:13:00:04",
    "IPv4Address": "172.19.0.4/16",
    "IPv6Address": ""
  }
}
```


Some Useful Commands

Command	Description
<code>docker network connect</code>	Connect a container to a network
<code>docker network create</code>	Create a network
<code>docker network disconnect</code>	Disconnect a container from a network
<code>docker network inspect</code>	Display detailed information on one or more networks
<code>docker network ls</code>	List networks
<code>docker network prune</code>	Remove all unused networks
<code>docker network rm</code>	Remove one or more networks

Example

- To connect two services, such as Wordpress and MySQL, we will modify the docker-compose.yaml file.

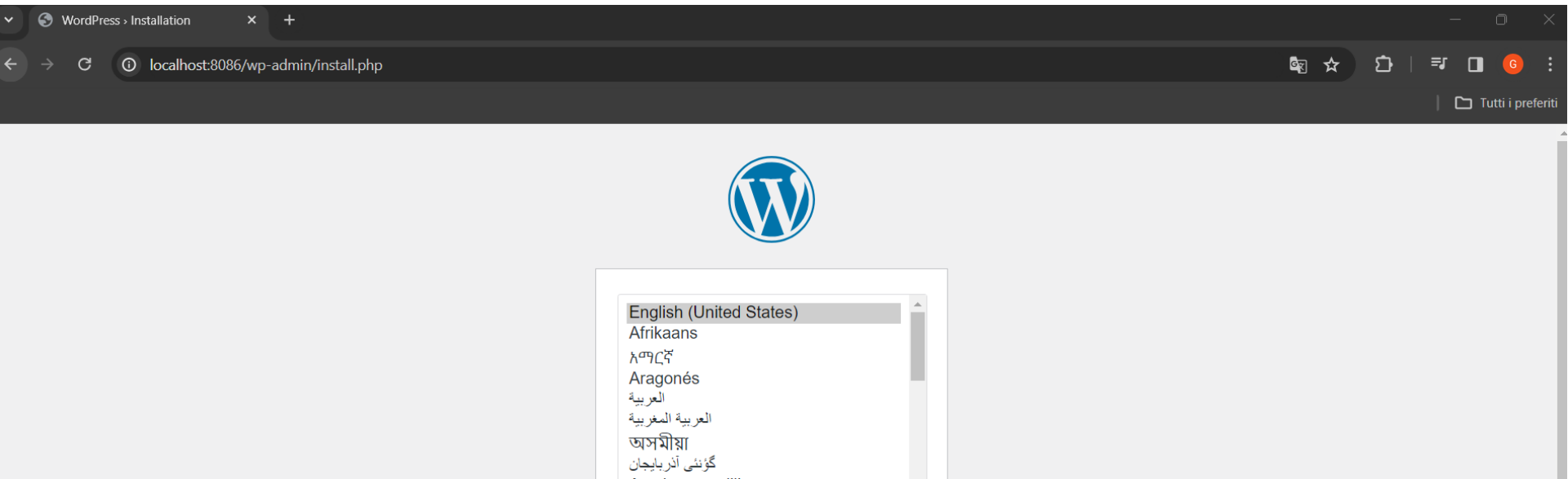
```
giammino@Giammino: ~/ese x + v
GNU nano 6.2 docker-compose.yaml *
image: wordpress
ports:
  - "8080:80"
restart: always
networks:
  - mynetwork
depends_on:
  - mysql-db
environment:
  WORDPRESS_DB_HOST: mysql-db
  WORDPRESS_DB_USER: myuser
  WORDPRESS_DB_PASSWORD: mypassword
  WORDPRESS_DB_NAME: mydatabase

mysql-db:
  image: mysql:latest
  environment:
    MYSQL_ROOT_PASSWORD: examplepassword
    MYSQL_DATABASE: mydatabase
    MYSQL_USER: myuser
    MYSQL_PASSWORD: mypassword
  restart: always
  networks:
    - mynetwork

networks:
  mynetwork:
    driver: bridge
```

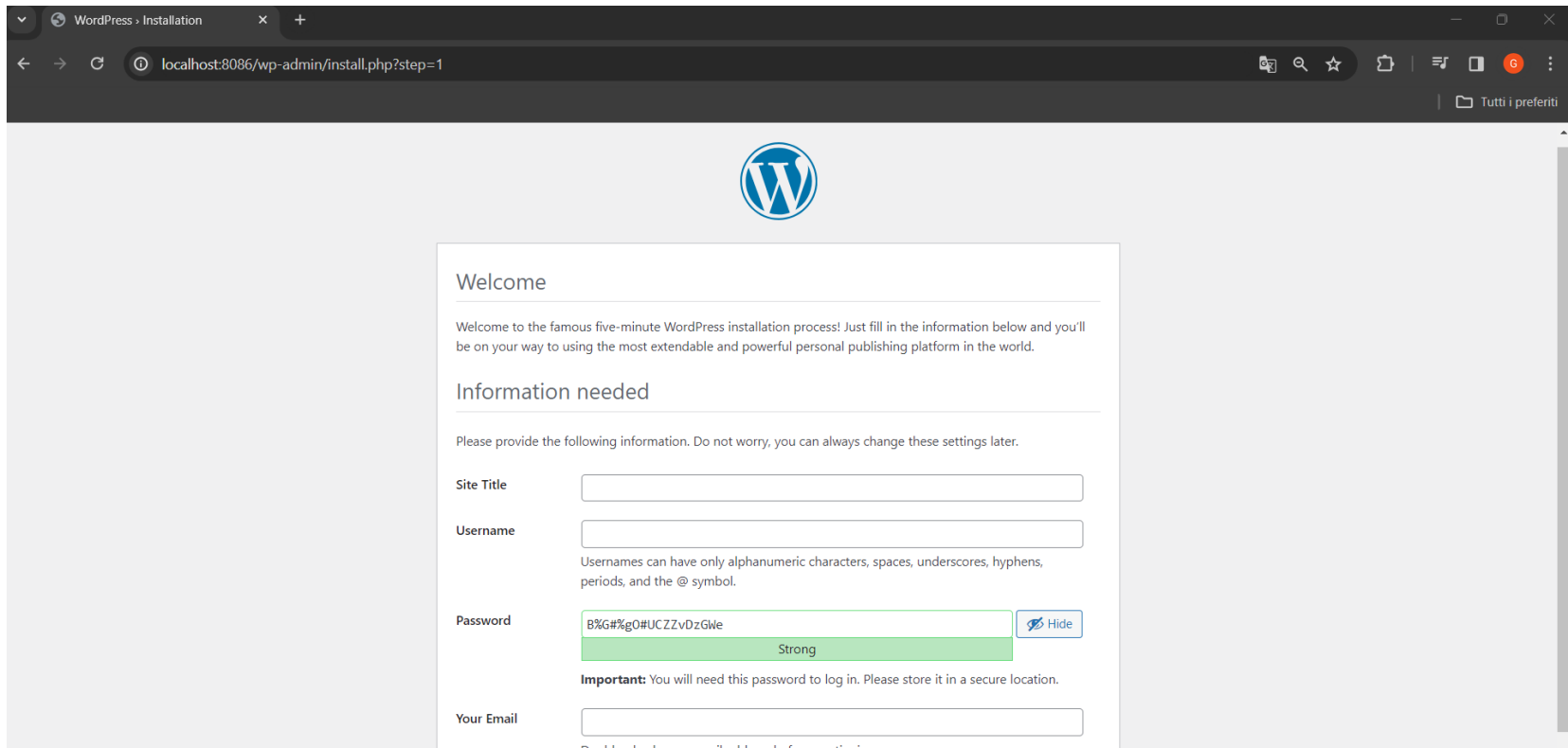
Example

➤ Now, if we go on localhost:8080, we can see Wordpress:



Example

- After choosing the language, you can enter the data to create a site.
- Subsequently, you can also log in to create your own projects.



The screenshot shows the WordPress installation interface in a web browser. The address bar indicates the URL is `localhost:8086/wp-admin/install.php?step=1`. The page features the WordPress logo at the top center. Below it, a 'Welcome' message states: 'Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.'

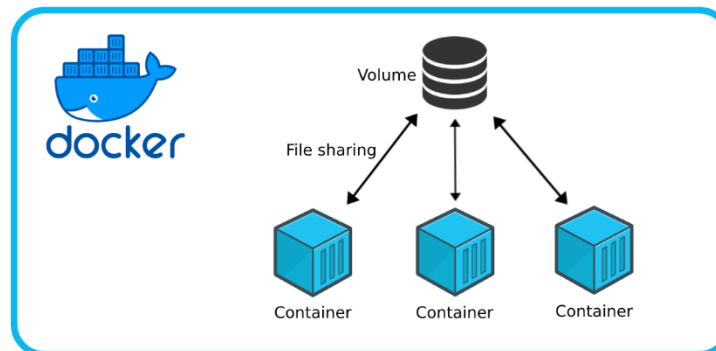
The 'Information needed' section prompts the user to provide the following information, noting that settings can be changed later:

- Site Title:** A text input field.
- Username:** A text input field. A note below states: 'Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.'
- Password:** A text input field containing the generated password `B%G#%g0#UCZZvDzGWe`. A strength indicator shows 'Strong' in a green bar. A 'Hide' button is next to the password field.
- Your Email:** A text input field. A note below states: 'Double check your email address before continuing.'

Example

- After choosing the language, you can enter the data to create a site.
- Subsequently, you can also log in to create your own projects.
- What happens if the container is stopped?
- All data is lost because it is not persistent.
- To make them persistent, we introduce **Docker volumes**.

- **Docker volumes** are a way to allow containers to persist and share data beyond the lifecycle of the container itself.
- Docker volumes are designed to resolve the challenge of data persistence in a container environment.
 - Data inside a container is ephemeral and is lost when the container is deleted. Volumes allow data to persist even when containers are stopped or removed.
 - With volumes, it's easier to back up data and restore it later. Volume data can be stored separately from containers, simplifying the backup process.



Example

- To see how Docker volumes work, let's once again modify the Docker Compose file.

```
giammino@Giammino: ~/ese  ×  +  ▾
GNU nano 6.2                                     docker-compose.yaml
version: "3"
services:
  wordpress:
    image: wordpress
    ports:
      - "8086:80"
    restart: always
    networks:
      - mynetwork
    depends_on:
      - mysql-db
    environment:
      WORDPRESS_DB_HOST: mysql-db
      WORDPRESS_DB_USER: myuser
      WORDPRESS_DB_PASSWORD: mypassword
      WORDPRESS_DB_NAME: mydatabase
    volumes:
      - wordpress_data:/var/www/html

  mysql-db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: examplepassword
      MYSQL_DATABASE: mydatabase
      MYSQL_USER: myuser
      MYSQL_PASSWORD: mypassword
    restart: always
    networks:
      - mynetwork
    volumes:
      - mysql_data:/var/lib/mysql

networks:
  mynetwork:
    driver: bridge

volumes:
  wordpress_data:
  mysql_data:
```

Example

- To see how Docker volumes work, let's once again modify the Docker Compose file.
- This changes has allowed us to save the volumes in the paths specified in the file.
- Even if we stop the container, the data, and in our case, the WordPress projects, are not deleted.
- If we want to view the volumes, we just need to execute this command.

```
giammino@Giammino:~/esempio$ docker volume ls
DRIVER      VOLUME NAME
local       1fc523612542e294bd6f7c949f53a770228cc94ba0958b0e6279dfe4dc080ad8
local       2c32b479d13d04cbce18e389fb31fb97e899d47c69ee29a943f762f6fb2563e2
local       4c1014a927be5514249afc5641e5b1d5339e114e208ecb2d2d0a5a58b060b078
local       09fb5a94e479c4848bc0f40b5d33658476d86e1c20cc11a2f789e3340f5375ce
local       18c34ebcc6dd8270a1e8331d564dea5b0fc778e93db44ff2fbb6d52ea5e81a7
local       60bb67f039e82c5128cbd2b74efc90a1815cdb3ca0b9ca222041951fb4f44863
local       78ec1d3be313964fb113d0f424d25332a312049694cbd06a2dee26f8afe5fab3
local       4764fa54b1c5cb72152aaea6e594cf24dd66196731383c7e8a14cf3749781f1d
local       a0dd179a5c740552d0c98f2bada2396ce89cd92f740ec398a0a1dee626227780
local       ad447bcd5342478ee64ec7d3237de88e67245b753a3d8bef07ee409b4f103432
local       b1e601757f018c2a4732c510ee70d6245aac71a07b5739ed995b23b3a18044ee
local       ccb7af9d2321057c98573deb96a60b1b3d466c68298c76226d632409955dea47
local       dc3dac3b067ab9b3a5e2b86a5f021147feeaab56c4d1cd7ef197a4d51c38054c
local       esempio_mysql_data
local       esempio_wordpress_data
local       fiware_mongo-db
local       siem-grafana-storage
local       siem-kafka-data
local       siem-logstash-data
local       siem-opensearch-data1
local       siem-zookeeper-data
local       siem-zookeeper-log-data
giammino@Giammino:~/esempio$
```


Conclusions

- It highlighted how the lighter and more efficient approach of containers provides a powerful method for isolating and deploying applications.
- The intuitiveness of Docker Desktop makes Docker accessible to a wide audience.
- We've seen the importance of a Dockerfile and how it determines the layers of a Docker image.
- The utility of Docker Compose in orchestrating multi-container services was underscored, simplifying the definition and management of complex applications.