
Docker architecture, images and containers

Basic concepts

Marica Antonacci (INFN Bari)
marica.antonacci@ba.infn.it



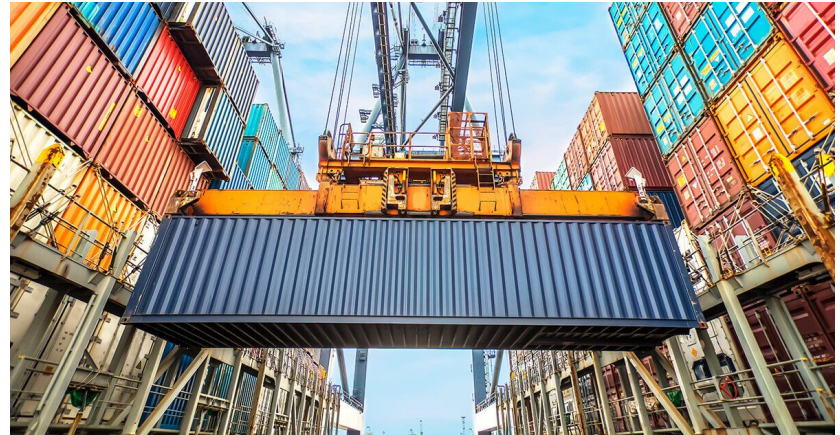
- What is a container?
- Docker architecture
- Docker main components
 - Images
 - Containers
 - Registries
- Docker CLI & GUI
- Hands-on#1
- References

What are containers?

Containers are a form of virtualization technology that allows you to **package an application and its dependencies together**, isolating it from the underlying system

□ Key concepts:

- **Isolation:** Containers provide a secure and isolated environment for applications, preventing conflicts with other applications or the host system.
- **Portability:** Containers can run consistently across different environments, making it easy to develop and deploy applications.
- **Efficiency:** Containers are efficient in terms of resource usage, as they share the host OS kernel.

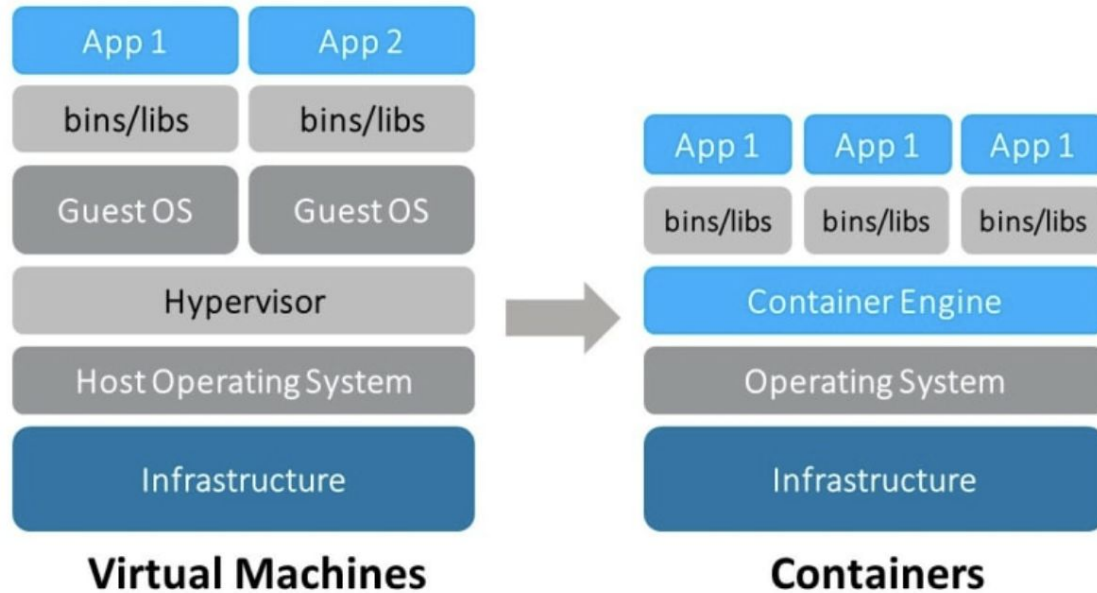


Benefits of containerization

Containerization offers several advantages in software development and deployment.

- ❑ **Build it once, run it anywhere:** Developers can easily and reliably run applications in different environments, such as local desktops, physical servers, virtual servers, production environments, and public and private clouds.
- ❑ **Improved Developer Productivity:** Containers allow developers to create predictable runtime environments. The old adage “it worked on my machine” is no longer a concern! In a containerized architecture, developers and operations teams spend less time debugging and diagnosing environmental differences, and can spend their time building and delivering new product features.
- ❑ **Smooth scaling:** Applications in containers can be easily scaled up or down to handle varying workloads. Containers support a true microservices approach to development.
- ❑ **Resource Optimization:** Containers optimize resource utilization, making efficient use of server resources.

Virtual Machines vs Containers



The Containers work on the concept of **OS-level virtualization**, i.e. the kernel's ability to make multiple isolated environments on a single host.

Virtual Machine

Pros:

VMs provide strong isolation and offer flexibility in choosing different operating systems.

Cons:

VMs are heavier, slower to start, and consume more resources due to their independent OS.

Container

Pros:

Lightweight and efficient, containers share the host OS kernel, resulting in faster startup and efficient resource usage.

Cons:

Limited OS compatibility and less isolation compared to VMs.

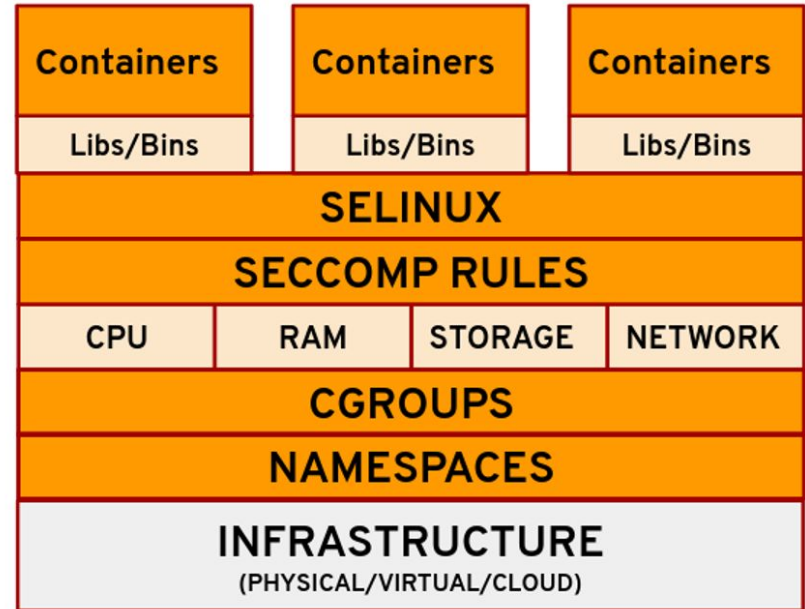
OS-level virtualization pillars

Namespaces provide process and resource isolation.

Control Groups (cgroups) are responsible for resource allocation and management.

Security Modules (AppArmor, SELinux) restrict a container's capabilities. They ensure that containers can only access the resources and actions they are explicitly allowed to, enhancing overall security.

Seccomp (Secure Computing Mode): Seccomp allows administrators to define a list of system calls that containers are allowed to make. It significantly reduces the attack surface by limiting the system calls available to containers. The default seccomp profile for Docker, disables around 44 syscalls out of 300+.



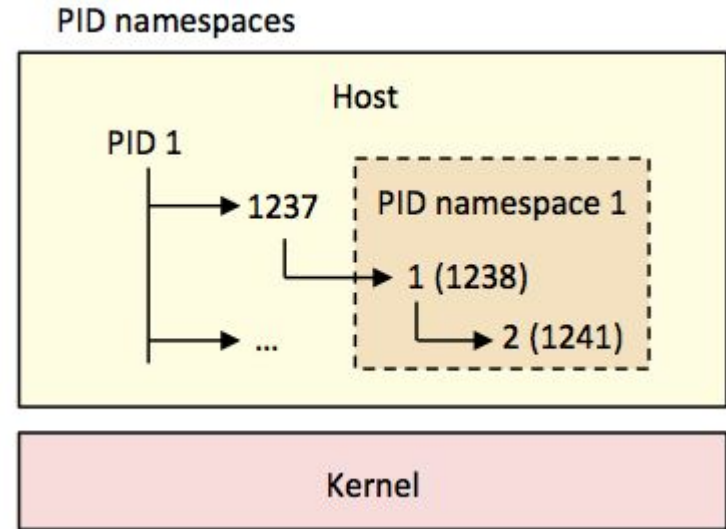
See this interesting [blog](#) about Namespaces and Cgroups

Restricting visibility: Namespaces

Linux namespaces: It is a feature of Linux kernel to isolate resources from each other. This allows one set of Linux processes to see one group of resources while allowing another set of Linux processes to see a different group of resources.

There are several kinds of namespaces in Linux: Mount (mnt), Process ID (PID), Network (net), User ID (user) and Interprocess Communication (IPC).

For example, two processes in two different mounted namespaces may have different views of what the mounted root file system is. Each container can be associated with a specific set of namespaces, and these namespaces are used inside these containers only.

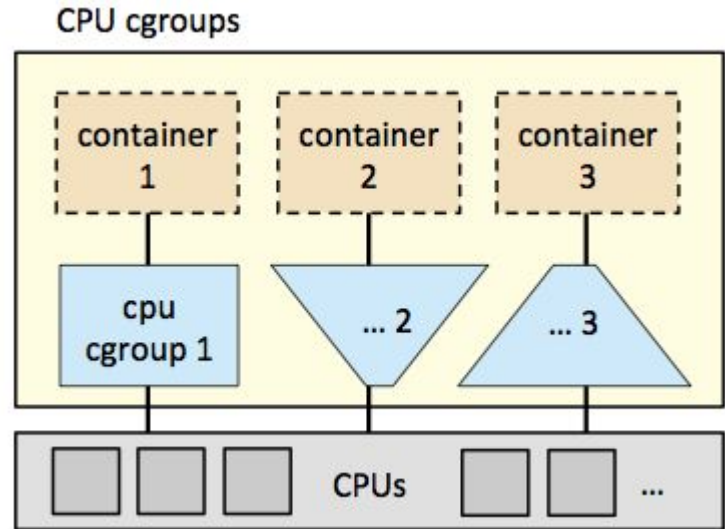


Restricting usage: Control groups

cgroups provide an effective mechanism for **resource limitation**.

With cgroups, you can control and manage system resources (CPU, Memory, Networking, disk I/O) per Linux process, increasing overall resource utilization efficiency.

Cgroups allow to control resource utilization per container.



Introduction to docker

- ❑ Docker is a leading containerization platform that simplifies the creation, deployment, and management of containers.
- ❑ Docker plays a pivotal role in modern software development and deployment practices.
- ❑ Notable companies like Netflix and Uber rely on Docker to enhance their application delivery processes.

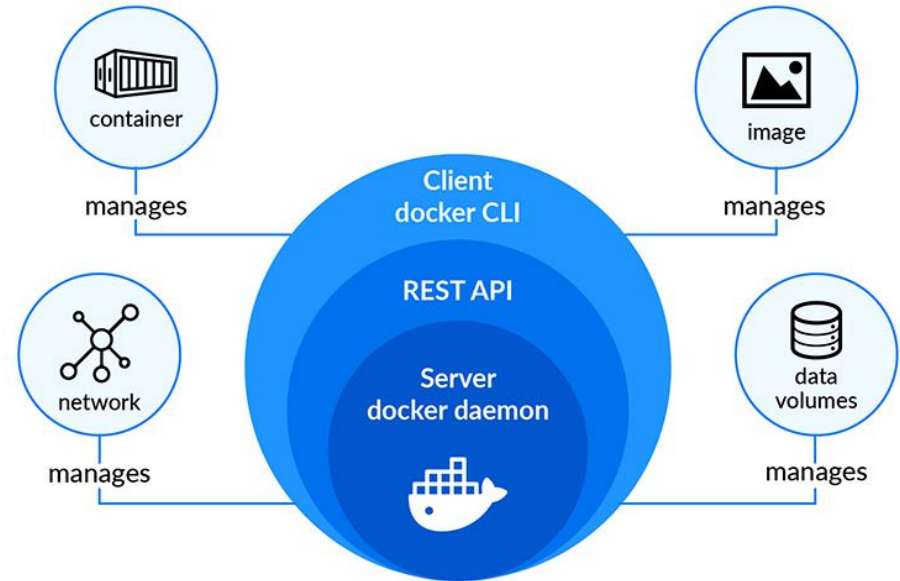
<https://www.linkedin.com/pulse/dockers-impact-how-leading-companies-scaled-business-using-sachin-adi/>

Docker architecture

Docker is an open source platform for building, deploying, and managing containerized applications

Docker works on a client-server architecture:

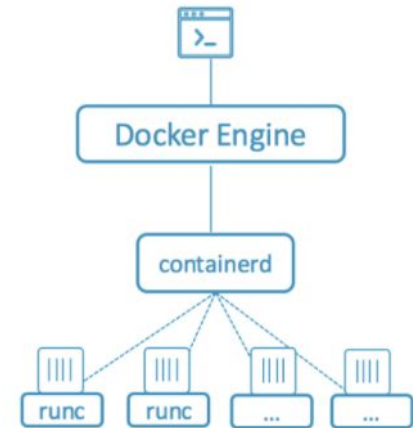
- a server with a long-running daemon process **dockerd**.
- **APIs** which specify interfaces that programs can use to talk to and instruct the Docker daemon.
- A command line interface (**CLI**) client docker.



docker, containerd, runc

When you run a container with *docker*, you're actually running it through the Docker daemon, containerd, and then runc.

- **containerd** is an industry standard high-level runtime for containers. It's main responsibility is to maintain the container's lifecycle (create/update/stop/restart or delete).
- **runc** is the runtime specification given by **OCI** (Open Container Initiative) for running containers, interacting with existing low-level Linux features, like namespaces and control groups.
 - after the creation of the container *runc* exits and the lifecycle of the container is managed by the *shim*^(*) process (that becomes parent of the container).



(*) In tech terms, a shim is a component in a software system, which acts as a bridge between different APIs, or as a compatibility layer. A shim is sometimes added when you want to use a third-party component, but you need a little bit of glue code to make it work.

Docker main components

- **Docker containers:** Isolated user-space environments running the same or different applications and sharing the same host OS kernel. Containers are created from Docker images.
- **Docker images:** Docker templates that include application libraries and applications. Images are used to create containers and you can bring up containers immediately. You can create and update your own custom images as well as download build images from Docker's public registry.
- **Docker registries:** This is an images store. Docker registries can be public or private, meaning that you can work with images available over the internet or create your own registry for internal purposes. One popular public Docker registry is [Docker Hub](#).

What is a docker registry?

A Docker registry is a **storage and distribution system** for named Docker images.

The same image might have multiple different versions, identified by their **tags**.

A Docker registry is organized into **Docker repositories** , where a repository holds all the versions of a specific image.

The registry allows Docker users to pull images locally, as well as push new images to the registry (given adequate access permissions when applicable).

By default, the Docker engine interacts with [DockerHub](https://hub.docker.com/), Docker's public registry instance.

Private registries

Use cases for running a private registry on-premise (internal to the organization) include:

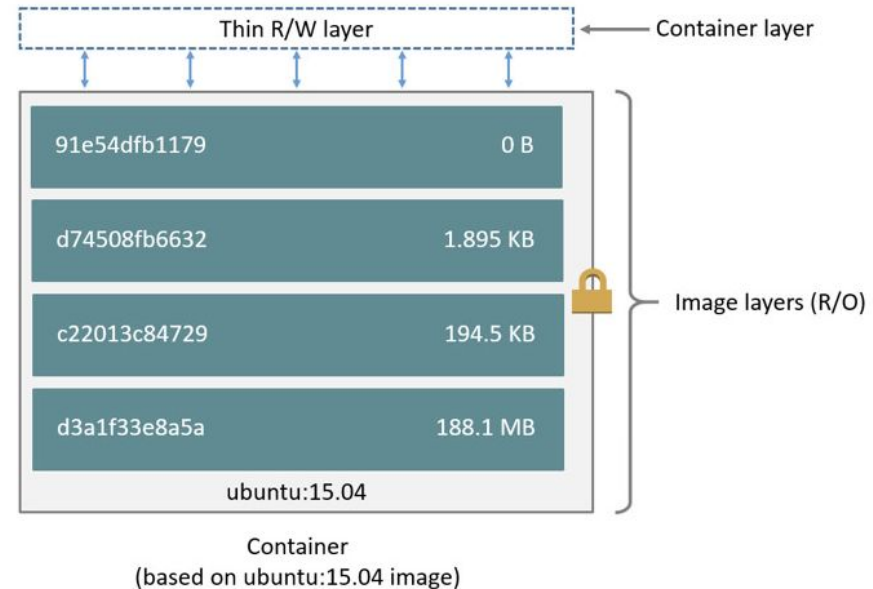
- **Distributing images inside an isolated network** (not sending images over the Internet)
- **Creating faster CI/CD pipelines** (pulling and pushing images from internal network), including faster deployments to on-premise environments
- **Deploying a new image over a large cluster of machines**
- **Tightly controlling where images are being stored**

Private registries: some open-source implementations

- **Docker Registry** is a stateless, highly scalable server side application that stores and lets you distribute Docker images .
- **GitLab Container Registry** is tightly integrated with GitLab CI's workflow, with minimal setup.
 - INFN SSNN provide a container registry as part of the platform baltiq.infn.it based on GitLab
- **Harbor** (CNCF Graduated project) is an open source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted.
 - INFN Cloud has implemented a docker registry based on Harbor: <https://harbor.cloud.infn.it/>
- **JFrog Container Registry** supporting Docker containers and Helm Chart repositories for Kubernetes deployments.

Docker image layers

- A Docker Image consists of read-only layers built on top of each other.
- Docker uses the **Union File System (UFS)** to build an image.
- The image is shared across containers.
- Each time Docker launches a container from an image, it adds a thin writable layer, known as the **container layer**, which stores all changes to the container throughout its runtime.

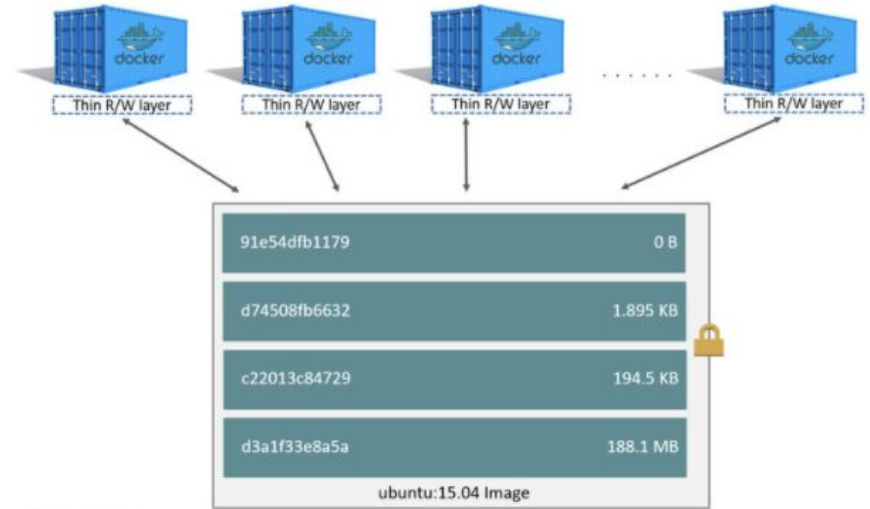


Docker image vs container

Each container has its own writable container layer, and all changes are stored in this container layer.

Multiple containers can share access to the same underlying image and yet have their own data state.

When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.



Copy-On-Write mechanism

COW is a standard UNIX pattern that provides a single shared copy of some data until the data is modified.

Docker makes use of copy-on-write technology with both images and containers. This **CoW strategy optimizes both image disk space usage and the performance of container start times**. At start time, Docker only has to create the thin writable layer for each container.

Containers that write a lot of data consume more space than containers that do not. This is because most write operations consume new space in the container's thin writable top layer.

Note: for write-heavy applications, you should not store the data in the container. Instead, use Docker volumes, which are independent of the running container and are designed to be efficient for I/O. In addition, volumes can be shared among containers and do not increase the size of your container's writable layer. (Source: [Docker docs](#))

Docker storage drivers

Storage drivers allow you to create data in the **writable layer** of your container. The files **won't be persisted** after the container is deleted, and both read and write speeds are **lower** than native file system performance.

Docker supports the following storage drivers:

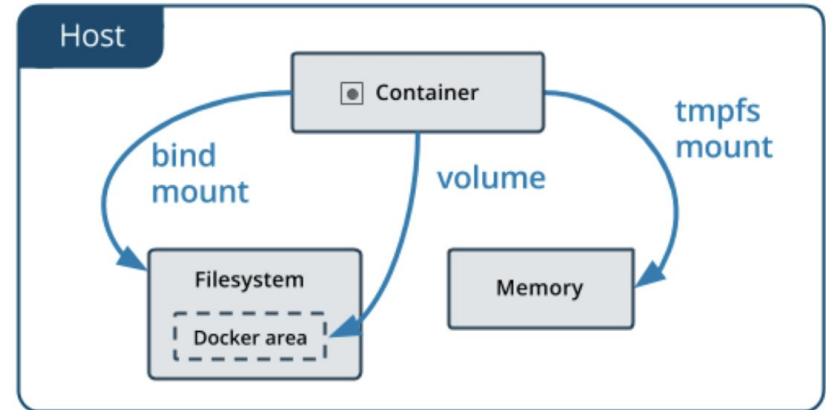
- **overlay2** is the preferred storage driver, for all currently supported Linux distributions, and requires no extra configuration.
- **fuse-overlayfs** is preferred only for running Rootless Docker on a host that does not provide support for rootless overlay2. On Ubuntu and Debian 10, the fuse-overlayfs driver does not need to be used as overlay2 works even in rootless mode.
- **devicemapper** is supported, but requires direct-lvm for production environments, because loopback-lvm, while zero-configuration, has very poor performance. devicemapper was the recommended storage driver for CentOS and RHEL, as their kernel version did not support overlay2. However, current versions of CentOS and RHEL now have support for overlay2, which is now the recommended driver.
- The **btrfs** and **zfs** storage drivers are used if they are the backing filesystem (the filesystem of the host on which Docker is installed). These filesystems allow for advanced options, such as creating “snapshots”, but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
- The **vfs** storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

More info at <https://docs.docker.com/storage/storagedriver/select-storage-driver/>

Persist data with volumes

Docker provides the following options for containers to store files in the host machine, so that the files are persisted even after the container stops

- ❖ volumes
- ❖ bind mounts
- ❖ tmpfs



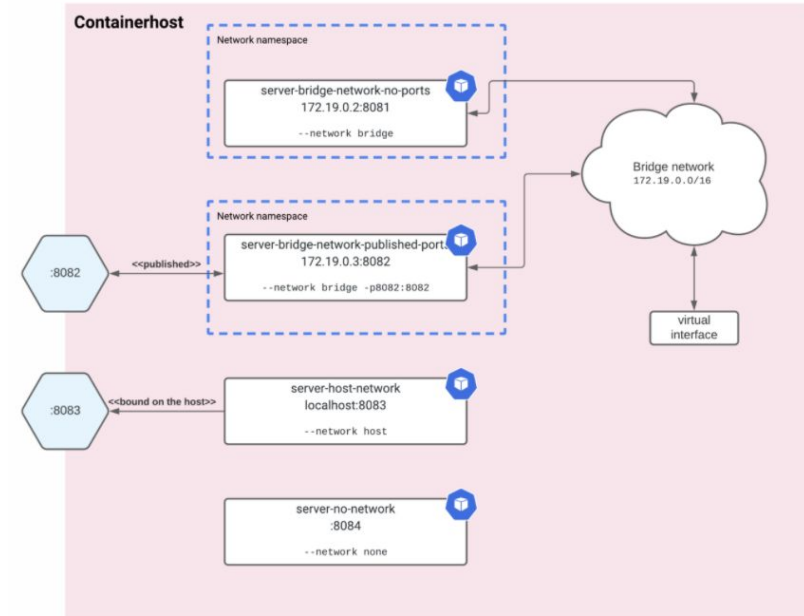
- **volumes** are stored in a part of the host filesystem which is **managed by Docker** (`/var/lib/docker/volumes/` on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- **bind mounts** may be stored **anywhere on the host system**. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
- **tmpfs** mounts are stored in the host system's memory only, and are never written to the host system's filesystem

Docker networking

A network in Docker is another logical object like a container and image.

By default Docker has the following networking drivers:

- **bridge**: the default networking driver in Docker. This can be used when multiple containers are running in standard mode and need to communicate with each other
- **host**: removes the network isolation completely. Any container running under a host network is basically attached to the network of the host system. Host mode networking can be useful to optimize performance, and in situations where a container needs to handle a large range of ports, as it does not require network address translation (NAT), and no “userland-proxy” is created for each port
- **none**: this driver disables networking for containers altogether
- **overlay**: this is used for connecting multiple Docker daemons across computers
- **macvlan**: it allows assignment of MAC addresses to containers, making them function like physical devices in a network
- **ipvlan**: similar to macvlan, the key difference being that the endpoints have the same MAC address.



Docker cli

```
$ docker help
```

```
Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "/home/tutor1/.docker")
  -c, --context string Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var and default context set with "docker context use")
  -D, --debug          Enable debug mode
  -H, --host list      Daemon socket(s) to connect to
  -l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
  --tls               Use TLS; implied by --tlsverify
  --tlscacert string  Trust certs signed only by this CA (default "/home/tutor1/.docker/ca.pem")
  --tlscert string    Path to TLS certificate file (default "/home/tutor1/.docker/cert.pem")
  --tlskey string     Path to TLS key file (default "/home/tutor1/.docker/key.pem")
  --tlsverify         Use TLS and verify the remote
  -v, --version       Print version information and quit

Management Commands:
  app*      Docker App (Docker Inc., v0.9.1-beta3)
  builder   Manage builds
  buildx*   Build with BuildKit (Docker Inc., v0.5.1-docker)
  config    Manage Docker configs
  container Manage containers
  context   Manage contexts
  image     Manage images
  manifest  Manage Docker image manifests and manifest lists
  network  Manage networks
  node     Manage Swarm nodes
  plugin   Manage plugins
  scan*    Docker Scan (Docker Inc., v0.7.0)
  secret   Manage Docker secrets
  service  Manage services
  stack    Manage Docker stacks
  swarm    Manage Swarm
  system   Manage Docker
  trust    Manage trust on Docker images
  volume   Manage volumes

https://docs.docker.com/engine/reference/commandline/cli/
```

Commands to manage docker objects

Usage: docker **container** COMMAND

Manage containers

Commands:

attach	Attach local standard input, output, and error streams to a running container
commit	Create a new image from a container's changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories on a container's filesystem
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
inspect	Display detailed information on one or more containers
kill	Kill one or more running containers
logs	Fetch the logs of a container
ls	List containers
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container
prune	Remove all stopped containers
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
run	Run a command in a new container
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
wait	Block until one or more containers stop, then print their exit codes

Usage: docker **image** COMMAND

Manage images

Commands:

build	Build an image from a Dockerfile
history	Show the history of an image
import	Import the contents from a tarball to create a filesystem image
inspect	Display detailed information on one or more images
load	Load an image from a tar archive or STDIN
ls	List images
prune	Remove unused images
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rm	Remove one or more images
save	Save one or more images to a tar archive (streamed to STDOUT by default)
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

Usage: docker **volume** COMMAND

Manage volumes

Commands:

create	Create a volume
inspect	Display detailed information on one or more volumes
ls	List volumes
prune	Remove all unused local volumes
rm	Remove one or more volumes

Usage: docker **network** COMMAND

Manage networks

Commands:

connect	Connect a container to a network
create	Create a network
disconnect	Disconnect a container from a network
inspect	Display detailed information on one or more networks
ls	List networks
prune	Remove all unused networks
rm	Remove one or more networks

Miscellaneous commands

- **docker ps**: list running containers
 - -a to list also stopped containers
 - -s to show container sizes
- **docker stats**: display container(s) usage statistics
- **docker system df**: show docker disk usage
- **docker system prune**: remove unused data

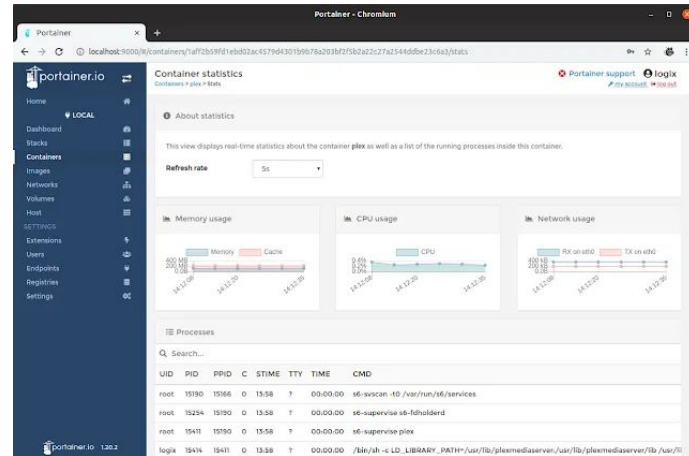
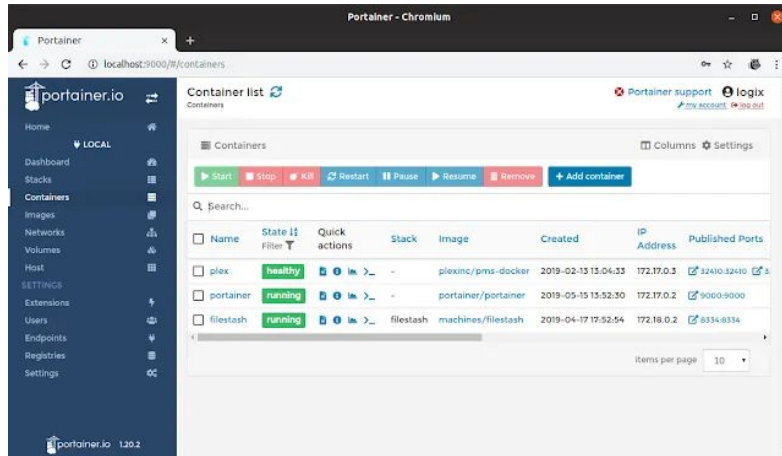
Docker Graphical Interface

Portainer is a lightweight management UI which allows you to easily manage your different Docker environments.

The tool, which is compatible with the standalone Docker engine, Docker Swarm, Nomad and Kubernetes, is simple to both use and deploy, being available as a Docker container itself. It can be used both on the local machine as well as a remote Docker GUI.

Portainer allows you to manage all your Docker resources (containers, images, volumes, networks and more)

For more details: <https://docs.portainer.io/>



Docker alternatives

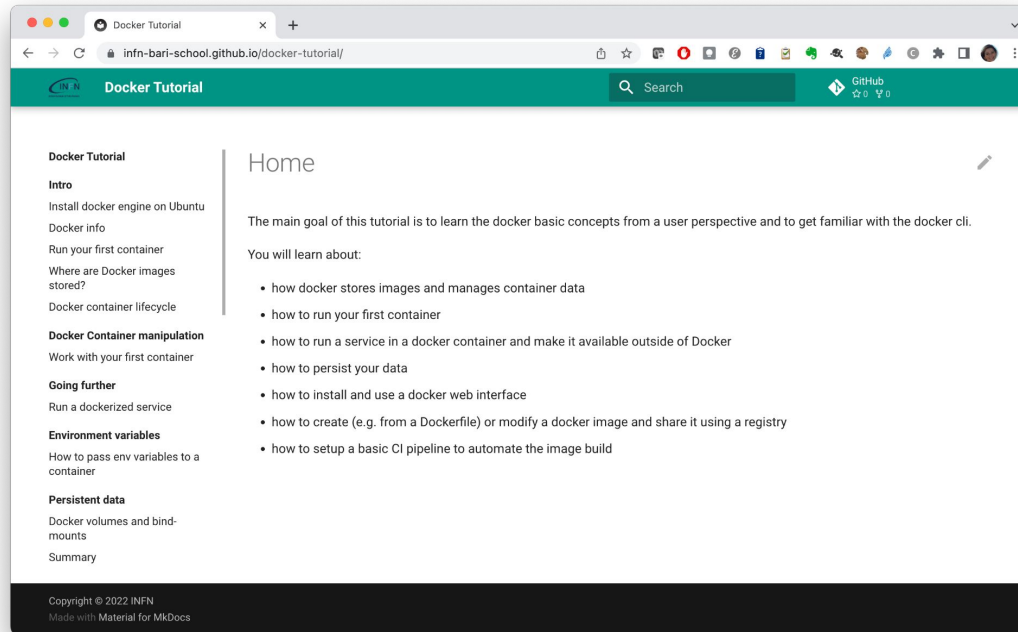
One of the drawbacks of Docker is that the Docker engine requires root privileges to run containers.

udocker is an open source project and provides the same basic functionality the Docker engine does but without root privileges.

It works by creating a chroot-like environment over the extracted container and uses various implementation strategies to mimic chroot execution with just user-level privileges. One of the execution environments you can use is runC, the same one used by Docker.

Podman is a daemon-less, open-source, Linux-native container engine developed by RedHat, that is used to build, run and manage Linux OCI containers and container images. Containers can either be run as root or in rootless mode

<https://infn-bari-school.github.io/docker-tutorial/>



References & credits

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

<https://medium.com/zero-equals-false/docker-introduction-what-you-need-to-know-to-start-creating-containers-8ffaf064930a>

<http://100daysofdevops.com/21-days-of-docker-day-21/>

<https://awesome-docker.netlify.app/>