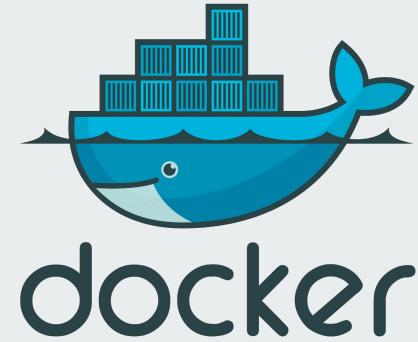


Programming and Communications III: Docker

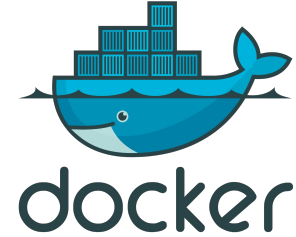


Jordi Ricard Onrubia Palacios

Departament d'Informàtica i Enginyeria Industrial Universitat de Lleida

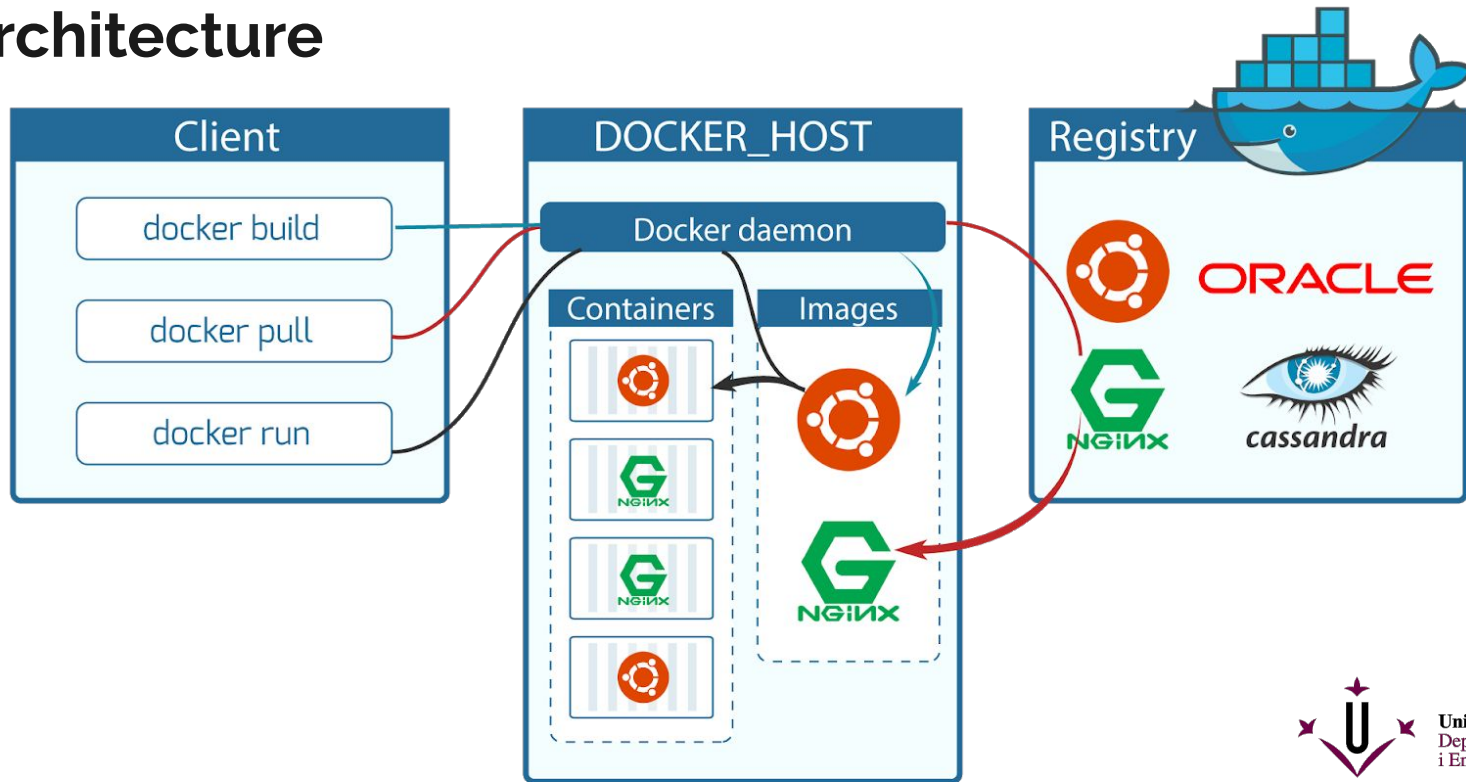


Overview



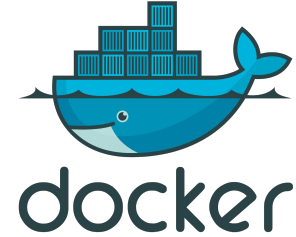
- Docker is an open platform for developing, shipping, and running applications.
- Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- Docker provides the ability to package and run an application in a loosely isolated environment called a container.
- Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host and allows you to run many containers simultaneously on a given host.
- Easily share containers, so everyone gets the same container with the same configuration.

Architecture





Architecture



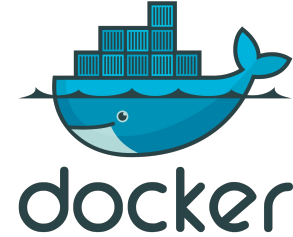
The Docker client: Primary way to interact with Docker, we can use it to send commands like run, build or pull. This way we can communicate with one or more docker daemons.

The Docker daemon: Listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker registries: Stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.



Architecture - Docker objects



Images:

Is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

Containers:

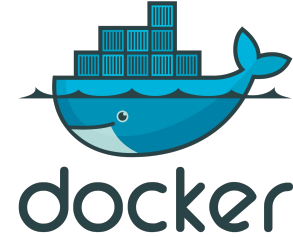
Is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.



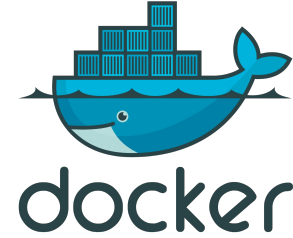
Architecture - More objects - Storage



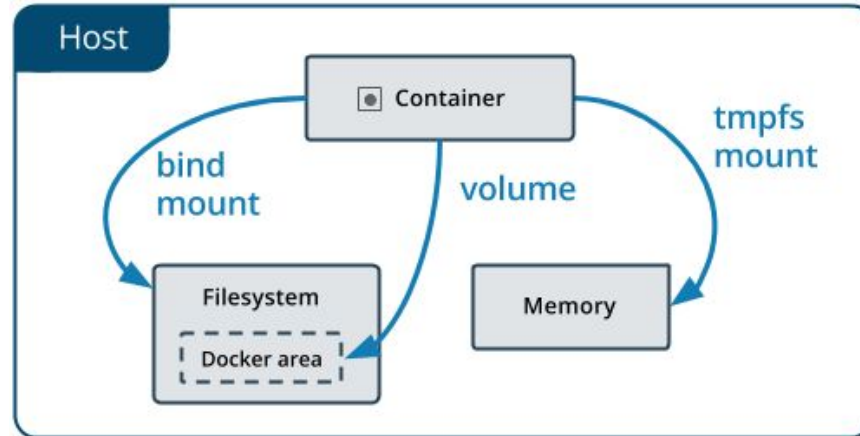
By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn't persist when that container no longer exists.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

Architecture - More objects - Storage

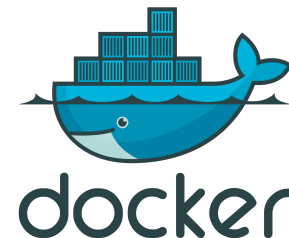


Docker has two options for containers to store files on the host machine, so that the files are persisted even after the container stops: volumes, and bind mounts.





Architecture - More objects - Storage

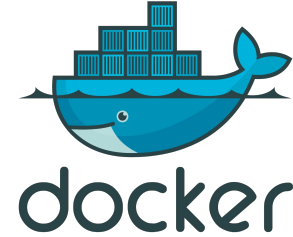


Volumes: The best way to persist data

- Stored in a part of the host filesystem which is managed by Docker.
- Created and managed by Docker although it can be created manually.
- When you mount the volume into a container, this directory is what is mounted into the container managed by Docker and isolated from the core functionality of the host machine.
- A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically.
- When you mount a volume, it may be named or anonymous.
- Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.



Architecture - More objects - Storage



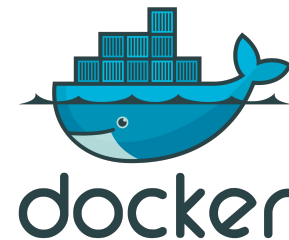
Bind mounts:

- 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.
- Limited functionality compared to volumes.
- When you use a bind mount, a file or directory on the host machine is mounted into a container.
- The file or directory is referenced by its full path on the host machine.
- The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist.
- Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available.
- Bind mounts allow access to sensitive files





Architecture - Docker objects - Networks



Networks: Connect containers and services together or with non-Docker workloads

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- bridge: The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- host: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.
- overlay: Overlay networks connect multiple Docker daemons together.
- ipvlan: IPvlan networks give users total control over both IPv4 and IPv6 addressing.
- macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
- none: For this container, disable all networking.
- network plugins: You can install and use third-party network plugins with Docker.



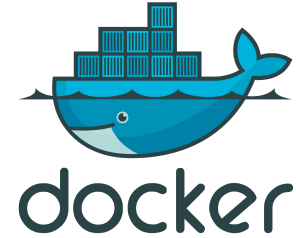
Installation

- For Macs: <https://docs.docker.com/desktop/mac/install/>
- For Windows: <https://docs.docker.com/desktop/windows/install/>
- For Linux: <https://docs.docker.com/desktop/linux/install/>



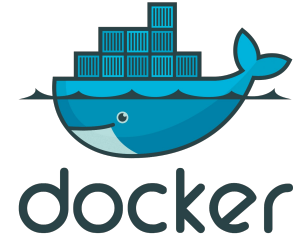
Creating and Using Containers

- Pre-existing Image
- Using Dockerfiles
- Using Docker Compose





Creating and Using Containers: Pre-existing Image

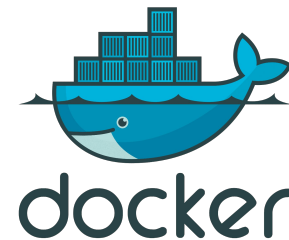


We can use images uploaded to Docker Hub, Docker Hub is a service provided by Docker for finding and sharing container images

```
docker run -d -p 80:80 docker/getting-started
```

- -d means detached, this way it will be executed in background and we will have our terminal free
- -p is used to map the port you will use outside the docker, with the port you will use inside the docker <outside_port>:<inside_port>

Creating and Using Containers: Pre-existing Image



```
~ via @base
→ docker run -d -p 80:80 docker/getting-started
Unable to find image 'docker/getting-started:latest' locally
latest: Pulling from docker/getting-started
df9b9388f04a: Pull complete
5867cba5fcdb: Pull complete
4b639e65cb3b: Pull complete
061ed9e2b976: Pull complete
bc19f3e8eeb1: Pull complete
4071be97c256: Pull complete
79b586f1a54b: Pull complete
0c9732f525d6: Pull complete
Digest: sha256:b558be874169471bd4e65bd6eac8c303b271a7ee8553ba47481b73b2bf597aee
Status: Downloaded newer image for docker/getting-started:latest
378be9a4239aa4b9fc45a653adbb7639877cdeecfdf0458b60b09219e81d320
```



NAME

STARTED

STATUS



docker/getting-started

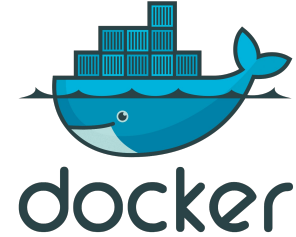
378be9a4239a (charming_mclaren)

less than a mi **running**





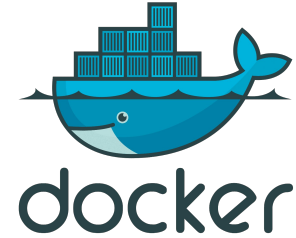
Creating and Using Containers: Dockerfile



A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Once we have the Dockerfile, we can use `docker build` to execute the Dockerfile and create our image.

Traditionally, the Dockerfile is called `Dockerfile` and located in the root of the context. You use the `-f` flag with `docker build` to point to a Dockerfile anywhere in your file system.

Creating and Using Containers: Dockerfile



```
Dockerfile

FROM python:3.9-slim

WORKDIR /example
COPY . /example

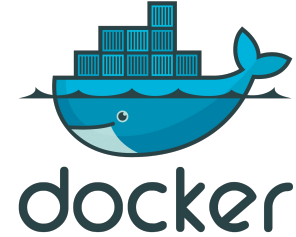
RUN apt-get update
RUN apt -y install build-essential
RUN pip install -r requirements.txt

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```



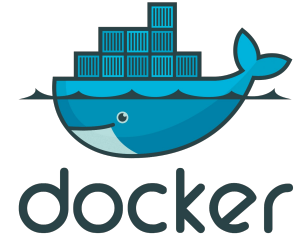


Creating and Using Containers: Dockerfile




- FROM: Base image we are using to build our image
- WORKDIR: Working directory we are using inside our container
- COPY: Copies our code into a folder inside the container
- RUN: Runs command when creating our image
 - In the example we use it to update the docker and install the required Python dependencies
- CMD: Runs a command when we run our image

Creating and Using Containers: Dockerfile



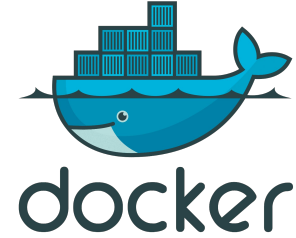
To build and run our Dockerfile:

- `docker build -t dockerfile_example .`
- `docker run --name dockerfile_example -p 80:80 dockerfile_example`

NAME		TAG	IMAGE ID	CREATED ↓	SIZE
dockerfile_example	IN USE	latest	cacaef801906	12 minutes ago	477.86 MB
<div> dockerfile_example dockerfile_exa... RUNNING PORT: 80</div>					



Creating and Using Containers: Dockerfile

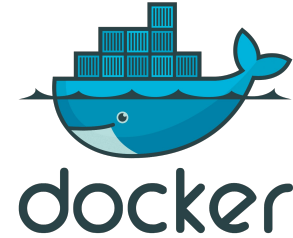


We can access to our container using a browser:

- <http://localhost:80/>



Creating and Using Containers: Docker Compose



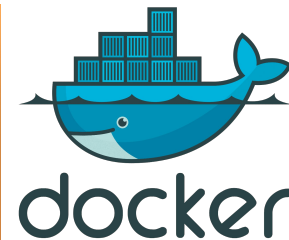
Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command.

The big advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repo (it's now version controlled), and easily enable someone else to contribute to your project.

Creating and Using Containers: Docker Compose

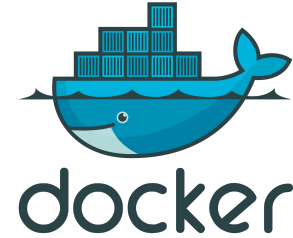
```
docker-compose.yml

version: '3.8'
services:
  db:
    container_name: pg_container
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root
      POSTGRES_DB: test_db
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
  pgadmin:
    container_name: pgadmin4_container
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: root
    ports:
      - "5050:80"
```



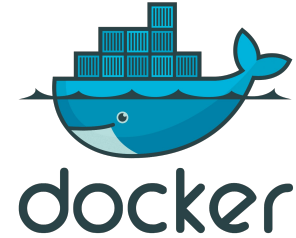


Creating and Using Containers: Docker Compose



- Services: Inside we will put all the services we need
- Each service will have a name it can be whatever we want, example: db, pgadmin...
- Container_name: We can add a name to our service container with
- Image: Name of the image we want to use for our services, if the image is present in our machine it will use it, otherwise will go to docker hub for it
- Restart: If our container goes down for any reason it will restart automatically
- Environment: To pass environment variables to our container if needed
- Ports: Ports we want to expose to access our service
- Volumes: To define volumes so we can persist our data

Creating and Using Containers: Docker Compose



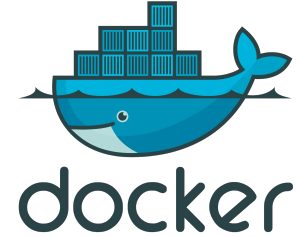
To build and run our docker-compose.yml:

- `docker-compose up -p <name>`
- `docker-compose -f <docker-compose-file> up` if you want to use an specific file

<input type="checkbox"/>	NAME	STARTED	STATUS	
<input type="checkbox"/>	desktop 2 containers		running (2/2)	Open
	dpage/pgadmin4 d5c1e453aef97 (pgadmin4_container)	less than a mi	running	
	postgres 5e9a1cfe4ed1 (pg_container)	less than a mi	running	



Creating and Using Containers: Docker Compose



To access our docker compose services we can use the same command as for the Dockerfile, but in this case we need to change the port, to access the pgadmin we need to use:

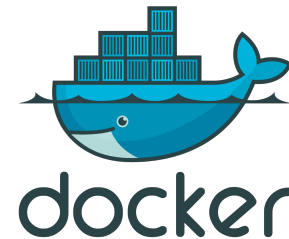
- <https://localhost:5050>

If we want to access to our db from outside docker, we can use:

- <https://localhost:5432>



Connection between Containers




The access to our containers have been always through a url, in this case localhost:<port> as we are running it in local.

But, what if we want to connect different containers? Docker gives us a simple solution, just use the container name. Docker will resolve the name as the proper connection string for us.

Connection between Containers

Welcome



pgAdmin

Management Tools for PostgreSQL

Feature rich | Maximises PostgreSQL | Open Source

pgAdmin is an Open Source administration and management tool for the PostgreSQL of developers, DBAs and system administrators alike.

Register - Server

General Connection SSL SSH Tunnel Advanced

Host name/address **pg_container**

Port 5432

Maintenance database postgres

Username root

Kerberos authentication? ☐

Password

Save password? ☐

Role

Service

Close Reset Save

```
docker-compose.yml

version: '3.8'
services:
  db:
    container_name: pg_container
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root
      POSTGRES_DB: test_db
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
  pgadmin:
    container_name: pgadmin4_container
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: root
    ports:
      - "5050:80"
```





More info at:

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

<https://docs.docker.com/storage/>

<https://docs.docker.com/network/>

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

<https://docs.docker.com/desktop/mac/install/>

<https://hub.docker.com/>

<https://github.com/JordiROP/ProgComm-III/tree/main/Docker/Examples>

