
taller de docker
COMISIÓN DE ESTUDIANTES DE
COMPUTACIÓN

Taller de Docker

21 de abril de 2026

DC · FCEyN · UBA

Motivación

¿Por qué aprender Docker?

- Herramienta clave para la vida profesional: reproducir entornos de desarrollo y producción sin dolores de cabeza.

Motivación

¿Por qué aprender Docker?

- Herramienta clave para la vida profesional: reproducir entornos de desarrollo y producción sin dolores de cabeza.
- No siempre es obligatoria en la currícula (aunque se esta empezando a incluir), pero es altamente esperada en la industria y un poco en investigación.

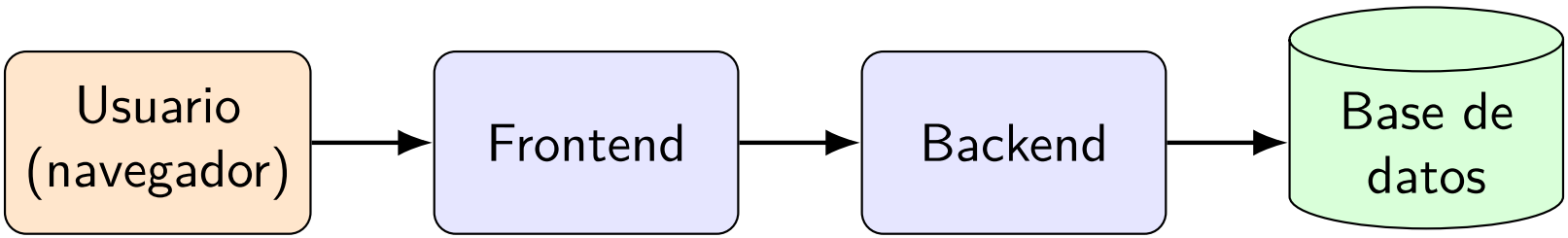
Motivación

¿Por qué aprender Docker?

- Herramienta clave para la vida profesional: reproducir entornos de desarrollo y producción sin dolores de cabeza.
- No siempre es obligatoria en la currícula (aunque se esta empezando a incluir), pero es altamente esperada en la industria y un poco en investigación.
- Nos ahorra tiempo: tareas repetitivas como instalar dependencias o configurar entornos pueden automatizarse fácilmente.

Arquitectura típica de una app web

¿Quién habla con quién?



- El **usuario** abre el navegador y entra a la página.
- El **frontend** muestra la interfaz y manda pedidos al backend.
- El **backend** procesa la lógica y pide/guarda datos.
- La **base de datos** persiste la información.

Frontend

Lo que ve el usuario

¿Qué es?

La parte de la aplicación que corre en el **navegador** y muestra la interfaz al usuario.

- Se encarga de mostrar botones, formularios, gráficos, etc.
- Captura la interacción del usuario (clicks, teclas) y la manda al backend.
- Tecnologías típicas: HTML, CSS, JavaScript, React, Vue, Angular.

Backend

El cerebro del lado del servidor

¿Qué es?

El **programa que corre en un servidor** y responde a los pedidos del frontend.

- Procesa la lógica de negocio (reglas, cálculos, autenticación).
- Habla con la base de datos para leer o guardar información.
- Tecnologías típicas: **Node.js**, Python (Flask, Django), Java (Spring), Ruby (Rails).

Base de datos

Donde queda guardada la info

¿Qué es?

Un sistema especializado en **almacenar y consultar información** de manera persistente.

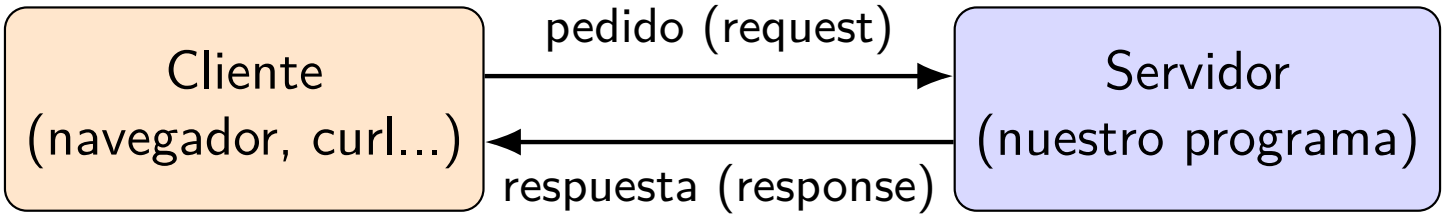
- Los datos sobreviven aunque el backend se reinicie.
- Ejemplos relacionales: **PostgreSQL**, MySQL, SQLite.
- Ejemplos no relacionales: MongoDB, **Redis** (cache, clave-valor).

¿Qué es un servidor?

El protagonista del ejemplo

Definición

Un **servidor** es un programa que está **siempre prendido**, esperando pedidos y respondiéndolos.



- No hace nada hasta que alguien lo "llama".
- Una misma máquina puede tener muchos servidores corriendo a la vez.

Ejemplo de Aplicación: Análisis de Imágenes Médicas

Objetivo

Sistema web que permite a profesionales médicos subir imágenes radiográficas y recibir un diagnóstico automático basado en un modelo de Deep Learning:

Backend

- Python (Flask, PyTorch, OpenCV, psycopg2)
- API para subir imágenes
- Procesamiento con modelo preentrenado
- Almacenamiento de metadatos en PostgreSQL

Otros Componentes

- **Base de datos (PostgreSQL)**: usuarios y resultados
- **Frontend (React + Axios)**: interfaz web
- **Redis**: cache de predicciones

Ejemplo de aplicacion: Requisitos

Python y librerías

- Python 3.10+ (con entorno virtual configurado)
- Flask (versiones compatibles con Python 3.10)
- PyTorch (versión que soporte la GPU/CPU del sistema)
- OpenCV (compilación con soporte de imágenes y video)
- psycopg2 (requiere librerías de PostgreSQL instaladas en el sistema)

Dependencias del sistema

- `gcc`, `g++`, `make`
- `libpq-dev`, `libjpeg-dev`, `zlib1g-dev`
- Configuración de drivers CUDA/cuDNN si se usa GPU

Ejemplo de aplicacion: Requisitos

Base de Datos

- PostgreSQL 14+ (instalación, usuarios, roles, configuración de puertos)
- Configuración de backups y permisos

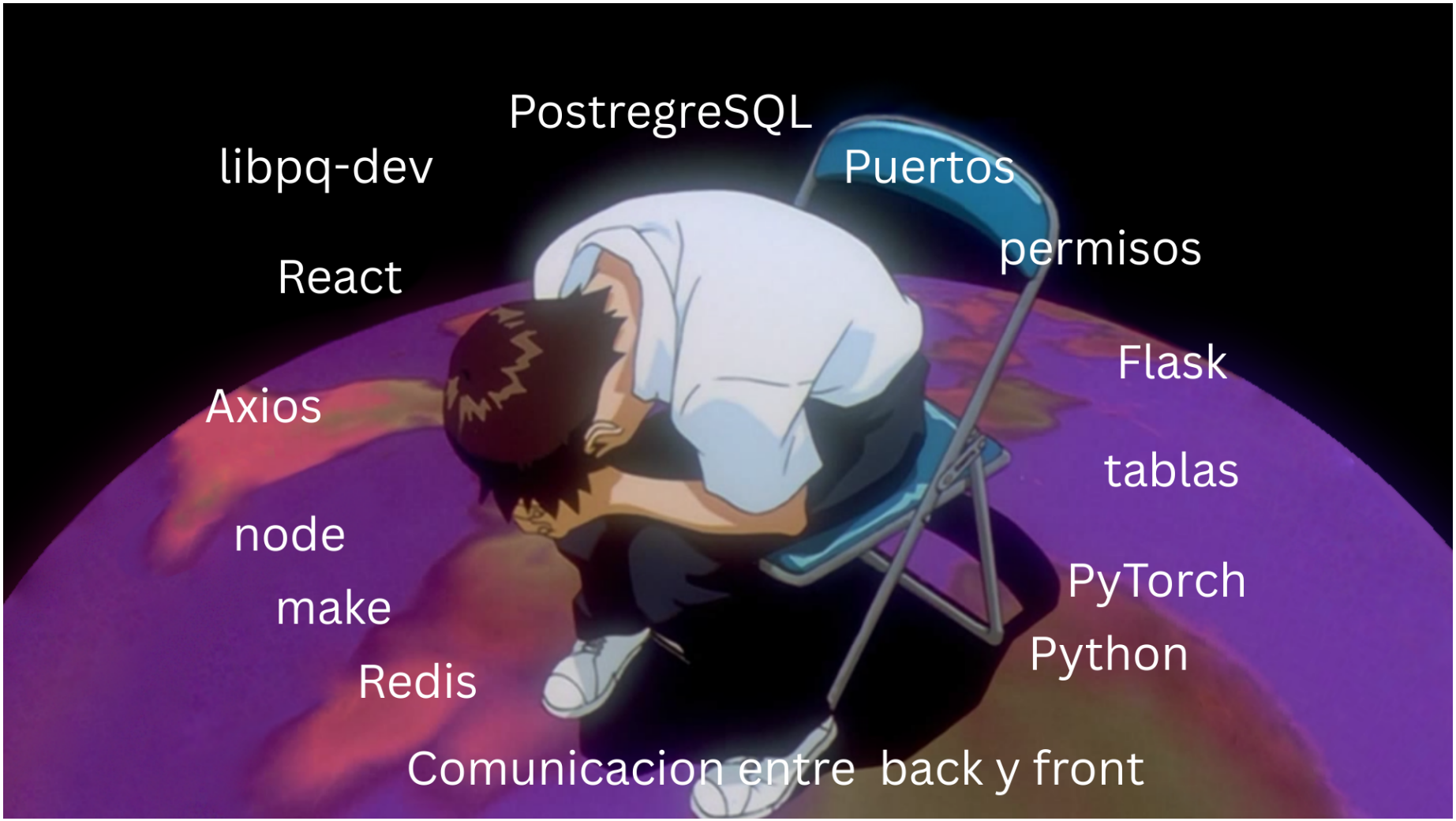
Frontend

- Node.js 18+ y npm
- Librerías de React y Axios (con dependencias propias)
- Posible conflicto de versiones entre proyectos

Cache

- Redis (instalación, configuración de persistencia y memoria)
- Gestión de puertos y compatibilidad con el backend

Ejemplo de aplicacion: Requisitos



Beneficios de Docker

Solución

- En lugar de instalar manualmente todas las dependencias en nuestra máquina, Docker permite levantar un entorno aislado con todo preconfigurado y listo para usar.
- Cada componente se ejecuta en su propio contenedor, aislado, y se comunica de manera controlada con los demás servicios.

Beneficios de Docker

Aislamiento

Cada componente corre en su propio contenedor, evitando conflictos de librerías y versiones.

Facilidad

Todo se inicia con un solo comando: sin instalaciones complejas.

Reproducibilidad

El entorno completo puede levantarse igual en cualquier máquina, garantizando que funcione siempre.

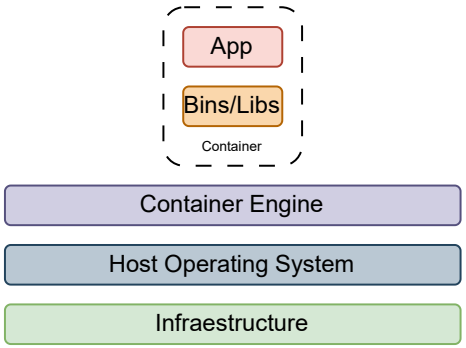
Escalabilidad

Contenedores separados permiten escalar y migrar servicios fácilmente.

Contenedores

¿Qué es un contenedor?

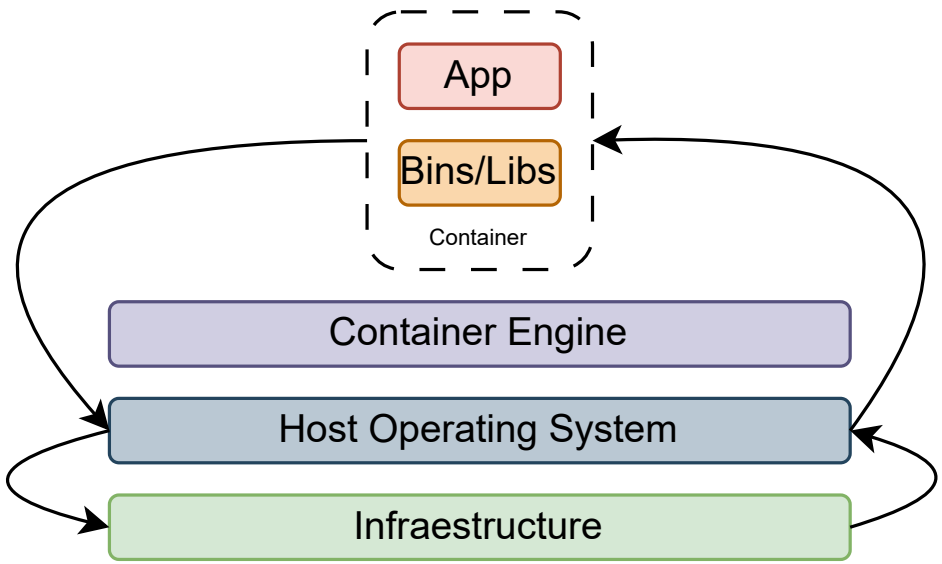
- Un contenedor es un **proceso** que corre aislado del resto del sistema, con la ilusión de tener su propio filesystem, red y lista de procesos.
- Analogía: un *departamento dentro de un edificio*. Comparte la estructura (el kernel del host) con los otros departamentos, pero cada uno tiene su espacio propio y no ve a los demás.
- Se arma arrancando un proceso común y pidiéndole al kernel que lo aísle con ciertas funciones.



Contenedores

Contenedor vs máquina virtual

- Una **VM** virtualiza una computadora entera: trae su propio kernel, su propio SO, emula hardware. Pesada y lenta de arrancar.
- Un **contenedor** reusa el kernel del host. No emula hardware ni arranca otro SO. Mucho más liviano y rápido.
- Por eso podés tener decenas de contenedores donde apenas entrarían unas pocas VMs.



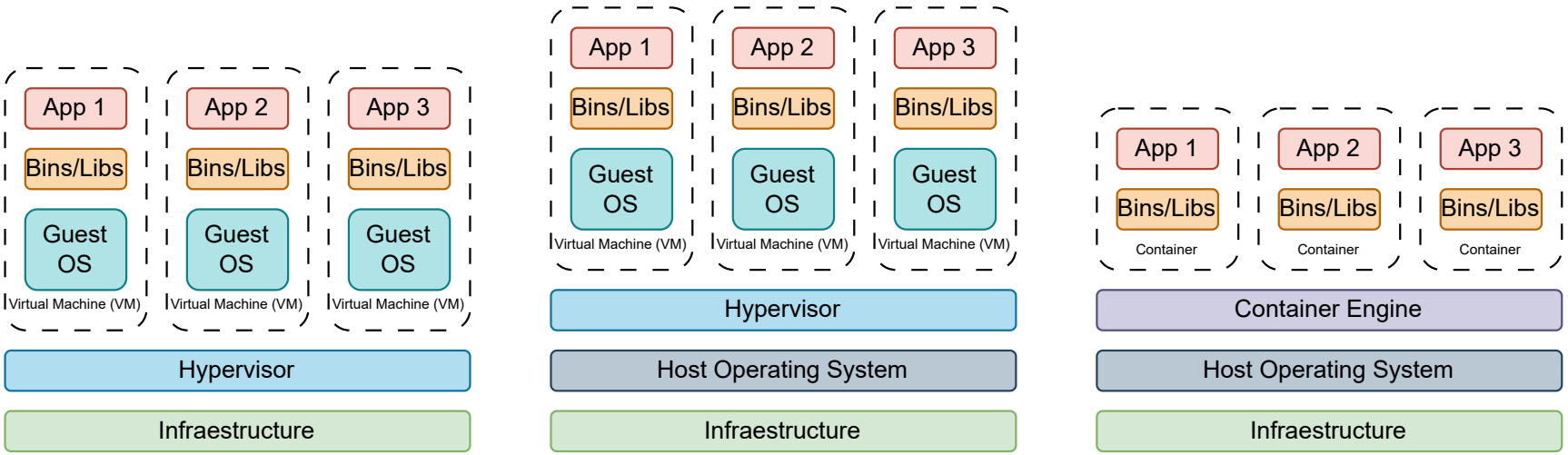
Contenedores

¿Cómo logra el aislamiento? (Linux)

- **Namespaces** $\rightarrow$ aíslan lo que el contenedor *ve*: procesos, red, usuarios, montajes. Dentro del contenedor no aparecen los procesos del host.
- **Cgroups** $\rightarrow$ limitan lo que el contenedor *consume*: CPU, memoria, disco, red. Evitan que un contenedor se coma toda la máquina.
- **OverlayFS** $\rightarrow$ permite armar filesystems en **capas** apiladas. Varias imágenes pueden compartir capas base y ahorrar espacio.

Contenedores

Vista general



Imágenes

¿Qué es una imagen de Docker?

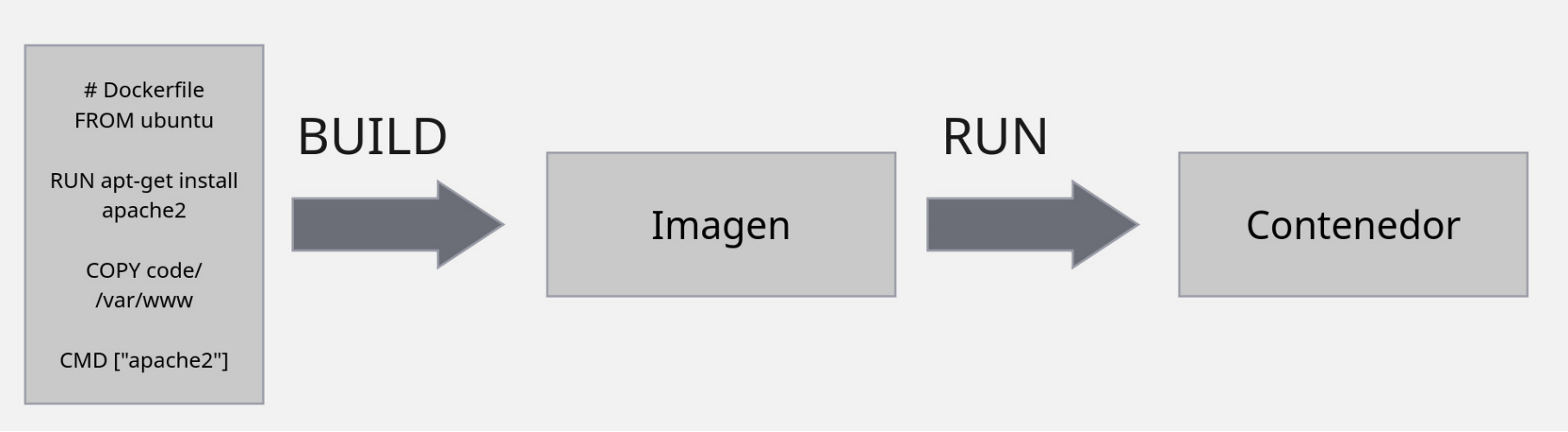
- Una **imagen** es un archivo (un paquete) que contiene **todo lo necesario** para correr una aplicación: código, librerías, binarios del sistema, dependencias y configuración.
- Es **estática**: no corre por sí sola. Es la "plantilla" a partir de la cual se crean los contenedores.
- Un **contenedor** es la **ejecución activa** de una imagen. De una sola imagen podés lanzar muchos contenedores.

Una imagen de Docker contiene típicamente

- Una base del sistema (ej: Debian, Alpine).
- Software y runtimes (ej: Python, Node).
- Tu aplicación y sus dependencias.

Imágenes

Definición



Contenedores

- Escribimos un dockerfile que es un recetario para crear una imagen.
- Generamos un build para generar la imagen.
- Un contenedor es la ejecucion activa de una imagen.

¿Cómo podemos compartir nuestras imágenes?

Registry

Registry de contenedores

Un **registry** es un servicio donde podemos:

- Almacenar imágenes de contenedores.
- Compartirlas con otros usuarios o equipos.
- Versionarlas y administrarlas de forma centralizada.

Así como un repositorio de código guarda programas, un registry guarda las imágenes listas para ejecutar.

Ejemplos de registries de imágenes

El rol de Docker Hub

- **Docker Hub**: el más popular.

Ejemplos de registries de imágenes

El rol de Docker Hub

- **Docker Hub**: el más popular.
- GitHub Container Registry (GHCR)
- GitLab Container Registry
- Amazon Elastic Container Registry (ECR)
- Google Artifact/Container Registry (GAR/GCR)
- Azure Container Registry (ACR)
- Quay.io, Harbor, JFrog Artifactory

Enfoque

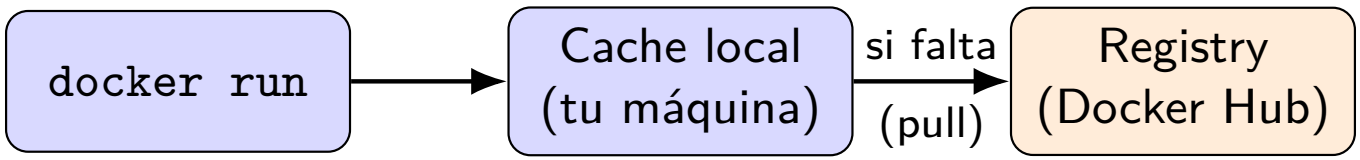
Aunque existen muchas opciones, **Docker Hub es el estándar de facto** y el que usaremos en este taller.

¿Cómo encuentra Docker las imágenes?

Autenticación y flujo de búsqueda

`docker login`

Autentica tu máquina contra un registry (por defecto Docker Hub). Hace falta para **subir** imágenes propias o para **bajar** de repositorios privados. Las imágenes públicas se pueden descargar sin loguearse.



1. Docker busca la imagen en el **cache local** (las que ya descargaste).
2. Si no está, la **descarga (pull)** del registry y la guarda en el cache.
3. **Ejecuta** el contenedor a partir de la imagen local.

La primera corrida tarda más porque se baja la imagen; las siguientes son inmediatas porque ya está cacheada.

Motor de contenedores

Almacenamiento de imágenes

- docker run <image_name> # *Crea y ejecuta un contenedor a partir de una imagen.*
- docker start|stop <container_name> # *Inicia o para un container.*
- docker rm <container_name> # *Remueve el contenedor.*
- docker ps # *Lista los contenedores que están ejecutando.*
- docker container stats # *Muestra el estado del uso de los recursos.*
- docker exec <container_name> <command> # *Ejecuta el comando dentro del contenedor.*
- docker logs <container_name> # *Muestra los logs del contenedor.*

[0] **https://docs.docker.com/reference/cli/docker/**

¿Qué es Node.js?

JavaScript fuera del navegador

Definición

Node.js es un entorno que permite ejecutar **JavaScript** afuera del navegador, típicamente para escribir **servidores** y herramientas de línea de comandos.

- Históricamente, JS solo corría dentro del navegador (frontend).
- Node trajo JS al backend: el mismo lenguaje en cliente y servidor.
- Viene con `npm`, su gestor de paquetes (librerías de terceros).
- En nuestro ejemplo usamos Node para levantar un servidor HTTP mínimo.

¿Qué es un puerto? (1/2)

Una analogía

La idea

Una computadora tiene **una sola dirección de red** (la IP), pero puede tener **muchos servicios** corriendo al mismo tiempo. ¿Cómo se distinguen?

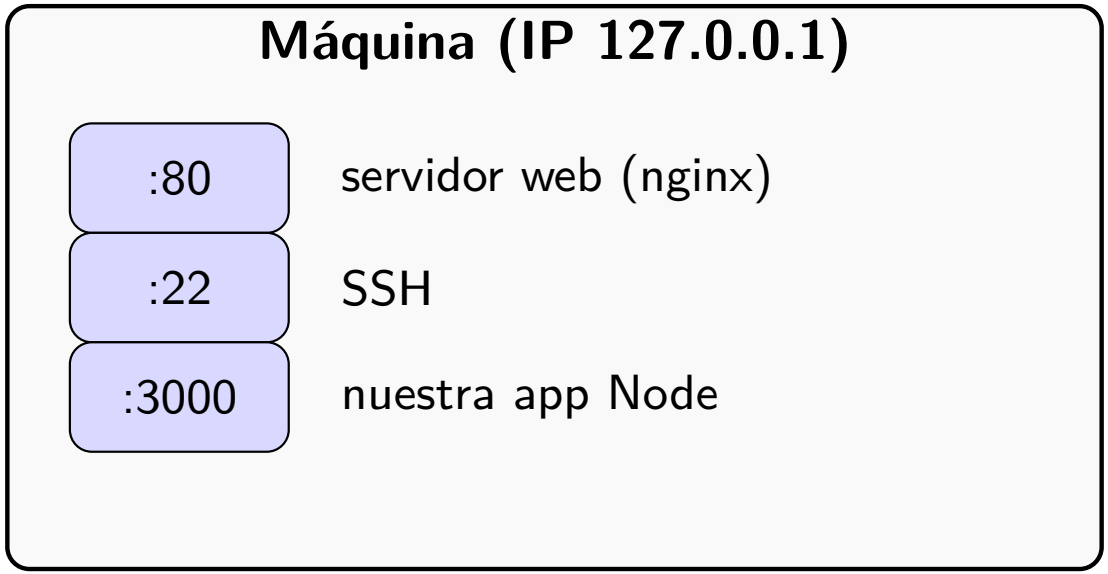
Analogía del edificio

- La **IP** es la dirección del edificio (*Av. Corrientes 1234*).
- El **puerto** es el número de departamento (*Depto 3000*).
- Para visitar a alguien no alcanza con la dirección del edificio: necesitás también el departamento.

Cada servicio "vive" en un puerto distinto de la misma máquina.

¿Qué es un puerto? (2/2)

Varios servicios en la misma máquina



Puertos típicos

- **80** → HTTP, **443** → HTTPS, **22** → SSH, **5432** → PostgreSQL.
- En desarrollo se suelen usar **3000**, **5000**, **8080** (no requieren permisos de root).

Dockerfile

Comandos

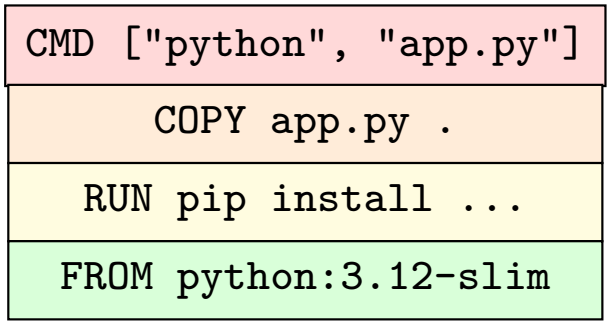
- FROM: Crear una nueva etapa de compilación a partir de una imagen base.
- WORKDIR: Cambiar directorio de trabajo.
- RUN: Ejecutar comandos de compilación.
- COPY: Copiar archivos y directorios.
- EXPOSE: Describe en qué puertos está escuchando tu aplicación.
- CMD: Especificar comandos predeterminados.
- ...

[0] **https://docs.docker.com/reference/dockerfile/**

Imágenes por capas (layers)

Cómo se construyen y reutilizan

Una imagen no es un archivo único: es una **pila de capas**. Cada instrucción del Dockerfile (`FROM`, `RUN`, `COPY`...) genera una capa sobre la anterior.



- Cada capa es **read-only**, identificada por un hash.
- Se **comparten entre imágenes**: la misma base se guarda una sola vez.
- Docker **cachea** las capas: si una instrucción no cambió, la reutiliza.
- **Tip**: poné lo que cambia poco (dependencias) antes que lo que cambia seguido (`COPY` del código).

Motor de contenedores

Puertos

- Los contenedores tambien manejan su propia red interna.
- Si queremos comunicarnos con el contenedor, podemos exponer puertos hacia nuestro host
- Una de las maneras de hacerlo es con "-p" en el comando de docker run.

¿Cómo funcionan los volúmenes en Docker?

Persistencia de datos

Definición

Un **volumen** es un mecanismo de Docker para almacenar datos fuera del ciclo de vida de un contenedor. Sirven para que la información **no se pierda** al detener o eliminar un contenedor.

- Los datos se almacenan en el host (por defecto en `/var/lib/docker/volumes/`).
- Se pueden compartir entre varios contenedores.
- Separan la **aplicación** (contenedor) de sus **datos**.

¿Cómo funcionan los volúmenes en Docker?

Persistencia de datos

Definición

Un **volumen** es un mecanismo de Docker para almacenar datos fuera del ciclo de vida de un contenedor. Sirven para que la información **no se pierda** al detener o eliminar un contenedor.

- Los datos se almacenan en el host (por defecto en `/var/lib/docker/volumes/`).
- Se pueden compartir entre varios contenedores.
- Separan la **aplicación** (contenedor) de sus **datos**.

Ejemplo

```
docker run -v  /data:/app/data myapp
```

- `/data` → carpeta en el host.
- `/app/data` → carpeta dentro del contenedor.

Variables de entorno en Docker

Parametrización de contenedores

¿Qué es una variable de entorno?

Una **variable de entorno** es un par `NOMBRE=VALOR` que se almacena en el sistema operativo y que las aplicaciones pueden leer para modificar su comportamiento sin cambiar el código.

- Permiten parametrizar la configuración de programas.
- Ejemplo típico: base de datos, puerto, modo debug, usuario, contraseña.
- Se pueden definir al crear un contenedor y son accesibles desde dentro del mismo.

Variables de entorno en Docker

Parametrización de contenedores

¿Qué es una variable de entorno?

Una **variable de entorno** es un par `NOMBRE=VALOR` que se almacena en el sistema operativo y que las aplicaciones pueden leer para modificar su comportamiento sin cambiar el código.

- Permiten parametrizar la configuración de programas.
- Ejemplo típico: base de datos, puerto, modo debug, usuario, contraseña.
- Se pueden definir al crear un contenedor y son accesibles desde dentro del mismo.

Uso en Docker

`docker run -e NOMBRE_VAR=valor imagen`

Ejemplo: `docker run -e MYSQL_ROOT_PASSWORD=secret mysql`

- `MYSQL_ROOT_PASSWORD` es la variable de entorno.
- `secret` es el valor que se le asigna.

¿Para qué sirven las variables de entorno en Docker?

Parametrización de contenedores por el usuario

Idea principal

Las variables de entorno permiten al usuario **configurar un contenedor** sin necesidad de modificar la imagen ni el código de la aplicación.

- Configurar contraseñas, nombres de usuario, puertos o rutas de archivos.
- Activar o desactivar modos de operación (ej. modo debug).
- Permiten que la misma imagen funcione en distintos entornos (desarrollo, testing, producción) solo cambiando las variables.

¿Qué es Docker Compose?

Automatización de contenedores

Definición

Docker Compose es una herramienta que permite definir y ejecutar aplicaciones multicontenedor mediante un archivo de configuración (`docker-compose.yml`).

- Permite describir contenedores, redes, volúmenes y variables de entorno de forma declarativa.
- Sustituye la necesidad de escribir múltiples comandos `docker run` a mano.
- Facilita levantar, detener y escalar aplicaciones completas con un solo comando: `docker-compose up` y `docker-compose down`.
- Mantiene toda la configuración de la aplicación en un solo archivo, fácil de versionar y compartir.

¿Cómo funcionan los networks en Docker?

Conexión entre contenedores

Definición

Un **network** en Docker es una red virtual que conecta contenedores entre sí y con el host, de manera aislada y controlada.

- **bridge** (por defecto): cada contenedor recibe una IP propia y puede comunicarse con otros usando **DNS interno de Docker**, es decir, por **nombre de contenedor** en lugar de IP.
- **host**: el contenedor comparte directamente la red del host (no tiene IP separada).
- **none**: sin acceso a red, solo `localhost`.

¿Cómo funcionan los networks en Docker?

Conexión entre contenedores

Definición

Un **network** en Docker es una red virtual que conecta contenedores entre sí y con el host, de manera aislada y controlada.

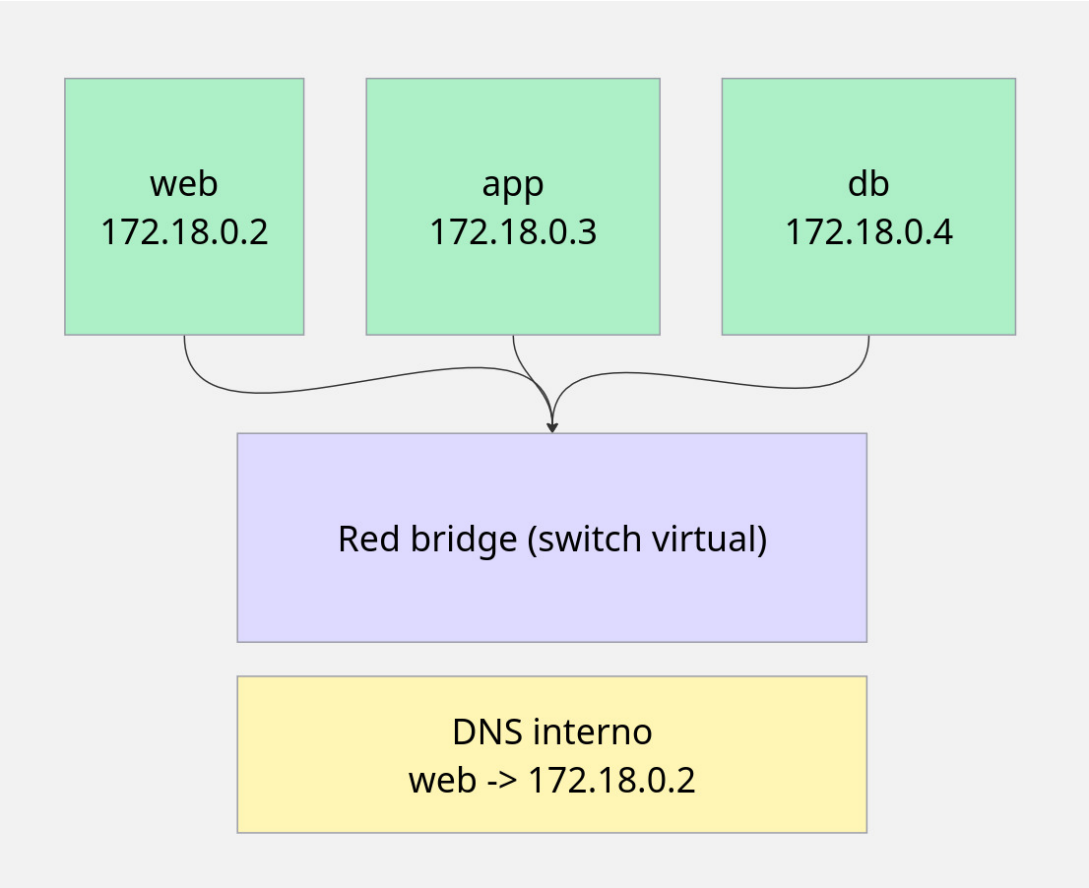
- **bridge** (por defecto): cada contenedor recibe una IP propia y puede comunicarse con otros usando **DNS interno de Docker**, es decir, por **nombre de contenedor** en lugar de IP.
- **host**: el contenedor comparte directamente la red del host (no tiene IP separada).
- **none**: sin acceso a red, solo `localhost`.

Ejemplo

```
docker network create mi_red
docker run -d --name web --network mi_red nginx
docker run -it --rm --network mi_red alpine ping web
```

¿Cómo funcionan los networks en Docker?

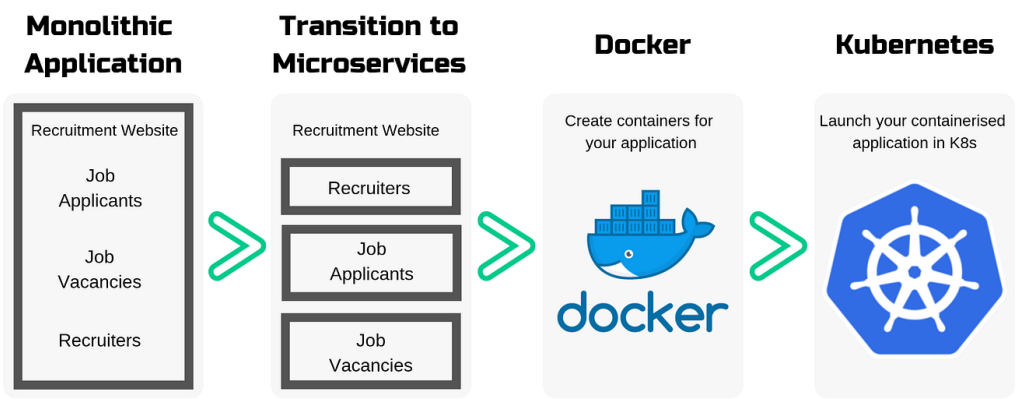
Conexión entre contenedores



Orquestación de contenedores: siguiente paso después de Docker Compose

- **¿Qué es la orquestación de contenedores?**
  - Gestión automatizada de múltiples contenedores en uno o varios servidores.
  - Asegura que los contenedores funcionen correctamente, se comuniquen y escalen según demanda.
- **Por qué es necesaria**
  - Docker Compose funciona localmente, pero tiene limitaciones al distribuir servicios en varios nodos.
  - Necesidad de recuperación automática, escalado y balanceo de carga.
  - Gestión centralizada de redes y almacenamiento persistente.
- **Qué hace la orquestación**
  - Automatiza la ejecución de contenedores.
  - Monitorea y reinicia contenedores que fallan.
  - Escala servicios según la demanda.
  - Balancea carga entre instancias.
  - Gestiona almacenamiento y redes de manera declarativa.

Orquestación de contenedores: siguiente paso después de Docker Compose



Bibliografía

- Documentación de Docker: https://docs.docker.com/reference/