

docker

DOCKER

INTRODUCCIÓN A DOCKER

1

DOCKER



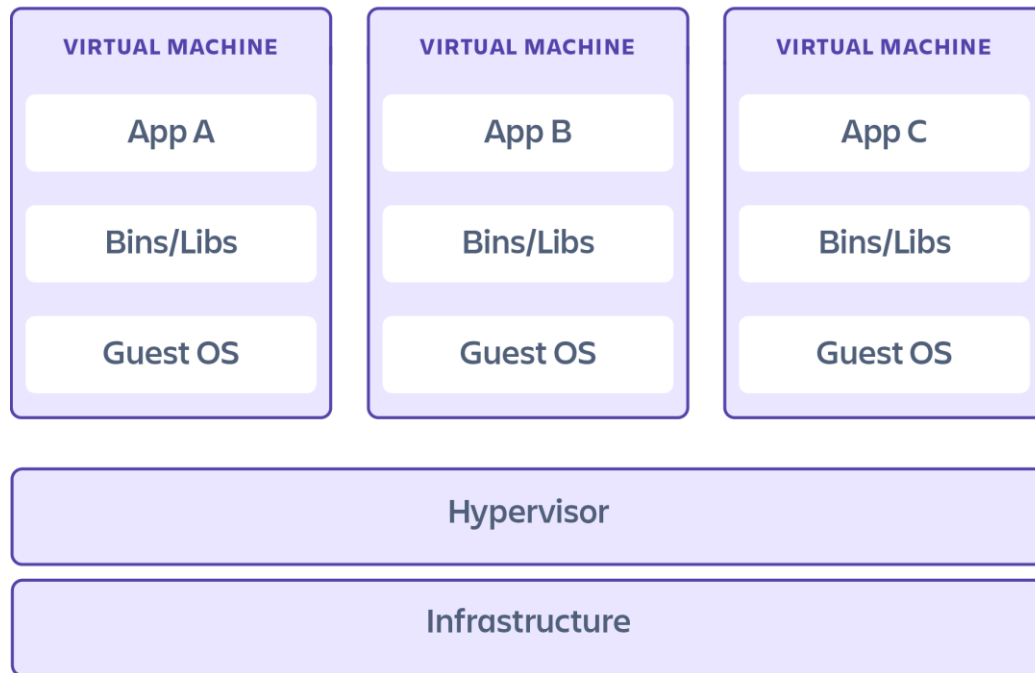
- ¿Qué es Docker?
- Docker es una plataforma de software que permite desarrollar, enviar y ejecutar aplicaciones dentro de contenedores. Un contenedor es una unidad ligera, portable y autosuficiente que incluye todo lo necesario para que una aplicación se ejecute: código, bibliotecas, dependencias, herramientas del sistema, etc.
- **Ventajas del de uso de contenedores:**
 - **Estandarizar entornos:** evita el clásico "en mi máquina funciona" porque el contenedor se ejecuta igual en cualquier lugar (local, servidor, nube).
 - **Aislamiento:** cada contenedor corre de forma independiente, sin interferir con otros.
 - **Portabilidad:** puedes mover contenedores entre diferentes sistemas fácilmente.
 - **Escalabilidad:** es ideal para arquitecturas como microservicios, donde cada componente corre en su propio contenedor.

DOCKER

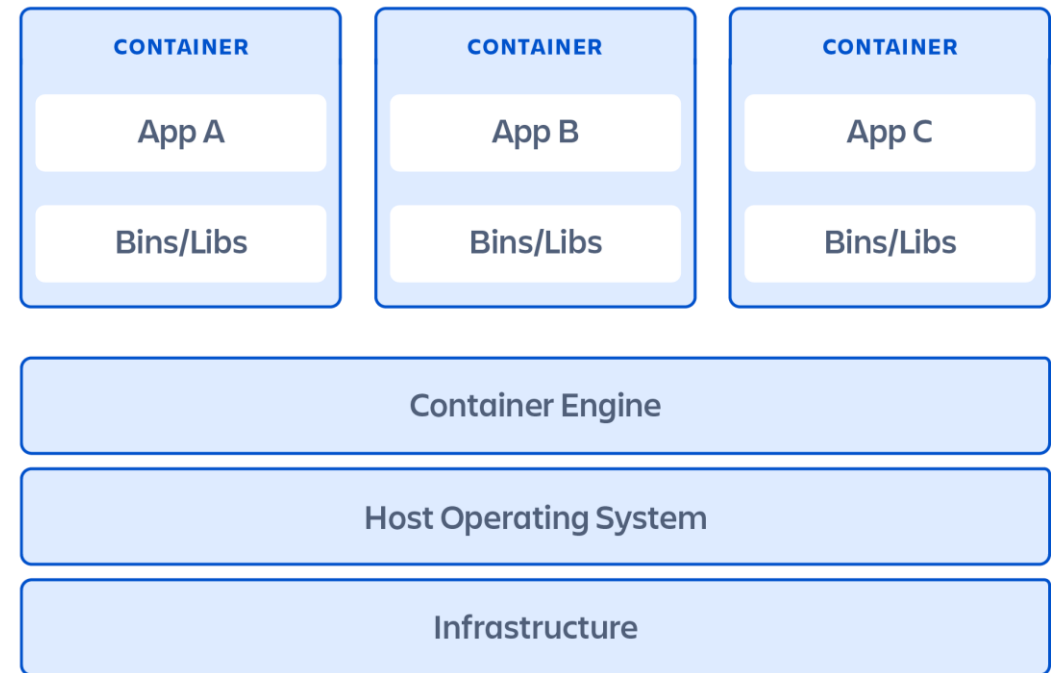


- Contenedores vs Virtualización

Virtual machines



Containers



<https://www.atlassian.com/es/microservices/cloud-computing/containers-vs-vms>

DOCKER



Componentes

■ Componentes:

- **Docker Engine:** Es el núcleo de Docker. Es un servicio cliente-servidor compuesto por:
 - **Docker Daemon** (dockerd): se ejecuta en segundo plano y gestiona contenedores, imágenes, redes, volúmenes, etc.
 - **Docker CLI** (docker): herramienta de línea de comandos para interactuar con el daemon.
 - **REST API:** interfaz para comunicar programas externos con el daemon.
- **Imágenes Docker:** Plantillas de solo lectura que definen el entorno y la aplicación que se ejecutará dentro del contenedor.
- **Contenedores Docker:** Instancias ejecutables de imágenes. Son aislados, portables y se destruyen fácilmente sin afectar el sistema anfitrión.
- **Dockerfile:** Archivo de texto que contiene instrucciones para construir una imagen Docker personalizada.
- **Docker Hub (o Docker Registry):** Repositorio en la nube donde puedes subir (push) o descargar (pull) imágenes Docker. Docker Hub es el registro público por defecto.

■ Componentes:

- **Volúmenes:** Mecanismo para almacenar datos persistentes y compartir información entre contenedores o entre el host y los contenedores.
- **Redes Docker:** Permiten la comunicación entre contenedores y con el exterior. Docker crea una red "bridge" por defecto, pero puedes definir redes personalizadas.
- **Docker Compose:** Herramienta para definir y correr múltiples contenedores con un solo archivo (docker-compose.yml). Muy útil para arquitecturas de microservicios.
- **Docker Swarm (opcional):** Sistema de orquestación nativo de Docker para desplegar y gestionar clústeres de contenedores.
- **Docker Desktop:** Aplicación para Windows y macOS que incluye Docker Engine, Docker CLI, Docker Compose, y una interfaz gráfica.
- **Plugins:** son componentes adicionales que extienden la funcionalidad del motor de Docker.

■ Dockerfile:

- Un Dockerfile es un archivo de texto que contiene instrucciones que Docker usa para construir una imagen personalizada. Este archivo define el paso a paso para crear una imagen Docker que contenga una aplicación o servicio con todas las dependencias necesarias. Automatiza la construcción de imágenes Docker, asegurando que se puede replicar el mismo entorno en cualquier lugar donde se ejecute Docker.
- Estructura :
 - FROM: Define la imagen base sobre la que se construirá la nueva imagen.
 - ENV: Asigna variables de entorno.
 - RUN: Ejecuta comandos dentro de la imagen durante el proceso de construcción (por ejemplo, instalar paquetes).
 - COPY o ADD: Copia archivos desde tu máquina local al contenedor.
 - WORKDIR: Define el directorio de trabajo donde se ejecutarán los comandos posteriores.
 - CMD o ENTRYPOINT: Define el comando que se ejecutará cuando el contenedor se inicie.

DOCKER



- **Dockerfile:**

- **Ejemplo:**

```
# Usa una imagen base  
FROM postgres:17
```

```
# Variables de entorno para la configuración inicial  
ENV POSTGRES_USER=admin  
ENV POSTGRES_PASSWORD=admin
```

```
# Copiar archivos SQL de inicialización (opcional)  
COPY ./init.sql /docker-entrypoint-initdb.d/
```


DOCKER



■ Componentes:

OBJETOS DE DOCKER

Ejemplo de comandos configuración básico de un archivo Dockerfile:

```

Docker
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install --production
COPY . .
CMD ["npm", "start"]
EXPOSE 3000

```

DOCKERFILE

- FROM: Define la imagen base desde la cual se construirá la nueva imagen.
- WORKDIR: Cambia el directorio de trabajo.
- COPY: Se utiliza para copiar archivos o directorios desde el host al sistema de archivos del contenedor.
- RUN: Se emplea para ejecutar comandos al momento de la construcción de la imagen.
- CMD: Especifica el comando por defecto que se ejecutará cuando se inicie el contenedor.
- EXPOSE: Describe en qué puertos está escuchando la aplicación.

IMÁGENES

Existen dos principios de las imágenes en Docker:

1. **Inmutables.** Al momento que se crea una imagen, no se puede modificar. Solo se puede crear una nueva imagen o agregarle cambios.
2. **Se componen de capas.** Cada capa simboliza un conjunto de cambios en el sistema de archivos que permite agregar, eliminar o modificar archivos.



CONTENEDORES

- Se ejecutan de forma aislada, separados de otros contenedores y de la máquina host.
- Se pueden ejecutar en cualquier máquina sin importar el sistema operativo.
- Los contenedores comparten el mismo kernel del sistema operativo anfitrión.

RED

- Soporte de políticas de seguridad para controlar el tráfico de red.
- La integración de un servidor DNS interno.
- Diferentes hosts físicos se pueden comunicar entre sí como si estuvieran en la misma red local.



VOLÚMENES

Cuenta con dos tipos de Montajes:

- **Montaje Volumen:** Docker gestiona completamente el volumen, y es el más común y recomendado para la mayoría de los casos de uso.
- **Montaje Bind:** Mapea un directorio o archivo del host a un contenedor, ofreciendo mayor flexibilidad.

PLUGINS

Los plugins permiten ampliar las capacidades de Docker fuera de sus funciones predeterminadas, estos permiten añadir nuevas funcionalidades que no están incluidas en la instalación estándar de Docker.



DOCKER



■ Comandos principales:

■ Información General

- **docker version** → Muestra las versiones del cliente y servidor de Docker instaladas en el sistema.
- **docker info** → Muestra detalles del entorno Docker: número de contenedores, imágenes, sistema operativo, drivers, recursos, etc.

■ Contenedores

- **docker ps** → Lista los contenedores en ejecución. Agrega -a para ver todos, incluyendo los detenidos.
- **docker stop [contenedor]** → Detiene un contenedor en ejecución de forma ordenada.
- **docker exec -it [contenedor] [comando]** → Ejecuta un comando dentro de un contenedor en ejecución. Comúnmente usado con /bin/bash para abrir una terminal interactiva.
- **docker rm [contenedor]** → Elimina un contenedor detenido. Usa -f para forzar la eliminación de uno en ejecución.
- **docker start [contenedor]** → Reinicia un contenedor ya creado (previamente detenido).

■ Comandos principales:

■ Imágenes

- `docker pull [imagen]` → Descarga una imagen desde Docker Hub (o un repositorio configurado) al sistema local.
- `docker run [imagen]` → Crea y ejecuta un contenedor a partir de una imagen. Si la imagen no está disponible localmente, Docker hace automáticamente un pull.
- `docker rmi [imagen]` → Elimina una imagen del sistema local. Solo si no está siendo usada por algún contenedor.
- `docker images` → Muestra una lista de las imágenes locales.
- `docker image ls` → Mismo resultado que `docker images`; muestra las imágenes almacenadas localmente.
- `docker build -t [nombre] .` → Construye una imagen a partir de un Dockerfile ubicado en el directorio actual (.). Usa -t para etiquetar la imagen con un nombre.
- `docker tag [imagen] [nuevo-nombre]` → Asigna una nueva etiqueta (nombre) a una imagen existente.
- `docker search [término]` → Busca imágenes públicas en Docker Hub con una palabra clave.
- `docker history [imagen]` → Muestra el historial de capas que componen una imagen (comandos usados para construirla).

■ Comandos principales:

■ Registros

- `docker logs [contenedor]` → Muestra los logs (salida estándar) de un contenedor.
- `docker logs -f [contenedor]` → Sigue mostrando los logs en tiempo real (modo follow, útil para monitoreo).

■ Guardado y Carga

- `docker save -o archivo.tar [imagen]` → Guarda una imagen en un archivo .tar. Útil para transferir entre sistemas sin conexión.
- `docker load -i archivo.tar` → Carga una imagen desde un archivo .tar previamente guardado con save.

■ Docker Compose

- `docker-compose up` → Levanta y ejecuta todos los servicios definidos en un archivo docker-compose.yml.

■ Autenticación

- `docker login` → Inicia sesión en Docker Hub (o en un registro privado) para poder hacer push de imágenes.

Contenedores – Primeros pasos

■ Nombres de los contenedores:

- Solo se permiten letras (a-z), números (0-9), guiones (-) y guiones bajos (_).
- No puede contener espacios ni caracteres especiales: !, /, @, #, etc.
- Debe comenzar con una letra o número.
- Es sensible a mayúsculas/minúsculas (MiContenedor ≠ micontenedor).
- Máximo recomendado: 255 caracteres.

■ Algunos nombres:

- postgres-db
- web_server_01
- app-backend
- myapp-db
- test_env_container

DOCKER



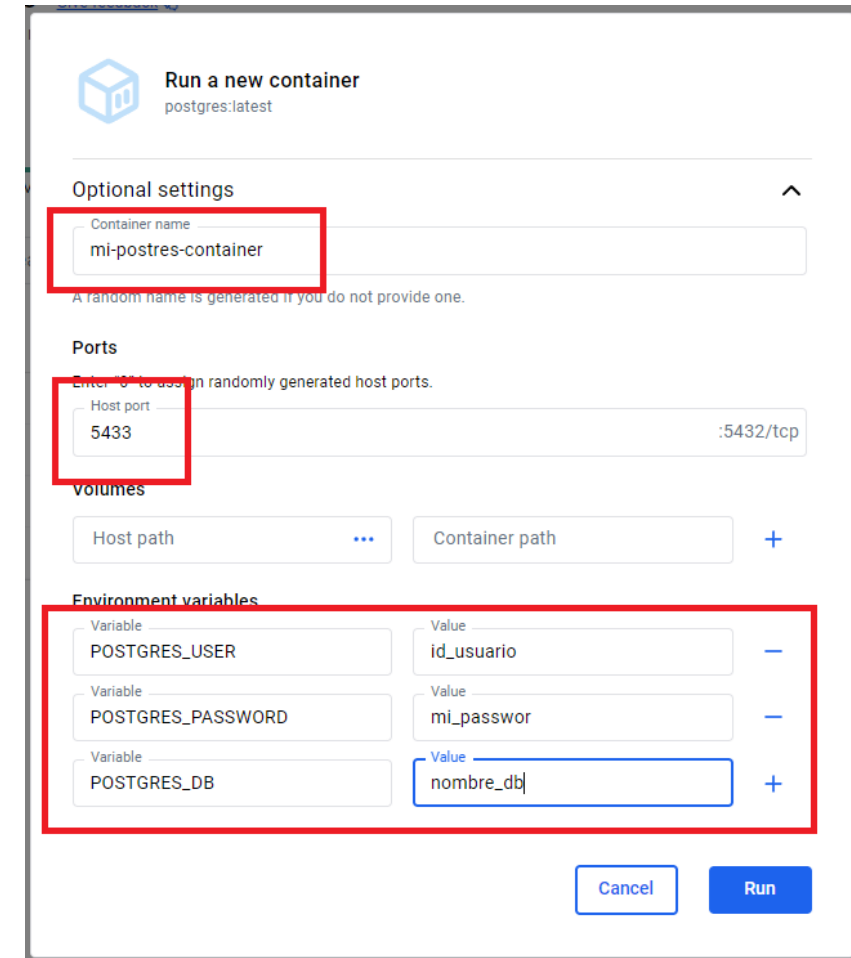
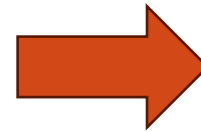
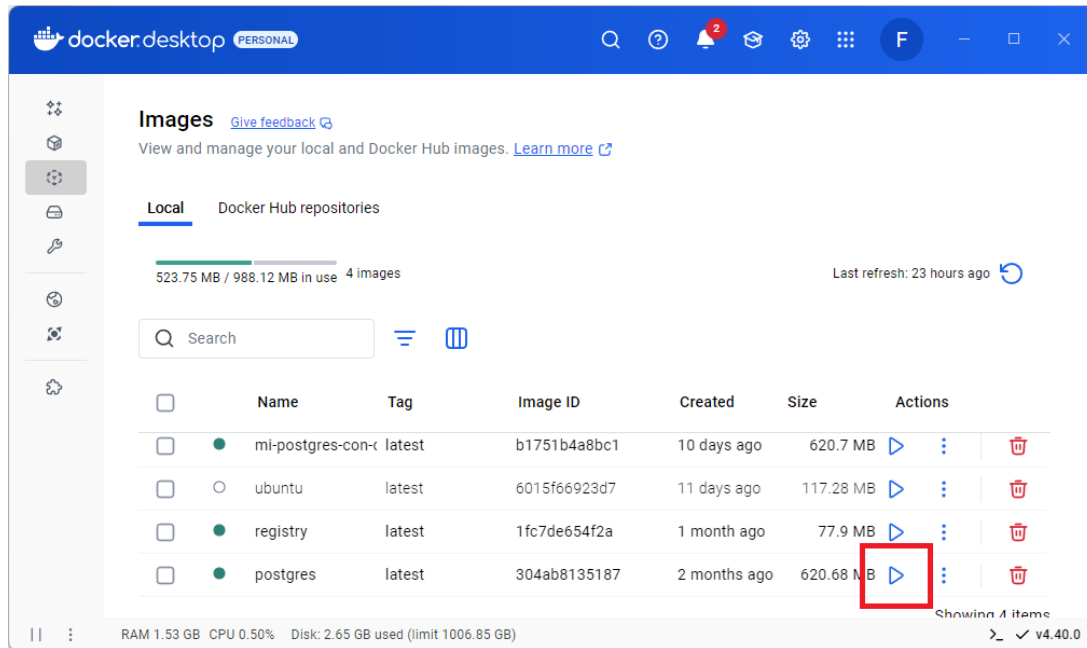
- Crear un contenedor a partir de una imagen descargada.
 - Tener arrancado el motor de Docker (*Docker engine*). Esto se consigue arrancando la aplicación Docker previamente instalada.
 - Descargar la imagen con el comando **docker pull**.
 - **docker pull postgres** → descarga la última versión de la imagen indicada.
 - Comprobar que la imagen está descargada en local.
 - **docker images**

- Crear un contenedor a partir de una imagen descargada.
 - Crear un contenedor con **docker run**. Sintaxis:
 - **docker run**
 - --name <container_name>
 - -e → Variables de entorno dentro del contenedor
 - POSTGRES_USER=<db_user>
 - -e
 - POSTGRES_PASSWORD=<db_password>
 - -e
 - POSTGRES_DB=<db_name>
 - -p 15432:5432 → Mapeo de puerto de la máquina y del contenedor
 - -d → Opcional. Indica si se quiere ejecutar en primer plano o no (modo **detached**)
 - postgres → Nombre de la imagen utilizada para crear el contenedor
 - Ejemplo:
 - docker run --name postgres_aves -e POSTGRES_USER=user_aves -e POSTGRES_PASSWORD=user_aves -e POSTGRES_DB=aves -p 15432:5432 -d postgres

DOCKER



- Crear un contenedor a partir de una imagen descargada.
 - Desde Docker Desktop:
 - Seleccionar la imagen



DOCKER



- **docker start *id/nombre*** arranca un contenedor.
- **docker ps** muestra la siguiente información de los contenedores en ejecución:
 - ID
 - Imagen
 - Comando ejecutado
 - Tiempo de ejecución
 - Estado
 - Puertos expuestos
 - Nombre asignado

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.26100.3775]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\ferna>docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED        STATUS        PORTS        NAMES
f2546d2a2b4c   mi-postgres-con-datos  "docker-entrypoint.s..."  12 seconds ago  Up 11 seconds  5432/tcp     goofy_colden

C:\Users\ferna>
```

DOCKER



- **docker ps** → Contenedores activos
- **docker ps -a** → Todos los contenedores
- **docker ps -q** → Sólo los id

```
Símbolo del sistema
C:\Users\ferna>docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f2546d2a2b4c	mi-postgres-con-datos	"docker-entrypoint.s..."	5 minutes ago	Up 5 minutes	5432/tcp	goofy_colden
8fb7d03f2c8e	postgres	"docker-entrypoint.s..."	5 minutes ago	Exited (1) 5 minutes ago		stupefied_mcclintock
203dd0296445	mi-postgres-con-datos	"docker-entrypoint.s..."	7 days ago	Exited (0) 7 days ago		elegant_hawking

```
C:\Users\ferna>
```

- **docker stop id/name** → Detiene el contenedor de manera ordenada.
- **docker kill id/name** → Detiene el contenedor de manera forzada.
- **docker restart id/name** → Reinicia el contenedor
- **docker rm id/name** → Elimina un contenedor

- **Modo interactivo:**

- Permite interaccionar con el contenedor.
- Para ejecutar un contenedor en modo interactivo, se utiliza la opción **-i** para habilitar la interacción, junto con **-t** para asignar una terminal (TTY). Esto te permite acceder a la terminal dentro del contenedor.

- **docker run -it** *ubuntu*
- **docker run -it** *ubuntu*
- **docker run -it** *debian bash*
- **docker run -it** *python*
- **docker run -it** *node*

DOCKER



- **Estadísticas:** muestra información de uso en tiempo real.
 - **docker stats** → Todos los contenedores
 - **docker stats *nombre/id*** → Un contenedor concreto

Parado

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
57196092b4c4	postgres-test	0.00%	0B / 0B	0.00%	0B / 0B	0B / 0B	0

En ejecución

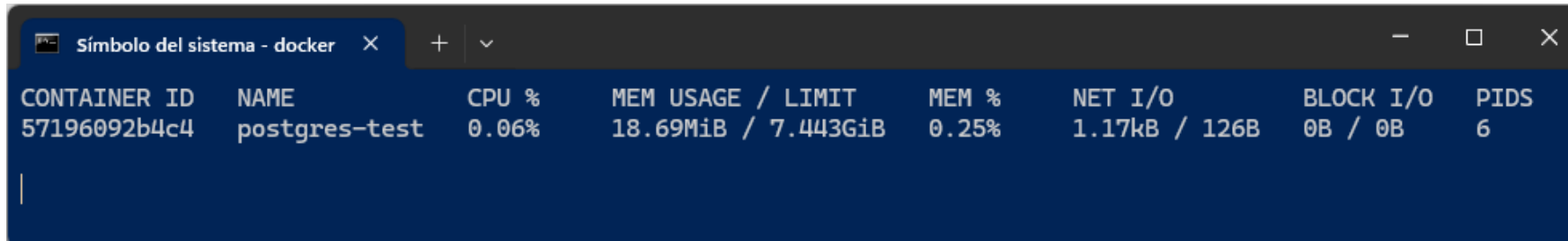
CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
57196092b4c4	postgres-test	0.06%	18.69MiB / 7.443GiB	0.25%	1.17kB / 126B	0B / 0B	6

■ Estadísticas:

- **docker stats --no-stream** → Hace una captura (no refresca)

■ Información proporcionada:

- CONTAINER ID → ID corto del contenedor (los primeros caracteres de su hash).
- NAME → Nombre asignado al contenedor.
- CPU % → Porcentaje de uso de CPU del contenedor respecto al total disponible en el host.
- MEM USAGE / LIMIT → Memoria usada por el contenedor en ese momento / límite máximo de memoria asignada. Si no se ha limitado, se muestra el total disponible en el sistema.
- MEM % → Porcentaje de memoria utilizada respecto al límite asignado (o total del sistema si no hay límite).
- NET I/O → Tráfico de red: cantidad de datos recibidos y enviados por el contenedor (en bytes o MB). Ejemplo: 1.2MB / 800kB (recibido / enviado).
- BLOCK I/O → Cantidad de operaciones de entrada/salida en disco realizadas por el contenedor (lecturas / escrituras de disco). Ejemplo: 10MB / 5MB (leídos / escritos).
- PIDS → Número de procesos en ejecución dentro del contenedor.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
57196092b4c4	postgres-test	0.06%	18.69MiB / 7.443GiB	0.25%	1.17kB / 126B	0B / 0B	6

DOCKER



Imágenes

- Una imagen de Docker es una plantilla inmutable y ligera que contiene todo lo necesario para ejecutar una aplicación: el sistema operativo base, bibliotecas, dependencias, archivos de código y scripts de configuración. Es la base a partir de la cual se crean los contenedores.
- **Características:**
 - **Capas (layers):** Cada instrucción en un Dockerfile (como FROM, COPY, RUN) crea una nueva capa. Estas capas se apilan para formar la imagen final. Las capas son cacheables, lo que hace que las reconstrucciones sean más rápidas. Si cambias una instrucción en el Dockerfile, solo las capas posteriores a esa instrucción se vuelven a construir.
 - **Inmutabilidad:** Una vez creada, una imagen no cambia. Si necesitas modificar algo, debes crear una nueva imagen.
 - **Versionamiento (tags):** Las imágenes pueden tener etiquetas (tags) para distinguir versiones.
 - **Portabilidad:** Puedes ejecutar la misma imagen en distintos entornos (desarrollo, pruebas, producción) sin preocuparte por diferencias de configuración del sistema.
 - **Optimización:** Es buena práctica construir imágenes livianas, usando imágenes base minimalistas.

DOCKER



- Listar imágenes:
 - `docker images`
 - `docker image ls` → (Atención al singular image. Mismo resultado que opción anterior)

```
Símbolo del sistema
C:\Users\ferna>docker images
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE
mi-postgres-con-datos latest      b1751b4a8bc1  8 days ago    621MB
ubuntu              latest      6015f66923d7  9 days ago    117MB
postgres            latest      304ab8135187  2 months ago  621MB
C:\Users\ferna>
```

DOCKER



- Buscar imágenes en repositorios oficiales:
 - `docker search nombre-imagen`
 - `docker search --filter "is-official=true" nombre-imagen`

```
Símbolo del sistema
C:\Users\ferna>docker search ubuntu
NAME                DESCRIPTION                STARS    OFFICIAL
ubuntu              Ubuntu is a Debian-based Linux operating sys... 17566    [OK]
ubuntu/squid        Squid is a caching proxy for the Web. Long-t... 113
ubuntu/nginx        Nginx, a high-performance reverse proxy & we... 129
ubuntu/cortex        Cortex provides storage for Prometheus. Long... 4
ubuntu/kafka         Apache Kafka, a distributed event streaming ... 53
ubuntu/prometheus    Prometheus is a systems and service monitori... 71
ubuntu/bind9         BIND 9 is a very flexible, full-featured DNS... 104
ubuntu/apache2       Apache, a secure & extensible open-source HT... 90
ubuntu/zookeeper     ZooKeeper maintains configuration informatio... 13
ubuntu/mysql         MySQL open source fast, stable, multi-thread... 67
ubuntu/postgres      PostgreSQL is an open source object-relatio... 41
ubuntu/jre           Distrosless Java runtime based on Ubuntu. Lon... 20
ubuntu/dotnet-aspnet  Chiselled Ubuntu runtime image for ASP.NET a... 26
ubuntu/redis         Redis, an open source key-value store. Long-... 23
ubuntu/python        A chiselled Ubuntu rock with the Python runt... 23
ubuntu/dotnet-deps    Chiselled Ubuntu for self-contained .NET & A... 16
ubuntu/grafana        Grafana, a feature rich metrics dashboard & ... 12
ubuntu/dotnet-runtime Chiselled Ubuntu runtime image for .NET apps... 20
ubuntu/cassandra      Cassandra, an open source NoSQL distributed ... 2
ubuntu/memcached      Memcached, in-memory keyvalue store for smal... 5
ubuntu/prometheus-alertmanager Alertmanager handles client alerts from Prom... 10
ubuntu/mlflow         MLFlow: for managing the machine learning li... 5
ubuntu/telegraf       Telegraf collects, processes, aggregates & w... 4
ubuntu/chiselled-jre  [MOVED TO ubuntu/jre] Chiselled JRE: distrol... 3
ubuntu/loki           Grafana Loki, a log aggregation system like ... 2

C:\Users\ferna>
```

```
Símbolo del sistema
C:\Users\ferna>docker search --filter "is-official=true" ubuntu
NAME                DESCRIPTION                STARS    OFFICIAL
ubuntu              Ubuntu is a Debian-based Linux operating sys... 17566    [OK]

C:\Users\ferna>
```

DOCKER



- Crear imagen de un proyecto web spring boot:
 - Fichero dockerfile (ubicar en la carpeta raíz del proyecto:

Usar una imagen base de JDK de Amazon Corretto
FROM amazoncorretto:17-alpine-jdk

Establecer el directorio de trabajo
WORKDIR /app

Copiar el archivo JAR generado en el contenedor
COPY target/*.jar app.jar

Exponer el puerto en el que la aplicación se ejecutará
EXPOSE 8081

Comando para ejecutar la aplicación
ENTRYPOINT ["java", "-jar", "app.jar"]

- Crear la imagen:

docker build -t nombre_imagen .

Ruta desde la que se va a construir la imagen (un punto en este caso)

DOCKER



- Copias de seguridad de imágenes:
 - Crear copia:
 - `docker save -o nombre-archivo.tar nombre-imagen`
 - Restaurar copia:
 - `docker load -i nombre-archivo.tar`
- Eliminar imágenes:
 - Si tiene contenedores asociados genera
 - `docker rmi nombre-imagen/id`
 - `docker rmi nombre-imagen1 nombre-imagen2 ...`

DOCKER



- Histórico de la imagen:
 - Muestra los pasos con los que se construyó la imagen a partir del Dockerfile.
 - **docker history** *nombre-imagen*

```
Símbolo del sistema x + v - □ x
E:\>docker history ubuntu
IMAGE          CREATED          CREATED BY          SIZE      COMMENT
6015f66923d7   9 days ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]   0B
<missing>      9 days ago      /bin/sh -c #(nop)  ADD file:ad85a9d7b0a74c214... 87.6MB
<missing>      9 days ago      /bin/sh -c #(nop)  LABEL org.opencontainers... 0B
<missing>      9 days ago      /bin/sh -c #(nop)  LABEL org.opencontainers... 0B
<missing>      9 days ago      /bin/sh -c #(nop)  ARG LAUNCHPAD_BUILD_ARCH    0B
<missing>      9 days ago      /bin/sh -c #(nop)  ARG RELEASE                0B
E:\>
```

DOCKER



- Conversión de un contenedor en una imagen:
 - `docker commit id-contenedor nombre-imagen:tag`
- Guarda estado del contenedor (instalación y ficheros) y algunos datos (no fiable).
- Alternativas:
 - Usar `pg_dump` dentro del contenedor para hacer un respaldo de la base de datos.
 - Usar volúmenes.

DOCKER



- Conversión de un contenedor en una imagen:
- Crear una imagen postgres con datos (usando pg_dump) y dockerfile:
 - 1. Generación de fichero con las sentencias de creación de la base de datos:
 - `docker exec -i nombre_contenedor pg_dumpall -U usuario_db > fichero_salida.sql`

■ Conversión de un contenedor en una imagen:

■ 2. Crear dockerfile

Usa una imagen base

FROM postgres:17

Variables de entorno para la configuración inicial

ENV POSTGRES_USER=admin

ENV POSTGRES_PASSWORD=admin

ENV POSTGRES_DB=mi_database

Hay variables comentadas porque
están en el fichero backup.sql

Copiar archivos SQL de inicialización (opcional)

COPY ./backup.sql /docker-entrypoint-initdb.d/

■ 3. Crear la imagen (desde la carpeta del dockerfile):

- **docker build** -t mi-postgres-con-datos .

■ Etiquetado:

- El etiquetado de imágenes en Docker es una forma de identificar y versionar imágenes para facilitar su manejo, distribución y despliegue.

- Sintaxis de las etiquetas:

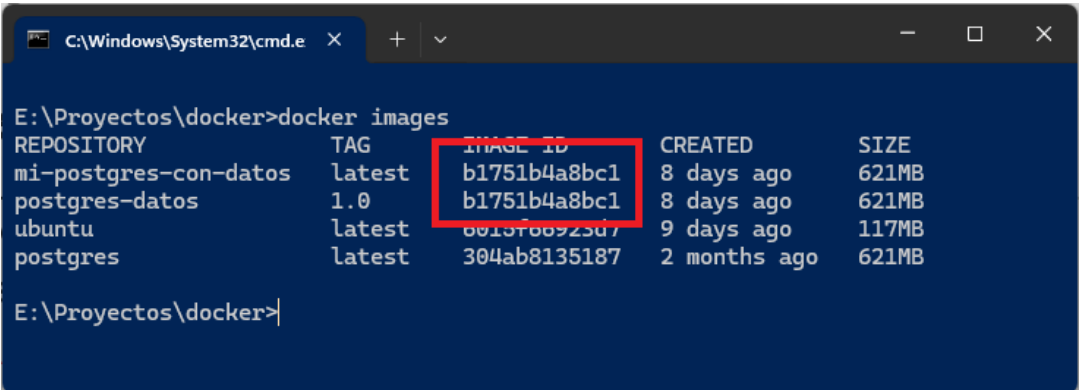
- nombre_imagen:etiqueta

- Uso:

- Construcción de una imagen:
 - `docker build -t tag .`

- Cambiar etiqueta:

- `docker tag id_imagen nombre_imagen:etiqueta`
 - **NOTA:** tag crea otra entrada en la lista de imágenes, pero hacen referencia al mismo id → Son la misma imagen.



```
C:\Windows\System32\cmd.e x + v
E:\Proyectos\docker>docker images
REPOSITORY          TAG          IMAGE ID      CREATED        SIZE
mi-postgres-con-datos latest       b1751b4a8bc1  8 days ago    621MB
postgres-datos      1.0         b1751b4a8bc1  8 days ago    621MB
ubuntu              latest      6813f68925d7  9 days ago    117MB
postgres            latest     304ab8135187  2 months ago  621MB
E:\Proyectos\docker>
```

■ Publicar una imagen en el repositorio oficial:

- Crear una cuenta en Docker Hub. <https://hub.docker.com/>
- `docker login` → Para hacer login desde el host
 - Para saber con qué usuario se está conectado:
 - `docker info`
 - Alternativa si no aparece el parámetro username (con Docker Desktop):
 - `docker-credential-desktop list`
- Construir la imagen (se necesita el fichero dockerfile):
 - `docker build -t miimagen:latest .`
- Etiquetar la imagen para Docker Hub:
 - `docker tag miimagen:latest miusuario/miimagen:latest`
- Subir la imagen:
 - `docker push miusuario/miimagen:latest`

■ Publicar una imagen en un repositorio local:

- Instalar el servidor local:
 - `docker run -d -p 5000:5000 --name registry registry`
- Crear la imagen
- Etiquetar la imagen para subirla al repositorio local
- Subir la imagen:
 - `docker push localhost:5000/nombre_imagen`
- Descargar desde el repositorio local:
 - `docker pull localhost:5000/nombre_imagen`
- Habilitar http:
 - En Linux/Unix:

Agregar a `/etc/docker/daemon.json`

```
{  
  "insecure-registries" : ["localhost:5000"]  
}
```
 - En Windows (Docker Desktop):
 - Agregar en Desktop → Settings → Docker Engine

```
"insecure-registries": ["localhost:5000"]
```

- **Buenas prácticas para la creación de los ficheros Dockerfile.**
 - Usar imágenes base apropiadas. Usar imágenes oficiales o mínimas como alpine, debian-slim, o ubuntu si se necesita más compatibilidad.
 - Minimizar la cantidad de capas. Combinar comandos RUN para reducir el número de capas.
 - Copiar solo lo necesario. Usar .dockerignore para excluir archivos innecesarios.
 - No ejecutar como root. Usar un usuario no privilegiado para ejecutar la aplicación.
 - Usar CMD o ENTRYPOINT correctamente. Usar CMD para proporcionar argumentos por defecto. Usar ENTRYPOINT si se quiere un comportamiento fijo del contenedor.
 - Hacer builds reproducibles. Fijar versiones en el sistema operativo y herramientas (evitar latest).
 - Organiza la estructura. Mantener el Dockerfile en la raíz del proyecto. Usar carpetas separadas si se tienen múltiples servicios o microservicios.
 - Usar etiquetas. Añadir LABEL para describir la imagen.
 - Reducir el tamaño de la imagen. Eliminar archivos temporales o herramientas innecesarias después de usarlas. Considerar usar multi-stage builds para compilar y luego copiar solo el resultado final.
 - Probar la imagen. Usar docker build --no-cache para detectar problemas que se ocultan con el cache. Usar linters como Hadolint para validar tu Dockerfile.

DOCKER



Redes

- En Docker, las redes son un componente fundamental que permite la comunicación entre contenedores, y entre contenedores y el mundo exterior (como la máquina anfitriona o Internet).
- Uso:
 - Aislar aplicaciones (cada red puede actuar como una mini red privada).
 - Controlar cómo se comunican los contenedores.
 - Conectar servicios que forman parte de una misma aplicación (por ejemplo: una app web, una base de datos y un servidor de caché).
 - Facilitar la escalabilidad y la seguridad.

■ Tipos de redes:

- Bridge/NAT – Por defecto para contenedores independientes.
 - Docker crea una red llamada bridge al instalarse.
 - Si no se especifica una red, los contenedores se conectan a esta.
 - Es útil cuando varios contenedores en la misma máquina necesitan comunicarse entre sí.
- Host – Comparte la red del sistema anfitrión.
 - El contenedor no tiene una pila de red propia.
 - Usa directamente la red del host (misma IP, mismos puertos).
 - Menor aislamiento, pero menos latencia.
 - Sólo funciona en Linux. No está soportado en Docker Desktop.
- None – Sin red.
 - El contenedor no está conectado a ninguna red.
 - Se usa cuando quieres controlar totalmente la red tú mismo.
- Overlay – Redes entre múltiples hosts Docker (en Swarm).
 - Permite que los contenedores se comuniquen, aunque estén en diferentes máquinas físicas.
 - Requiere que Docker esté en modo Swarm.
- Macvlan – Contenedor con su propia dirección MAC e IP de la red local.
 - Útil si necesitas que el contenedor aparezca como un dispositivo físico en la red local.
 - La tarjeta de red del host esté en modo promiscuo.

Docker Swarm es una herramienta integrada en Docker que permite **orquestrar múltiples contenedores** distribuidos en varias máquinas (nodos), como si fueran un solo sistema. Convierte un conjunto de máquinas en un **clúster de Docker** coordinado. **Kubernetes** es alternativa a Swarm.

■ Redes creadas por defecto:

```
E:\Proyectos\docker>docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
5f37f325bf61	bridge	bridge	local
f1422fa44ebc	host	host	local
fee3857540ef	none	null	local

■ Creación de redes:

- Tipo bridge: `docker network create --driver bridge mi_red_bridge`
- Tipo host: `docker network create --driver host mi_red_host`
- Tipo overlay: `docker network create --driver overlay mi_red_overlay`
- Tipo macvlan: `docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 mi_red_macvlan`
- Tipo none: `docker network create --driver none mi_red_none`
- Eliminar redes:
 - `docker network rm mi_red_bridge`
- Consultar redes:
 - `docker network ls`
- Inspeccionar redes:
 - `docker network inspect mi_red_bridge`
- Conectar un contenedor a una red:
 - `docker network connect <nombre_o_id_red> <nombre_o_id_contenedor>`
- Desconectar un contenedor de una red:
 - `docker network disconnect <nombre_o_id_red> <nombre_o_id_contenedor>`
- Especificar redes al crear un contenedor:
 - `docker run -d --name contenedor_1 --network mi_red_bridge nginx`

■ Rangos autogenerados:

- Al crear una red personalizada con Docker, si no se definen explícitamente la subred o el gateway, Docker autogenera esos valores a partir de un pool interno.

- Ejemplo:

- `docker network create mynet`

16 bits fijos y 32-16 para direcciones.

- Docker genera esta configuración:

- Subred: 172.18.0.0/16
 - Gateway: 172.18.0.1
 - Rango de IPs para contenedores: 172.18.0.2 en adelante

■ Rango específico:

- `docker network create --subnet=192.168.100.0/24 --gateway=192.168.100.1 mynet`

■ Ejemplo:

- Contenedor con la base de datos.
- Contenedor con la aplicación.
- Crear la red:
 - `docker network create --driver bridge mi_red_bridge`
- Configurar la aplicación para que conecte con el contenedor de la base de datos:
 - `application.properties`
 - `spring.datasource.url=jdbc:postgresql://videogames-docker-database:5432/videogames`
- Arrancar el contenedor de la base de datos en la red:
 - `docker run -d --name videogames-docker-database --network mi_red_bridge fouya/ifct0062_postgresql_videogames:latest`
- Arrancar el contenedor de la aplicación en la red:
 - `docker run --name videogames-docker-app --network mi_red_bridge -p 8081:8081 mi-app-supergames:latest`

Importante: es el puerto interno del contenedor, no el expuesto

DOCKER



Almacenamiento

- Docker utiliza contenedores para ejecutar aplicaciones de forma aislada. Sin embargo, los contenedores son efímeros, lo que significa que sus datos se pierden cuando el contenedor se detiene o se elimina. Para resolver esto, Docker ofrece mecanismos de almacenamiento persistente, permitiendo que los datos sobrevivan incluso si el contenedor desaparece.
- ¿Por qué es importante el almacenamiento en Docker?
 - Los datos importantes (como bases de datos, archivos de usuario, logs) necesitan persistencia.
 - Permite compartir datos entre contenedores.
 - Facilita respaldos y migraciones.

■ Tipos de persistencia:

■ **Volumes** (Volúmenes de Docker).

Recomendado

- Son la forma más recomendada y flexible de persistir datos en Docker.
- Se almacenan en el host, en el directorio `/var/lib/docker/volumes/`.
- Gestionados por Docker (puedes crearlos, listarlos, eliminarlos, etc.)
- Independientes del ciclo de vida del contenedor.
- Se pueden compartir entre contenedores.

■ **Bind Mounts** (Montajes de enlace o ligados).

- Permiten montar un directorio o archivo específico del sistema de archivos del host dentro del contenedor.
- Más control, pero menos portables.
- Útiles en desarrollo o cuando se necesita acceso directo a archivos del host.

■ **Tmpfs mounts** (Montajes en memoria).

- Montan un sistema de archivos temporal en la RAM.
- Los datos se pierden al detener el contenedor.
- Usados cuando se necesita velocidad y no persistencia.
- Ideales para datos temporales o sensibles

■ Sistemas de archivos:

- **AUFS:** es un sistema de archivos tipo *union file system* que Docker utilizaba en versiones anteriores como *storage driver* para gestionar las capas de las imágenes y contenedores.
- **OverlayFS:** es un *unión file system* que ha sido ampliamente adoptado por Docker debido a su eficiencia y rendimiento superior en comparación con otros sistemas de archivos de unión como AUFS.
- **Overlay2:** es una versión **mejorada** y más eficiente de **OverlayFS**. Docker cambió de **OverlayFS** a **Overlay2** como el controlador de almacenamiento predeterminado, ya que ofrece una serie de mejoras en términos de rendimiento y capacidad de manejar un mayor número de capas.
 - Un Union File System (UFS) es un tipo de sistema de archivos en el que varios directorios (o sistemas de archivos) se combinan en una sola jerarquía. La principal característica es que estos sistemas de archivos "**se unen**" para formar una estructura que aparece como un único directorio, mientras que cada uno de ellos sigue siendo accesible de forma independiente.

Predeterminado

- **Sistemas de archivos:**

- **Btrfs (B-tree File System)** en DockerBtrfs es un sistema de archivos avanzado de Linux que ha sido diseñado para ser un sistema de archivos de copia en escritura (copy-on-write o COW) con soporte para instantáneas (snapshots), compresión, autorreparación, y otras características avanzadas. Aunque no es el controlador de almacenamiento predeterminado en Docker (como overlay2), se ofrece como una opción viable para ciertos entornos.
- **Device Mapper** es un sistema de administración de dispositivos en Linux que permite crear volúmenes lógicos (LVM, Logical Volume Management). Docker utiliza Device Mapper para gestionar la creación, el almacenamiento y la manipulación de las capas de las imágenes de los contenedores.

■ Volúmenes:

- En Docker, los **volúmenes son una forma de persistir datos fuera de los contenedores**. Un volumen es un directorio o archivo especial que se utiliza para **almacenar datos**, y se puede **compartir entre contenedores** o **persistir datos incluso después de que el contenedor haya sido eliminado**.
- Los volúmenes son muy útiles cuando necesitas mantener datos entre ejecuciones de contenedores o cuando múltiples contenedores necesitan acceder a los mismos datos.

DOCKER



- **Volúmenes:**

- Creación:
 - `docker volume create` nombre_del_volumen
- Listado de volúmenes:
 - `docker volume ls`
- Consulta de detalles:
 - `docker volume inspect` nombre_del_volumen
- Eliminar un volumen:
 - `docker volume rm` nombre_del_volumen
- Eliminar volúmenes no utilizados:
 - `docker volume prune`

- **Volúmenes:**

- Montar (y crear si no existe) un volumen en el contenedor:

- Con la opción `-v`:

- `docker run -v nombre_del_volumen:/ruta/en/el/contenedor imagen`

- Con la opción `--mount`:

- `docker run --mount source=nombre_del_volumen,target=/ruta/en/el/contenedor imagen`

Recomendado

- **Volúmenes:**

- Plugins:

- Es un software que **implementa el API de volumen de Docker** y le dice al motor de Docker cómo crear, montar y administrar volúmenes usando un sistema externo o personalizado.
 - Los plugins de volúmenes permiten extender el sistema de almacenamiento de Docker más allá de los volúmenes locales por defecto.
 - Gracias a estos plugins, puedes usar sistemas de almacenamiento externos como NFS, Ceph, Amazon EFS, GlusterFS, NetApp, entre muchos otros, directamente como volúmenes montables por contenedores.

DOCKER



Docker compose

- Docker Compose es una herramienta que permite definir y administrar aplicaciones multi-contenedor en Docker. Utiliza un archivo de configuración, generalmente llamado `docker-compose.yml`, para describir los servicios, redes y volúmenes que componen una aplicación.
- ¿Qué hace Docker Compose?
 - Definir varios contenedores (servicios) en un solo archivo YAML.
 - Ejecutarlos todos con un solo comando: `docker-compose up`.
 - Automatizar redes entre contenedores, montajes de volúmenes y variables de entorno.
 - Escalar servicios fácilmente (por ejemplo, varias instancias de un contenedor).
- Advertencias:
 - Los contenedores que levanta Docker Compose no deben existir.
 - El nombre del contenedor creado se toma de la carpeta en la que está el fichero `Docker-compose.yml`.

DOCKER



- Ejemplo docker-compose.yml:

```
version: '3.8'
services:
  web:
    image: nginx
    ports:
      - "80:80"
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: ejemplo123
```

- Este ejemplo lanza dos contenedores:
 - Uno con Nginx accesible en el puerto 80.
 - Uno con PostgreSQL con una contraseña definida.

DOCKER



- Ejemplo con los contenedores utilizados en la sección de redes.
- Crear el docker-compose.yml. La carpeta en la que se encuentre dará nombre al contenedor en Docker.
- Ejecutar:
 - `docker-compose up -d`

```
services:
  database:
    image: fouya/ifct0062_postgresql_videogames:latest
    container_name: videogames-docker-database
    networks:
      - mi_red_bridge
    # ports:
      # - "5432:5432"

  app:
    image: mi-app-supergames:latest
    container_name: videogames-docker-app
    depends_on:
      - database
    networks:
      - mi_red_bridge
    ports:
      - "8088:8081"
    # environment:
      # -
    SPRING_DATASOURCE_URL=jdbc:postgresql://videogames-docker-
    database:5432/videogames
    # Agrega otras variables si fueran necesarias,
    como usuario y contraseña

networks:
  mi_red_bridge:
    driver: bridge
```

Crea una nueva red. Consultar con
Docker network ls

DOCKER



docker-compose up → Levanta los servicios definidos en el archivo

docker-compose up -d → Levanta los servicios en segundo plano (detached)

docker-compose down → Detiene y elimina contenedores, redes y volúmenes

docker-compose build → Construye las imágenes de los servicios

docker-compose restart → Reinicia todos los servicios

docker-compose stop → Detiene los servicios sin eliminar contenedores

docker-compose start → Inicia servicios previamente detenidos

docker-compose ps → Lista los contenedores en ejecución

docker-compose logs → Muestra los logs de todos los servicios

docker-compose exec <servicio> <comando> → Ejecuta un comando dentro de un contenedor