



Summer School 2025

# INTRODUCCIÓN A DOCKER

Aritz Madariaga ([eifersucht](#))

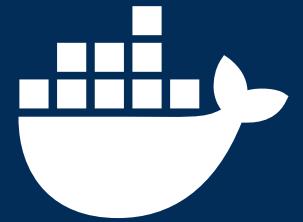


///  
**0xDecode**  
 Deusto Electronic Club Of  
Developers & Engineers

# Contenidos

- Introducción a Docker
- Flujo Build → Ship → Run + comandos esenciales
- Dockerfile: tu primera imagen personalizada
- Docker Compose: orquestación básica
  - Opciones avanzadas en Compose
- Recursos y buenas prácticas
- Orquestadores y plataformas modernas



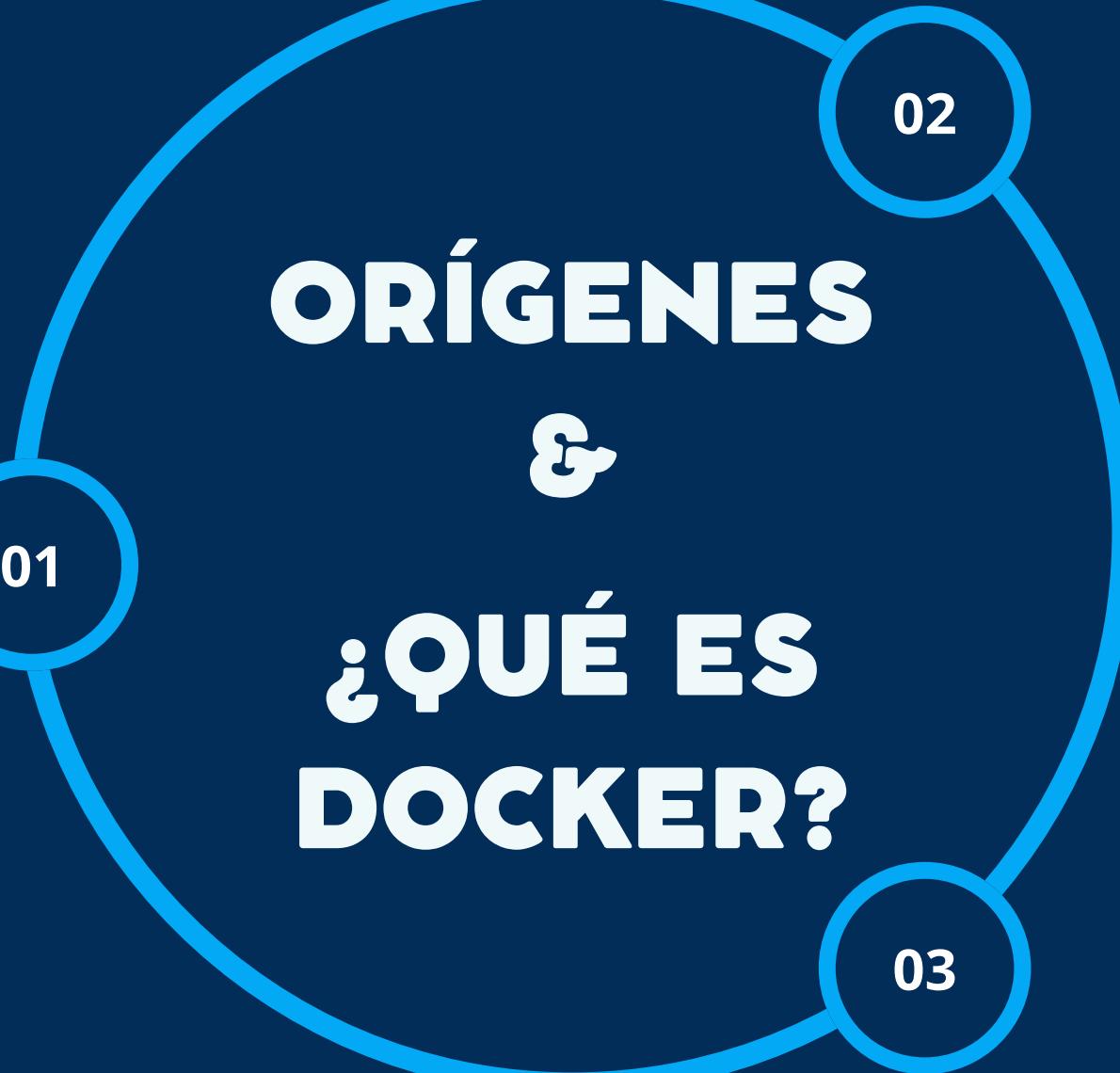


**Solomon Hykes** inicia el proyecto Docker dentro de dotCloud, el proyecto fue presentado internamente como evolución de la PaaS de dotCloud y lanzado públicamente como Docker en marzo de 2013 en la PyCon

## ORÍGENES



## ¿QUÉ ES DOCKER?



Docker es una plataforma de contenedores ligera y portable (open-source desde marzo de 2013), escrita principalmente en Go, que permite empaquetar aplicaciones y todas sus dependencias para ejecutarlas en cualquier entorno.

Docker es una herramienta que puede empaquetar una aplicación y sus dependencias en un contenedor virtual que se puede ejecutar en cualquier servidor Linux. Esto ayuda a permitir la flexibilidad y portabilidad en donde la aplicación se puede ejecutar, ya sea en instalaciones físicas, la nube pública, nube privada, etc.

451 Research

# ¿Por qué docker? Qué problemas resuelve

①

Con Docker, lo que funciona en tu ordenador funciona igual en producción: empaquetas tu app y sus dependencias en una única imagen portable

②

Los contenedores aprovechan el núcleo del host, compartiendo recursos de forma eficiente; arrancan en segundos y ocupan solo megabytes, frente a las VMs que necesitan gigabytes y minutos

③

Gracias a cgroups y namespaces del kernel de Linux, Docker aísla procesos y limita recursos (CPU, memoria, I/O) de forma eficiente sin la capa extra de VMs

④

contenedor ≠ VM

# Contenedor vs Máquina Virtual

## ARQUITECTURA

Se ejecuta directamente sobre el host OS, gestionado por el Docker daemon, compartiendo el núcleo y usando solo las dependencias de la aplicación.

vs

Implica un guest OS completo sobre un hypervisor. Cada VM arranca un sistema operativo completo (kernel + librerías) independiente del host.

## CONSUMO DE RECURSOS

Arranca en milisegundos, ocupa MB y sólo solicita recursos según demanda.

vs

Arranca más lento, usa GB de espacio y requiere recursos preasignados.

## AISLAMIENTO

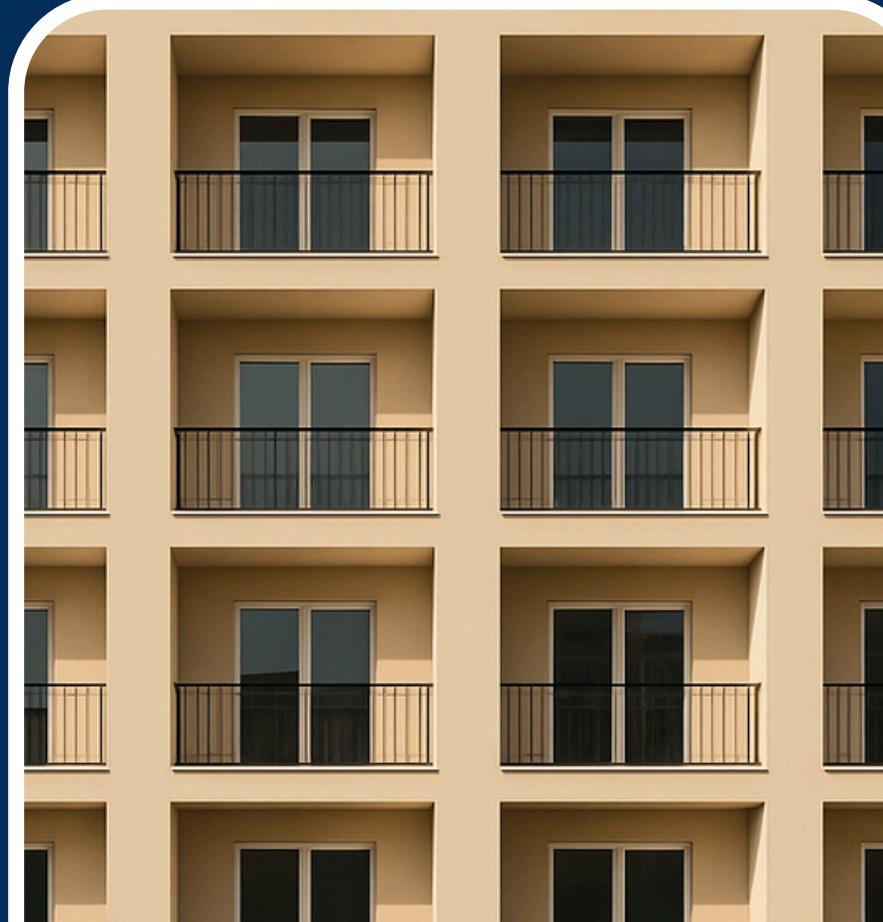
Usa namespaces (para PID, red, FS...) y cgroups para aislar procesos y controlar recursos.

vs

Proporciona aislamiento total



# Podríamos decir que....



## CONTENEDORES

Como un apartamento dentro de un edificio (todos en el mismo edificio, pero cada uno con su puerta y sus propias reglas).



## MÁQUINAS VIRTUALES

Como casas independientes, cada una con su propio terreno y servicios completos (agua, electricidad).

2

**FLUJO BUILD → SHIP → RUN**  
**&**  
**COMANDOS ESENCIALES**

# Build → Ship → Run

Docker es una plataforma para desarrollar, empaquetar y ejecutar aplicaciones de manera rápida y consistente

## Build

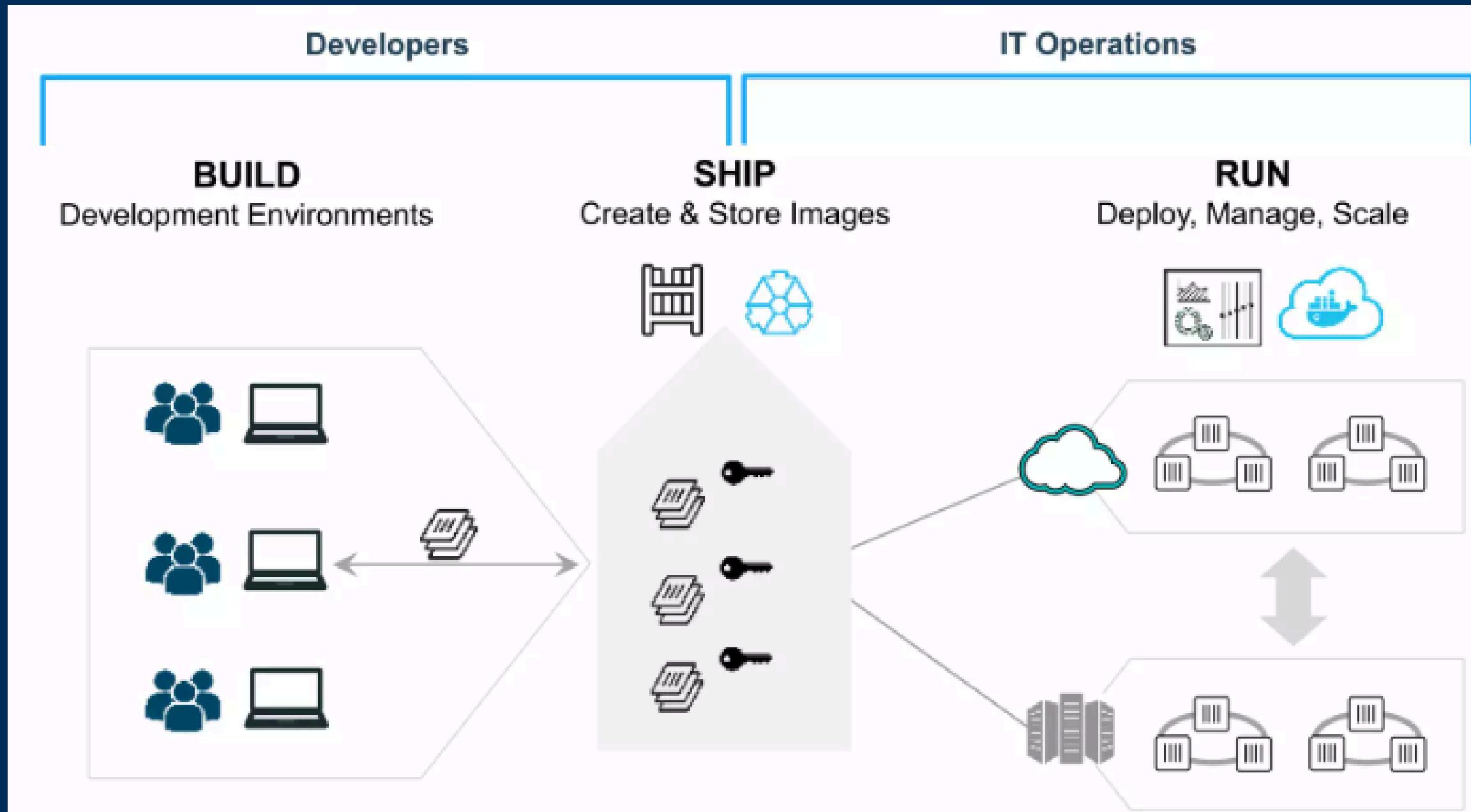
Creas una imagen con tu proyecto mediante un Dockerfile — empaqueta todo lo necesario para ejecutarlo en cualquier entorno.

## Ship

Subes esa imagen a un registro (como Docker Hub) para compartirlo o utilizarla en otros sistemas.

## Run

Ejecutas la imagen localmente o en producción, levantando el contenedor con tu aplicación lista para funcionar.



# Comandos

## # 1. Ejecutar un contenedor de prueba

```
docker run --rm hello-world
```

## # 2. Descargar y correr una imagen nginx

```
docker run -d -p 8080:80 nginx
```

## # 3. Ver contenedores en ejecución

```
docker ps
```

## # 4. Parar un contenedor

```
docker stop <container-id>
```

## # 5. Eliminar el contenedor

```
docker rm <container-id>
```

Comando	Descripción
docker images / docker rmi	Listar o eliminar imágenes locales
docker pull / docker push	Descargar o subir imágenes
docker ps -a	Ver todos los contenedores (incluidos parados)
docker exec -it	Abrir una terminal dentro de un contenedor
docker logs -f	Ver los registros de un contenedor
docker build -t miapp .	Construir una imagen desde un Dockerfile
docker system prune	Limpiar recursos no utilizados

# Ejemplo práctico 1/2

# 1. Ejecuta un contenedor de prueba (se auto-elimina al terminar)

```
docker run --rm hello-world
```

# 2. Inicia un servidor web nginx en segundo plano y mapea el puerto 8080

```
docker run -d -p 8080:80 --name web nginx
```

# 3. Lista contenedores activos

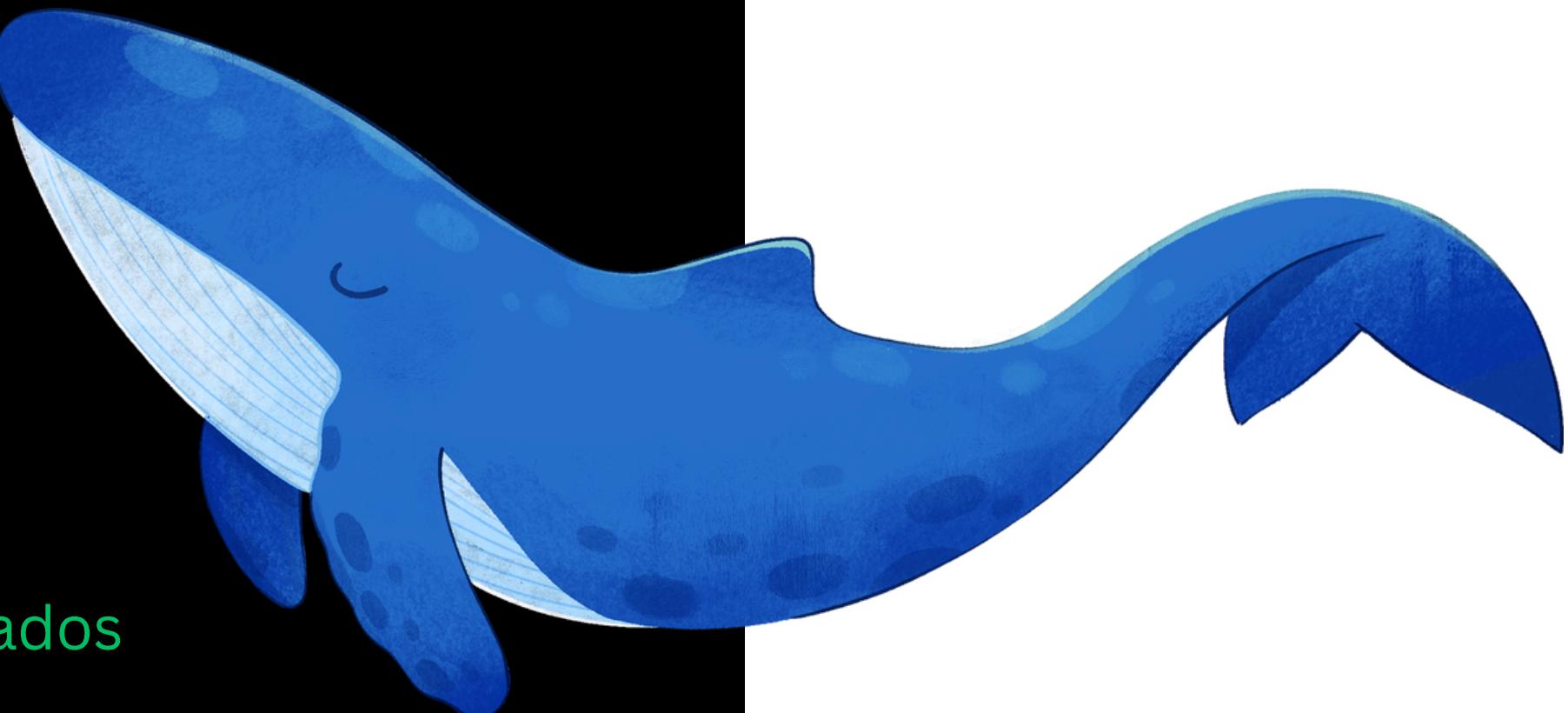
```
docker ps
```

# 4. Detiene el contenedor llamado "web"

```
docker stop web
```

# 5. Lista todos los contenedores, incluidos los parados

```
docker ps -a
```



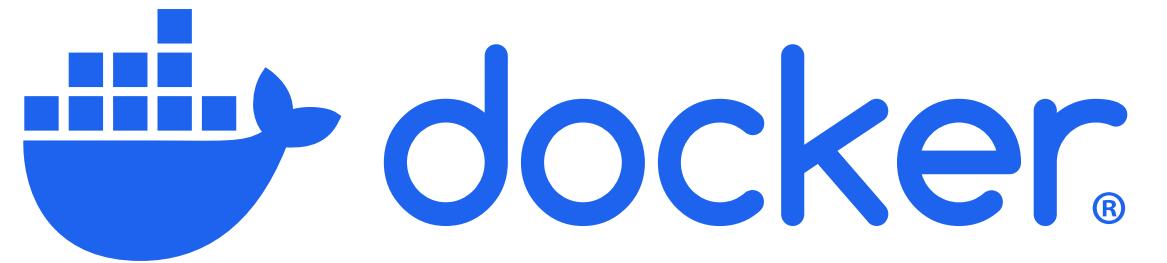
# Ejemplo práctico 2/2

# 6. Elimina el contenedor "web"  
docker rm web

# 7. Elimina imágenes locales específicas  
docker rmi nginx

# 8. Elimina todas las imágenes locales (forzando eliminación)  
docker rmi -f \$(docker images -a -q)

# 9. Limpieza automática de recursos no utilizados  
docker system prune -a -f --volumes



3

# DOCKERFILE: TU PROPIA IMAGEN

“

*Un Dockerfile es un archivo de texto  
con instrucciones para crear una  
imagen Docker, ejecutando comandos,  
copiando archivos y definiendo el  
entorno.*

”

James Walker

# Requisitos

- Docker instalado
  - Debes tener instalado Docker (o Docker Desktop) y funcionando
- Python 3 instalado localmente para generar dependencias.
- Código base en una carpeta con:
  - app.py o similar (ej. Flask/FastAPI)
  - requirements.txt con las dependencias
  - .dockerignore (exclude \_\_pycache\_\_, .venv, \*.pyc, etc.)
- Editor de código o IDE
  - Cualquier editor (VS Code, Sublime, etc.) para editar tu Dockerfile y archivos de configuración.

# Dockerfile (buenas prácticas)

```
FROM python:3.10-slim

WORKDIR /app

# Copia sólo el archivo de dependencias
COPY requirements.txt .

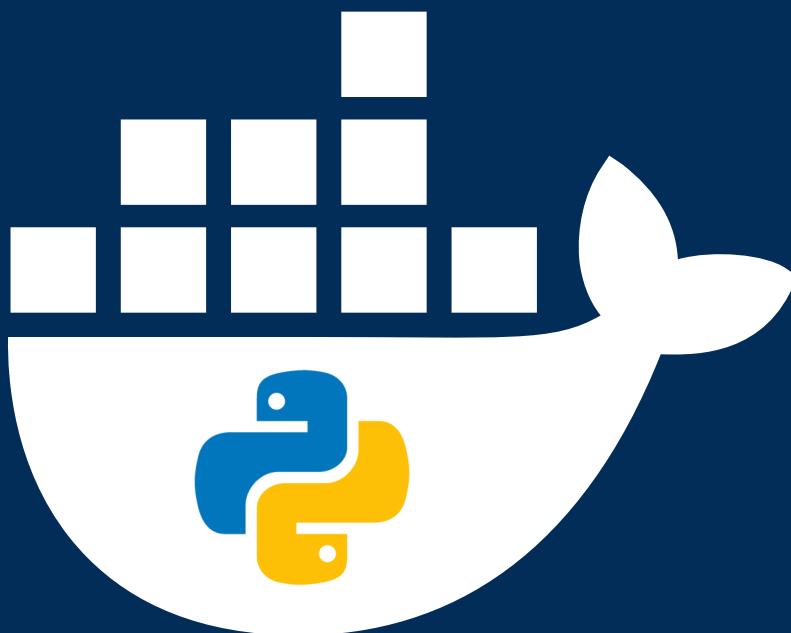
# Instala dependencias usando cache
RUN pip install --no-cache-dir -r requirements.txt

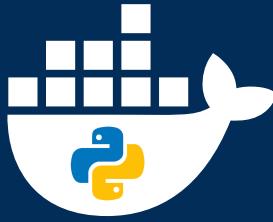
# Copia el resto del código
COPY ..

EXPOSE 8000

ENTRYPOINT ["python", "app.py"]
```

- Utilizar imágenes base ligeras (python:3.10-slim o :3.11-alpine)
- Separar instalación de dependencias para aprovechar el cache (COPY requirements.txt + RUN pip install)
- Incluir .dockerignore para reducir el contexto de construcción
- Especificar EXPOSE, utilizar ENTRYPOINT + CMD en forma de array





# Construimos pero no juzgamos

Ejemplo de proyecto Python Flask (demo)

```
File Edit Selection View Go Run Terminal Help
EXPLORER ... summer_school
SUMMER SCHOOL [WSL: UBUNTU]
.dockerignore
app.py
Dockerfile
requirements.txt

app.py
1 from flask import Flask
2 import flask
3 import werkzeug
4
5 app = Flask(__name__)
6
7 @app.route("/")
8 def hello():
9     return "¡Hola desde Flask en Docker!"
10
11 @app.route("/version")
12 def version():
13     return {
14         "flask_version": flask.__version__,
15         "werkzeug_version": werkzeug.__version__
16     }
17
18 if __name__ == "__main__":
19     app.run(host="0.0.0.0", port=8000)
```

[Descargar código de ejemplo](#)

docker build -t python-app:1.0 .

# Vemos las imágenes en el sistema  
docker images

# Lanzamos la imagen en un contenedor  
docker run -d -p 8000:8000 --name pyapp python-app:1.0

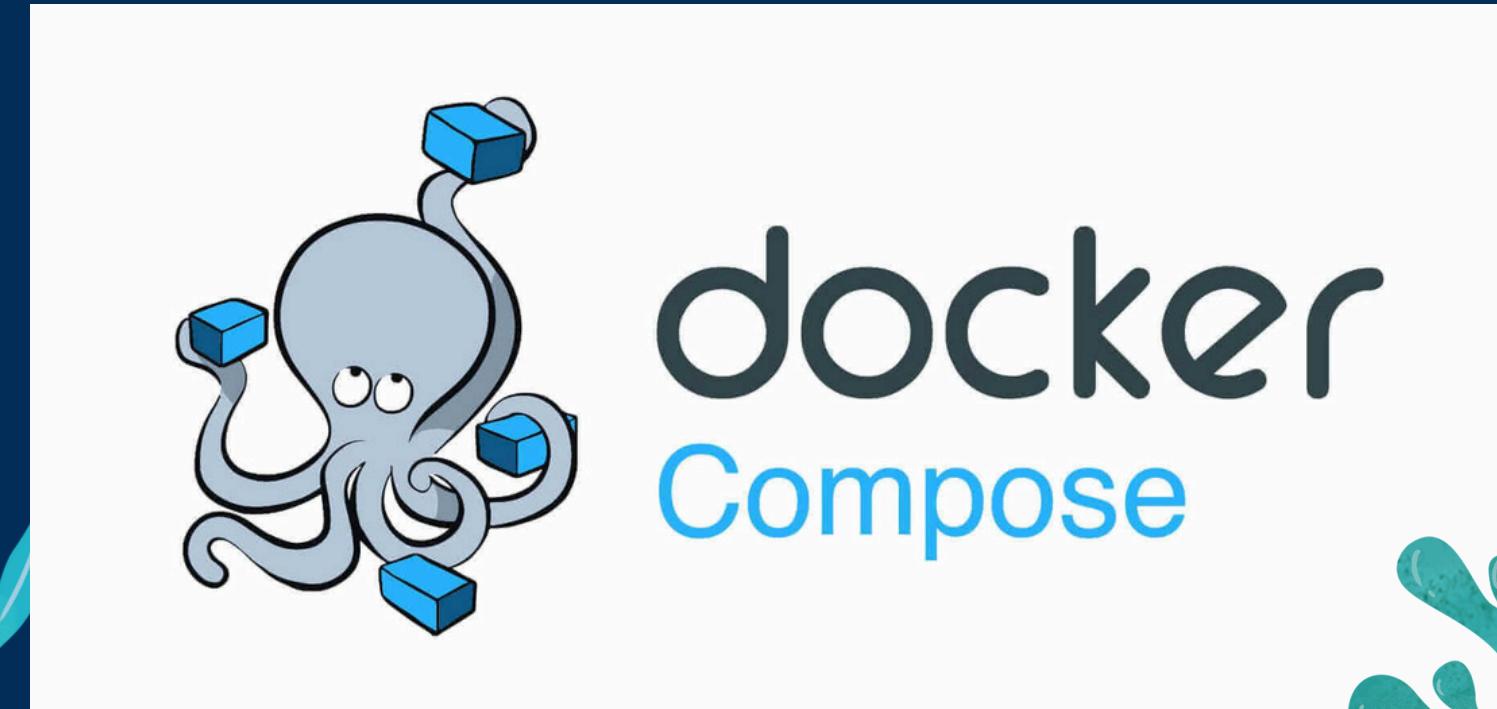
#logs  
docker logs pyapp  
# Paramos y borramos el contenedor  
docker stop pyapp  
docker rm pyapp

4

# DOCKER COMPOSE

# ¿Por qué usar Docker Compose?

- Permite definir múltiples contenedores y su configuración en un solo archivo YAML.
- Facilita levantar entornos completos con un solo comando (docker compose up).
- Asegura consistencia entre desarrollo, testing y producción



# Docker Compose (docker-compose.yml)

```
services:  
  web:  
    build: .  
    ports:  
      - "8000:8000"  
    depends_on:  
      - db  
    environment:  
      - DATABASE_URL=postgresql://postgres:password@db:5432/myapp  
    volumes:  
      - ./app  
  
  db:  
    image: postgres:14-alpine  
    environment:  
      - POSTGRES_USER=postgres  
      - POSTGRES_PASSWORD=password  
      - POSTGRES_DB=myapp  
    ports:  
      - "5432:5432"  
    volumes:  
      - pgdata:/var/lib/postgresql/data  
  
volumes:  
  pgdata:
```

## Operar con un compose

- docker compose up
- docker compose up -d
- docker compose up --build
- docker compose up --build -d
- docker compose down
- docker compose restart
- docker compose ps
- docker compose logs
- docker compose rm
- docker compose exec web sh
- docker compose build
- docker compose pull
- docker compose push

# Docker Compose (Algunos detalles) 1/2

```
services:  
  web:  
    build: .  
    ports:  
      - "8000:8000"  
    depends_on:  
      - db  
    environment:  
      - DATABASE_URL=postgresql://postgres:password@db:5432/myapp  
    volumes:  
      - ./app  
  
  db:  
    image: postgres:14-alpine  
    environment:  
      - POSTGRES_USER=postgres  
      - POSTGRES_PASSWORD=password  
      - POSTGRES_DB=myapp  
    ports:  
      - "5432:5432"  
    volumes:  
      - pgdata:/var/lib/postgresql/data  
  
volumes:  
  pgdata:
```

## Servicio **web**

- **build**: .: crea la imagen Docker en base al Dockerfile del directorio actual.
- **ports**: "8000:8000": expone el puerto 8000 del contenedor hacia el 8000 del host.
- **depends\_on**: [db]: garantiza que el servicio db se inicie primero, aunque no espera a que esté listo para conexiones .
- **environment**: define variables de entorno accesibles desde la app; DATABASE\_URL indica cómo conectar con Postgres usando el alias db como host.
- **volumes**: ./app: vincula tu código local al contenedor para desarrollo en caliente, permitiendo ver cambios sin reconstruir la imagen.

# Docker Compose (Algunos detalles) 2/2

```
services:  
  web:  
    build: .  
    ports:  
      - "8000:8000"  
    depends_on:  
      - db  
    environment:  
      - DATABASE_URL=postgresql://postgres:password@db:5432/myapp  
    volumes:  
      - ./app  
  
  db:  
    image: postgres:14-alpine  
    environment:  
      - POSTGRES_USER=postgres  
      - POSTGRES_PASSWORD=password  
      - POSTGRES_DB=myapp  
    ports:  
      - "5432:5432"  
    volumes:  
      - pgdata:/var/lib/postgresql/data  
  
volumes:  
  pgdata:
```

## Servicio db

- **image:** postgres:14-alpine: usa la imagen oficial de PostgreSQL versión 14 en variante ligera Alpine.
- **Variables de entorno:**
  - POSTGRES\_USER, POSTGRES\_PASSWORD, POSTGRES\_DB: configuran credenciales y base de datos iniciales.
- **ports:** "5432:5432": expone Postgres en el host, útil para conectarte desde el exterior.
- **volumes:** pgdata:/var/lib/postgresql/data: persiste los datos en un volumen Docker llamado pgdata, para que sobrevivan reinicios.

# Opciones avanzadas (healthcheck)

- Levantar distintos servicios a la vez y que unos dependan de otros genera dependencias a nivel temporal
- Solucion: **Healthchecks**
  - Permiten verificar que un servicio no solo esté en ejecución, sino que esté realmente listo para aceptar conexiones o cumplir su función

```
db:  
  healthcheck:  
    test: ["CMD-SHELL", "pg_isready -U postgres -d myapp"]  
    interval: 10s  
    timeout: 5s  
    retries: 5  
    start_period: 20s
```

```
web:  
  depends_on:  
    db:  
      condition: service_healthy
```

# Opciones avanzadas (redes)

- Cuando ejecutas docker compose up, Docker crea automáticamente una red llamada <proyecto>\_default, basada en el controlador bridge. Todos los contenedores definidos en tu docker-compose.yml se conectan a esta red sin necesidad de configurarla explícitamente.
- En esta red, los contenedores pueden comunicarse entre sí usando el nombre del servicio como host (por ejemplo, db) en lugar de usar IPs fijas
- El bridge funciona como un switch virtual: permite la comunicación interna, pero aísla tu aplicación del resto del sistema, excepto por los puertos que pubiques

```
networks:  
  mi_red:  
  
services:  
  web:  
    networks:  
      - mi_red  
  db:  
    networks:  
      - mi_red
```

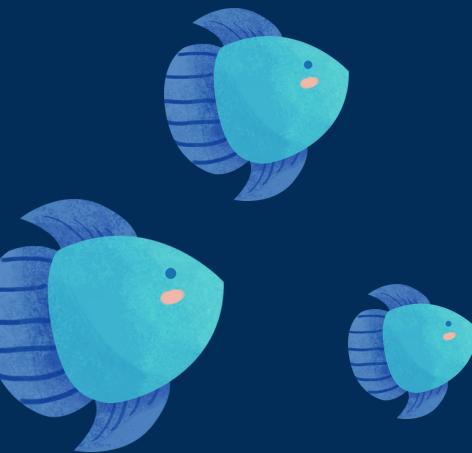


5

## RECURSOS Y BUENAS PRÁCTICAS

# Buenas prácticas

- Emplea imágenes base ligeras (-slim, -alpine)
- Aprovecha el cache en Dockerfile para tiempos de build más rápidos
- Utiliza .dockerignore para reducir el contexto y tamaño de las imágenes
- Define healthchecks robustos (HTTP, pg\_isready)
- Segmenta servicios y tráfico con redes personalizadas
- Controla tu entorno con docker compose down --volumes --rmi all
- Cuida la seguridad: haz escaneos de vulnerabilidades y evita ejecutar como root



# Algunos recursos

- [Docker Curriculum](#): tutorial paso a paso con ejemplos reales — ideal para aprender mediante práctica guiada
- [Docker Docs - Compose](#): guía oficial sobre orquestación, redes, volúmenes y healthchecks
- [Play-with-Docker](#): entorno online gratuito que permite practicar sin instalar nada
- [Docker For The Absolute Beginner \(KodeKloud\)](#): curso gratuito con prácticas interactivas sobre Docker y Compose
- [freeCodeCamp Docker Handbook](#): guía profunda desde conceptos básicos hasta usos avanzados con ejemplos

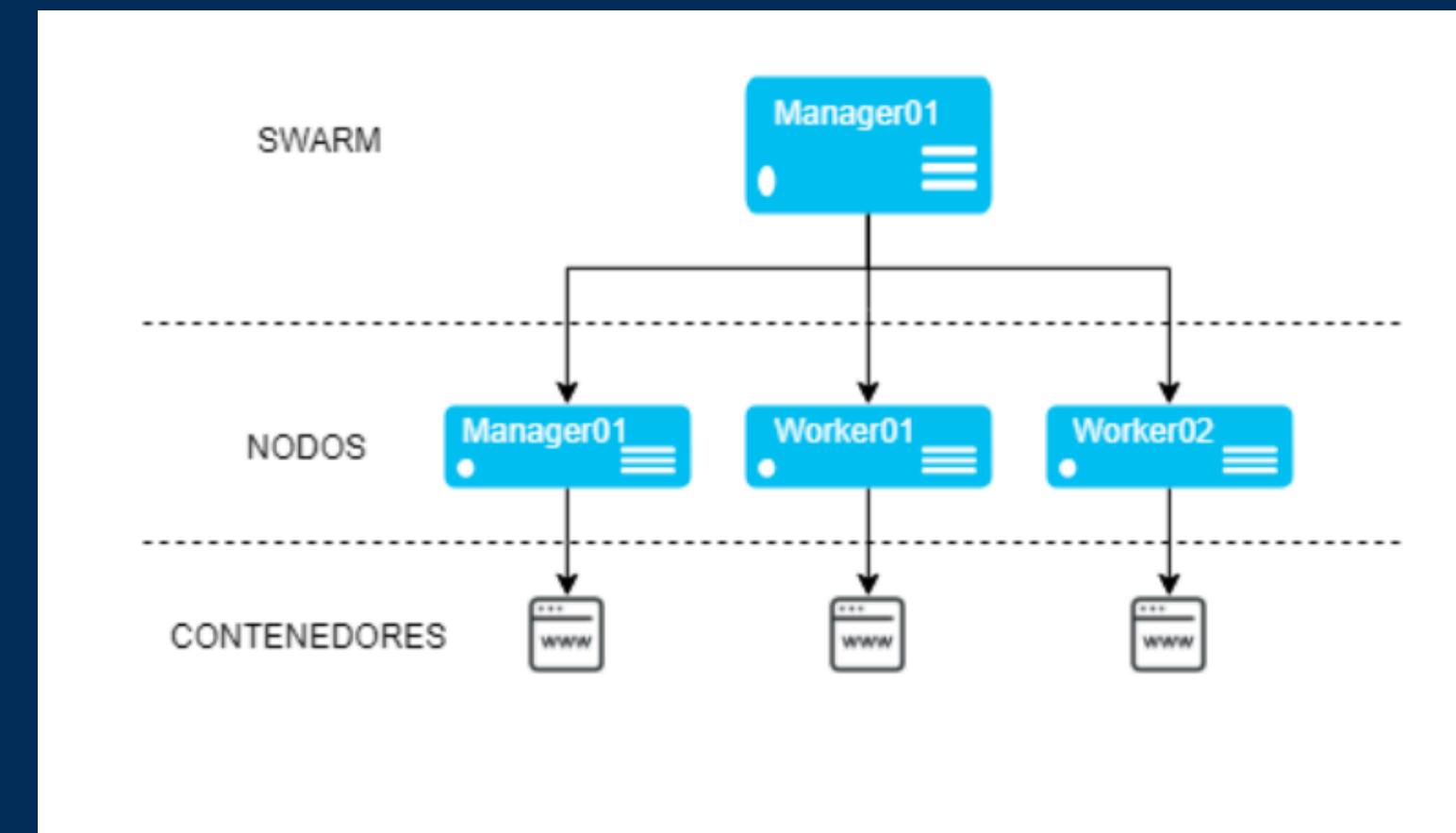


6

# ORQUESTRADORES Y PLATAFORMAS MODERNAS

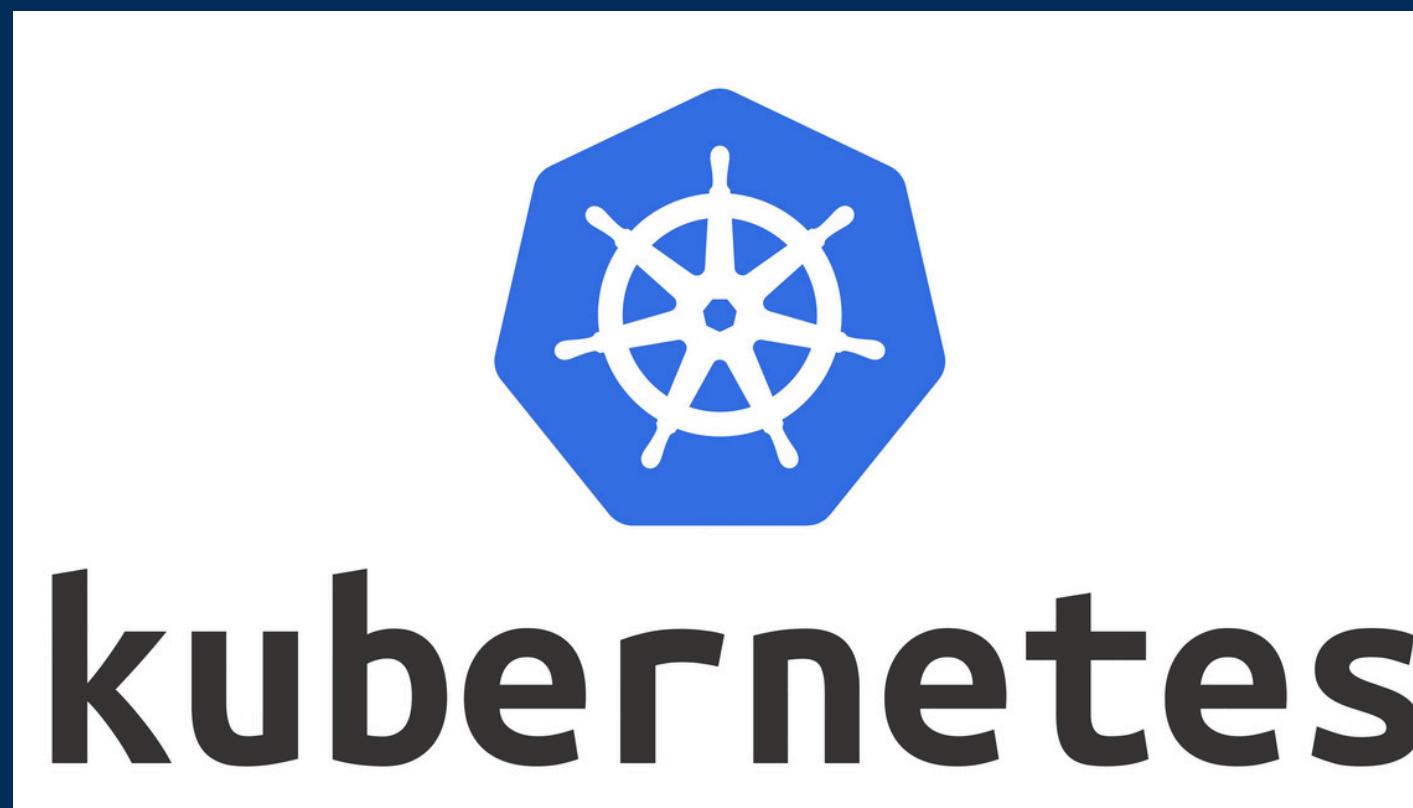
# DOCKER SWARM

- Es la funcionalidad nativa de orquestación en Docker Engine, que permite crear y gestionar un cluster de contenedores con múltiples nodos (managers y workers).
- Puedes iniciar un swarm fácilmente con *docker swarm init* y añadir nodos con *docker swarm join*
- Gestión de clúster integrada: no requiere herramientas externas; todo con Docker CLI
- Modelo declarativo y escalable:
  - Definir réplicas deseadas
  - Equilibrado automático
  - Chequeos de estado y rescheduling tras fallos
- Red overlay y servicio DNS interno para comunicación entre nodos
- Seguridad por defecto: cifrado TLS, autenticación y RBAC opcional



# KUBERNETES

- Plataforma de orquestación de contenedores open source, creada por Google en 2014 y mantenida por la CNCF
- Maneja despliegue, escalado y gestión de aplicaciones contenedorizadas en clústeres heterogéneos (on-prem, cloud, híbridos).



- Autoscaling automático (Horizontal y Vertical)
- Balanceo de carga integrado con detección de servicio (DNS interno)
- Self-healing: reinicia pods fallidos y reprograma en nodos sanos
- Rollouts y rollbacks automáticos
- Gestión declarativa con pods, deployments, services, volumes, configmaps, secrets

# PORTAINER

- Herramienta open-source y ligera con interfaz web para gestionar entornos Docker, Docker Swarm y Kubernetes
- Ideal si deseas:
  - Visualizar contenedores, imágenes, redes y volúmenes.
  - Ofrecer una consola gráfica a quienes no trabajan con CLI.
  - Gestionar múltiples entornos o clústeres con Portainer CE o Premium



# RANCHER

- Plataforma enfocada en Kubernetes, con gestión de multi-clúster, RBAC, políticas, monitoreo y catálogo de aplicaciones.
- Es una solución empresarial robusta, ideal si necesitas:
  - Gestionar múltiples clústeres (on-prem, cloud, edge).
  - Controlar accesos y permisos por usuario o grupo.
  - Simplificar implementaciones de aplicaciones (Helm, GitOps)



# PORTAINER vs RANCHER

Característica	Portainer	Rancher
<b>Enfoque</b>	GUI para Docker y Kubernetes	Gestión completa de Kubernetes corporativo
<b>Escalabilidad</b>	Clústeres pequeños y medianos	Multi-clúster, políticas avanzadas
<b>Curva de aprendizaje</b>	Baja (ideal para principiantes)	Alta (necesita conocimiento profundo de K8s)
<b>Ámbito</b>	Local / Swarm / pequeño K8s	Kubernetes en producción + multi-clúster



# ESKERRIK ASKO

aritzmadariaga@deusto.es

///  
0x  
□