

Docker: Tu app en una caja

Cómo los contenedores eliminaron el "en mi máquina funciona" — y por qué toda empresa tech usa Docker hoy

EL PROBLEMA QUE RESUELVE

LA FRASE MÁS FAMOSA DEL DESARROLLO DE SOFTWARE

"En mi máquina funciona."

Imaginá esta situación real. Vos terminás tu app, la probás, todo anda perfecto. La subís al servidor de la empresa y... no funciona. O se lo mandás a una colega para que lo pruebe y tampoco funciona. ¿Qué pasó?

- Tu laptop tiene Python 3.12, el servidor tiene Python 3.9
- Instalaste una librería en tu máquina que no está en el servidor
- Desarrollás en macOS, el servidor corre Ubuntu Linux
- Una variable de entorno que definiste localmente no existe en producción
- Una desarrolladora nueva tarda **dos o tres días** en tener el entorno configurado

Docker resuelve todo esto de una sola vez: empaqueta tu app *junto con todo su entorno*. Si corre en tu máquina, corre en cualquier servidor. Siempre. Sin excepciones.

UN EJEMPLO DE LA VIDA REAL

Sofía es desarrolladora. Creó una API en Node.js para su trabajo. Quiere que su colega Luciana la pruebe desde su computadora.

SIN DOCKER

Sofía tiene que decirle a Luciana: "instalá Node 20, después npm install, fijate que tengas la variable DATABASE_URL seteada, ah y necesitás Redis corriendo en el puerto 6379, y también que la versión de npm sea al menos 9.x..." — Luciana pasa la tarde entre errores y mensajes.

CON DOCKER

Sofía le manda el repo con el Dockerfile adentro. Luciana hace `docker run sofía/mi-api` y en 30 segundos tiene todo funcionando exactamente igual que en la máquina de Sofía.

¿QUÉ ES DOCKER EXACTAMENTE?

DEFINICIÓN

Docker es un programa — un software — que instalás en tu computadora o en un servidor. Una vez instalado, Docker es capaz de leer un archivo especial llamado `Dockerfile` y hacer dos cosas fundamentales:

- **Construir imágenes** (`docker build`) — empaquetar tu app y todo lo que necesita en una "caja" portable
- **Correr contenedores** (`docker run`) — abrir esa caja y ejecutar la app dentro de ella, aislada del resto del sistema

Se instala en Windows, Mac o Linux desde **docker.com** como "Docker Desktop" (trae una interfaz visual). En servidores Linux se instala como "Docker Engine" (solo terminal). Una vez instalado, lo usás con el comando `docker` desde la terminal.

ANALOGÍA: DOCKER ES UNA CHEF QUE SABE LEER RECETAS

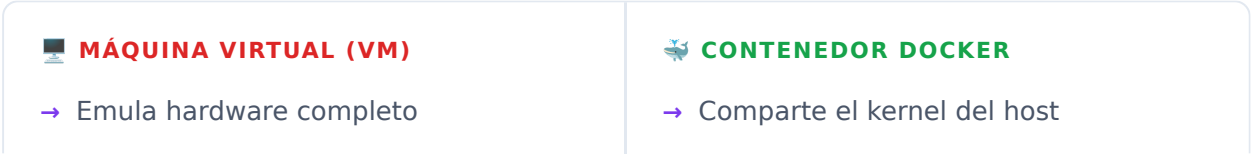
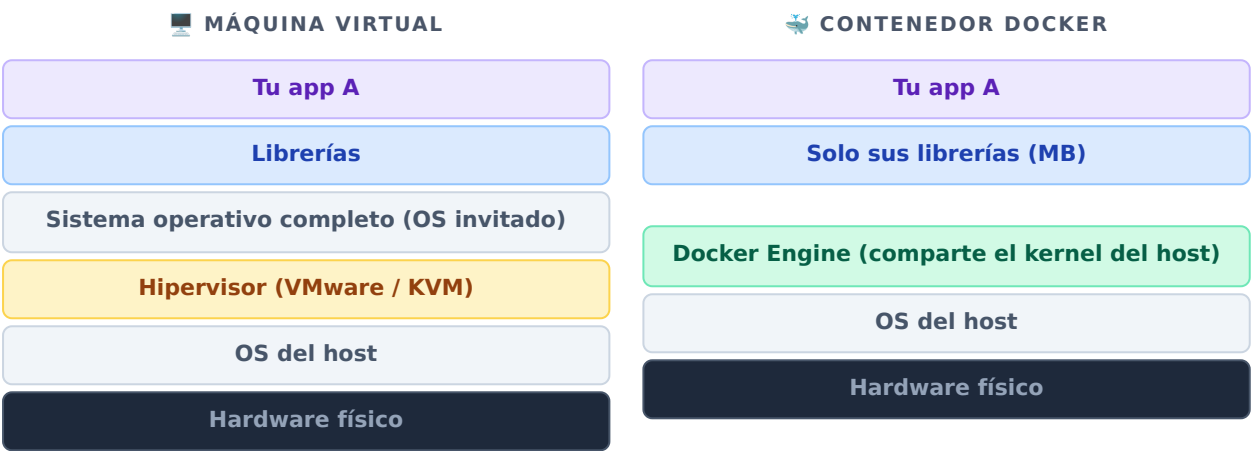
El **Dockerfile es la receta escrita** — dice qué ingredientes usar y qué pasos seguir. Docker (la chef) lee esa receta y cocina el plato — eso es la **imagen**. Cuando querés servir, Docker toma el plato ya preparado y lo pone en la mesa — eso es el **contenedor**. Podés servir el mismo plato veinte veces sin volver a cocinar.

UN MOMENTO: ¿QUÉ ES VIRTUALIZACIÓN?

En la sesión 2 usamos **EC2**, que es una máquina virtual en AWS. Antes de entender Docker mejor, vale la pena cerrar ese concepto.

Virtualización significa hacer que una sola máquina física se comporte como varias máquinas independientes. Hay una capa de software llamada **hipervisor** (como VMware o VirtualBox) que simula el hardware completo — procesador, memoria, disco — y sobre eso corre un sistema operativo entero. Cada VM tiene su propio Windows o Linux adentro. Por eso una VM ocupa varios GB y tarda minutos en arrancar.

Docker hace algo diferente y más liviano: **no virtualiza el hardware**. En cambio, usa directamente el sistema operativo del servidor pero aísla cada app en un espacio separado usando funciones del kernel de Linux. El resultado es que los contenedores arrancan en segundos y pesan megabytes en lugar de gigabytes.



- | | |
|--|---|
| <ul style="list-style-type: none"> → OS entero adentro (varios GB) → Tarda minutos en arrancar → Consume mucha RAM aunque no haga nada → Máximo decenas por servidor | <ul style="list-style-type: none"> → Solo la app y sus dependencias (MB) → Arranca en segundos o menos → Muy liviano — bajo consumo en idle → Podés correr cientos en un mismo servidor |
|--|---|

¿CÓMO AÍSLA DOCKER SI NO TIENE SU PROPIO OS?

Un contenedor es, a nivel técnico, **un proceso normal corriendo en el procesador del host** — igual que cuando abrís Chrome o cualquier otra app. Lo que Docker hace es arrancar ese proceso usando dos mecanismos que vienen dentro del kernel de Linux:

NAMESPACES — EL AISLAMIENTO

Le dan al proceso su propia "vista" del sistema. Desde adentro, el contenedor ve solo:

- Sus propios procesos (no puede ver los del host ni de otros contenedores)
- Su propio sistema de archivos
- Su propia interfaz de red con su propia IP

El proceso *cree* que es el único en la máquina. En realidad comparte todo con el host.

✂ CGROUPS — LOS LÍMITES

Le ponen un techo a los recursos que puede consumir:

- Máximo X MB de RAM
- Máximo N% de CPU
- Si se pasa, el kernel lo frena

Esto evita que un contenedor descontrolado se coma todos los recursos del servidor.

LA CONSECUENCIA PRÁCTICA

Cuando hacés `docker run mi-app`, Linux crea un proceso nuevo con estas reglas aplicadas. No hay boot, no hay detección de hardware, no pasa nada especial. El proceso arranca en milisegundos — igual que cualquier otro programa — solo que vive en su propia burbuja aislada.

ACLARACIÓN: ¿LOS CONTENEDORES TIENEN OS O NO?

El diagrama de capas puede confundir. La respuesta correcta es: **tienen parte del OS, pero no el kernel.**

- Los contenedores **sí tienen un espacio de usuario de Linux** — filesystem, librerías del sistema. Es lo que trae el `FROM` del Dockerfile (`FROM python:3.12-slim` incluye un Debian mínimo)
- Lo que **no tienen** es su propio kernel — usan el del host directamente
- El kernel es la parte que habla con el hardware (CPU, RAM, red). Ese lo comparten todos los contenedores con el host



¿QUÉ ES UN DOCKERFILE?

Un Dockerfile es **un archivo de texto plano sin extensión**, que se llama literalmente `Dockerfile`. Contiene instrucciones escritas en un lenguaje muy simple que Docker entiende. Docker lo lee de arriba hacia abajo y ejecuta cada instrucción en orden para construir el entorno de tu app.

Cada línea empieza con una **palabra clave en mayúsculas** que indica qué tiene que hacer Docker en ese paso:

FROM	Siempre la primera línea. Define de qué imagen base partir. No construís de cero — Docker Hub tiene miles de imágenes ya preparadas. <code>python:3.12-slim</code> significa "un Linux mínimo con Python 3.12 ya instalado".
WORKDIR	Crea una carpeta dentro del contenedor y la establece como directorio de trabajo. Todo lo que viene después pasa ahí adentro. Similar a hacer <code>mkdir /app && cd /app</code> .
COPY	Copia archivos desde tu máquina (host) al contenedor. <code>COPY . .</code> copia todo el contenido de la carpeta actual al WORKDIR dentro del contenedor.
RUN	Ejecuta un comando <i>durante la construcción de la imagen</i> . Se usa para instalar dependencias, compilar código, crear carpetas, etc. Corre una sola vez al hacer <code>docker build</code> .
EXPOSE	Documenta en qué puerto escucha la app. No abre el puerto automáticamente — eso lo hacés cuando corrés el contenedor con <code>-p</code> . Es solo documentación.
CMD	El comando que se ejecuta cuando el contenedor arranca. Es lo que "corre" tu app. Solo puede haber uno. Si Docker no encuentra esto, el contenedor arranca y se detiene inmediatamente.
ENV	Define variables de entorno dentro del contenedor. <code>ENV NODE_ENV=production</code> hace que la app sepa que está en producción.

Un Dockerfile es solo texto — lo abrís con cualquier editor, lo guardás en Git, lo versionás. No es magia. **Es la fuente de verdad de cómo se construye tu app.** Sin Docker instalado, es solo texto. Con Docker instalado, ese texto se convierte en un entorno completamente reproducible.

UN DOCKERFILE REAL, LÍNEA POR LÍNEA

Este es el Dockerfile de una app Python con Flask — muy similar a la que deployamos en la sesión 2:

```
# Partimos de un Linux mínimo que ya tiene Python 3.12
FROM python:3.12-slim

# Crear /app dentro del contenedor y trabajar desde ahí
WORKDIR /app

# Primero copiamos solo requirements.txt (truco de caché)
COPY requirements.txt .
RUN pip install -r requirements.txt

# Ahora copiamos el resto del código de la app
COPY . .

# Documentamos que la app usa el puerto 5000
EXPOSE 5000

# Cuando el contenedor arranque, ejecutar app.py
CMD ["python", "app.py"]
```

¿POR QUÉ COPIAR REQUIREMENTS.TXT ANTES QUE EL CÓDIGO?

Docker construye en capas — cada instrucción es una capa. Si las dependencias no cambiaron, Docker reutiliza la capa de `pip install` del caché y se ahorra minutos de instalación. Si primero copiamos todo junto, cualquier cambio en el código (aunque sea un comentario) forzaría reinstalar todas las dependencias desde cero.

DOCKER BUILD: CONSTRUIR LA IMAGEN

Cuando ejecutás `docker build`, Docker lee tu Dockerfile de arriba a abajo y ejecuta cada instrucción. El resultado es una **imagen** guardada en tu máquina:

```
# Desde la carpeta donde está tu Dockerfile:
docker build -t mi-app:1.0 .
```

→ `-t mi-app:1.0` — le da nombre y versión a la imagen (`ag`). El formato `nombre:versión` es

- `.` — el punto indica la carpeta actual como "contexto": los archivos que Docker puede copiar al contenedor.

Lo que pasa adentro cuando ejecutás ese comando:

- Docker descarga la imagen base del `FROM` (si no la tiene ya en caché)
- Ejecuta cada instrucción y crea una **capa** por cada una
- Guarda todas las capas apiladas como una imagen lista para usar
- La imagen queda en tu máquina — podés verla con `docker images`

SISTEMA DE CAPAS Y CACHÉ

Cada instrucción del Dockerfile genera una capa inmutable. Si nada cambió desde el último build, Docker reutiliza la capa del caché. Por eso el primer build puede tardar minutos (descarga todo) y los siguientes tardan segundos.

DOCKER RUN: CORRER UN CONTENEDOR

Una imagen es un molde estático. Cuando la "corrés", Docker crea una instancia viva — un proceso aislado ejecutándose en tu máquina. Eso es un **contenedor**:

```
# Lo más básico: correr la imagen
docker run mi-app:1.0

# Con puerto mapeado: máquina:5000 → contenedor:5000
docker run -p 5000:5000 mi-app:1.0

# En segundo plano y con nombre
docker run -d -p 5000:5000 --name mi-flask mi-app:1.0

# Con variables de entorno (para secrets, conexiones, etc.)
docker run -d -p 5000:5000 -e DATABASE_URL=postgres://... mi-app:1.0
```

- `-p 5000:5000` — mapea **puerto-de-tu-máquina:puerto-del-contenedor**. Sin esto, la app corre aislada y no podés accederla desde el browser. El formato es **puerto-de-tu-máquina:puerto-del-contenedor**
- `-d` — modo detached (segundo plano). El contenedor sigue corriendo y recuperarás la terminal.
- `--name mi-flask` — le da un nombre legible para referenciarlo después (stop, logs, etc.).
- `-e CLAVE=valor` — pasa variables de entorno al contenedor. Así inyectás configuración sin hardcodearla en el código.

LA TRAMPA DEL PUERTO

Si la app escucha en el puerto 5000 pero no hacés `-p 5000:5000`, el contenedor corre pero es inaccesible desde afuera. Es como si la habitación estuviera cerrada con llave. El flag `-p` abre esa puerta. Si no podés acceder a la app, casi siempre es esto.

COMANDOS ESENCIALES

Ver qué hay corriendo

<code>docker ps</code>	Lista los contenedores actualmente en ejecución
<code>docker ps -a</code>	Lista todos los contenedores (incluyendo los detenidos)
<code>docker images</code>	Lista las imágenes descargadas en tu máquina

Gestionar contenedores

<code>docker stop <nombre></code>	Detiene un contenedor en ejecución
<code>docker rm <nombre></code>	Elimina un contenedor detenido
<code>docker logs <nombre></code>	Ve la salida (logs) de un contenedor
<code>docker logs -f <nombre></code>	Sigue los logs en tiempo real (<code>-f</code> = follow)
<code>docker exec -it <nombre> bash</code>	Abre una terminal interactiva <i>dentro</i> del contenedor

Imágenes y Docker Hub

<code>docker pull nginx</code>	Descarga una imagen desde Docker Hub (sin correrla)
<code>docker run -p 80:80 nginx</code>	Descarga (si no existe) y corre nginx en el puerto 80
<code>docker rmi <imagen></code>	Elimina una imagen de tu máquina
<code>docker tag mi-app usuario/mi-app:1.0</code>	Le da un nombre de Docker Hub a tu imagen
<code>docker push usuario/mi-app:1.0</code>	Sube la imagen a Docker Hub (o a ECR en AWS)

DOCKER HUB: EL REPOSITORIO DE IMÁGENES

Docker Hub es el registro público de imágenes — pensalo como GitHub pero para imágenes Docker. Miles de imágenes oficiales listas para usar:

- `nginx` — servidor web. Con `docker run -p 80:80 nginx` tenés un servidor funcionando en 10 segundos.
- `postgres:16` — base de datos PostgreSQL sin instalar nada
- `node:20-alpine` — Node.js en una imagen ultra liviana (Alpine Linux)
- `python:3.12-slim` — Python en una imagen recortada, sin extras
- `redis` — cache en memoria, un comando y corre

En AWS, el equivalente privado de Docker Hub se llama **ECR — Elastic Container Registry**. Ahí las empresas guardan las imágenes de sus apps propias, sin que sean públicas.

- **Consistencia total:** el mismo Dockerfile produce el mismo resultado en cualquier máquina. El "en mi máquina funciona" desapareció.
- **Onboarding en minutos:** en vez de un documento de 30 páginas para configurar el entorno, una desarrolladora nueva hace `docker run` y en minutos tiene todo corriendo.
- **Aislamiento:** si la app de pagos falla, la app de usuarios sigue corriendo. Cada servicio en su propio contenedor.
- **Base del CI/CD moderno:** el pipeline construye la imagen, la sube a ECR, y la despliega en producción. Sin manos humanas, sin errores de entorno.
- **Base de Kubernetes:** K8s (próximo tema) orquesta contenedores Docker a escala — decide dónde correr cada uno, cuántos, y qué hacer si uno cae.