

# Apunte 22 - Introducción a Docker

## Introducción

Docker es una plataforma que permite crear, distribuir y ejecutar aplicaciones dentro de contenedores, garantizando que se comporten igual en cualquier entorno.

A diferencia de las máquinas virtuales, los contenedores no replican un sistema operativo completo, sino que comparten el kernel del host y aíslan cada aplicación junto con sus dependencias.

Esto simplifica la instalación, acelera el despliegue, reduce conflictos de configuración y hace posible reproducir entornos de desarrollo y producción de forma confiable.

En este apunte introducimos los conceptos esenciales de Docker, su arquitectura, comandos básicos y la creación de imágenes y contenedores, preparando el terreno para el despliegue de aplicaciones completas en el siguiente bloque.

## Origen

Una frase que se ha vuelto meme en el desarrollo de software es la siguiente: "... en mi máquina funciona! ..."

A raíz de esto, vale la siguiente pregunta: ¿Por qué en mi máquina funciona y en el servidor no? La respuesta a esto, se puede deber por tres razones:

- Falta uno o más Archivos  
La aplicación se encuentra incompleta en producción.
- Configuraciones Distintas  
La aplicación en producción contiene un archivo de configuración o una variable de entorno distinta a la aplicación local.
- Discordancia Del Software  
Se puede dar el caso de que en el sistema local se tenga instalada la versión de Java 21 y en el servidor se tenga instalada la versión de Java 17 o incluso 11.

Por estas razones, surge la creación de Docker. Docker es una plataforma que permite ejecutar y desplegar aplicaciones sin preocuparte por los problemas de compatibilidad y dependencias.

## Contenedores y Paquetes de Aplicaciones

En el desarrollo de software es frecuente la frase:

**"En mi máquina funciona, pero en el servidor no..."**

Las diferencias entre entornos —versiones de Java, librerías faltantes o configuraciones distintas— suelen provocar errores difíciles de reproducir.

**Docker** surge justamente para eliminar esas diferencias, ofreciendo una forma de ejecutar las aplicaciones dentro de **contenedores**: entornos aislados que incluyen todo lo necesario para que una aplicación funcione igual en cualquier lugar.

Un contenedor empaqueta el sistema operativo base, las dependencias, las variables de entorno y el propio código de la aplicación.

De esta manera, la aplicación puede desplegarse y ejecutarse de forma predecible, sin importar si el host es Windows, Linux o macOS.

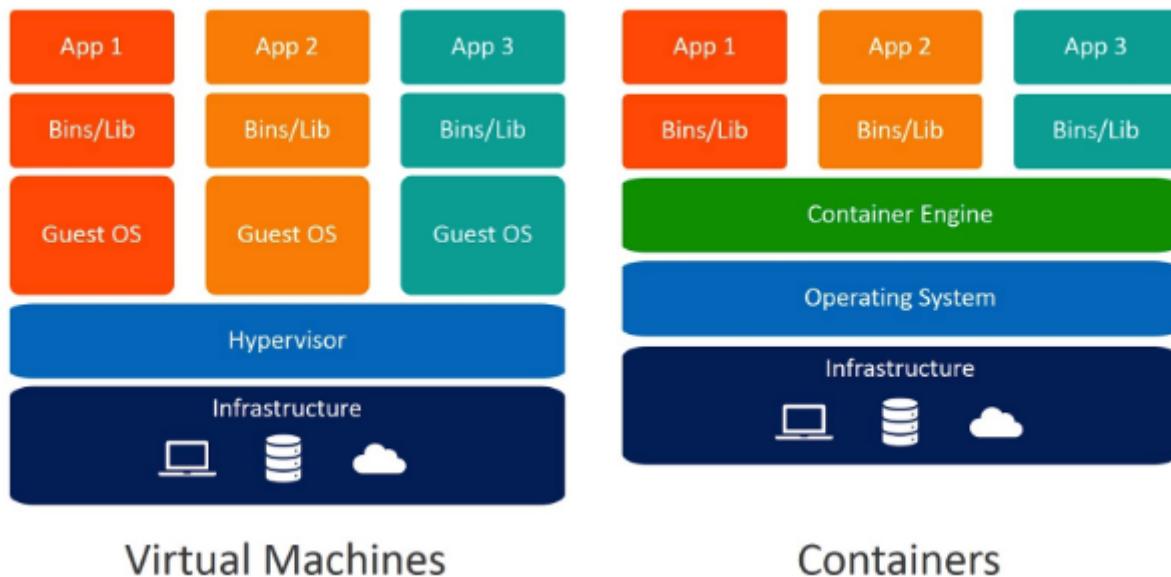
Cuando necesitamos distribuir o compartir una aplicación, no enviamos los archivos sueltos, sino una **imagen Docker**, que actúa como un **paquete de aplicación**.

Este paquete puede contener, por ejemplo, una app Java 17 con PostgreSQL 16.1 y su configuración, lista para ser levantada con:

```
docker compose up
```

Gracias a este modelo, podemos ejecutar múltiples aplicaciones con diferentes versiones o dependencias en el mismo equipo, manteniendo cada una en su propio entorno aislado.

## Máquinas Virtuales Vs Contenedores



Un contenedor es entonces, un ambiente aislado para poder ejecutar aplicaciones. Por otro lado, una máquina virtual es una abstracción de hardware físico donde se cuenta con una computadora física que ejecuta un sistema operativo y que dentro de la misma tiene otra computadora que virtualiza el hardware físico y posibilita instalar un sistema operativo igual o distinto al sistema operativo anfitrión. La ventaja de las máquinas virtuales, es que se pueden tener más de una máquina virtual en ejecución en el sistema operativo anfitrión.

## Comparación entre Máquinas Virtuales y Contenedores

Una de las decisiones más importantes al momento de desplegar aplicaciones modernas es elegir el entorno adecuado para su ejecución. Tradicionalmente, se utilizaban **Máquinas Virtuales (VMs)**, pero actualmente los **Contenedores** han ganado popularidad por sus ventajas en eficiencia y portabilidad.

A continuación, se ofrece una explicación visual, conceptual y analógica para comprender las principales diferencias.

### ■ Máquinas Virtuales (VMs)

Una máquina virtual es una emulación completa de un sistema informático. Cada VM contiene su propio sistema operativo invitado, bibliotecas y binarios, además de la aplicación en sí.

◆ **Infraestructura** → Sistema operativo anfitrión → **Hypervisor** → Sistema operativo invitado + App

## Hypervisor

Para poder hacer uso de las máquinas virtuales, es necesario instalar una aplicación que lleva el nombre de Hypervisor. Un hypervisor es un monitor de máquina virtual que crea y gestiona máquinas virtuales, siendo entornos virtuales aislados que pueden ejecutar sistemas operativos y aplicaciones de manera independiente. Existen diversas aplicaciones de hypervisor, como ser:

- VirtualBox
- VMWare
- Parallels (Exclusivo para MacOS)
- Hyper-V (Exclusivo para Windows)

Por ejemplo, se puede tener en la misma computadora anfitriona, una máquina virtual que tenga instalada la versión 17 de Java y la versión 16.1 de PostgreSQL para dar soporte a la aplicación uno y, a su vez, otra máquina virtual que tenga instalada la versión 11 de Java y la versión 5.5 de MySQL para dar soporte a la aplicación dos. Esta situación permite que ambas máquinas virtuales se ejecuten en la misma computadora anfitriona y además permite que cada máquina virtual sea un ambiente totalmente aislado entre todas las máquinas virtuales, pero esto trae consigo ciertas desventajas:

📦 **Ejemplo real:** Cada VM es como alquilar un departamento completo con cocina, baño y sala. Tarda más en estar listo, cuesta más y requiere más recursos.

## Desventajas:

- Cada VM necesita un **sistema operativo completo** (más espacio y tiempo de inicio).
- **Mayor consumo de recursos** (CPU/RAM asignadas estáticamente).
- Lento para iniciar y mover entre equipos.

## 📦 Contenedores

Un contenedor es un entorno liviano que contiene solo la aplicación y sus dependencias. Todos los contenedores comparten el mismo sistema operativo del host.

◆ **Infraestructura** → Sistema operativo → **Container Engine** → Contenedores con Apps + Dependencias

📦 **Ejemplo real:** Un contenedor es como un **food truck**: liviano, portátil, rápido de mover y con los recursos justos para ofrecer su servicio.

## Ventajas:

- **Arranque rápido** y liviano.
- Requiere menos espacio (no incluye sistema operativo completo).
- Utiliza de forma más eficiente los recursos del host.

## Analogía: Departamento vs Food Truck

Comparación	 Máquinas Virtuales	 Contenedores
Sistema operativo	Uno por cada VM	Compartido
Aislamiento	Total, incluso del SO	Aislamiento a nivel de aplicación
Tiempo de inicio	Minutos	Segundos
Uso de recursos	Alto	Eficiente
Portabilidad	Menor (depende del hypervisor)	Alta (puede moverse entre hosts)
Tamaño del entorno	GBs (5GB o más por imagen)	MBs (imágenes reducidas)
Ejemplo	 Departamento completo	 Food Truck

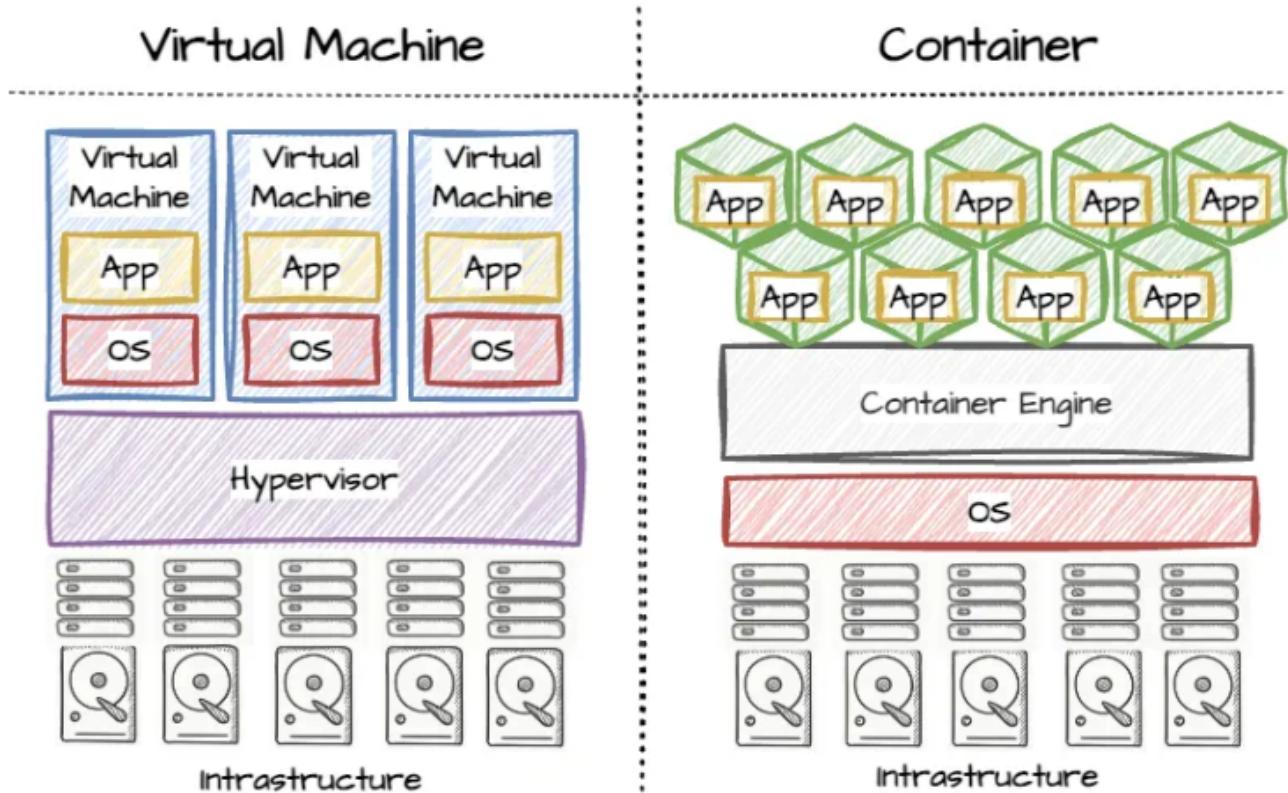
## Herramientas para Virtualización y Contenedores

Categoría	Herramientas comunes
Hypervisores	VirtualBox, VMWare, Parallels, Hyper-V
Contenedores	Docker, Podman, Containerd

## Conclusión

Los contenedores son la opción preferida para desarrollos modernos que necesitan agilidad, portabilidad y eficiencia. Sin embargo, las VMs siguen siendo útiles cuando se requiere un sistema operativo completo por aplicación.

El uso de Docker y otras herramientas de contenedores permite a los desarrolladores replicar entornos de forma rápida, segura y reproducible, facilitando tanto el desarrollo como el despliegue en producción.

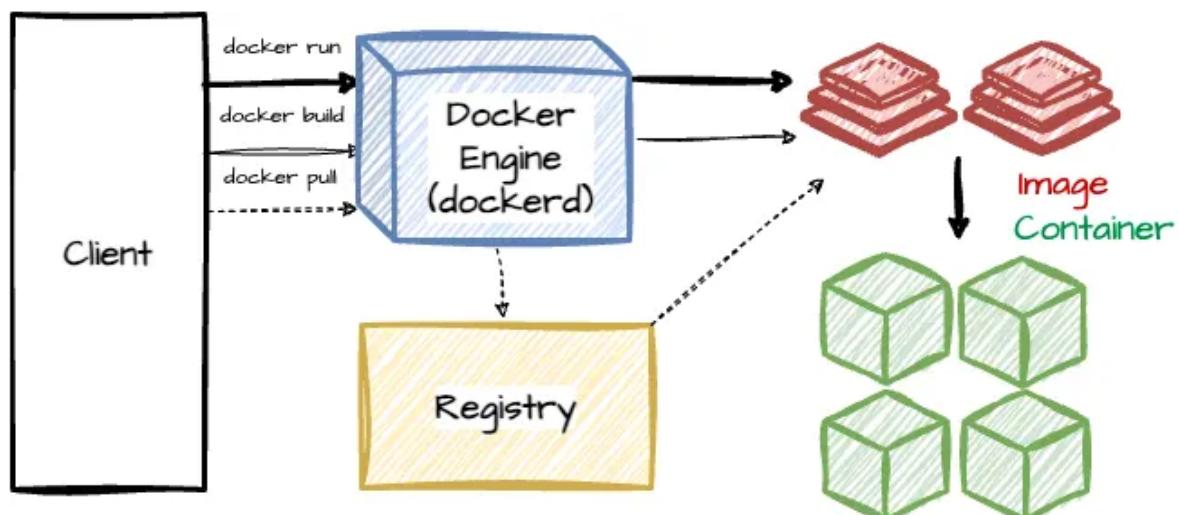


## Arquitectura de Docker

La arquitectura de Docker introduce un enfoque revolucionario en la forma en que desarrollamos, desplegamos y ejecutamos aplicaciones. A continuación, exploramos sus componentes clave, su funcionamiento en relación con el sistema operativo anfitrión, y cómo se diferencia de las máquinas virtuales tradicionales.

### 📦 Componentes principales

- Brief



- **Sistema Operativo Anfitrión (Host OS):** Es el sistema operativo real donde Docker se encuentra instalado (Linux, Windows, macOS). Es quien provee los recursos físicos y servicios de bajo nivel.

- **Docker Engine:** Plataforma que permite construir, ejecutar y gestionar contenedores. Incluye:
  - **dockerd:** daemon que escucha peticiones y gestiona objetos como imágenes y contenedores.
  - **containerd:** runtime de bajo nivel que se encarga de la ejecución real de los contenedores.
  - CLI (**docker**): herramienta de línea de comandos para interactuar con el daemon.
- **Contenedores:** Instancias en ejecución de imágenes Docker. Son procesos aislados que incluyen todo lo necesario para ejecutar una aplicación: código, dependencias, librerías, configuraciones, etc.
- **Sistema de archivos en capas (Union File System):** Las imágenes se componen de capas apiladas de solo lectura. Al crear un contenedor, se le agrega una capa superior de lectura-escritura. Esto permite eficiencia en almacenamiento y reutilización de capas comunes.

## 🔍 Otro enfoque: Contenedores vs. Máquinas Virtuales

Característica	Máquinas Virtuales	Contenedores Docker
Aislamiento	Completo (sistema operativo completo por VM)	A nivel de proceso
Peso	Pesado (varios GB)	Ligero (MB)
Tiempo de inicio	Lento (minutos)	Rápido (segundos)
Recursos	Requiere CPU, RAM y disco por adelantado	Comparte recursos con el host dinámicamente
Sistema operativo incluido	Sí	No (usa el kernel del host)
Ideal para	Simular entornos completos	Microservicios, CI/CD, despliegues portables

## ⚙️ Funcionamiento en el Host OS

Cada contenedor corre como un **proceso aislado** dentro del host, con:

- **su propio sistema de archivos** (montado desde la imagen)
- **su espacio de red virtual** (puertos, IP, reglas de firewall)
- **sus variables de entorno y configuración**

Pero todos comparten el mismo **kernel del sistema operativo anfitrión**, lo que evita la sobrecarga de tener múltiples kernels como sucede en las máquinas virtuales.

## 📦 ¿Por qué Containers?

Para entender mejor la propuesta de Docker, podemos usar una analogía con el transporte marítimo:

🚢 **Contenedores marítimos** revolucionaron el transporte de carga porque establecieron un formato único, estandarizado y apilable para todo tipo de mercancías, sin importar el contenido.

De forma similar:

- Un contenedor Docker **puede ejecutarse en cualquier sistema** con Docker Engine, como los contenedores físicos en cualquier barco.
- Las aplicaciones se empacan con todas sus dependencias, **aisladas del entorno**, evitando problemas de configuración cruzada.
- Se pueden **apilar, mover y desplegar fácilmente** en distintos entornos (dev, staging, prod).

 **Dato curioso:** El logo de Docker representa una ballena cargando contenedores... y no es casualidad 😊

Esta analogía resume por qué los contenedores se volvieron una pieza clave en la evolución hacia la nube, microservicios y DevOps.

## Imágenes y Contenedores

Para entender Docker en profundidad, vamos a usar otra analogía útil: **planos y edificios**.

- Una **imagen Docker** es como el plano de un edificio: define los materiales, el diseño y los pasos para construir.
- Un **contenedor Docker** es un edificio construido a partir de ese plano: es una instancia ejecutable, real, viva.

Una imagen contiene todas las instrucciones necesarias para construir un contenedor: sistema de archivos base, dependencias, configuraciones, puertos expuestos, etc. Estas instrucciones se definen generalmente en un archivo llamado **Dockerfile**, que actúa como la receta de la imagen.

Las imágenes pueden provenir de:

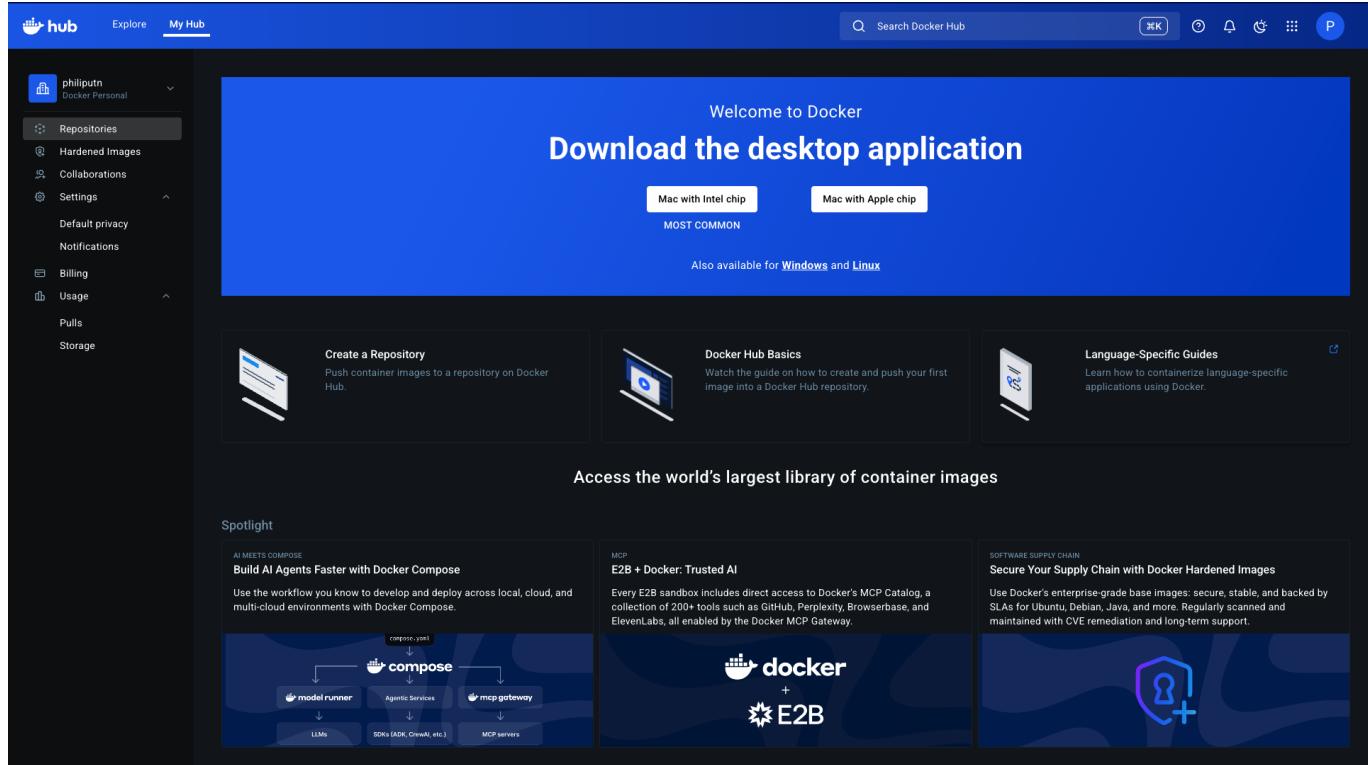
- Repositorios públicos como [Docker Hub](#), o
- Ser construidas localmente a partir de nuestro propio Dockerfile.

Una vez creada, una imagen puede usarse múltiples veces para levantar contenedores idénticos pero aislados entre sí. Esto es especialmente útil en ambientes de desarrollo, integración continua y despliegue a producción.

## El Docker Registry 📁

Cuando ejecutamos un comando como `docker run`, Docker necesita encontrar la imagen a partir de la cual levantar el contenedor. Para eso, busca en un **registry** (registro) de imágenes.

El más conocido es [Docker Hub](#), pero también existen registros privados que se pueden montar en empresas u organizaciones.



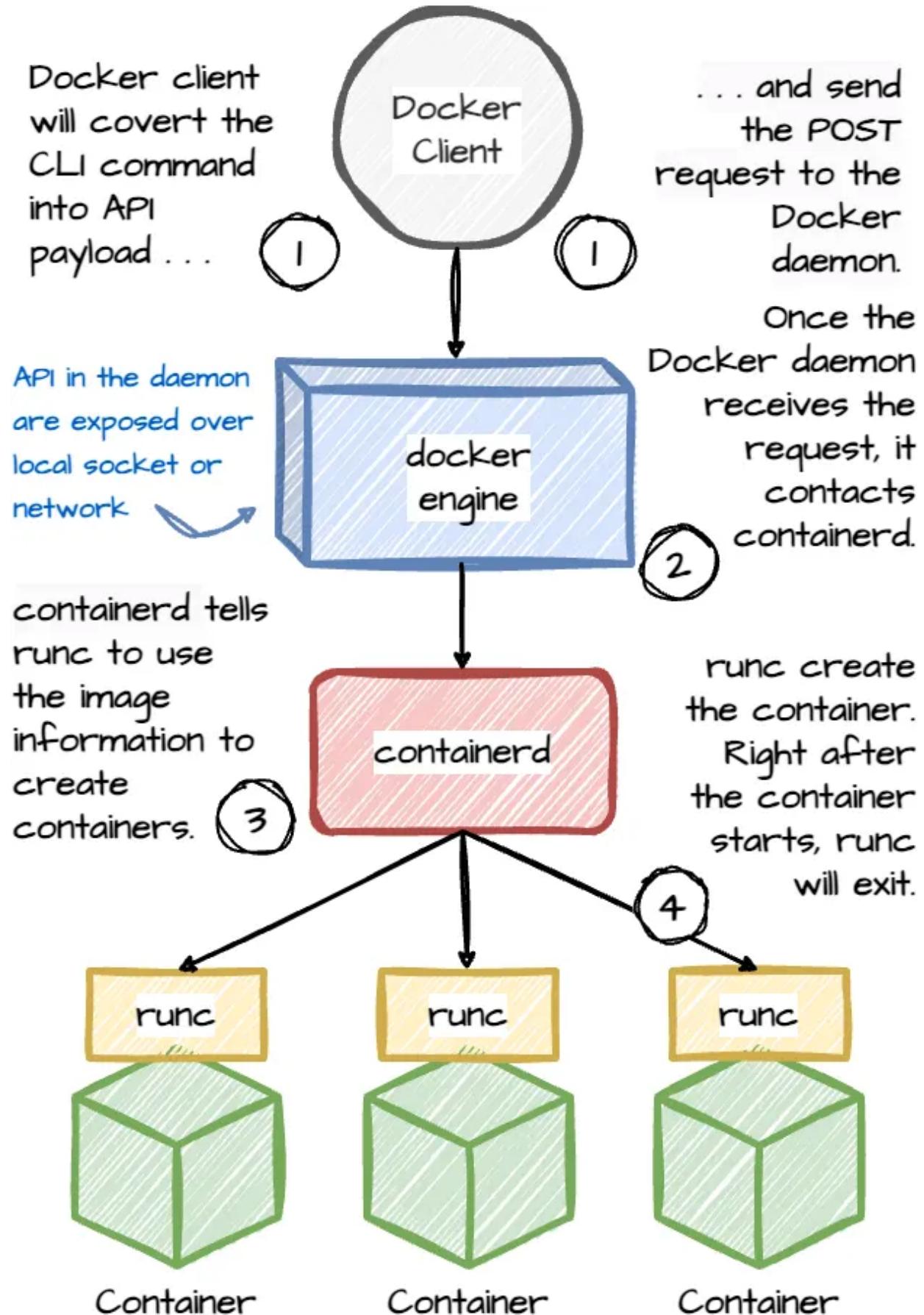
El flujo habitual es:

- Subimos nuestras imágenes con `docker push`
- Las traemos con `docker pull`

¿Qué pasa cuando ejecutamos `docker run`? 🧠⚙️

Si bien la instalación se documenta en el siguiente apartado imaginemos que ya hemos resuelto ese paso para ver el esquema de funcionamiento.

Cuando ejecutamos `docker run hello-world`, se pone en marcha un flujo muy interesante entre varios componentes internos de Docker. Lo detallamos a continuación y lo acompañamos con el siguiente diagrama:



1. **Docker Client:** Es el CLI (Command Line Interface) que utilizamos para interactuar con Docker.  
Convierte el comando en una llamada HTTP y lo envía al demonio de Docker.

2. **Docker Engine**: Es el demonio de Docker que escucha las solicitudes. Cuando recibe la instrucción, delega en **containerd** la gestión del contenedor.
3. **containerd**: Es un proceso de larga duración que gestiona el ciclo de vida del contenedor. Se encarga de buscar la imagen, convertirla a un formato compatible y pedir a **runc** que cree el contenedor.
4. **runc**: Es el ejecutor que finalmente crea el contenedor a partir de la imagen. Se comunica con el kernel del sistema operativo para establecer el espacio aislado donde se ejecutará el contenedor. Apenas inicia el contenedor, **runc** finaliza.

Este flujo permite levantar un contenedor liviano, rápido y con aislamiento, sin necesidad de levantar una máquina virtual completa. Esta diferencia es la que hace que Docker sea tan popular y eficiente en entornos de desarrollo y producción.

En resumen, Docker permite definir nuestras aplicaciones como imágenes, almacenarlas en registros, y luego ejecutarlas como contenedores usando un pipeline muy optimizado compuesto por **docker engine**, **containerd** y **runc**.

### 3.6 Sistema de Archivos en Docker: Capas, Volúmenes y Compartición

Uno de los aspectos más sorprendentes para quienes venimos del mundo de las máquinas virtuales tradicionales es cómo Docker gestiona el **sistema de archivos**. A diferencia de una VM, que requiere un disco virtual completo con sistema operativo y datos, Docker implementa un **modelo híbrido y por capas**, altamente eficiente y modular.

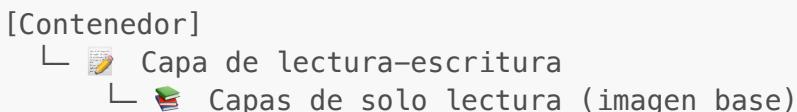
#### Capas de Sistema de Archivos: Union File System (UnionFS)

Cada imagen Docker está construida por **capas superpuestas** que forman un sistema de archivos unificado conocido como **UnionFS**. Estas capas:

- Son **de solo lectura** y se generan a partir de instrucciones del **Dockerfile** (**FROM**, **RUN**, **COPY**, etc.).
- Se **apilan secuencialmente**: cada instrucción genera una nueva capa.
- Al crear un contenedor a partir de una imagen, se agrega una **capa superior de lectura-escritura** (llamada *container layer*) donde se almacenan los cambios propios del contenedor en ejecución.

Este enfoque tiene múltiples ventajas:

- Ahorro de espacio: múltiples contenedores pueden compartir las mismas capas base.
- Rapidez: construir imágenes y levantar contenedores es mucho más rápido.
- Reproducibilidad: cada capa es inmutable y versionable.



#### Volúmenes Docker: Persistencia y Compartición

Por defecto, los cambios realizados en un contenedor **no persisten** una vez que este se elimina. Para resolver este problema, Docker ofrece los **volúmenes**, que permiten:

- **Persistir datos** más allá del ciclo de vida del contenedor.
- **Compartir archivos** entre contenedores.
- **Mapear carpetas del host** dentro del contenedor.

Existen dos formas principales de definir volúmenes:

- **Volúmenes gestionados por Docker:**

```
volumes:
  - nombre_del_volumen:/ruta/dentro/del/contenedor
```

- **Bind mounts (montaje directo desde el host):**

```
volumes:
  - ./datos-locales:/app/data
```

Esto resulta en una arquitectura mixta donde el contenedor accede a:

- Su propio sistema de archivos por capas.
- Directorios persistentes que pueden ser externos o internos al host.

## Diferencia Clave con Máquinas Virtuales

Aspecto	Máquina Virtual	Contenedor Docker
Sistema de archivos	Disco virtual completo (e.g., VDI)	Capas superpuestas + volúmenes
Tamaño base	5–15 GB mínimo	100s de MB (imagen mínima)
Acceso al sistema anfitrión	Limitado (vía carpetas compartidas)	Directo (mediante volúmenes)
Aislamiento	Completo (OS propio)	Parcial (mismo kernel, procesos aislados)

## En resumen

Docker redefine la gestión de archivos mediante un sistema híbrido y modular. Gracias al uso de capas inmutables, volúmenes y bind mounts, permite combinar:

- Portabilidad y aislamiento.
- Persistencia y colaboración.
- Flexibilidad para entornos de desarrollo, pruebas y producción.

Este enfoque lo convierte en una herramienta extremadamente poderosa y eficiente, que supera muchas de las limitaciones tradicionales del enfoque basado en máquinas virtuales.

## Instalación De Docker

Ya hemos documentado la instalación de Docker en distintas plataformas, sin embargo lo agregamos aquí también para completar el presente apunte.

A continuación, se detallan los pasos para instalar Docker en cada sistema operativo.

### 1.1 En Windows 10/11

1. Descargar Docker Desktop desde: <https://www.docker.com/products/docker-desktop/>
2. Ejecutar el instalador y seguir los pasos.
3. Al finalizar, reiniciar la PC si es necesario.
4. Verificar desde terminal (PowerShell o CMD):

```
docker --version  
docker compose version
```

 Si Docker está correctamente instalado, los comandos anteriores deberían devolver algo como:

- Docker version 28.3.x, build abc1234
- Docker Compose version v2.38.x

 Requiere tener habilitado **WSL2** (Subsistema de Windows para Linux versión 2), que permite ejecutar un entorno Linux directamente sobre Windows. Se recomienda instalar **Ubuntu** como distro por su compatibilidad, soporte extendido y facilidad de uso. Docker Desktop guía automáticamente en la instalación y configuración inicial si aún no está configurado.

### 1.2 En macOS

#### Opción recomendada: Docker Desktop

1. Descargar Docker Desktop desde: <https://www.docker.com/products/docker-desktop/>
2. Abrir el archivo **.dmg** y arrastrar Docker a la carpeta **Aplicaciones**.
3. Ejecutar Docker Desktop y seguir las instrucciones del asistente de configuración inicial.

Verificamos instalación:

```
docker --version  
docker compose version
```

#### Alternativa: Instalación con Homebrew

Para quienes prefieren usar herramientas desde la terminal y minimizar recursos de sistema:

```
brew install docker docker-compose
brew install colima
colima start
```

⚠️ **Colima** permite correr contenedores sin necesidad de Docker Desktop, utilizando el backend de virtualización nativo (**qemu**, **vz**, etc.). Ideal para Apple Silicon.

### Otras alternativas "Mac style"

Si buscás una experiencia más nativa, liviana y fluida, especialmente en Mac M1/M2/M3, se puede usar:

**OrbStack** → <https://orbstack.dev> Interfaz moderna, súper liviana, con excelente soporte para contenedores y máquinas virtuales.

### Comparativa: OrbStack vs Docker Desktop en macOS

Característica	OrbStack	Docker Desktop
 <b>Ligereza</b>	Muy liviano, bajo consumo de recursos	Pesado, puede consumir muchos recursos
 <b>Velocidad de arranque</b>	Muy rápido (menos de 1 segundo)	Lento (varios segundos)
 <b>Instalación</b>	Desde <b>.dmg</b> o vía <b>brew install orbstack</b>	Desde <b>.dmg</b> oficial o <b>brew install</b>
 <b>Docker Engine incluido</b>	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> Sí
 <b>Tiempo de inicio de contenedores</b>	Instantáneo (con pre-warmed VM)	Lento en frío, mejora luego del primer arranque
 <b>Compatibilidad con ARM/M1/M2/M3/M4</b>	Total, nativo para Apple Silicon	Compatible, pero más pesada con Intel/AMD64
 <b>Soporte para redes personalizadas</b>	Avanzado, integración con host mejorada	Limitado, requiere hacks o workarounds
 <b>Soporte para volúmenes</b>	Integración más rápida y eficiente con macOS	Más lento al montar volúmenes
 <b>Licencia</b>	Gratis y de código cerrado	Gratis con restricciones (requiere login)
 <b>Interfaz de usuario (UI)</b>	Minimalista, limpia, moderna	Cargada, más compleja
 <b>Ideal para estudiantes</b>	<input checked="" type="checkbox"/> Sí, especialmente en Apple Silicon	<input checked="" type="checkbox"/> Sí, aunque más pesado
 <b>Soporte empresarial</b>	 No oficial	<input checked="" type="checkbox"/> Sí, con planes pagados

### Conclusión:

Si estás trabajando en macOS (especialmente con chip Apple Silicon), **OrbStack** ofrece una experiencia más liviana, rápida y amigable para entornos de desarrollo educativo o personal. En cambio, **Docker Desktop** puede ser más completo para entornos empresariales que requieran soporte oficial o características avanzadas de Kubernetes.

### 1.3 En Linux (Ubuntu/Debian)

```
sudo apt update  
sudo apt install docker.io docker-compose -y  
sudo systemctl enable docker  
sudo systemctl start docker  
sudo usermod -aG docker $USER
```

 Reiniciar sesión para que se aplique el grupo **docker**

Verificar:

```
docker --version  
docker compose version
```

## Laboratorio 1 - Primeros pasos con contenedores Docker

 Probemos la instalación

No podía faltar el **hello-world** que ya mencionamos

Esta imagen de Docker Hub nos permite comprobar que todo está funcionando de forma correcta

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

[!INFO] La respuesta lo dice todo jeje

```
philip ~
$ docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
198f93fd5094: Pull complete
Digest: sha256:56433a6be3fda188089fb548eae3d91df3ed0d6589f7c2656121b911198df065
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Ahora sí nos pongamos serios, o no

En este primer laboratorio vamos a experimentar con un contenedor simple y liviano utilizando la imagen **busybox**, que es una de las más pequeñas y rápidas disponibles en Docker Hub. A través de este laboratorio vamos a:

- Analizar qué es **busybox** y qué contiene.
- Entender cómo se construye, ejecuta, detiene y reinicia un contenedor.
- Explorar el comportamiento de un contenedor desde adentro y desde afuera.
- Jugar con las herramientas incluidas en **busybox** para empezar a familiarizarnos con el entorno.

### ¿Qué es BusyBox? 📦

**BusyBox** es una imagen mínima que combina pequeñas versiones de muchas utilidades de UNIX en un solo binario. Incluye comandos como **ls**, **cp**, **mv**, **sh**, **echo**, entre muchos otros. Es ampliamente utilizada en sistemas embebidos, y en nuestro caso nos sirve como entorno de pruebas para comprender el funcionamiento básico de los contenedores Docker.

Desde el punto de vista de las **capas**:

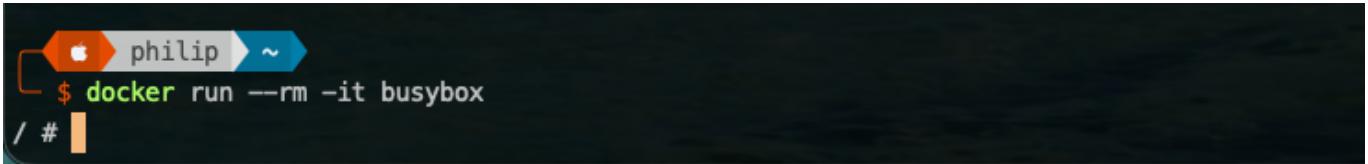
- Está basada en una estructura mínima, sin un sistema de paquetes completo.
- No incluye un gestor de servicios ni entorno gráfico, solo herramientas básicas de consola.
- Es ideal para explorar el ciclo de vida de un contenedor con bajo consumo de recursos.

## 1 Ejecución inicial

```
docker run --rm -it busybox
```

- **--rm**: elimina el contenedor una vez que termina la ejecución.
- **-it**: activa modo interactivo con terminal.
- **busybox**: nombre de la imagen.

Con este comando accedemos a una shell (**sh**) dentro del contenedor. Probá ejecutar:



```
Philip ~$ docker run --rm -it busybox
/ #
```

```
ls
whoami
cat /proc/version
```

## 2 Desde adentro del contenedor 🧠

Algunas acciones que podemos probar dentro del contenedor:

- Crear archivos: **touch archivo.txt**
- Consultar procesos: **ps**

Verificamos también las herramientas disponibles:

```
busybox --list
```



```
Philip ~$ docker run --rm -it busybox
/ # ls
bin dev etc home lib lib64 proc root sys tmp usr var
/ # whoami
root
/ # cat /proc/version
Linux version 6.10.14-linuxkit (root@buildkitsandbox) (gcc (Alpine 13.2.1_git20240309) 13.2.1 20240309, GNU ld (GNU Binutils) 2.42) #1 SMP Tue Oct 14 07:32:13 U
TC 2025
/ # busybox --list
[
[[
acpid
add-shell
```

## 3 Desde afuera del contenedor 🛡

Creamos un contenedor en modo pausado:

```
docker run -td --name prueba-busy busybox
```

**--name:** le damos un nombre al contenedor para identificarlo luego **-t: TTY** agrega la capacidad de terminal con shell al contenedor **-d: detached** es decir en segundo plano, el contenedor queda creado pero la consola vuelve al sistema operativo host.

Abrimos una nueva terminal y ejecutamos:

```
docker ps
```

```
philip ~ $ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8f433062ee6f busybox "sh" 4 minutes ago Exited (0) 4 minutes ago
e3fc7d2e5c56 ghcr.io/project-osrm/osrm-backend:latest "osrm-routed /data/a..." 24 hours ago Exited (0) 14 seconds ago
```

O también podemos ver su existencia en Docker Desktop:

The Docker Desktop interface shows the 'Containers' tab. It displays a search bar with the ID 'ic347b997e73bd09e69c962ec7f8732ee'. Below it, there's a toggle switch for 'Only show running containers'. A table lists one container:

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	prueba-busy	8f433062ee6f	busybox		N/A	8 minutes ago	

Y también podemos ver la imagen que se descargó con el primer **docker run** y luego se reutilizó para todos los demás casos

The Docker Desktop interface shows the 'Images' tab. It displays a search bar and a table listing three images:

	Name	Tag	Image ID	Created	Size	Actions
<input type="checkbox"/>	ghcr.io/project-osrm/osrm-backend	latest	729461bcc9ae	6 months ago	101.37 MB	
<input type="checkbox"/>	hello-world	latest	56433a6be3fd	3 months ago	16.91 KB	
<input type="checkbox"/>	busybox	latest	e3652a00a2fa	1 year ago	6.2 MB	

que también podemos inspeccionar por línea de comandos con:

```
docker inspect busybox
```

Notar que usamos el nombre de la imagen y no del contenedor

```
philip ~
$ docker inspect busybox
[
  {
    "Id": "sha256:e3652a00a2fabd16ce889f0aa32c38eec347b997e73bd09e69c962ec7f8732ee",
    "RepoTags": [
      "busybox:latest"
    ],
    "RepoDigests": [
      "busybox@sha256:e3652a00a2fabd16ce889f0aa32c38eec347b997e73bd09e69c962ec7f8732ee"
    ],
    "Parent": "",
    "Comment": "",
    "Created": "2024-09-26T21:31:42Z",
    "DockerVersion": "",
    "Author": "",
    "Architecture": "arm64",
    "Variant": "v8",
    "Os": "linux",
    "Size": 1913613,
    "GraphDriver": {
      "Data": null,
      "Name": "overlayfs"
    },
    "RootFS": {
      "Type": "layers",
      "Layers": [
        "sha256:1a382740c5642e4607412a341df3716c22287ffa6adf92eaff54e079a1902f05"
      ]
    },
    "Metadata": {
      "LastTagTime": "2025-10-31T01:03:58.037070763Z"
    },
    "Descriptor": {
      "mediaType": "application/vnd.oci.image.index.v1+json",
      "digest": "sha256:e3652a00a2fabd16ce889f0aa32c38eec347b997e73bd09e69c962ec7f8732ee",
      "size": 9535
    },
    "Config": {
      "Cmd": [
        "sh"
      ],
      "Entrypoint": null,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
      ],
      "Labels": null,
      "OnBuild": null,
      "User": "",
      "Volumes": null,
      "WorkingDir": ""
    }
  }
]
```

También podemos inspeccionar el contenedor:

```
docker inspect prueba-busy
```

La salida es demasiado extensa pero arroja toda la información que podemos necesitar acerca del contenedor inspeccionado

Iniciemos ahora el contenedor:

```
docker start -ai prueba-busy
```

-ai: reconecta a la consola del contenedor

En la consola del contenedor vamos a crear un archivo para comprobar el sistema de archivos del contenedor

```
touch /home/prueba.txt
```

```
ls /home/
```

Con ls comprobamos que el archivo prueba.txt se creó correctamente.

#### 4 Detenemos y reiniciamos

```
docker stop prueba-busy  
docker start -ai prueba-busy
```

El parámetro -ai nos reconecta a la terminal del contenedor.

Luego podemos comprobar la persistencia del archivo prueba.txt

```
ls /home/
```

#### 5 Finalmente eliminamos el contenedor

El contenedor que reamos sin --rm se persistió y por más que lo detengamos sigue residiendo en nuestro sistema operativo host, por lo que lo ideal es eliminarlo si es una prueba que no vamos a usar

```
docker stop prueba-busy  
docker container rm prueba-busy
```

Primero nos aseguramos que esté detenido Y luego lo eliminamos con docker container rm

E incluso podemos eliminar la imagen, en ese caso la próxima vez la tendrá que volver a descargar

```
docker rmi busybox
```

```

philip ~
$ docker rmi busybox
Untagged: busybox:latest
Deleted: sha256:e3652a00a2fabd16ce889f0aa32c38eec347b997e73bd09e69c962ec7f8732ee

philip ~
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
hello-world         latest   56433a6be3fd  2 months ago  16.9kB
ghcr.io/project-osrm/osrm-backend  latest   729461bcc9ae  6 months ago  101MB

```

## 6 Bonus: docker/whalesay 🐋

Para hacerlo un poco más entretenido vamos ahora a crear nuestra propia imagen, para jugar un poco

### Paso 1: Crear un nuevo Dockerfile

Creamos una carpeta para trabajar:

```

mkdir docker-whalesay
cd docker-whalesay

```

Y luego creamos un archivo llamado **Dockerfile** con el siguiente contenido:

```

FROM debian:bullseye-slim

# Instalamos cowsay y fortune
RUN apt-get update && \
    apt-get install -y cowsay fortune figlet && \
    ln -s /usr/games/cowsay /usr/bin/cowsay && \
    ln -s /usr/games/fortune /usr/bin/fortune

# Copiamos un script de entrada
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

ENTRYPOINT ["./entrypoint.sh"]

```

### 📘 Paso 2: Crear el script de entrada `entrypoint.sh`

Creamos un archivo llamado `entrypoint.sh` en el mismo directorio, con el siguiente contenido:

```

#!/bin/sh

MESSAGE="$@"

if [ -z "$MESSAGE" ]; then
  echo "⚠️ No se recibió ningún mensaje. Usá el contenedor pasando un mensaje como argumento."
  exit 1

```

```
fi

figlet "🐳 Contenedor Dice:"
echo ""
echo "$MESSAGE" | cowsay
```

Este script toma el texto pasado como argumento al ejecutar el contenedor (\$@), y si no recibe nada, da un mensaje de error. Si recibe texto, lo transforma en arte ASCII.

### 💡 Paso 3: Construir la imagen

Desde la carpeta donde se encuentran el **Dockerfile** y el **entrypoint.sh**, ejecutamos:

```
docker build -t whalesay-custom .
```

### ✍️ Paso 4: Ejecutar el contenedor con un mensaje

Ahora ejecutamos el contenedor con un mensaje personalizado:

```
docker run --rm whalesay-custom "Hola Backend de Aplicaciones 🖥🚀"
```

### 🧹 Limpieza opcional

Si queremos eliminar la imagen:

```
docker rmi whalesay-custom
```

## Resumen

Este laboratorio nos permitió:

- Explorar una primera visión del ciclo de vida de un contenedor.
- Conocer una imagen minimalista como **busybox** desde Docker Hub.
- Jugar desde la terminal como si fuera un mini sistema operativo.
- Construir una imagen para el whalesay desde un archivo **Dockerfile**

### [!NOTE]

En todos los primeros ejemplos utilizamos **docker run** que entiende que la imagen está en el caché local o la busca desde el repositorio, y en el último caso utilizamos **docker build** que compone la imagen desde los comandos vertidos en el **Dockerfile**.

## Construcción y Gestión de Imágenes

Docker nos permite definir, construir, gestionar y publicar imágenes de contenedor que contienen todo lo necesario para ejecutar nuestras aplicaciones. En este apartado vamos a profundizar en la construcción de imágenes mediante Dockerfiles, su gestión local y la publicación en Docker Hub.

## Dockerfile: La Receta de la Imagen

Un **Dockerfile** es un archivo de texto que contiene un conjunto de instrucciones que Docker utiliza para construir una imagen. Cada instrucción agrega una capa a la imagen final.

Instrucciones comunes:

- **FROM**: Imagen base desde la cual se construye.
- **WORKDIR**: Directorio de trabajo dentro del contenedor.
- **COPY / ADD**: Copian archivos del host al contenedor. **ADD** permite descargar desde URLs o descomprimir.
- **RUN**: Ejecuta comandos durante la construcción de la imagen.
- **ENV**: Define variables de entorno.
- **CMD / ENTRYPOINT**: Define el comando que se ejecuta al iniciar el contenedor.
- **EXPOSE**: Documenta los puertos que usará la aplicación (no los expone en el host).

## Ejemplo

```
FROM openjdk:21
WORKDIR /app
COPY target/app.jar app.jar
CMD ["java", "-jar", "app.jar"]
```

**FROM** indica la imagen base sobre la cual se construirá nuestra imagen personalizada.

En este caso, usamos la imagen oficial de OpenJDK versión 21, que ya contiene el JDK preinstalado y listo para ejecutar aplicaciones Java.

Este paso es clave para evitar tener que instalar Java manualmente dentro del contenedor.

**WORKDIR** establece el directorio de trabajo dentro del sistema de archivos del contenedor.

Todas las instrucciones siguientes (como COPY, CMD, etc.) se ejecutarán desde este directorio.

Si no existe, Docker lo crea automáticamente.

**COPY** toma un archivo desde el sistema host y lo copia dentro del contenedor.

En este caso, estamos copiando el archivo .jar resultante del proceso de empaquetado Maven (target/app.jar) al directorio de trabajo dentro del contenedor (/app/app.jar).

Esta es la aplicación que vamos a ejecutar dentro del contenedor.

**CMD** indica el comando por defecto que se ejecutará cuando el contenedor se inicie.

En este caso, estamos diciendo que queremos ejecutar:

```
java -jar app.jar
```

Usamos la forma JSON de CMD, que es la más recomendada porque evita la ejecución de una shell intermedia.

El Dockerfile es reproducible y declarativo: define de forma simple qué necesitamos y cómo queremos ejecutar nuestra aplicación sin importar el entorno del host.

## Detalle de Instrucciones Clave

### 1. FROM

Define la imagen base. Debe estar en la primera línea. Se recomienda evitar `latest` y usar versiones explícitas (e.g., `openjdk:21-jdk-slim`).

Es importante tener en cuenta que las imágenes de Docker suelen estar etiquetadas con versiones, lo que permite identificar las distintas variantes disponibles de la imagen. Estas versiones se especifican mediante tags que siguen al nombre de la imagen, por ejemplo, en la instrucción `FROM nombre-imagen: <version>`, `<version>` indica la versión específica de la imagen que se utilizará en la construcción del contenedor Docker. No olvidemos que nuestro objetivo es que nuestra aplicación se ejecute correctamente independientemente del lugar donde se decida ejecutar y por esto es crítico que fijemos la versión precisa de los distintos componentes de nuestra aplicación.

### 2. WORKDIR

Establece un directorio de trabajo para instrucciones posteriores.

```
WORKDIR /app
```

### 3. COPY / ADD

Copian archivos desde el host al contenedor. `ADD` permite URLs y archivos comprimidos.

```
COPY archivo.txt /app/
ADD https://sitio.com/data.csv /app/data.csv
```

`COPY` copia el archivo `archivo.txt` que va a buscar en el directorio local del sistema de archivos del sistema operativo host al directorio `/app/` del sistema de archivos del contenedor. `ADD` descarga el contenido del archivo `data.csv` desde `https://sitio.com/` y lo guarda en `/app/data.csv` en el sistema de archivos de contenedor.

Estos comandos soportan wildcards y expresiones regulares.

### 4 . RUN

Ejecuta comandos durante la construcción.

```
RUN apt-get update && apt-get install -y curl \
&& rm -rf /var/lib/apt/lists/*
```

**RUN** en este caso va a instalar los comandos `apt-get update` y luego el comando `apt-get install -y curl` y finalmente `rm -rf /var/...`

Utilizando el operador de concatenación, se asegura que el siguiente comando solo se ejecute si la ejecución del comando anterior tuvo éxito.

## 5. ENV

La instrucción **ENV** se utiliza en un **Dockerfile** para definir variables de entorno dentro de la imagen del contenedor. Estas variables pueden ser utilizadas por la aplicación o por otros comandos dentro del contenedor.

```
ENV SPRING_PROFILES_ACTIVE=dev
```

En este ejemplo, se está definiendo la variable de entorno **SPRING\_PROFILES\_ACTIVE** con el valor **dev**. Esta variable será visible dentro del contenedor cuando la aplicación se ejecute, y en el caso de una aplicación Spring Boot, indicará que debe activarse el perfil **dev**.

### Valor por defecto y sobreescritura

El valor establecido con **ENV** en el **Dockerfile** actúa como **valor por defecto**. Sin embargo, al momento de ejecutar el contenedor, es posible sobreescribir este valor utilizando la opción `-e` o `--env` del comando `docker run`:

```
docker run -e SPRING_PROFILES_ACTIVE=prod miimagen
```

En este caso, aunque el **Dockerfile** tiene el valor **dev**, se usará **prod** como valor efectivo dentro del contenedor.

Este mecanismo permite mantener valores por defecto en el **Dockerfile**, pero adaptarlos según el entorno de despliegue: desarrollo, testing, producción, etc.

Es una práctica recomendada definir valores seguros o representativos por defecto con **ENV**, y luego sobreescribirlos externamente según sea necesario.

## 6. CMD vs ENTRYPOINT

En Docker, tanto **CMD** como **ENTRYPOINT** se utilizan para definir el comando por defecto que se ejecutará cuando se inicie un contenedor a partir de la imagen. Sin embargo, tienen diferencias importantes que conviene entender claramente.

### CMD

- Especifica el **comando por defecto** que se ejecutará al iniciar el contenedor.
- **Puede ser sobreescrito** si al momento de ejecutar el contenedor (`docker run`) se proporciona un nuevo comando.

- Si se define más de una vez en el Dockerfile, **solo se toma en cuenta la última**.
- Puede tener dos formas:
  - Forma tipo **shell**: `CMD npm start`
  - Forma tipo **exec** (recomendada): `CMD ["npm", "start"]`

Ejemplo:

```
CMD ["npm", "start"]
```

Si luego ejecutamos:

```
docker run mi-imagen npm test
```

Docker ignorará el `CMD` definido y ejecutará `npm test`.

## ENTRYPOINT

- Define el **comando principal** que siempre se ejecutará al iniciar el contenedor.
- **No puede ser sobreescrito** con argumentos pasados en `docker run`, salvo que se use la opción `--entrypoint`.
- Sirve para establecer un comportamiento fijo del contenedor, por ejemplo, que siempre ejecute una determinada aplicación.
- Al igual que `CMD`, puede definirse en forma tipo shell o exec (se recomienda exec).

Ejemplo:

```
ENTRYPOINT ["npm"]
CMD ["start"]
```

Este patrón permite que:

```
docker run mi-imagen          # ejecuta: npm start
docker run mi-imagen test      # ejecuta: npm test
```

## Recomendación

Usar `ENTRYPOINT` para establecer el comando principal (que difícilmente cambiará), y `CMD` para definir parámetros por defecto que sí puedan ser modificados. Esta combinación brinda flexibilidad y claridad al construir imágenes reutilizables y configurables.

📌 **Tip:** Si solo se necesita un comando modificable, usar **CMD**. Si queremos que el contenedor actúe como un ejecutable, usar **ENTRYPOINT**.

## 7. EXPOSE

Documenta el puerto utilizado.

```
EXPOSE 8080
```

## Construcción de Imágenes

Para construir la imagen, vamos a suponer que estamos dentro del proyecto maven app, y que app es un proyecto que se va a empaquetar en app.jar, si tomamos este supuesto sabemos que el siguiente comando:

```
mvn clean package # si es una app Java
```

Va a generar el archivo **target/app.jar**

Luego:

```
docker build -t mi-imagen .
```

### [!NOTE]

El último parámetro de docker build es el directorio de trabajo o destino por lo que el **.** final del comando corresponde con el directorio actual y no puede ser omitido.

Esto crea una imagen, que puede ser verificada con:

```
docker image ls
```

Columnas importantes:

- REPOSITORY: Nombre de la imagen
- TAG: Versión
- IMAGE ID: Identificador único
- CREATED: Fecha de creación
- SIZE: Tamaño en disco

## Gestión de Imágenes

### Ver imágenes locales

```
docker images
```

## Eliminar imágenes

```
docker image rm <image>
```

Puede usarse nombre, ID completo o los primeros caracteres del ID.

## Limpiar imágenes sin uso

```
docker image prune
```

## Docker Hub: Publicar y Compartir

[Docker Hub](#) es la plataforma de registro de imágenes Docker más conocida. Permite almacenar, compartir y descargar imágenes públicas o privadas.

### Publicar una Imagen

1. Crear cuenta y loguearse:

```
docker login
```

2. Crear repositorio en Docker Hub.

3. Taggear la imagen:

```
docker build -t usuario/repo:tag .
```

4. Subirla:

```
docker push usuario/repo:tag
```

5. Descargarla desde otro host:

```
docker pull usuario/repo:tag
```

## Creación de Contenedores Docker

En esta sección vamos a trabajar con los contenedores Docker: cómo se crean, se ejecutan, se detienen, se eliminan y cómo gestionamos sus datos persistentes. Mientras que las imágenes son plantillas, los contenedores son instancias vivas de esas plantillas. Vamos a recorrer paso a paso todo lo necesario para manipularlos.

### Recordemos ¿Qué es un Contenedor?

Como ya mencionamos repetidas veces los contenedores son instancias ejecutables de imágenes Docker. Cuando creamos una imagen, todavía no está corriendo ni es capaz de correr a no ser que se cree un contenedor a partir de ella. Es solo un conjunto de instrucciones. Para que la aplicación "viva", necesitamos ejecutarla en un contenedor. Cada contenedor es un proceso aislado, con su propio sistema de archivos, red y entorno, pero que se ejecuta sobre el kernel del sistema operativo anfitrión.

### Crear y Ejecutar Contenedores

Podemos iniciar un contenedor de dos formas:

- En dos pasos:

```
docker create --name mi-contenedor mi-imagen  
docker start mi-contenedor
```

- O directamente con `docker run`, que combina ambos pasos:

```
docker run --name mi-contenedor mi-imagen
```

Opcionalmente, podemos agregar:

- `-d` para ejecutarlo en segundo plano (detached mode)
- `-it` para abrir una terminal interactiva
- `-p` para mapear puertos del host al contenedor
- `-v` para montar volúmenes
- `-e` para pasar variables de entorno

### Detener y Eliminar Contenedores

Una vez que el contenedor está corriendo, podemos detenerlo y eliminarlo fácilmente:

```
# Detener un contenedor  
docker stop mi-contenedor  
  
# Eliminar un contenedor detenido  
docker rm mi-contenedor
```

```
# O eliminar forzadamente un contenedor en ejecución  
docker rm -f mi-contenedor
```

## Logs y Estado del Contenedor

Podemos ver qué está ocurriendo dentro del contenedor utilizando los logs:

```
# Ver logs del contenedor  
docker logs mi-contenedor  
  
# Seguir logs en tiempo real  
docker logs -f mi-contenedor  
  
# Ver solo las últimas líneas  
docker logs --tail 20 mi-contenedor
```

Para ver el estado de todos los contenedores:

```
docker ps -a
```

## Puertos y Comunicación

Cuando queremos que una aplicación dentro del contenedor sea accesible desde el exterior, debemos mapear puertos:

```
docker run -d -p 9001:9001 --name api-spring api-spring
```

Esto mapea el puerto **9001** del host al **9001** del contenedor. Si no hacemos esto, el servicio dentro del contenedor no podrá ser accedido desde fuera.

## Volúmenes y Datos Persistentes

Los volúmenes nos permiten guardar datos fuera del ciclo de vida del contenedor. Si eliminamos un contenedor, los datos en un volumen permanecen intactos.

```
# Crear un volumen  
docker volume create datos-persistentes  
  
# Usarlo en un contenedor  
docker run -v datos-persistentes:/app/data mi-imagen  
  
# Ver volúmenes existentes  
docker volume ls
```

En un apartado específico trataremos el uso de volúmenes para trabajar los datos persistentes en el sistema operativo anfitrión.

### Docker Cheat Sheet: Comandos Clave

Comando	Descripción	Ejemplo
<code>docker create</code>	Crea un contenedor sin ejecutarlo	<code>docker create --name prueba ubuntu</code>
<code>docker run</code>	Crea y ejecuta un contenedor	<code>docker run -it ubuntu</code>
<code>docker start</code>	Inicia un contenedor detenido	<code>docker start prueba</code>
<code>docker stop</code>	Detiene un contenedor en ejecución	<code>docker stop prueba</code>
<code>docker rm</code>	Elimina un contenedor detenido	<code>docker rm prueba</code>
<code>docker logs</code>	Muestra los logs del contenedor	<code>docker logs prueba</code>
<code>docker ps -a</code>	Lista todos los contenedores (activos e inactivos)	<code>docker ps -a</code>
<code>docker run -p</code>	Mapea puertos entre host y contenedor	<code>docker run -p 8080:80 nginx</code>
<code>docker run -v</code>	Monta un volumen persistente	<code>docker run -v datos:/data mi-imagen</code>
<code>docker volume create</code>	Crea un volumen	<code>docker volume create datos</code>
<code>docker volume ls</code>	Lista los volúmenes disponibles	<code>docker volume ls</code>
<code>docker exec -it</code>	Ejecuta un comando dentro del contenedor en modo interactivo	<code>docker exec -it prueba /bin/sh</code>

Esta tabla es un excelente punto de referencia para reforzar los comandos vistos en clase y tener a mano los más usados al trabajar con contenedores. ¡Seguimos avanzando!

## Laboratorio 2 - Ejecutando una Aplicación Java en un Contenedor Docker

En este laboratorio vamos a construir y ejecutar una aplicación Java sencilla desde un contenedor Docker. El objetivo es consolidar los conocimientos sobre:

- Construcción de imágenes Docker desde un `.jar` empaquetado
- Uso de variables de entorno
- Interacción entre el host y el contenedor
- Exploración del entorno base `openjdk`

La aplicación está empaquetada en Maven y responde con una frase curiosa basada en variables de entorno.

## 1. Estructura del Proyecto

```
.  
└── Dockerfile  
└── pom.xml  
└── src  
    └── main  
        └── java  
            └── utnfcc  
                └── isi  
                    └── back  
                        └── docker  
                            └── app  
                                └── DockerFunApp.java
```

## 2. Código Fuente

El archivo `pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
http://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <groupId>utnfcc.isi.back.docker</groupId>  
    <artifactId>docker-fun-app</artifactId>  
    <version>1.0.0</version>  
    <properties>  
        <maven.compiler.source>21</maven.compiler.source>  
        <maven.compiler.target>21</maven.compiler.target>  
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
  
    <configuration.main.class>utnfcc.isi.back.docker.app.DockerFunApp</configur  
    ation.main.class>  
    </properties>  
  
<build>  
    <plugins>  
        <!-- Ejecutar con: mvn -q exec:java -->  
        <plugin>  
            <groupId>org.codehaus.mojo</groupId>  
            <artifactId>exec-maven-plugin</artifactId>  
            <version>3.5.1</version>  
            <configuration>  
                <mainClass>${configuration.main.class}</mainClass>  
            </configuration>  
        </plugin>  
        <!-- (Opcional) empaquetar jar ejecutable -->  
        <plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jar-plugin</artifactId>
<version>3.4.2</version>
<configuration>
    <archive>
        <manifest>
            <mainClass>${configuration.main.class}</mainClass>
        </manifest>
    </archive>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

La clase principal se encuentra en [DockerFunApp.java](#):

```
package utnfc.isi.back.docker.app;

import java.time.LocalDateTime;
import java.util.Random;

public class DockerFunApp {

    public static void main(String[] args) {
        System.out.println("🌐 iHola desde Docker!");
        System.out.println("📅 Fecha y hora del contenedor: " +
LocalDateTime.now());
        System.out.println("🎲 Número aleatorio dentro del contenedor: " +
new Random().nextInt(100));
        System.out.println("💻 Usuario del sistema dentro del contenedor:
" + System.getProperty("user.name"));
        System.out.println("📁 Directorio de trabajo: " +
System.getProperty("user.dir"));
        String usr = System.getenv().getOrDefault("APP_USER",
"BackendDev");
        System.out.println("Hola " + usr);
    }
}
```

### 3. Construcción del .jar

```
mvn clean package
```

Esto genera el archivo [target/docker-fun-app-1.0.0.jar](#), que usaremos para construir la imagen.

Podemos probar la aplicación ejecutando en el entorno local:

```
java -jar target/docker-fun-app-1.0.0.jar
```

```
[philip ~ ... material-docker > projects > docker-java-fun-app >]
$ java -jar target/docker-fun-app-1.0.0.jar
iHola desde Docker!
[17] Fecha y hora del contenedor: 2025-10-31T01:51:25.398010
[77] Número aleatorio dentro del contenedor: 77
[philip] Usuario del sistema dentro del contenedor: philip
[material-docker] Directorio de trabajo: /Users/philip/Developer/Facu/Back/material-docker/projects/docker-java-fun-app
Hola BackendDev
```

#### 4. Dockerfile explicado paso a paso

```
FROM openjdk:21
WORKDIR /app
COPY target/docker-fun-app-1.0.0.jar app.jar
ENV APP_USER="Feli Steffo"
CMD ["java", "-jar", "app.jar"]
```

- **FROM openjdk:21:** Imagen base oficial de Java 21. Contiene un sistema Linux con Java preinstalado.
- **WORKDIR /app:** Directorio de trabajo dentro del contenedor.
- **COPY target/docker-fun-app-1.0.0.jar app.jar:** Copia el .jar compilado al contenedor.
- **ENV APP\_USER=...**: Variable de entorno accesible desde el código Java.
- **CMD:** Comando que se ejecuta al iniciar el contenedor.

¿Qué contiene la imagen **openjdk:21**? Es una imagen basada en Debian que incluye OpenJDK 21 preinstalado. No es un sistema operativo completo como en una VM, pero tiene lo suficiente para ejecutar aplicaciones Java. Podés inspeccionarla conectándote al contenedor y usando comandos como `cat /etc/os-release` o `java -version`.

#### 5. Construcción de la Imagen

```
docker build -t docker-java-app .
```

Nota: no olvidar el . del final!

#### 6. Ejecución del Contenedor

```
docker run --rm docker-java-app
```

```
philip ~ ... material-docker projects docker-java-fun-app
$ docker run --rm docker-java-app
iHola desde Docker!
Fecha y hora del contenedor: 2025-10-31T05:00:43.476305004
Número aleatorio dentro del contenedor: 8
Usuario del sistema dentro del contenedor: root
Directorio de trabajo: /app
Hola Feli Steffo
```

Para probar con otra variable:

```
docker run --rm -e APP_USER="Backend 3K2" docker-java-app
```

**-e**: permite establecer un valor en el momento de ejecución para la variable de entorno.

```
philip ~ ... material-docker projects docker-java-fun-app
$ docker run --rm -e APP_USER="Backend 3K2" docker-java-app
iHola desde Docker!
Fecha y hora del contenedor: 2025-10-31T05:03:08.503430752
Número aleatorio dentro del contenedor: 63
Usuario del sistema dentro del contenedor: root
Directorio de trabajo: /app
Hola Backend 3K2
```

## 7. Exploración del Entorno

Podemos correr el contenedor en modo interactivo para inspeccionarlo:

```
docker run -it --entrypoint sh docker-java-app
```

Comandos útiles dentro del contenedor:

```
java -version
cat /etc/os-release
ls -l /app
```

## 8. Conclusión

Este laboratorio nos permite:

- Repasar la construcción de imágenes desde proyectos Java
- Explorar el entorno base que provee una imagen oficial ([openjdk](#))
- Comprender la diferencia entre imagen y contenedor
- Usar variables de entorno y modificar la ejecución

# Proceso de Desarrollo con Docker y Spring boot

Para implementar el despliegue de una aplicación dentro de un contenedor Docker, comenzamos creando un archivo **Dockerfile** en la raíz del proyecto. Este archivo define el proceso de construcción de la **Imagen Docker**, que incluirá todo lo necesario para ejecutar nuestra aplicación:

- Una versión ligera del sistema operativo (por ejemplo, Alpine Linux).
- Un ambiente de ejecución adecuado (como una JVM o JDK).
- Los archivos de la aplicación (por ejemplo, el `.jar` construido con Maven).
- Cualquier dependencia adicional.
- Variables de entorno necesarias para la ejecución.

Una vez creada, esta imagen puede publicarse en un **registro de imágenes** como Docker Hub, permitiendo que otros desarrolladores o servidores de despliegue puedan descargarla y ejecutarla con facilidad.

Este enfoque garantiza un entorno de ejecución **consistente y reproducible**, independientemente del sistema operativo subyacente, simplificando el proceso de colaboración y despliegue entre diferentes equipos.

## Panorama Completo

Antes de profundizar en los detalles, veamos una visión general del proceso completo con un ejemplo simple y concreto.

Contamos con una **aplicación Java** utilizando el framework **Spring**, la cual expone un endpoint mediante la anotación `@RestController` en la URI:

```
/messages/hello-world
```

Este endpoint devuelve un mensaje simple:

```
return ResponseEntity.ok("Hello World");
```

La aplicación escucha en el puerto **9001** bajo el context path **/docker-spring**, por lo que el endpoint completo será:

```
GET http://localhost:9001/docker-spring/messages/hello-world
```

Para ejecutar esta aplicación de forma tradicional, necesitaríamos:

- Un sistema operativo (Windows, Linux, macOS).
- Una JVM (Java Virtual Machine), versión 21.
- El archivo `.jar` compilado del proyecto.
- Ejecutar manualmente `java -jar docker-spring.jar` desde la terminal.

¿Cómo simplificamos esto con Docker?

Agrupamos todo lo necesario dentro de un contenedor. Para ello, creamos un **Dockerfile** con las siguientes instrucciones:

```
FROM openjdk:21-alpine
WORKDIR /app
COPY target/docker-spring.jar app.jar
EXPOSE 9001
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Recordemos: ¿Qué hace cada instrucción?

- **FROM openjdk:21-alpine**: Usa como base una imagen que incluye Alpine Linux (una distro minimalista y segura) y Java 21.
- **WORKDIR /app**: Define el directorio de trabajo dentro del contenedor.
- **COPY target/docker-spring.jar app.jar**: Copia el **.jar** generado por Maven al contenedor.
- **EXPOSE 9001**: Informa a Docker que la aplicación usará ese puerto.
- **ENTRYPOINT**: Define el comando que se ejecutará al iniciar el contenedor.

## Resultado

Con esta imagen ya construida, cualquier usuario o servidor puede:

- Descargar la imagen.
- Crear un contenedor.
- Ejecutar la aplicación sin preocuparse por las versiones de Java, dependencias o sistema operativo.

Este enfoque nos permite empaquetar nuestra aplicación como una unidad de despliegue autosuficiente y portable, eliminando los clásicos problemas de "funciona en mi máquina".

En los siguientes bloques desarrollaremos paso a paso este flujo como parte del tercer laboratorio de Docker.

## 🏗 Construcción de la Imagen Docker para una Aplicación Spring Boot

Bien, ya hemos creado contenedores en base a imágenes de Docker Hub, y también construimos una imagen con una aplicación Java y el contenedor para poder ejecutarla. Ahora es momento de saltar al primer contenedor de un microservicio spring.

Una vez desarrollada nuestra aplicación Spring Boot, el siguiente paso es **compilarla, empaquetarla y contenerizarla**. A continuación, describimos el proceso completo, desde la generación del **.jar** hasta la ejecución del contenedor en Docker.

### 1. 📦 Empaquetar la Aplicación

Desde la raíz del proyecto, ejecutamos el siguiente comando Maven para compilar y empaquetar la aplicación:

```
mvn clean compile package
```

💡 Este paso genera un archivo `.jar` dentro del directorio `target/`, el cual será utilizado para construir la imagen Docker.

## 2. 🛠️ Construir la Imagen Docker

Con el `.jar` generado, procedemos a construir la imagen usando `docker build`. Desde la raíz del proyecto, ejecutamos:

```
docker build -t docker-spring .
```

💡 `-t docker-spring` define la etiqueta (tag) de la imagen. El punto `.` indica que el `Dockerfile` se encuentra en el directorio actual.

### ✓ Verificar la Imagen

Para verificar que la imagen se construyó correctamente:

```
docker image ls
```

### 📋 Significado de las columnas

- **REPOSITORY**: nombre de la imagen.
- **TAG**: versión de la imagen.
- **IMAGE ID**: identificador único de la imagen.
- **CREATED**: fecha de creación.
- **SIZE**: espacio en disco que ocupa.

## 3. 🚀 Ejecutar la Aplicación en un Contenedor

Con la imagen creada, ejecutamos un contenedor basado en ella con el siguiente comando:

```
docker run -d -p 9001:9001 --name docker-spring docker-spring
```

🔧 Explicación de parámetros:

- `-d`: ejecuta el contenedor en segundo plano (modo detached)
- `-p 9001:9001`: mapea el puerto 9001 del host al puerto 9001 del contenedor
- `--name docker-spring`: nombre asignado al contenedor
- `docker-spring`: nombre de la imagen a utilizar

### ✓ Verificar el Contenedor

```
docker ps -a
```

### 📋 Recordemos el significado de las columnas

- **CONTAINER ID:** identificador único del contenedor
- **IMAGE:** imagen usada para crear el contenedor
- **COMMAND:** comando que se ejecutó al iniciar
- **CREATED:** fecha de creación
- **STATUS:** estado del contenedor (ejecutando, detenido, etc.)
- **PORTS:** puertos expuestos y mapeados
- **NAMES:** nombre asignado al contenedor

## 4. 📄 Consultar los Logs del Contenedor

Para monitorear el log de ejecución de la aplicación:

```
docker logs -f docker-spring
```

🔍 La opción `-f` permite seguir los logs en tiempo real.

## 5. 🌐 Probar la Aplicación

Una vez que el contenedor está en ejecución, podemos acceder a los endpoints de nuestra aplicación a través de un navegador web o herramienta como `curl`:

```
http://localhost:9001/docker-spring/messages/hello-world
```

## 📚 Para seguir aprendiendo

### • Docker Docs

Guía oficial y documentación completa sobre la instalación, comandos y arquitectura de Docker.

👉 <https://docs.docker.com/>

### • Docker Hub

Registro público de imágenes de contenedores. Permite publicar, compartir y descargar imágenes.

👉 <https://hub.docker.com/>

### • Play With Docker

Entorno de laboratorio online para experimentar con Docker sin necesidad de instalación local.

👉 <https://labs.play-with-docker.com/>