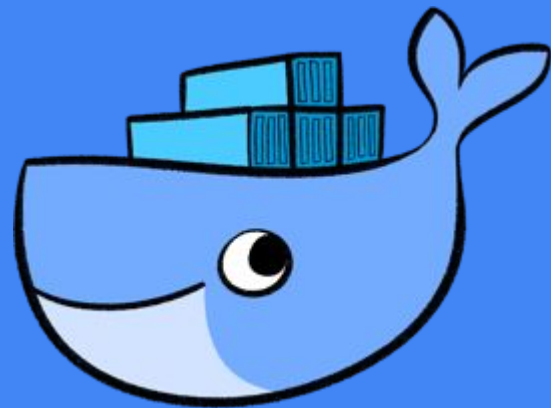


# Dockers 2024

Raúl Pucheta Barranco



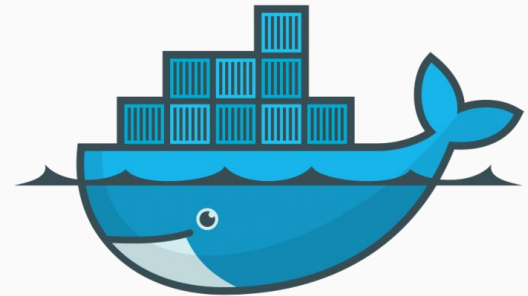
# Índice

- 1.- Introducción - ¿Qué es Docker?
- 2.- ¿Que es una imagen?
- 3.- ¿Que es un contenedor?
- 4.- Contenedores vs Máquinas Virtuales
- 5.- Instalación
- 6.- Esenciales Dockerfile.
- 7.- Arrancando un contenedor
- 8.- Docker Compose
- 9.- Docker Volumes
- 10.- Redes Docker
- 11.- Arquitectura Dockers
- 12.- Capas en Docker
- 13.- Referencias



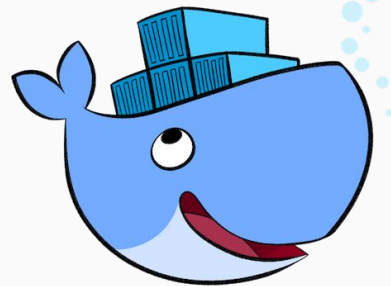
# 1.- Introducción ¿Qué es Docker?

Docker es una plataforma que facilita la creación, despliegue y ejecución de aplicaciones en contenedores. Estos contenedores empaquetan la aplicación y todo lo necesario para que funcione consistentemente en cualquier entorno. Docker ayuda a resolver el problema de "funciona en mi máquina", mejorando la eficiencia y facilitando la colaboración entre desarrolladores y operaciones al mantener los entornos aislados y consistentes.



## 2.- ¿Qué es una imagen?

Una imagen en Docker es una plantilla de solo lectura con instrucciones para crear un contenedor Docker. Contiene el código de la aplicación, las librerías, las herramientas necesarias, y otras dependencias requeridas para que la aplicación se ejecute. Las imágenes se utilizan para construir contenedores y asegurar la consistencia en los entornos de desarrollo, pruebas y producción.



### 3.- ¿Qué es un contenedor?

Un contenedor es una unidad estándar de software que empaqueta el código de una aplicación y todas sus dependencias para que la aplicación se ejecute de manera rápida y confiable en cualquier entorno de computación. Los contenedores ofrecen aislamiento de recursos, pero son más eficientes que las máquinas virtuales tradicionales porque comparten el núcleo del sistema operativo del anfitrión y utilizan menos recursos.



# 4.-Contenedores

Vs

- Eficiencia de Recursos: Utilizan menos recursos porque comparten el núcleo del sistema operativo del host y arrancan casi instantáneamente.
- Aislamiento de Procesos: Ofrecen aislamiento a nivel de proceso, ejecutándose en el mismo núcleo del sistema operativo pero manteniendo los procesos separados.
- Despliegue Rápido: Ideal para aplicaciones basadas en microservicios debido a su rápida capacidad de despliegue y escalabilidad.

# Máquinas Virtuales

- Aislamiento Completo: Proporcionan aislamiento completo a través de la virtualización del hardware, ejecutando un sistema operativo completo para cada instancia.
- Recursos Dedicados: Consumen más recursos ya que cada máquina virtual necesita su propio sistema operativo y un conjunto de recursos dedicados.
- Arranque más lento: Tienen tiempos de arranque más lentos comparados con los contenedores, lo que puede afectar la eficiencia en entornos de despliegue rápidos.

## 5.- Instalación

Linux 1/3



- 1) Actualizar el índice de paquetes del sistema:  
Ejecuta `sudo apt-get update` para asegurarse de que tu lista de paquetes esté actualizada.
- 2) Instalar paquetes para permitir el uso de un repositorio sobre HTTPS:  
`sudo apt-get install ca-certificates curl gnupg lsb-release`
- 3) Agregar la clave oficial de Docker GPG:  
`curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg`

## 5.- Instalación

Linux 2/3



4) Establecer el repositorio de Docker:

```
echo \  
"deb [arch=amd64  
signed-by=/usr/share/keyrings/docker-archive-  
keyring.gpg]  
https://download.docker.com/linux/ubuntu \  
$(lsb_release -cs) stable" | sudo tee  
/etc/apt/sources.list.d/docker.list > /dev/null
```

5) Instalar Docker Engine:

Actualiza el índice de paquetes de nuevo:

```
sudo apt-get update
```

Instala Docker Engine:

```
sudo apt-get install docker-ce docker-ce-cli  
containerd.io
```



## 5.- Instalación

Linux 3/3



6) Verificar la instalación: Ejecuta `sudo docker run hello-world` para comprobar que Docker se haya instalado correctamente y esté funcionando.

## 5.- Instalación

### Windows



- 1) Descarga Docker Desktop para Windows desde la página oficial: Docker Hub.
- 2) Ejecuta el instalador descargado y sigue las instrucciones. Asegúrate de tener habilitadas las características de virtualización en la BIOS de tu sistema.
- 3) Inicia Docker Desktop una vez completada la instalación.
- 4) Verifica la instalación: Abre una terminal (puedes usar PowerShell) y ejecuta `docker run hello-world` para comprobar que Docker funciona correctamente.

## 5- Instalación

macOs



1) Descarga Docker Desktop para Mac desde la página oficial: Docker Hub.

2) Abre el archivo .dmg descargado y arrastra el icono de Docker a la carpeta de Aplicaciones.

3) Inicia Docker Desktop desde la carpeta de Aplicaciones.

4) Dale los permisos necesarios a Docker si tu sistema operativo lo solicita durante el primer inicio.

5) Verifica la instalación: Abre una terminal y ejecuta

`docker run hello-world`

para asegurarte de que Docker está funcionando correctamente.

## 6.- Esenciales Dockerfile

### ¿Qué es Dockerfile?

Dockerfile, es básicamente un archivo de texto que contiene todas las instrucciones necesarias para construir una imagen de Docker.

### Tipos instrucciones básicas Dockerfile:

**FROM:** Especifica la imagen base desde la cual se construye la nueva imagen. Por ejemplo, FROM ubuntu:18.04 indica que la imagen se basará en Ubuntu 18.04.

**RUN:** Ejecuta comandos en la capa superior de la imagen y confirma los resultados. Se usa comúnmente para instalar software necesario en la imagen. Por ejemplo, RUN apt-get update && apt-get install -y nginx.

**CMD:** Proporciona los comandos por defecto para un contenedor en ejecución. Estos comandos se pueden sobrescribir desde la línea de comandos cuando el contenedor se inicia. Por ejemplo, CMD ["echo", "Hello World"].

## 6.- Esenciales Dockerfile

**EXPOSE:** Informa a Docker que el contenedor escuchará en los puertos especificados en tiempo de ejecución. No publica el puerto. Se usa junto con RUN para mapear el puerto. Por ejemplo, EXPOSE 80.

**ENV:** Establece variables de entorno. Por ejemplo, ENV NGINX\_VERSION 1.14.

**ADD:** Copia nuevos archivos, directorios o URL desde <src> y los añade al sistema de archivos del contenedor en la ruta <dest>. Por ejemplo, ADD https://example.com/big.tar.xz /usr/src/things/.

**COPY:** Copia nuevos archivos o directorios desde <src> y los añade al sistema de archivos del contenedor en la ruta <dest>. A diferencia de ADD, COPY es más transparente y se prefiere para copiar archivos locales. Por ejemplo, COPY ./app /app.

**ENTRYPOINT:** Configura un contenedor para que se ejecute como un ejecutable. Cualquier comando CMD se convierte en argumentos para ENTRYPOINT. Por ejemplo, ENTRYPOINT ["python", "/code/app.py"].

**WORKDIR:** Establece el directorio de trabajo para las instrucciones RUN, CMD, ENTRYPOINT, COPY y ADD que siguen en el Dockerfile. Por ejemplo, WORKDIR /app.

## 6.- Esenciales Dockerfile

**USER:** Establece el UID (ID de usuario) o GID (ID de grupo) que se usará para ejecutar el contenedor y para cualquier instrucción RUN, CMD, y ENTRYPOINT que le siga en el Dockerfile. Por ejemplo, USER 1001.

**VOLUME:** Crea un punto de montaje con el nombre especificado y marca la ruta como que contiene volúmenes externos desde el host o de otros contenedores. Por ejemplo, VOLUME /data.



## 6.- Esenciales Dockerfile

```
# Usar una imagen base de Python 3.8
FROM python:3.8-slim

# Establecer el directorio de trabajo en el contenedor
WORKDIR /app


# Copiar los archivos de requisitos primero, para aprovechar la caché de capas
COPY requirements.txt .

# Instalar las dependencias
RUN pip install --no-cache-dir -r requirements.txt

# Copiar el resto del código de la aplicación
COPY . .

# Exponer el puerto en el que la app se ejecuta
EXPOSE 5000

# Comando para ejecutar la aplicación
CMD ["python", "./app.py"]
```



## 6.- Esenciales Dockerfile

### Ejecutando Dockerfile

Una vez creado Dockerfile, por medio de terminal nos desplazamos hasta la carpeta donde se encuentra y de esta forma procederemos a construir la imagen definida dentro de dicho fichero:

`docker build -t nombre-de-tu-imagen .`

Si el Dockerfile se encuentra en otro lugar que donde te encuentras en el terminal debes especificar la ruta en lugar del .

En caso que el nombre del dockerfile no sea el genérico “Dockerfile” debemos ejecutarlo con el siguiente comando:

`docker build -t nombre-de-tu-imagen -f ruta/al/Dockerfile.personalizado .`

### Comprobar la creación de la imagen

Para verificar si la imagen se ha creado con éxito podemos hacerlo mediante el siguiente comando:

`docker images`

Veremos la imagen en la lista creada.

En este punto solo nos queda arrancar nuestra imagen en un contenedor.



## 6.- Esenciales Dockerfile

### Ejercicio Dockerfile

Vamos con un ejercicio práctico para crear un Dockerfile. El objetivo será construir una imagen de Docker que ejecute una pequeña aplicación web escrita en Python que muestra "¡Hola, mundo Dockerizado!" en el navegador.

#### Paso 1: Preparar el entorno de desarrollo

Crea un nuevo directorio para tu proyecto y navega a él en tu terminal. Dentro de este directorio, crearás dos archivos: app.py y Dockerfile.

#### Paso 2: Crear la aplicación web

app.py: Este archivo contendrá el código de tu aplicación web. Utilizaremos Flask, un micro framework de Python, para servir una página web simple. El contenido de app.py será:

```
from flask import Flask  
app = Flask(__name__)
```

```
@app.route('/')  
def hello_world():  
    return '¡Hola, mundo Dockerizado!'
```

```
if __name__ == '__main__':  
    app.run(host='0.0.0.0', port=80)
```



## 6.- Esenciales Dockerfile

requirements.txt: Este archivo contendrá las dependencias necesarias para tu aplicación. En este caso, solo necesitamos Flask. Crea el archivo requirements.txt con el siguiente contenido:

```
Flask==2.0.1
```

### Paso 3: Crear el Dockerfile

Ahora, crea el Dockerfile en el mismo directorio que app.py. El Dockerfile contendrá las instrucciones para construir la imagen de tu aplicación. Aquí tienes un ejemplo básico:

```
# Establecer la imagen base  
FROM python:3.9-slim
```

```
# Establecer el directorio de trabajo en el contenedor  
WORKDIR /app
```

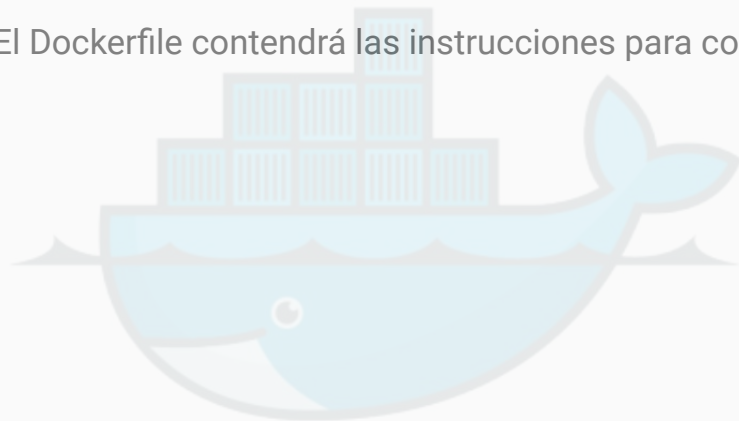
```
# Copiar los archivos de requisitos primero, para aprovechar la caché de Docker  
COPY requirements.txt .
```

```
# Instalar las dependencias  
RUN pip install --no-cache-dir -r requirements.txt
```

```
# Copiar el resto del código de la aplicación  
COPY . .
```

```
# Hacer que el puerto 80 esté disponible para el mundo exterior  
EXPOSE 80
```

```
# Ejecutar la aplicación cuando se inicie el contenedor  
CMD ["python", "app.py"]
```



## 6.- Esenciales Dockerfile

### **Paso 4:** Construir y ejecutar tu contenedor

Construir la imagen:

Abre una terminal y navega al directorio donde se encuentra tu Dockerfile.

Ejecuta el siguiente comando para construir tu imagen Docker:

```
docker build -t mi-app-flask .
```

Ejecutar la imagen en un contenedor:

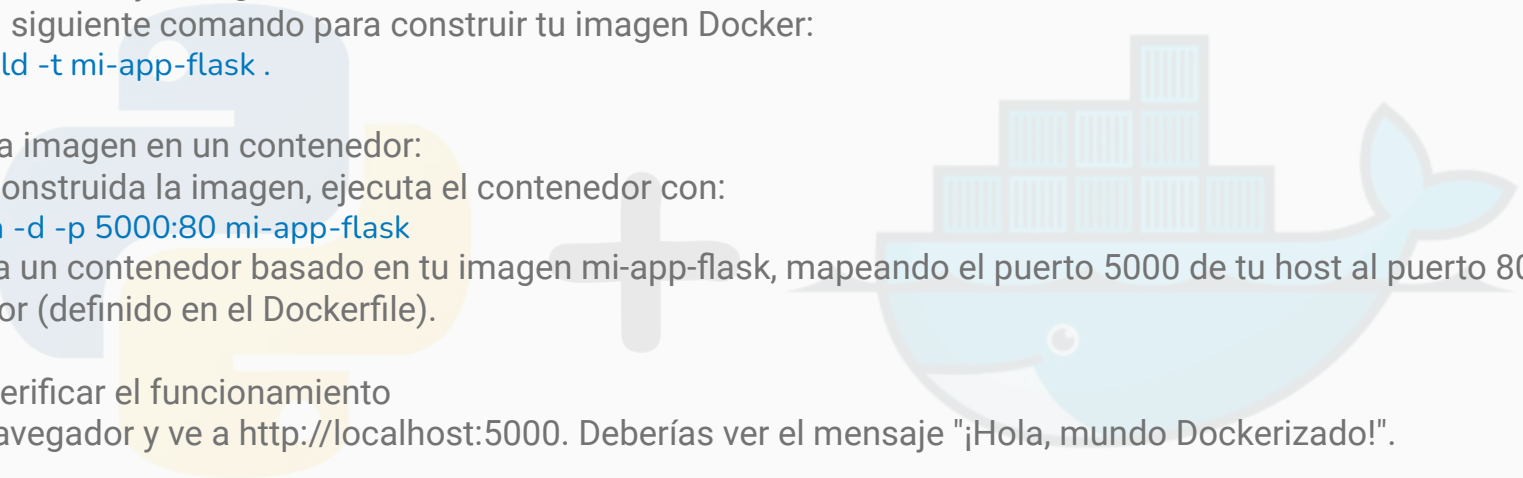
Una vez construida la imagen, ejecuta el contenedor con:

```
docker run -d -p 5000:80 mi-app-flask
```

Esto inicia un contenedor basado en tu imagen mi-app-flask, mapeando el puerto 5000 de tu host al puerto 80 del contenedor (definido en el Dockerfile).

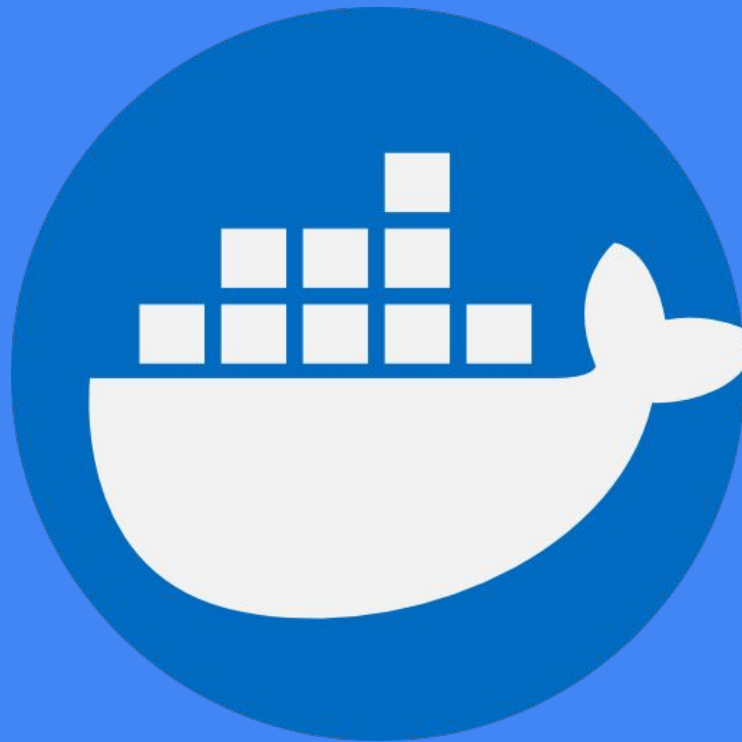
### **Paso 5:** Verificar el funcionamiento

Abre tu navegador y ve a <http://localhost:5000>. Deberías ver el mensaje "¡Hola, mundo Dockerizado!".



# 7.- Arrancando un contenedor

Arrancar un contenedor en Docker es un proceso sencillo y directo, pero fundamental para trabajar con contenedores. Aquí te dejo una explicación resumida junto con comandos básicos y un ejercicio práctico para fortalecer tu comprensión.



## Conceptos Básicos

**Imagen:** Es una plantilla de solo lectura que contiene el sistema operativo, el software y los archivos necesarios para ejecutar una aplicación. Piensa en ella como la receta para crear un contenedor.

**Contenedor:** Es una instancia ejecutable de una imagen. Puedes tener muchos contenedores corriendo basados en la misma imagen.

## Comandos Básicos:

- Listar imágenes disponibles: Para ver qué imágenes tienes disponibles localmente, usa:

`docker images`

- Ejecutar un contenedor: Para iniciar un contenedor a partir de una imagen, utiliza:

`docker run [opciones] imagen [comando] [argumentos]`

Algunas opciones útiles son:

-d: Ejecuta el contenedor en segundo plano (modo "detached").

--name: Asigna un nombre al contenedor.

-p: Mapea un puerto del contenedor a un puerto del host (ej., -p 8080:80).

· Listar contenedores en ejecución: Para ver los contenedores activos, ejecuta:

`docker ps`

Para ver todos los contenedores, incluso los detenidos, agrega la opción -a.

· Detener un contenedor: Para detener un contenedor que está corriendo, utiliza:

`docker stop nombre_contenedor`

· Iniciar un contenedor detenido: Si quieres volver a iniciar un contenedor que has detenido, usa:

`docker start nombre_contenedor`

## Ejercicio Práctico

Objetivo: Ejecutar un contenedor de Nginx, hacer que sirva una página web en el puerto 8080 de tu máquina local, y luego detener el contenedor.

### 1.- Buscar la imagen de Nginx:

No necesitas hacer esto en tu terminal. Solo asegúrate de que existe una imagen oficial de Nginx en Docker Hub.

### 2.- Ejecutar un contenedor Nginx:

Ejecuta el siguiente comando para iniciar un contenedor de Nginx en segundo plano y mapear el puerto 8080 del host al puerto 80 del contenedor (el puerto por defecto de Nginx).

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

### 3.- Verificar la ejecución:

Abre tu navegador y ve a <http://localhost:8080>. Deberías ver la página por defecto de Nginx.

#### 4.- Detener el contenedor:

Una vez que hayas terminado, puedes detener el contenedor con:

`docker stop mi-nginx`

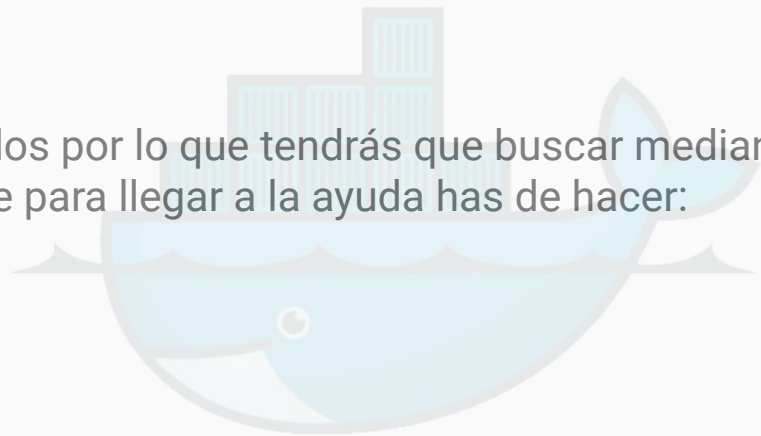
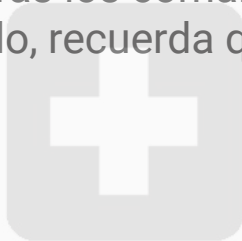
Para los siguientes 3 puntos no tendrás los comandos por lo que tendrás que buscar mediante la ayuda la forma de hacer lo que te pido, recuerda que para llegar a la ayuda has de hacer:

`docker -help`

5.- Reinicia el contenedor.

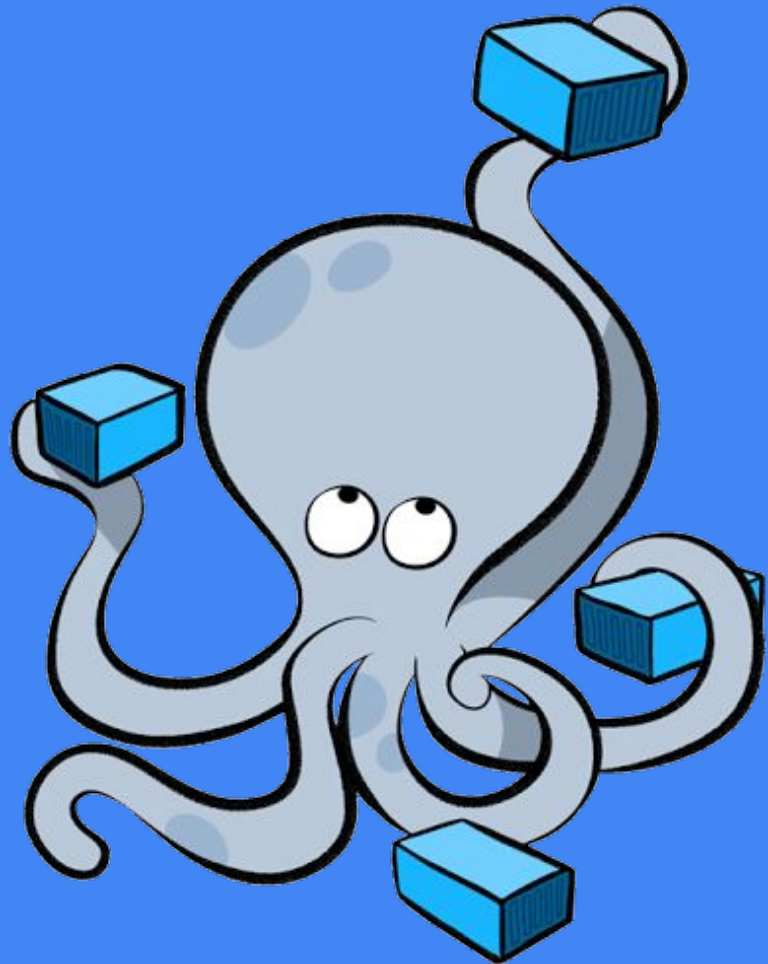
6.- Borra el contenedor.

7.- Borra la imagen.

The NGINX logo, featuring a green hexagon with a white 'N' inside, and the word 'NGINX' in a green, stylized font below it.The word 'docker' in a light gray, lowercase, sans-serif font.

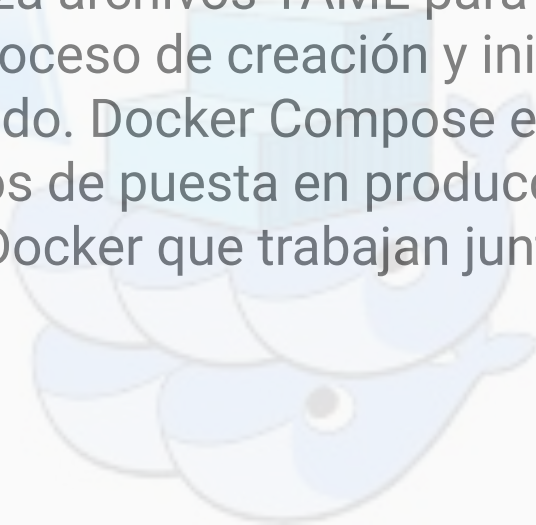


# 8.-Docker Compose



### ¿Que es Compose?

Docker Compose es una herramienta que permite definir y ejecutar aplicaciones Docker multi-contenedor. Utiliza archivos YAML para configurar los servicios de la aplicación y ejecuta el proceso de creación y inicio de todos los contenedores con un solo comando. Docker Compose es ideal para el desarrollo, las pruebas y los entornos de puesta en producción, ya que simplifica el manejo de contenedores Docker que trabajan juntos.



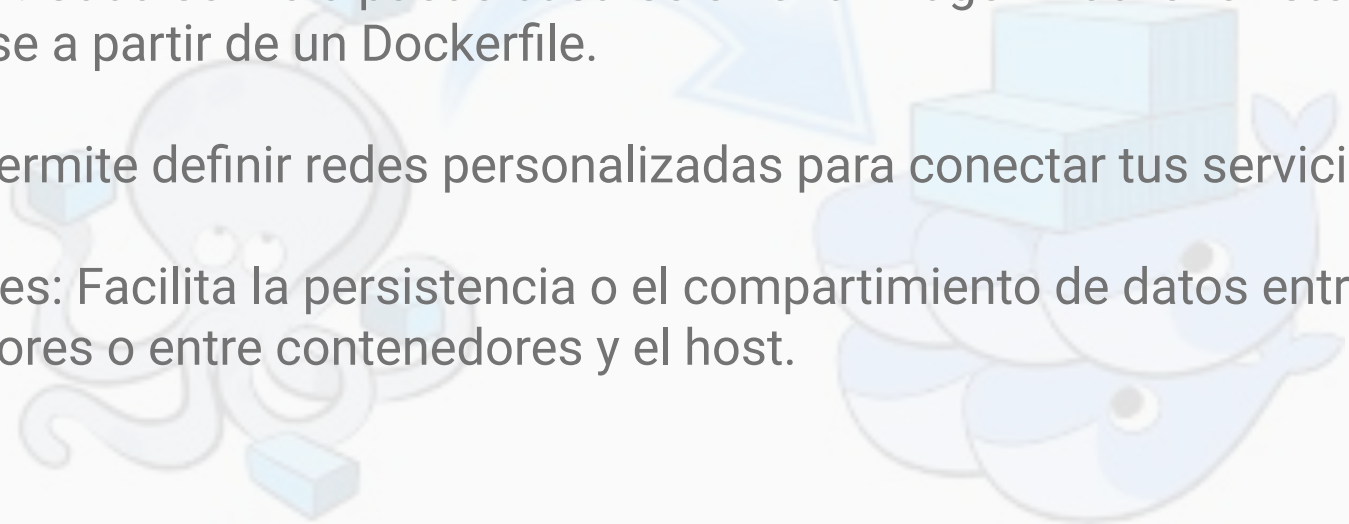
## 8.- Docker Compose

### Características clave:

- Definición de servicios en un archivo: Permite describir en un archivo YAML los servicios, redes y volúmenes necesarios para tu aplicación, facilitando su gestión como un conjunto.
- Aislamiento: Aunque todos los servicios definidos en `docker-compose.yml` se ejecutan en una única red, cada uno opera en un entorno aislado.
- Desarrollo y despliegue simplificados: Con un solo comando, puedes construir o reconstruir servicios, iniciar y detener servicios, ver el estado de ejecución de los servicios y mucho más.
- Compatibilidad y portabilidad: Las definiciones de Compose pueden trasladarse entre entornos de desarrollo, pruebas y producción, asegurando la consistencia en todos ellos.

### Componentes básicos de un archivo Docker Compose:

- Servicios: Define los contenedores que se deben ejecutar como parte de tu aplicación. Cada servicio puede basarse en una imagen Docker existente o construirse a partir de un Dockerfile.
- Redes: Permite definir redes personalizadas para conectar tus servicios entre sí.
- Volúmenes: Facilita la persistencia o el compartimiento de datos entre contenedores o entre contenedores y el host.



## 8.- Docker Compose

### Ejemplo básico de un docker-compose.yml

Supongamos que quieres ejecutar una aplicación web sencilla que utiliza Flask para el backend y Redis como base de datos en memoria. Tu archivo docker-compose.yml podría verse así:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ../code
    depends_on:
      - redis
  redis:
    image: "redis:alpine"
```

## 8.- Docker Compose

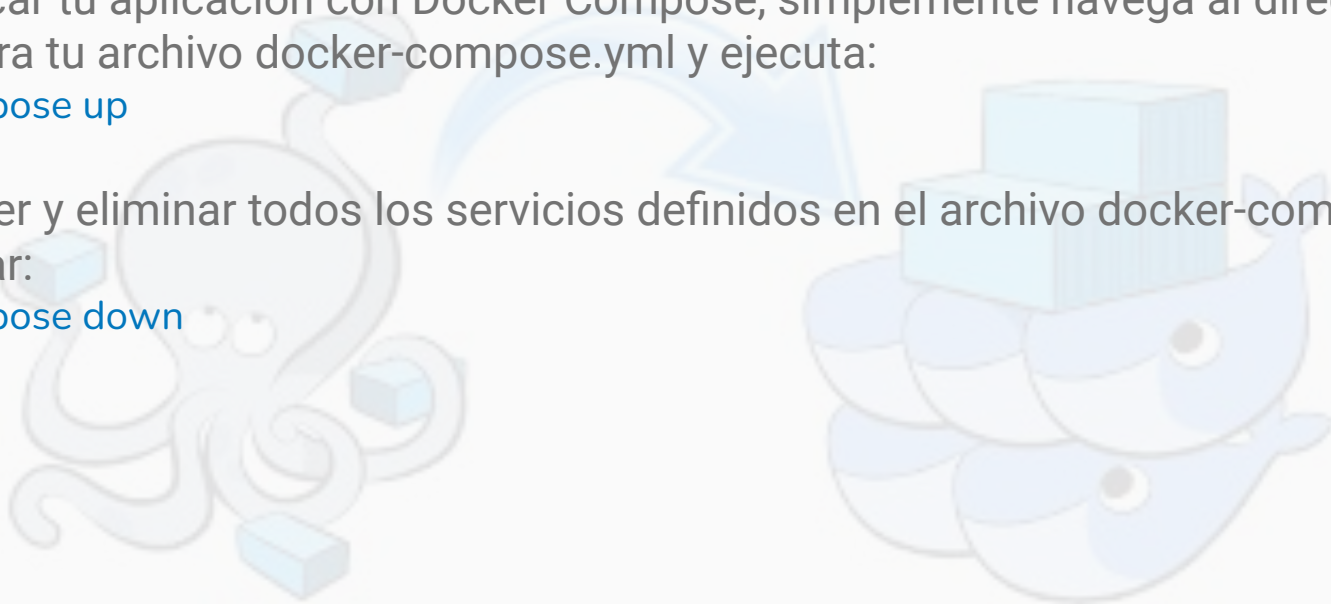
### Cómo ejecutar Docker Compose:

Para arrancar tu aplicación con Docker Compose, simplemente navega al directorio donde se encuentra tu archivo `docker-compose.yml` y ejecuta:

`docker-compose up`

Para detener y eliminar todos los servicios definidos en el archivo `docker-compose.yml`, puedes usar:

`docker-compose down`



## 8.- Docker Compose

### Ejercicio Básico:

Para introducirte en Docker Compose, vamos a realizar un ejercicio práctico simple. Crearemos una aplicación con dos servicios: una aplicación web Python Flask sencilla y una base de datos Redis. Utilizaremos Docker Compose para definir y ejecutar estos servicios.

#### Paso 1: Preparar la aplicación Flask

Primero, necesitamos una aplicación Flask simple. Crea un directorio para tu proyecto y añade los siguientes archivos:

app.py (una aplicación Flask que se conecta a Redis y aumenta un contador cada vez que se accede a la página):

## 8.- Docker Compose

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Conectar a Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route('/')
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>no se puede conectar a Redis, contador deshabilitado</i>"

    html = "<h3>Hola {name}</h3>" \
        "<b>Visitas:</b> {visits}" \
        "<br/>"
    return html.format(name=os.getenv("NAME", "mundo"), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```





## 8.- Docker Compose

requirements.txt (las dependencias de tu aplicación):

flask

redis

### Paso 2: Crear un Dockerfile

En el mismo directorio, crea un Dockerfile para tu aplicación Flask:

# Usa una imagen base oficial de Python

FROM python:3.7-alpine

# Establece el directorio de trabajo en /code

WORKDIR /code

# Instala las dependencias de Python

COPY requirements.txt .

RUN pip install -r requirements.txt

# Copia el resto del directorio actual en /code

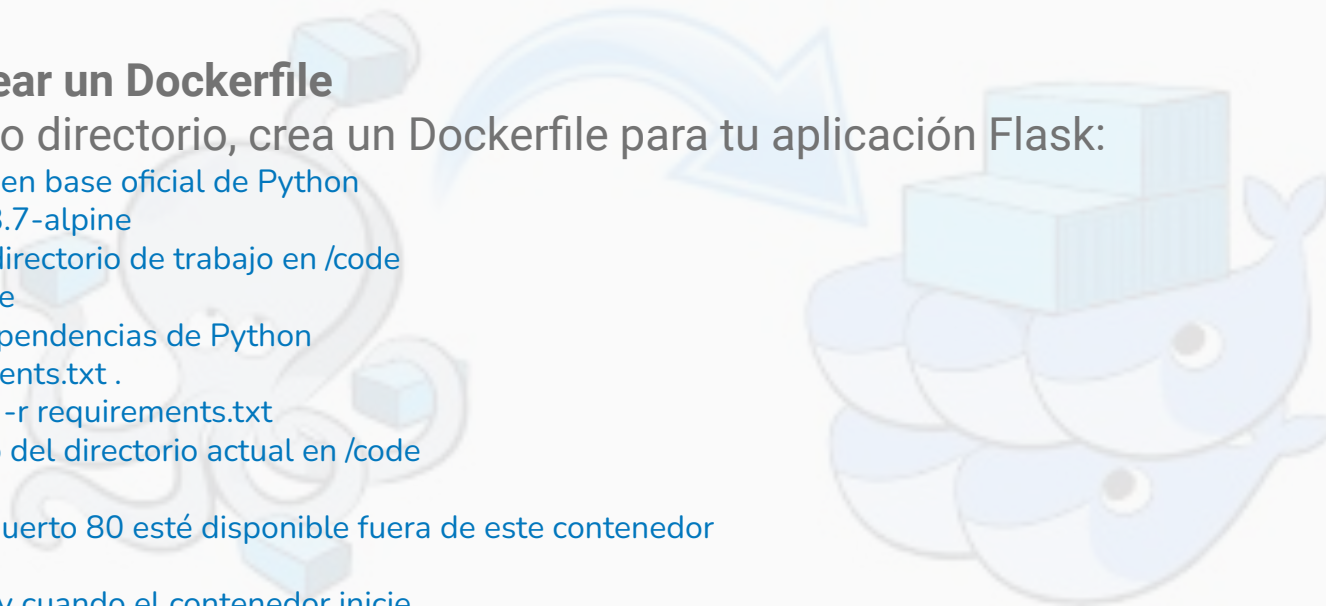
COPY . .

# Hace que el puerto 80 esté disponible fuera de este contenedor

EXPOSE 80

# Ejecuta app.py cuando el contenedor inicie

CMD ["python", "app.py"]



## 8.- Docker Compose

### Paso 3: Definir servicios con Docker Compose

Crea un archivo docker-compose.yml en el mismo directorio, que defina los servicios web y redis

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:80"
    environment:
      NAME: Docker Compose
  redis:
    image: "redis:alpine"
```

Este archivo indica a Docker Compose cómo construir la imagen de tu aplicación Flask y cómo conectarla a un servicio Redis.



## 8.- Docker Compose

### Paso 4: Construir y ejecutar con Docker Compose

Desde el terminal, en el directorio de tu proyecto, ejecuta:

`docker-compose up`

Este comando construirá la imagen para el servicio web, descarga la imagen Redis (si es necesario) y arrancará los contenedores.

### Paso 5: Verificar la aplicación

Abre tu navegador y visita `http://localhost:5000`. Deberías ver un mensaje de bienvenida y un contador de visitas que incrementa con cada recarga de la página.

### Paso 6: Limpiar

Cuando hayas terminado, puedes detener y eliminar los contenedores creados por Docker Compose con:

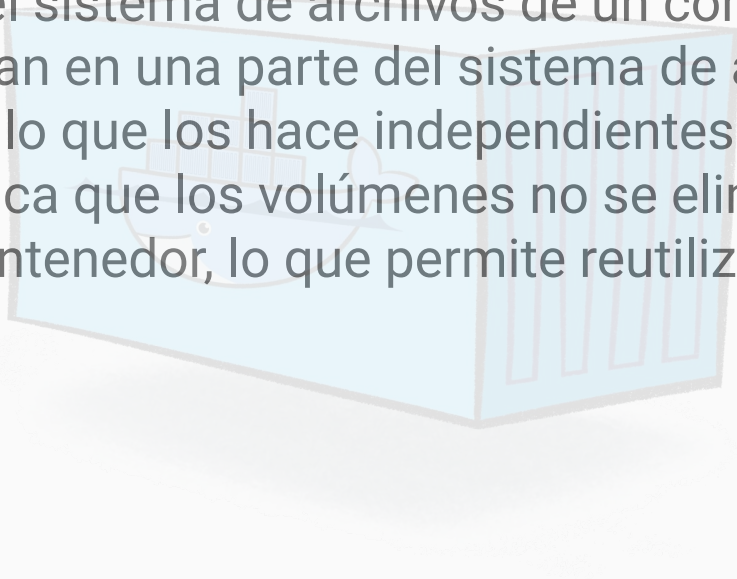
`docker-compose down`

# 9.- Docker Volumes



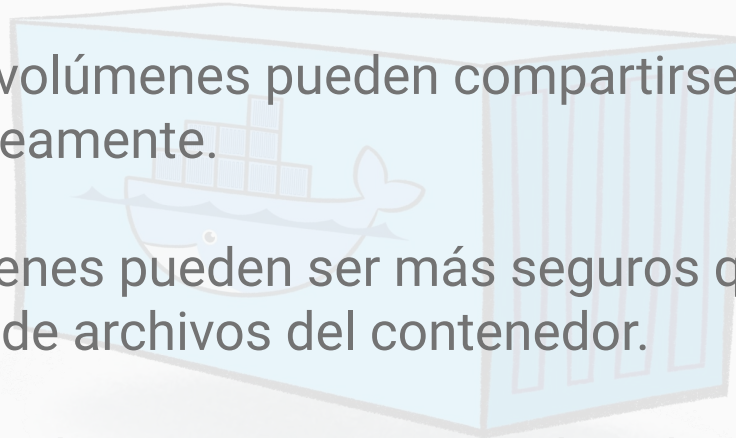
## ¿Que son los Volumes?

Los volúmenes en Docker son una forma de persistir y compartir datos entre contenedores y entre un contenedor y el host. A diferencia de los datos almacenados dentro del sistema de archivos de un contenedor, los volúmenes se almacenan en una parte del sistema de archivos del host gestionada por Docker, lo que los hace independientes del ciclo de vida del contenedor. Esto significa que los volúmenes no se eliminan cuando se elimina o detiene un contenedor, lo que permite reutilizar y compartir datos.



## Características clave de los volúmenes:

- Persistencia de datos: Los datos almacenados en volúmenes persisten incluso después de que los contenedores se eliminen.
- Compartir datos: Los volúmenes pueden compartirse entre múltiples contenedores simultáneamente.
- Seguridad: Los volúmenes pueden ser más seguros que el almacenamiento de datos en el sistema de archivos del contenedor.
- Gestión por Docker: Docker gestiona el ciclo de vida de los volúmenes, facilitando su inspección, eliminación o limpieza.



# 1.- Ejercicio básico: Crear y usar un volumen con Docker

Objetivo: Crear un volumen para persistir los datos generados por un contenedor de Redis.

## Paso 1: Crear un volumen

Primero, crea un volumen en Docker utilizando el comando `docker volume create`:

```
docker volume create mi-volumen-redis
```

## Paso 2: Ejecutar un contenedor de Redis usando el volumen

Ahora, inicia un contenedor de Redis y monta el volumen creado en el paso anterior en el contenedor. Esto asegurará que los datos de Redis se almacenen en el volumen y persistan más allá del ciclo de vida del contenedor.



```
docker run -d --name mi-redis -v mi-volumen-redis:/data redis redis-server --appendonly yes
```

En este comando, `-v mi-volumen-redis:/data` monta el volumen `mi-volumen-redis` en la ruta `/data` dentro del contenedor. La opción `--appendonly yes` le indica a Redis que use el modo "Append Only File" (AOF), que persiste los datos escribiéndolos en un archivo de log.

### Paso 3: Verificar la persistencia de datos

Conectar al contenedor de Redis:

```
docker exec -it mi-redis redis-cli
```

Crear algunos datos en Redis:

```
set mi_clave "Hola Docker Volumes"
```





Salir de Redis y detener el contenedor:

`exit`

`docker stop mi-redis`

Reiniciar el contenedor de Redis:

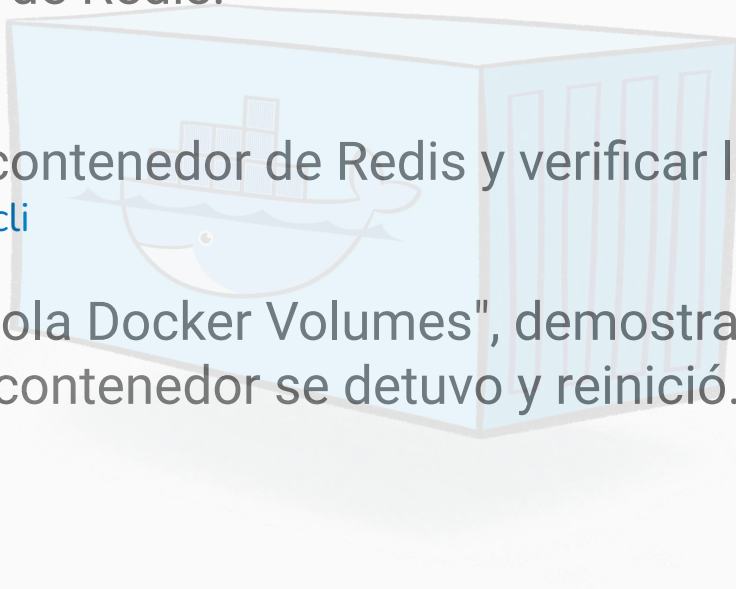
`docker start mi-redis`

Volver a conectarte al contenedor de Redis y verificar los datos:

`docker exec -it mi-redis redis-cli`

`get mi_clave`

Deberías ver el valor "Hola Docker Volumes", demostrando que los datos persistieron aunque el contenedor se detuvo y reinició.



## Paso 4: Limpiar

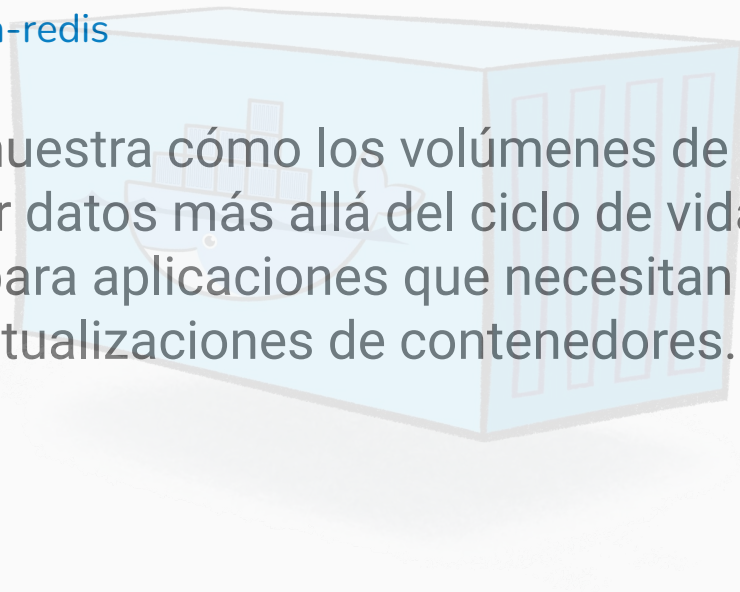
Cuando hayas terminado, puedes detener y eliminar el contenedor, y luego eliminar el volumen si lo deseas:

```
docker stop mi-redis
```

```
docker rm mi-redis
```

```
docker volume rm mi-volumen-redis
```

Este ejercicio básico muestra cómo los volúmenes de Docker pueden ser utilizados para persistir datos más allá del ciclo de vida de un contenedor, una práctica esencial para aplicaciones que necesitan mantener datos a través de reinicios o actualizaciones de contenedores.



## 2.- Ejercicio básico: Demostración Volúmenes.

Objetivo: Entender cómo los volúmenes pueden ser utilizados para compartir datos entre contenedores.

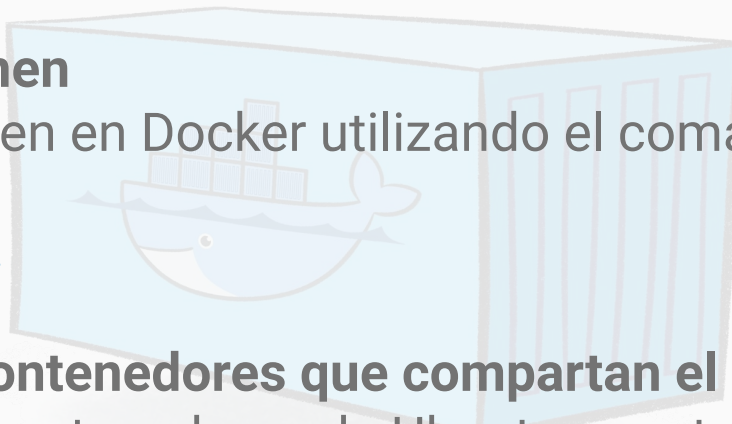
### Paso 1: Crear un volumen

Primero, crea un volumen en Docker utilizando el comando `docker volume create`:

```
docker volume create data_vol
```

### Paso 2: Ejecutar dos contenedores que compartan el mismo volumen.

Vamos a ejecutar dos contenedores de Ubuntu, montando el volumen data\_vol en ambos. En el primer contenedor, crearemos un archivo en el volumen compartido, y luego veremos si podemos acceder a ese mismo archivo desde el segundo contenedor.



## Crea el primer contenedor:

```
docker run -it --name ubuntu1 --mount source=data_vol,target=/app ubuntu bash -c "echo 'Hola desde ubuntu1' > /app/saludo.txt; bash"
```

## Crea el segundo contenedor:

```
docker run -it --name ubuntu2 --mount source=data_vol,target=/app ubuntu bash -c "cat /app/saludo.txt; bash"
```

## Paso 3: Ejecuta dos contenedores sin compartir el volumen:

### Crea el primer contenedor:

```
docker run -it --name ubuntu3 ubuntu bash -c "echo 'Hola desde ubuntu3' > /saludo.txt; bash"
```

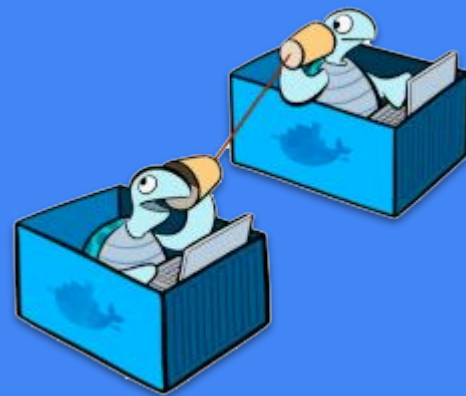
### Crea el segundo contenedor:

```
docker run -it --name ubuntu4 ubuntu bash -c "cat /saludo.txt || echo 'Archivo no encontrado'; bash"
```



# 10.- Redes Docker

La red de Docker se refiere al conjunto de funcionalidades y tecnologías que Docker utiliza para proporcionar conectividad de red a los contenedores. Utiliza su propio stack de red que permite a los usuarios configurar cómo los contenedores se comunican entre sí y con otras redes. Docker encapsula diferentes modelos de red en "drivers" de red, cada uno de los cuales ofrece diferentes capacidades y se adapta a diferentes escenarios de uso.



### 1.- Principales Drivers y componentes de las redes docker:

Docker utiliza redes virtuales para permitir la comunicación entre contenedores y también entre contenedores y el host externo. Estos son los tipos principales de redes en Docker:

#### Bridge

- **Predeterminada:** Cuando creas un contenedor sin especificar una red, Docker lo asigna a una red bridge predeterminada.
- **Aislamiento:** Ideal para ejecutar contenedores que no necesitan ser accesibles desde la red externa.
- **Comunicación:** Permite la comunicación entre contenedores en la misma red bridge.

## 10.- Redes Docker

### Overlay:

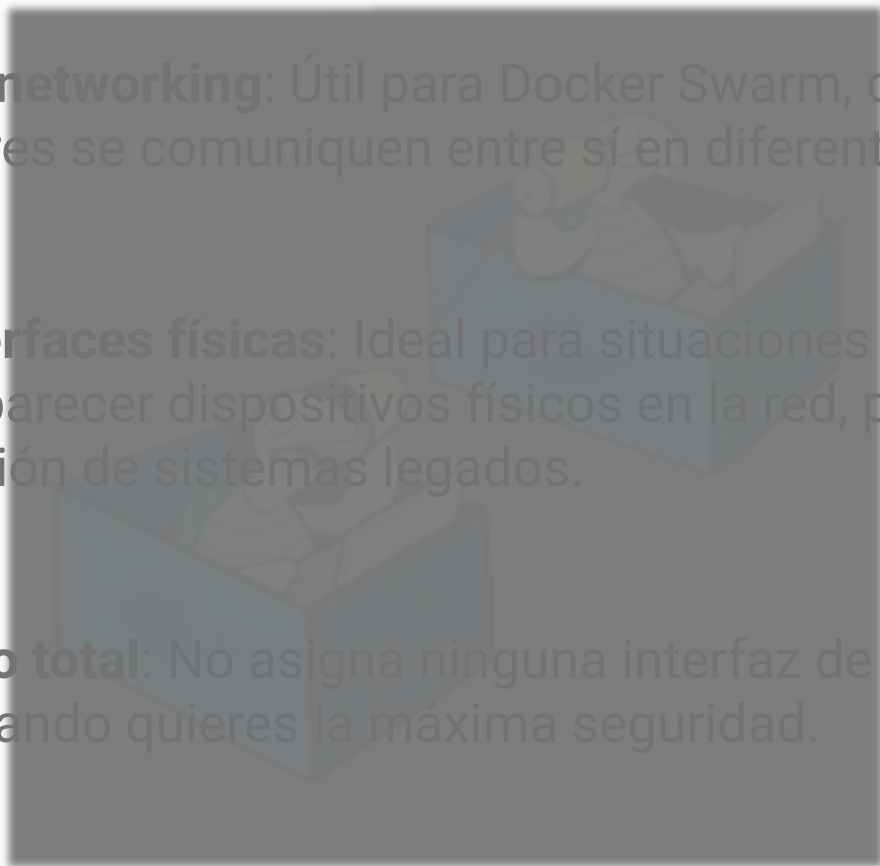
- **Multi-host networking:** Útil para Docker Swarm, donde necesitas que los contenedores se comuniquen entre sí en diferentes nodos.

### Macvlan:

- **Simula interfaces físicas:** Ideal para situaciones donde tus contenedores necesitan parecer dispositivos físicos en la red, posiblemente para fines de integración de sistemas legados.

### None:

- **Aislamiento total:** No asigna ninguna interfaz de red a los contenedores, útil para cuando quieres la máxima seguridad.



### 2.- Funcionalidades Avanzadas:

- **Redes personalizadas:** Los usuarios pueden crear sus propias redes, especificar cuál driver de red usar, y configurar opciones avanzadas como políticas de IPAM (IP Address Management).
- **Enlace de puertos:** Docker permite mapear puertos de contenedores a puertos del host, facilitando la comunicación con el mundo exterior.
- **Conectividad segura:** Algunos drivers de red, especialmente overlay, soportan la encriptación de tráfico, proporcionando una conectividad segura entre contenedores distribuidos.



## 10.- Redes Docker

### 3.- Ejemplos uso:

#### Driver Bridge:

**Uso Común:** Aislamiento de red entre contenedores que sólo necesitan comunicarse entre sí o con el host.

#### Ejemplo Práctico:

Supongamos que quieres conectar dos contenedores Apache y un contenedor MySQL en la misma red:

```
bash
```

```
# Crea una red bridge personalizada
```

```
docker network create --driver bridge my_bridge
```

```
# Ejecuta un contenedor MySQL en la red my_bridge
```

```
docker run -d --name mysql-server --network my_bridge mysql
```

```
# Ejecuta dos contenedores Apache en la misma red
```

```
docker run -d --name apache1 --network my_bridge httpd
```

```
docker run -d --name apache2 --network my_bridge httpd
```

## 10.- Redes Docker


### **Driver Host:**

**Uso Común:** Maximizar el rendimiento de la red eliminando la sobrecarga de la red virtual.

### **Ejemplo Práctico:**

Utilizar el driver host para ejecutar un servidor Nginx que escuche directamente en el puerto 80 del host:

bash

 Copy code

```
# Ejecuta un contenedor Nginx usando el espacio de nombres de red del host  
docker run -d --name nginx --network host nginx
```

## 10.- Redes Docker

### Driver Overlay:

**Uso Común:** Conectar contenedores a través de múltiples hosts, especialmente en entornos orquestados como Docker Swarm o Kubernetes.

### Ejemplo Práctico:

Crear una red overlay para conectar contenedores en diferentes hosts en un clúster de Docker Swarm:

```
bash Copy code  
  
# Crea una red overlay en un ambiente de Docker Swarm  
docker network create --driver overlay --attachable my_overlay  
  
# Ejecuta contenedores en diferentes nodos pero en la misma red overlay  
docker run -d --name service1 --network my_overlay alpine sleep 3600  
docker run -d --name service2 --network my_overlay alpine sleep 3600
```

Contenedores service1 y service2 podrán comunicarse entre sí, incluso estando en diferentes nodos físicos.

## 10.- Redes Docker

### Driver Macvlan:

**Uso Común:** Integración con redes físicas existentes donde los contenedores necesitan aparecer como dispositivos físicos.

### Ejemplo Práctico:

Configurar un contenedor que obtenga una dirección IP de la red física del host:

```
bash Copy code

# Crea una red macvlan
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 my_macvlan

# Ejecuta un contenedor en esta red
docker run -d --name my-container --network my_macvlan alpine
```

El contenedor my-container aparecerá en la red como un dispositivo más, con una dirección IP propia dentro del rango 192.168.1.0/24

## 10.- Redes Docker

### Driver None:

**Uso Común:** Total aislamiento de red para contenedores que no deben comunicarse ni ser accesibles.

### Ejemplo Práctico:

Ejecutar un contenedor sin ninguna conectividad de red:

```
bash
```

[Copy code](#)

```
# Ejecuta un contenedor con red deshabilitada
```

```
docker run -d --name isolated-container --network none alpine
```

Este contenedor estará completamente aislado desde un punto de vista de red, ideal para pruebas de seguridad o comportamiento de aplicaciones sin ninguna influencia externa.

## 10.- Redes Docker

### EJERCICIOS:

#### Ejercicio 1: Explorando el Driver Bridge

**Objetivo:** Comprender cómo funciona la red Bridge y la comunicación entre contenedores.

**1.- Crea una red Bridge:** Crear una red Bridge personalizada.

```
docker network create --driver bridge my-bridge-net
```

**2.- Lanza contenedores:** Iniciar dos contenedores de Ubuntu que pertenezcan a esta red y que uno pueda hacer ping al otro.

```
docker run -dit --name ubuntu1 --network my-bridge-net ubuntu
```

```
docker run -dit --name ubuntu2 --network my-bridge-net ubuntu
```

**3.- Interacción:** Acceder al primer contenedor (ubuntu1) y usar ping para comprobar la conectividad con ubuntu2 usando su nombre de contenedor.

```
docker exec -it ubuntu1 bash
```

```
ping ubuntu2
```

## 10.- Redes Docker

### Ejercicio 2: Utilizando el Driver Host

**Objetivo:** Demostrar cómo los contenedores pueden compartir la red del host.

**1.- Ejecutar un servidor HTTP Básico** en el contenedor que use la red del host:

```
docker run -d --name mynginx --network host nginx
```

**2.- Verificación:** Acceder desde el navegador al localhost o usar “curl localhost” desde el host para ver la página de bienvenida de nginx.



## 10.- Redes Docker

### Ejercicio 3: Configurando Redes Overlay

**Objetivo:** Entender cómo los contenedores pueden comunicarse a través de múltiples hosts.

**1.- Crear una red Overlay** requiere un entorno de Docker Swarm:

```
docker network create --driver overlay --attachable my-overlay-net
```

**2.- Desplegar contenedores** en diferentes nodos o simularlo en un solo host:

```
docker run -dit --name service1 --network my-overlay-net alpine sleep infinity
```

```
docker run -dit --name service2 --network my-overlay-net alpine sleep infinity
```

**3.- Comunicación:** Ingresar a “service1” y hacer ping a “service2” para validar.

```
docker exec -it service1 ping service2
```

¿Que casos de uso encontráis?



## 10.- Redes Docker

### Ejercicio 4: Aislamiento con Driver None

**Objetivo:** Observar el efecto del completo aislamiento de red en un contenedor.

#### 1.- Ejecutar un contenedor sin red:

```
docker run -dit --name no-net-container --network none alpine sleep infinity
```

**2.- Verificar aislamiento:** Intentar hacer ping a cualquier dirección externa o interna desde dentro del contenedor fallará.

```
docker exec -it no-net-container ping google.com
```

¿Que casos de uso encontráis?

# 11.- Arquitecturas Docker



## **Definición de Arquitecturas en Docker**

Cuando hablamos de "arquitecturas" en el contexto de Docker, nos referimos a dos aspectos principales:

### **1.- Arquitecturas de Procesador (Hardware):**

Docker puede ejecutarse en diversas arquitecturas de hardware como x86-64 (la más común en PCs y servidores), arm64 (utilizada en dispositivos más nuevos de Raspberry Pi, smartphones y como servidores alternativos), y s390x (usada en mainframes)

### **2.- Arquitecturas de Software (Orquestación de Contenedores):**

Además de las arquitecturas de hardware, Docker también es flexible en términos de las arquitecturas de software que puede formar parte, especialmente en entornos orquestados como Docker Swarm y Kubernetes.

## Expansión sobre Arquitecturas de Procesador en Docker

Docker se diseñó con la portabilidad entre diferentes sistemas y arquitecturas de hardware en mente. Aquí están las arquitecturas de procesador más comunes compatibles con Docker y algunos detalles específicos:

### **x86-64 (AMD64/Intel 64)**

**Predominio:** La mayoría de los desarrollos y despliegues de Docker se realizan en esta arquitectura debido a su prevalencia en servidores y estaciones de trabajo modernos.

**Características:** Amplio soporte de herramientas de desarrollo y operación, incluyendo integraciones con IDEs, sistemas de CI/CD, y monitoreo.

### **ARM (Arm32v7/Arm64v8)**

**Emergencia:** Con la creciente popularidad de dispositivos IoT y servidores de bajo consumo energético, ARM se ha vuelto una arquitectura importante para Docker.

**Especificaciones:** A menudo requiere imágenes de Docker especializadas que están optimizadas para el bajo poder de procesamiento y eficiencia energética de estos chips.

### **IBM Z (s390x)**

**Uso:** Predominantemente usado en grandes sistemas empresariales como mainframes, donde la estabilidad y el procesamiento de grandes volúmenes de transacciones son críticos.

**Características:** Soporte para Docker está disponible, permitiendo modernizar aplicaciones y aprovechar contenedores en un entorno de mainframe.

### **PowerPC (ppc64le)**

**Aplicación:** Utilizado en computadoras de alto rendimiento y servidores que requieren capacidades de procesamiento paralelo avanzadas.

**Ventajas:** Soporte de Docker en esta arquitectura permite a las empresas integrar aplicaciones legacy con tecnologías modernas de contenedores.

# Expansión sobre Arquitecturas de Software en Docker

En cuanto a las arquitecturas de software, Docker proporciona flexibilidad no solo a nivel de sistema operativo sino también en la manera de gestionar y orquestar contenedores a gran escala.

## Docker Swarm

**Descripción:** Docker Swarm es una herramienta de orquestación nativa de Docker que permite a los usuarios gestionar un clúster de contenedores Docker como un único sistema virtual.

### Características:

**Facilidad de uso:** Comparativamente más sencillo de configurar y mantener que Kubernetes.

**Integración directa:** Totalmente integrado con el ecosistema Docker, incluyendo Docker CLI y Docker Compose.

**Load balancing:** Automático entre los contenedores para distribuir las cargas de trabajo de manera eficiente.

## Kubernetes

**Descripción:** Kubernetes es un orquestador de contenedores de código abierto que automatiza la implementación, el escalado y la gestión de aplicaciones contenerizadas.

### Características:

**Extensibilidad:** Soporte para múltiples proveedores de servicios en la nube, así como soluciones on-premise.

**Auto-curación:** Capacidad para reiniciar contenedores que fallan, reemplazar y reprogramar contenedores cuando los nodos mueren, y matar contenedores que no responden a las verificaciones de estado definidas por el usuario.

**Gestión de configuraciones:** Gestiona secretos y la configuración de aplicaciones sin reconstruir las imágenes de contenedor.

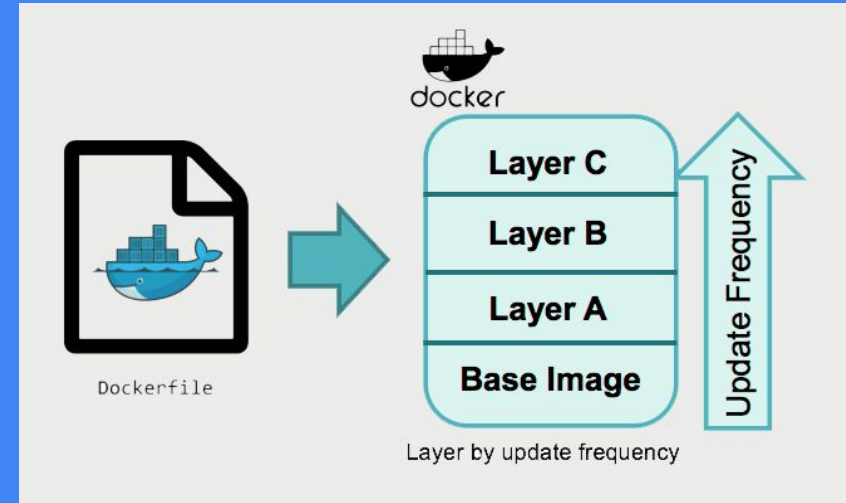
# Expansión sobre Arquitecturas de Software en Docker

Característica	Docker Swarm	Kubernetes
Instalación	Más sencilla y menos componentes	Más compleja, especialmente en configuraciones de alta disponibilidad
Escalabilidad	Escala bien hasta cientos de nodos	Diseñado para escalar a miles de nodos
Gestión de Configuración	Usa Compose y Swarm stack files	Usa objetos YAML más complejos y versátiles
Gestión de Configuración	Amplio, pero menor que Kubernetes	Extremadamente amplio y creciente
Gestión de Configuración	Preferido para aplicaciones más simples o donde la integración de Docker es prioritaria	Estándar de facto para aplicaciones complejas y microservicios a gran escala

# 12.- Dockers

## Capas

Las capas en Docker son un concepto fundamental que facilita la gestión eficiente de las imágenes de contenedores. Este concepto no solo optimiza el almacenamiento y la transferencia de imágenes, sino que también acelera el proceso de despliegue de contenedores.



# 12.-Docker Capas

## Definición de Capas en Docker

Docker utiliza un sistema de archivos de capas para construir imágenes de contenedores. Cada capa representa una instrucción en el Dockerfile, y cada capa se apila sobre la anterior. Las capas son inmutables, lo que significa que una vez que una capa es creada, no puede ser alterada.

## Cómo Funcionan las Capas en Docker

### Creación de Imágenes

- Cuando se construye una imagen de Docker a partir de un Dockerfile, cada instrucción genera una nueva capa:
  - FROM crea una capa base.
  - RUN ejecuta comandos y los cambios resultantes forman una nueva capa.
  - COPY y ADD añaden archivos desde el contexto de construcción y crean nuevas capas.
  - CMD y ENTRYPOINT definen comandos para ejecutar al iniciar el contenedor y pueden ajustar capas existentes.
- Estas capas se almacenan en el sistema de archivos del host que ejecuta Docker y se comparten entre múltiples contenedores, reduciendo el uso de espacio y aumentando la velocidad de lanzamiento de nuevos contenedores.

### Ejemplo práctico usando Dockerfile:

Supongamos que tenemos el siguiente Dockerfile para una aplicación web simple:

```
FROM ubuntu:18.04
```

```
RUN apt-get update && apt-get install -y nginx
```

```
COPY index.html /var/www/html/index.html
```

```
CMD ["nginx", "-g", "daemon off;"]
```



## 12.-Docker Capas

- **Capa 1:** Basada en ubuntu:18.04.
- **Capa 2:** Contiene los cambios realizados por la instalación de nginx.
- **Capa 3:** Añade el index.html al directorio correcto.
- **Capa 4:** Configura el comando que se ejecutará al iniciar el contenedor.

### Beneficios del Uso de Capas

- **Reutilización de Capas:** Si varios Dockerfiles utilizan la misma instrucción base o capas intermedias idénticas, estas capas se comparten, reduciendo el tiempo de construcción y el espacio en disco necesario.
- **Eficiencia en Transferencias:** Cuando se actualizan las imágenes, solo las capas que han cambiado se descargan o envían a través de la red, lo que resulta en un proceso de actualización más rápido.

### Copy-on-Write (CoW)

Docker utiliza el principio de "copy-on-write" para gestionar las capas. Cuando un contenedor modifica un archivo, Docker copia este archivo desde la capa en la que se encuentra a una nueva capa perteneciente al contenedor actual, garantizando que las capas subyacentes nunca sean modificadas.

### Consideraciones de Seguridad y Mantenimiento

- **Seguridad:** Es vital asegurarse de que las capas intermedias no contengan datos sensibles, ya que estas capas pueden ser compartidas entre diferentes imágenes.
- **Mantenimiento:** Las capas que no se utilizan regularmente deben ser limpiadas para evitar el uso innecesario de espacio en disco, lo cual se puede hacer con comandos como `docker system prune`.

# 12.-Docker Capas

## Conclusión

El sistema de capas de Docker no solo optimiza el almacenamiento y la distribución de las imágenes de contenedores, sino que también facilita la gestión de versiones y la eficiencia operativa. Este diseño permite a los desarrolladores y administradores de sistemas maximizar los recursos y mejorar la seguridad al aislar las dependencias y reducir la redundancia. Comprender este sistema es crucial para cualquiera que trabaje con Docker a nivel profesional.

## Ejercicio Práctico: Explorando las Capas en Docker

### Objetivo del Ejercicio:

Demostrar la eficiencia en el uso de almacenamiento y tiempo de construcción mediante la reutilización de capas en Docker.

### Paso 1: Preparación

**Limpiar el Entorno (opcional):** Todos deben comenzar con el mismo entorno limpio para destacar la eficacia del almacenamiento de capas.

```
docker system prune -a
```

### Paso 2: Crear un Dockerfile Básico

#### 2.1: Dockerfile A:

Crea un Dockerfile llamado DockerfileA con el siguiente contenido:

```
FROM ubuntu:18.04  
RUN touch /file1.txt
```

# 12.-Docker Capas

## 2.2: Construir la Imagen A:

Construye una imagen a partir de DockerfileA:

```
docker build -t image-a -f DockerfileA .
```

## 2.3: Observar las capas:

Utiliza el comando para ver las capas de la imagen:

```
docker history image-a
```

## Paso 3: Modificar el Dockerfile y Reconstruir:

### 3.1: Dockerfile B:

Modifica DockerfileA para crear un nuevo Dockerfile llamado DockerfileB que añada un nuevo archivo:

```
FROM ubuntu:18.04
```

```
RUN touch /file1.txt
```

```
RUN touch /file2.txt
```

### 3.2: Construir la imagen B:

Construye una imagen a partir de DockerfileB:

```
docker build -t image-b -f DockerfileB .
```

### 3.3: Observar la Capas Nuevamente:

```
docker history image-b
```

# 12.-Docker Capas

## Paso 4: Discusión y Análisis:

### 4.1.Comparar las Capas:

Discute qué capas son compartidas entre image-a e image-b y cómo esto afecta el almacenamiento y el tiempo de construcción.

### 4.2.Ventajas de la Reutilización de Capas:

Explora cómo la reutilización de capas puede reducir significativamente el tiempo de descarga y espacio en disco cuando se despliegan nuevas versiones de contenedores que comparten bases comunes.

### 4.3.Copy-on-Write (CoW):

Explica cómo funciona el mecanismo de copy-on-write (CoW) en este contexto y por qué es beneficioso.

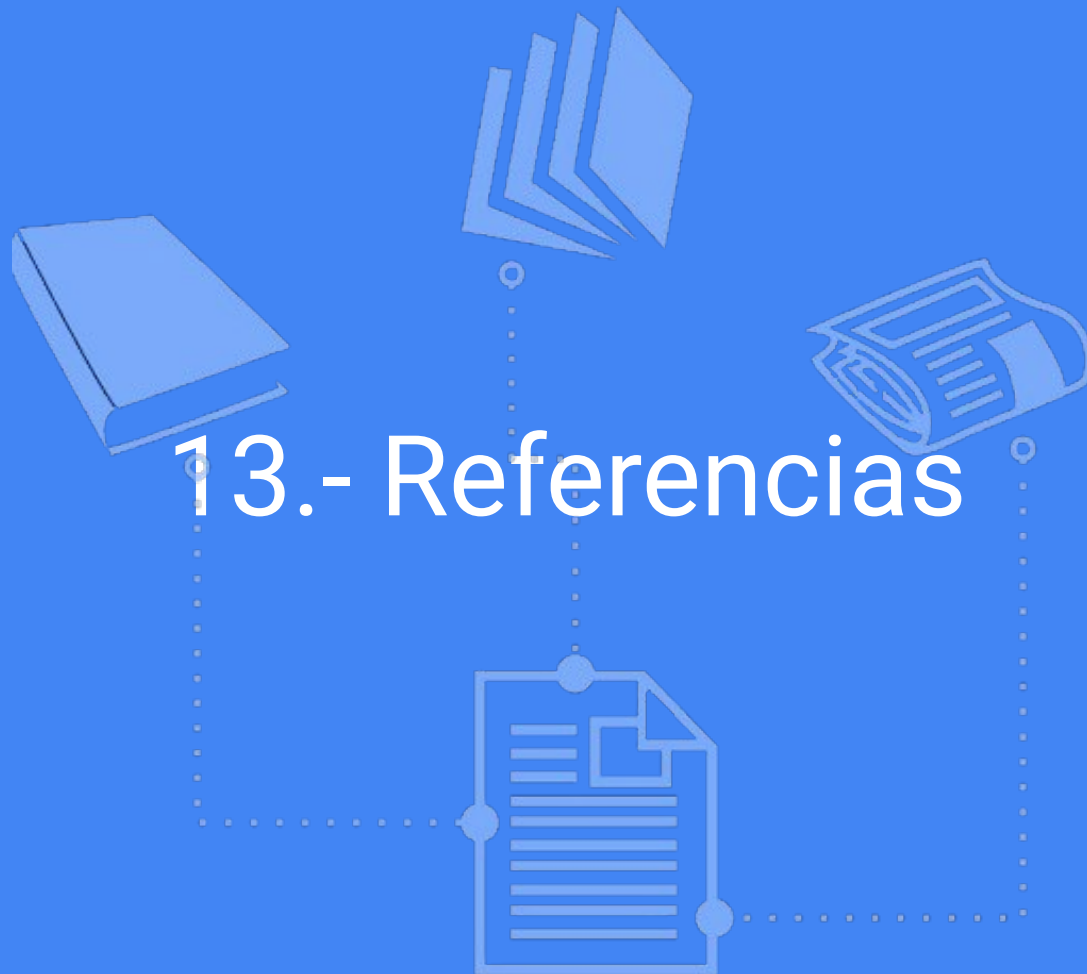
## Paso 5: Ejercicio Extendido (Opcional):

### 5.1.Experimentar con Capas Base Diferentes:

Crea variaciones de Dockerfile cambiando la imagen base (por ejemplo, alpine vs ubuntu) y observa cómo cambian las capas y el tamaño total de la imagen.

### 5.2.Uso de Caché en la Construcción:

Modifica los Dockerfiles para añadir comandos que invaliden la caché (como apt-get update) y observa cómo afecta al uso de capas.



## 13.- Referencias

## Documentación Oficial y Tutoriales

### Documentación Oficial de Docker

Descripción: Contiene la documentación completa de Docker, incluyendo guías de inicio rápido, tutoriales avanzados, y referencia técnica.

URL: [Docker Documentation](#)

### Get Started with Docker

Descripción: Un tutorial introductorio proporcionado por Docker para comenzar rápidamente con Docker y entender sus principios básicos.

URL: [Get Started](#)

Cursos y Tutoriales en Video

### Docker Mastery: with Kubernetes +Swarm from a Docker Captain

Descripción: Un curso muy popular en Udemy que cubre Docker así como orquestación usando Docker Swarm y Kubernetes, impartido por Bret Fisher, uno de los capitanes de Docker.

URL: [Docker Mastery on Udemy](#)

### Learn Docker in 12 Minutes

Descripción: Un video corto de Jake Wright que proporciona una introducción rápida a Docker, ideal para una comprensión inicial rápida.

URL: [Learn Docker in 12 Minutes - YouTube](#)

## Libros

### "Docker Deep Dive" por Nigel Poulton

Descripción: Este libro es excelente para aquellos que desean profundizar en Docker y comprender los detalles técnicos de cómo funciona Docker y cómo se puede utilizar de manera efectiva.

URL: [Docker Deep Dive on Amazon](#)

### "Docker Up & Running: Scaling and Managing Containerized Applications" por Sean P. Kane y Karl Matthias

Descripción: Ideal para entender la operación y escalado de aplicaciones contenerizadas en producción.

URL: [Docker Up & Running on Amazon](#)

## Blogs y Artículos

### Docker Blog

Descripción: El blog oficial de Docker donde publican regularmente actualizaciones, guías, y estudios de caso que son muy útiles para mantenerse al día con las últimas prácticas y características.

URL: [Docker Blog](#)

### "A Beginner's Guide to Docker — How to Create a Client/Server Side with Docker-Compose" en freeCodeCamp

Descripción: Un artículo introductorio que explica cómo configurar un ambiente de desarrollo cliente/servidor usando Docker Compose.

URL: [Beginner's Guide to Docker on freeCodeCamp](#)

## Comunidades y Foros

### Stack Overflow - Docker Tag

Descripción: Una comunidad activa donde puedes hacer preguntas específicas y obtener respuestas de profesionales en el área.

URL: [Stack Overflow Docker Tag](#)

### Docker Community Forums

Descripción: Los foros de la comunidad Docker son un buen lugar para discutir problemas, características y mejores prácticas con otros usuarios de Docker.

URL: [Docker Forums](#)

Gracias ;)