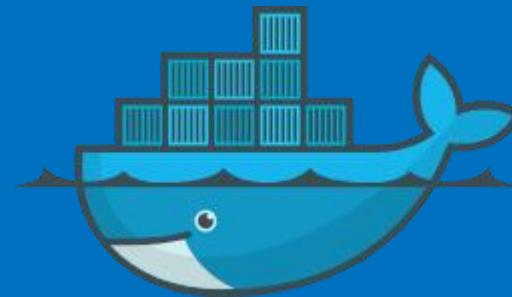


Desarrollo avanzando de código

# Docker y DockerHub

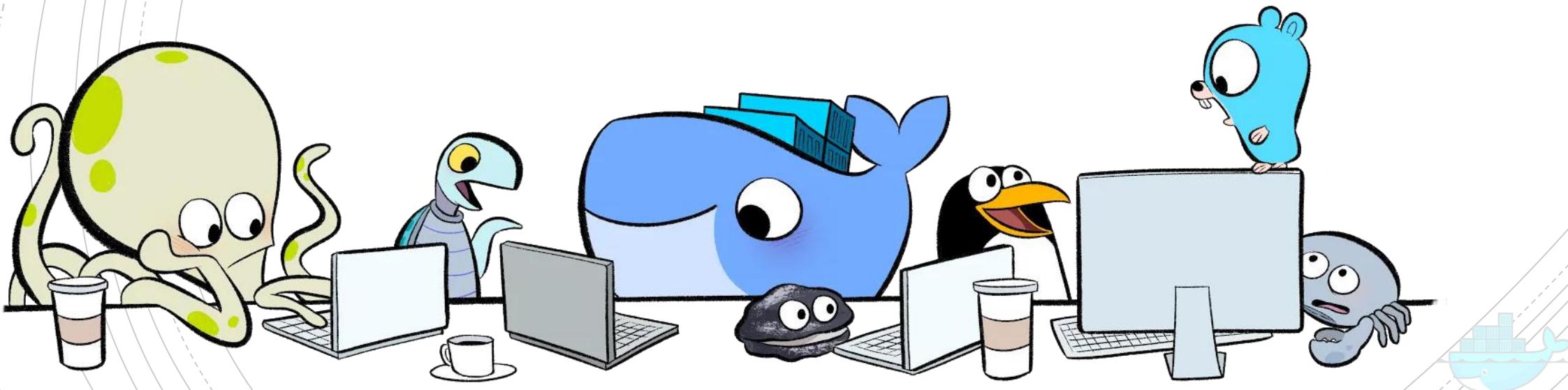
Manejo de contenedores

José Luis González Sánchez  
<https://github.com/joseluisgs>



# ¿Qué vamos a aprender?

- Aprenderemos a manejar imágenes y contenedores y cómo aplicarlos para mejorar en el desarrollo de software
- Siempre con el objetivo de poder crear un entorno que podamos compartir y facilitar el despliegue de nuestro proyecto



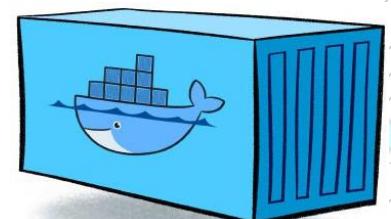
# ¿Qué es Docker?

- Docker es una plataforma para desarrollar, lanzar y ejecutar aplicaciones. Permite separar las aplicaciones desarrolladas de la infraestructura donde se desarrollan y trabajar así más comoda y rápidamente.
- Docker permite empaquetar y lanzar una aplicación en un entorno totalmente aislado llamado **contenedor**. Estos contenedores se ejecutan directamente sobre el kernel de la máquina por lo que son mucho más ligeros que las máquinas virtuales.
- Con Docker podemos ejecutar mucho más contenedores para el mismo equipo que si éstos fueran máquinas virtuales. De esta manera, podemos probar rápidamente nuestra aplicación web, por ejemplo, en múltiples entornos distintos al de donde nos encontramos desarrollando. Realmente es mucho más rápido que hacerlo en una máquina virtual, puesto que reduce el tiempo de carga y el espacio requerido por cada uno de estos contenedores o máquinas.



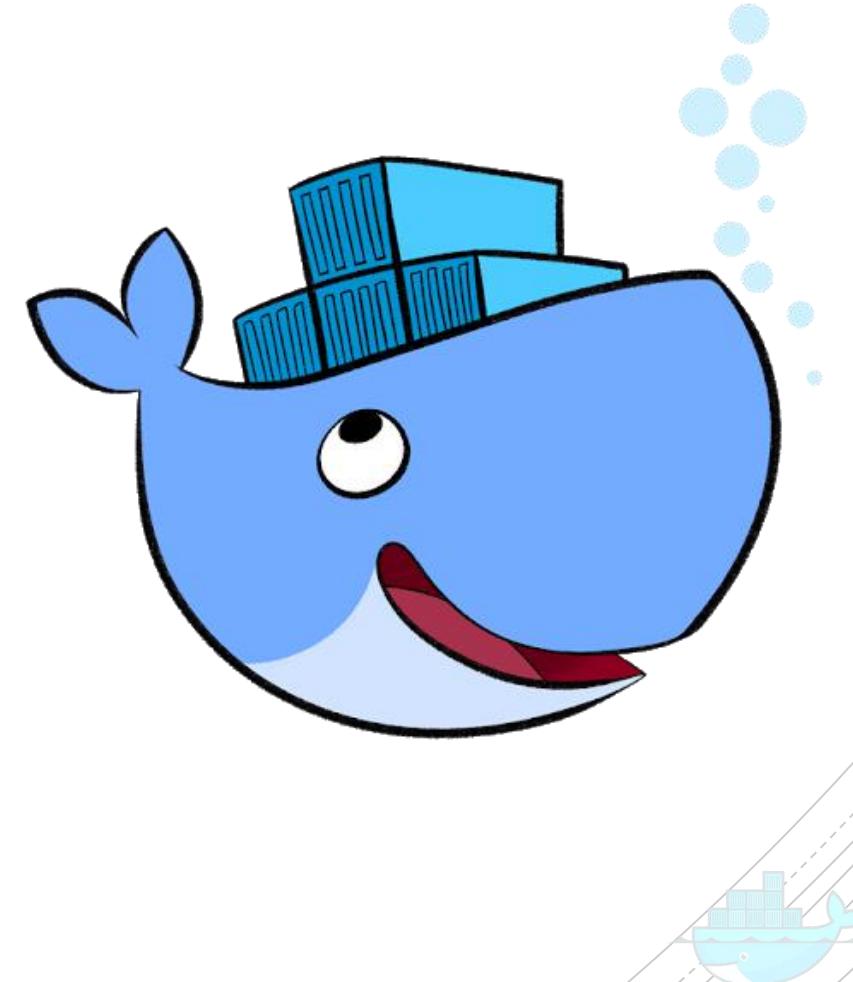
# Contenedores

- Los **contenedores** se distinguen de las máquinas virtuales en que las máquinas virtuales emulan un ordenador físico en el que se instala un sistema operativo completo, mientras que los contenedores usan el kernel del sistema operativo anfitrión pero contienen las capas superiores (sistema de ficheros, utilidades, aplicaciones).
- Al ahorrarse la emulación del ordenador y el sistema operativo de la máquina virtual, los contenedores son más pequeños y rápidos que las máquinas virtuales. Pero al incluir el resto de capas de software, se consigue el aislamiento e independencia entre contenedores que se busca con las máquinas virtuales.



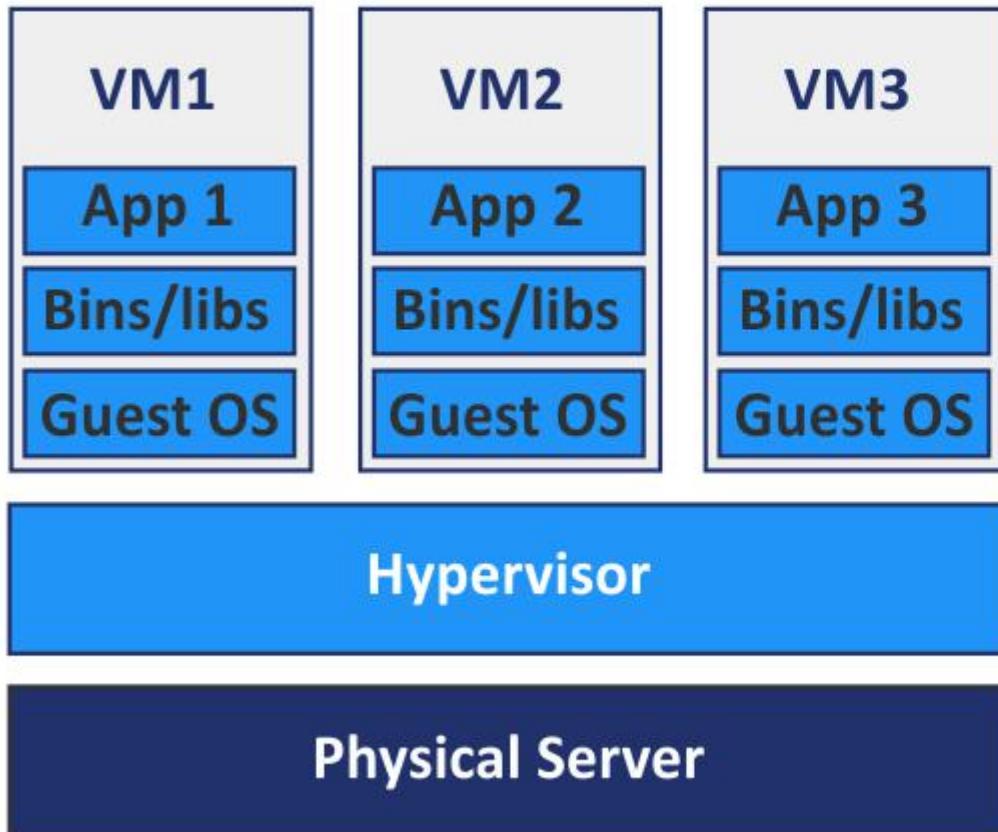
# Contenedores

- **Docker** no es virtualizado, no hay un hipervisor. Los procesos que corren dentro de un contenedor de docker se ejecutan con el mismo kernel que la máquina anfitrión. Linux lo que hace es aislar esos procesos del resto de procesos del sistema, ya sean los propios de la máquina anfitrión o procesos de otros contenedores.
- Además, es capaz de controlar los recursos que se le asignan a esos contenedores (cpu, memoria, etc).
- Internamente, el contenedor no sabe que lo es y a todos los efectos es una distribución GNU/Linux independiente, pero sin la penalización de rendimiento que tienen los sistemas virtualizados.
- Así que, cuando ejecutamos un contenedor, estamos ejecutando un servicio dentro de una distribución construida a partir de una "receta". Esa receta permite que el sistema que se ejecuta sea siempre el mismo, independientemente de si estamos usando Docker en Ubuntu, Fedora o, incluso, sistemas privativos compatibles con Docker.
- De esa manera podemos garantizar que estamos desarrollando o desplegando nuestra aplicación, siempre con la misma versión de todas las dependencias.

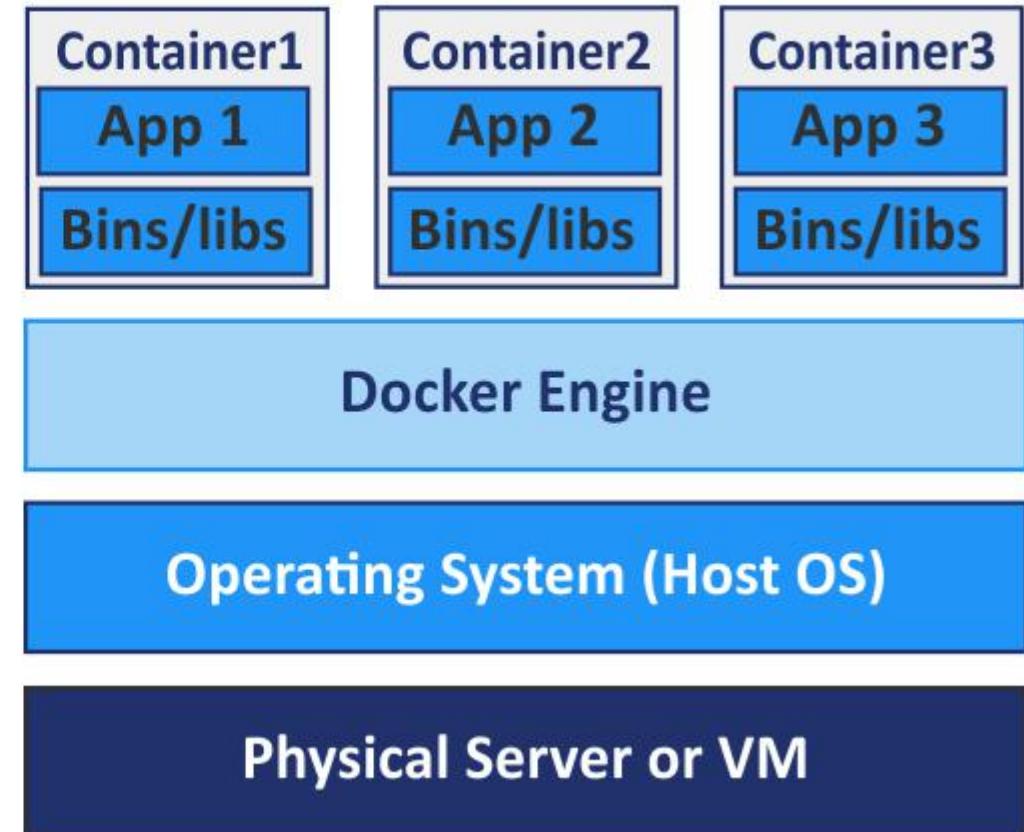


# Contenedores

## Virtual Machines

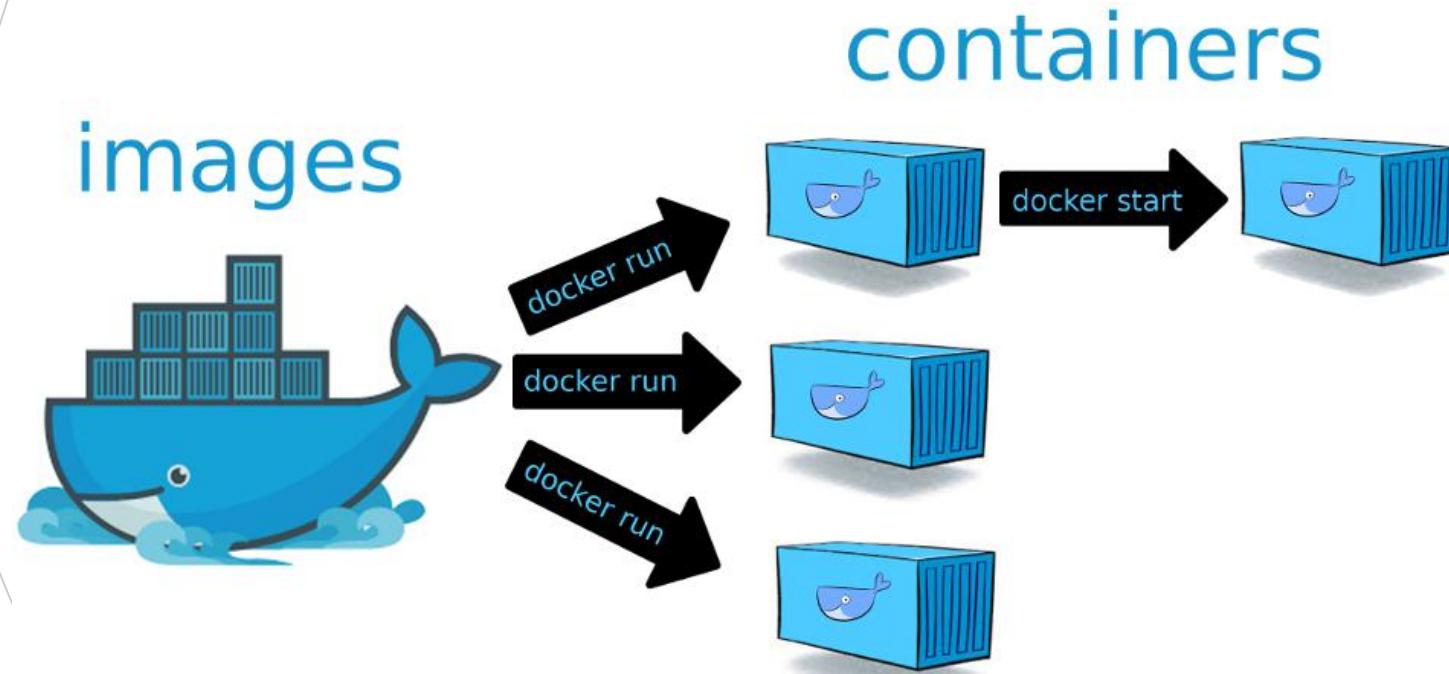


## Containers



# Imagenes

- Una Imagen es una plantilla de solo lectura que contiene las instrucciones para crear un contenedor Docker. Pueden estar basadas en otras imágenes, lo cual es habitual. Para ello usaremos distintos ficheros como el dockfile.
- Por ejemplo una imagen podría contener un sistema operativo Ubuntu con un servidor Apache y tu aplicación web instalada.



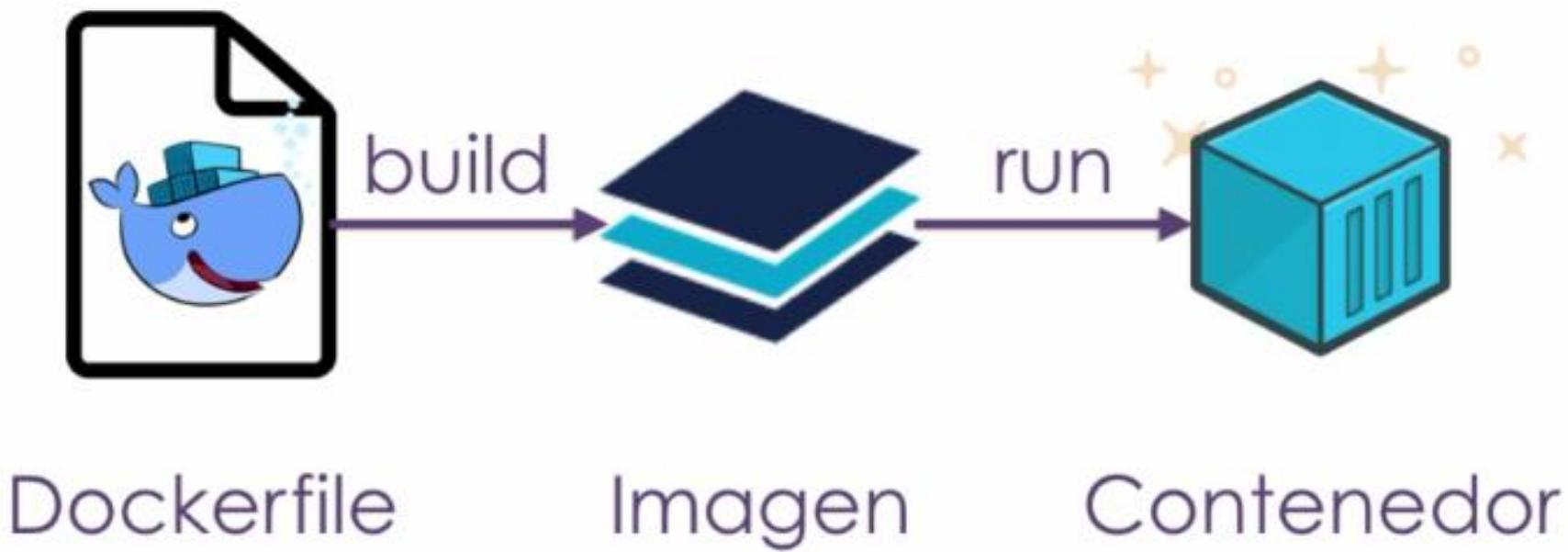
# Imágenes y contenedores

- Por lo tanto, un contenedor es una instancia ejecutable de una imagen.
- Esta instancia puede ser creada, iniciada, detenida, movida o eliminada a través del cliente de Docker o de la API.
- Las instancias se pueden conectar a una o más redes, sistemas de almacenamiento, o incluso se puede crear una imagen a partir del estado de un contenedor.
- Se puede controlar cómo de aislado está el contenedor del sistema anfitrión y del resto de contenedores.
- El contenedor está definido tanto por la imagen de la que procede como de las opciones de configuración que permita.
- Por ejemplo, la imagen oficial de MariaDb permite configurar a través de opciones la contraseña del administrador, de la primera base de datos que se cree, del usuario que la maneja, etc.

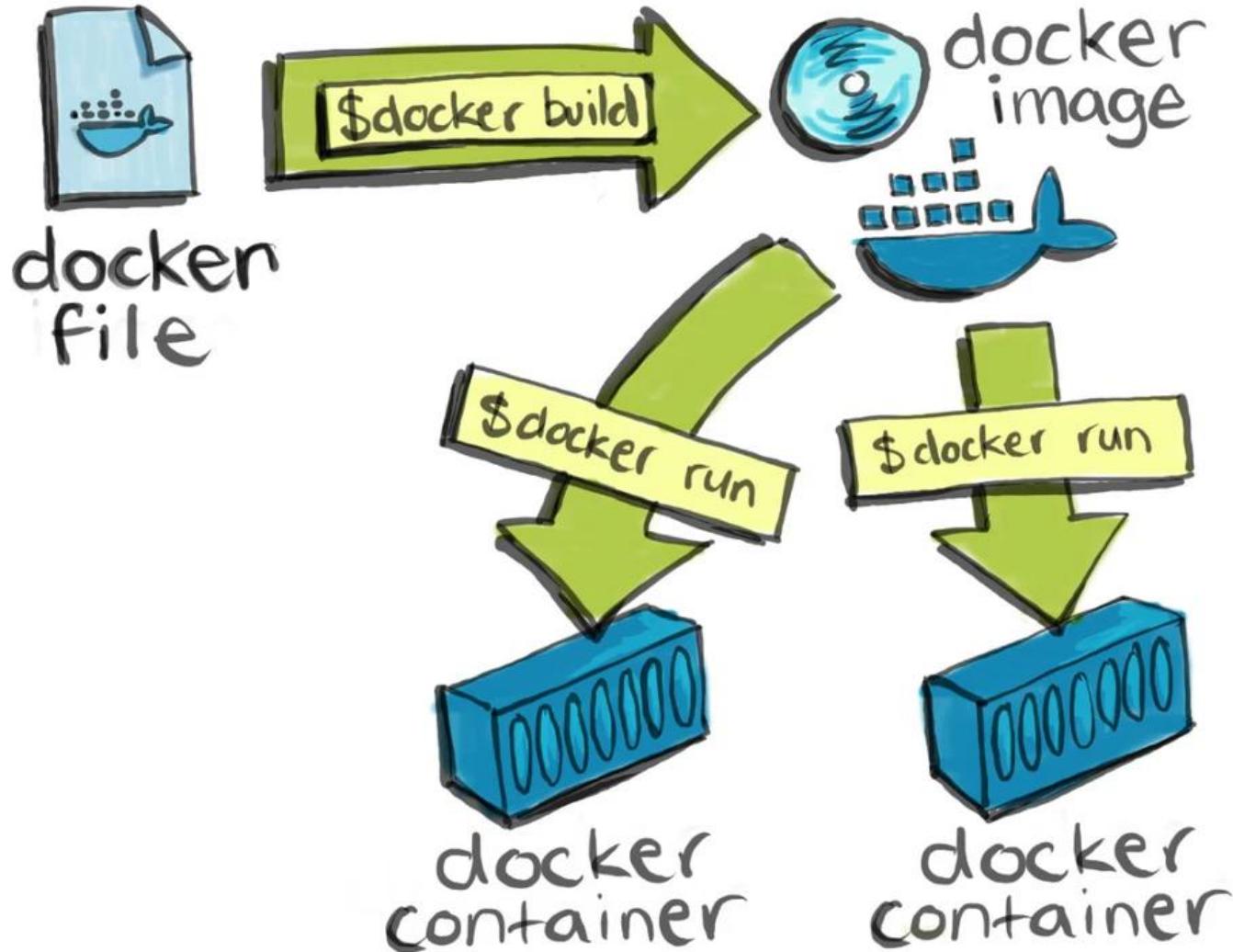


# Dockerfile

- Un Dockerfile es un archivo de texto plano que contiene una serie de instrucciones necesarias para crear una imagen que, posteriormente, se convertirá en los contenedores que ejecutamos en el sistema
- Es como la receta para definir la imagen, que posteriormente lanzaremos como contenedor

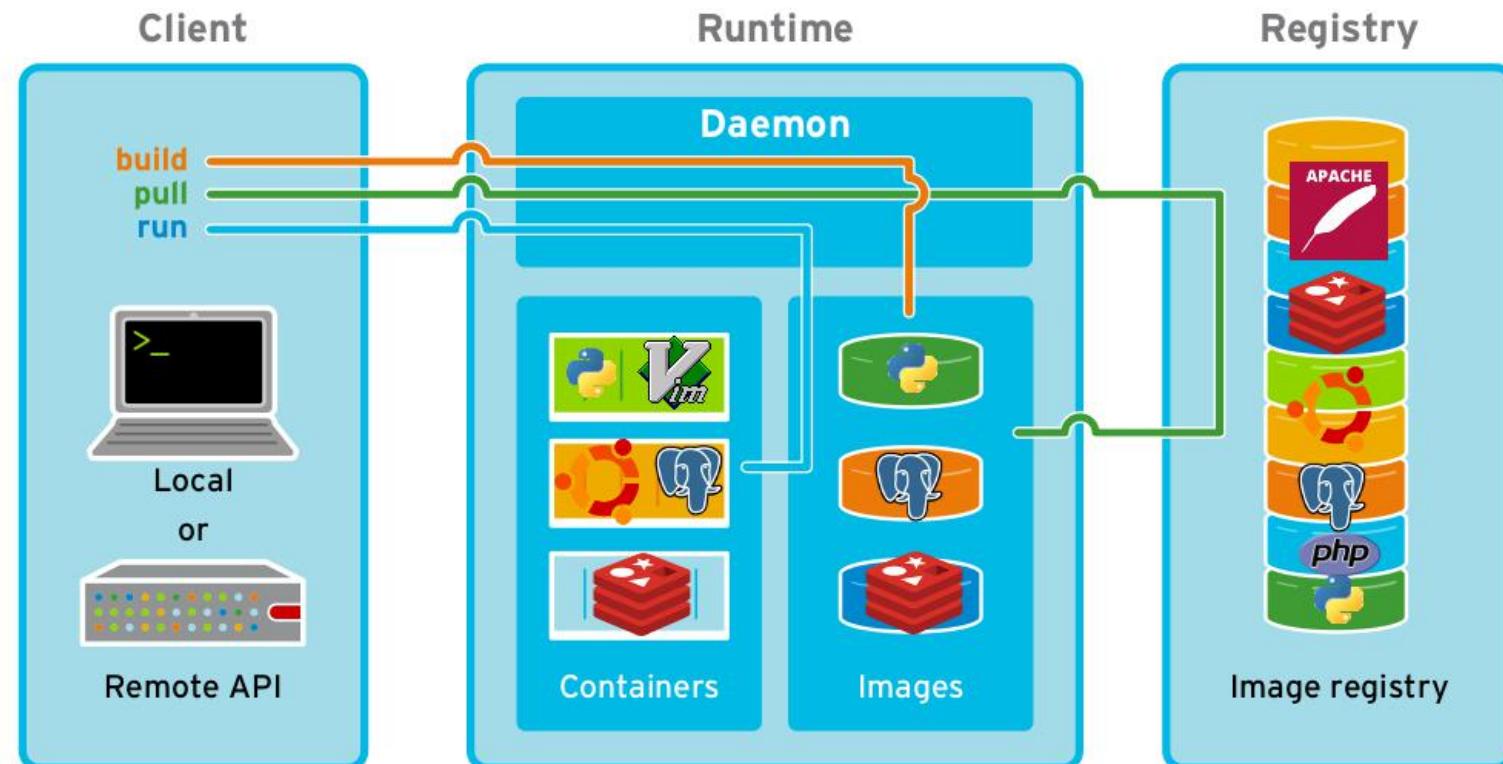


# Resumiendo



# DockerHub

- DockerHub es el registro de imágenes donde podemos subir nuestras imágenes o encontrar imágenes hechas con la que trabajar sobre ellas. Podemos usarlo de manera muy similar a la que usamos GitHub y nos podemos dar de alta en dicho servicio para almacenar nuestras imágenes



# Instalar Docker

- Debido a que, dependiendo de la distribución, la forma de instalarlo difiere, es mejor consultar la documentación oficial para saber como instalar Docker en tu máquina.
- <https://docs.docker.com/get-docker/>

- Desistalar versiones anteriores

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

- Configurando el repositorio

```
sudo apt-get update  
sudo apt-get install \  
apt-transport-https \  
ca-certificates \  
curl \  
gnupg-agent \  
software-properties-common
```



# Instalar Docker

- Añadimos la clave GPG

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

- Añadimos el repositorio

```
sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

- Instalamos

```
sudo apt-get update
```

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



# Instalar Docker

- Añadimos el usuario, de esta manera nos ahorramos hacer sudo siempre

```
sudo usermod -aG docker $USER
```

- Instalamos Docker Compose

```
sudo apt install docker-compose
```

- Probamos el primer contenedor

```
sudo docker run hello-world
```

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adb7777e9aacf18357296e799f81cabc9fde470971e499788
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.  
(amd64)
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:

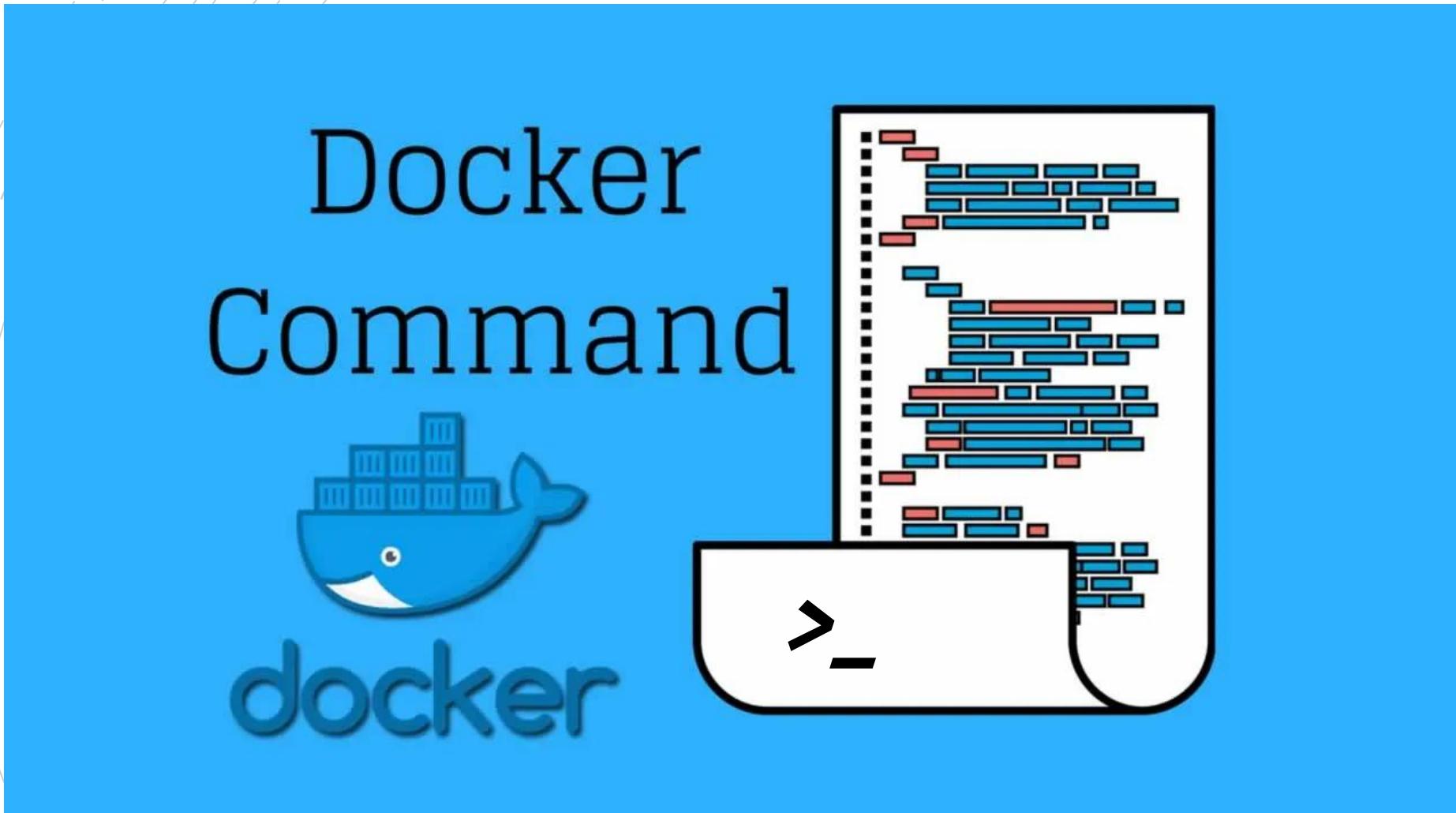
```
https://hub.docker.com/
```

For more examples and ideas, visit:

```
https://docs.docker.com/get-started/
```



# Comandos Docker



# Comandos Docker

- Para ver los comandos disponibles (en general y en particular):

```
sudo docker
```

```
sudo docker image
```

```
sudo docker network
```

- Para ver las opciones de cada comando:

```
sudo docker cp --help
```

- Los comandos tienen a su vez opciones. Los nombres de las opciones van precedidas de los caracteres -- y entre la opción y su valor se puede escribir un espacio o el carácter =.

```
sudo docker COMANDO --OPCIÓN=VALOR
```

```
sudo docker COMANDO --OPCIÓN VALOR
```

- Si el valor de una opción contiene espacios, escriba el valor entre comillas

```
sudo docker COMANDO --OPCIÓN="VALOR CON ESPACIOS"
```

```
sudo docker COMANDO --OPCIÓN "VALOR CON ESPACIOS"
```



# Comandos con imágenes

- Para gestionar las imágenes, se utiliza el comando:

```
sudo docker image OPCIONES
```

- Para descargar una imagen:

```
sudo docker image pull REPOSITORIO
```

```
sudo docker pull REPOSITORIO
```

- Para ver las imágenes ya descargadas:

```
sudo docker image ls
```

```
sudo docker images
```

- Para borrar una imagen (se deben borrar previamente los contenedores basados en esa imagen):

```
sudo docker image rm IMAGEN
```

```
sudo docker rmi IMAGEN
```



# Comandos con contenedores

- Para crear un contenedor (y ponerlo en marcha):

```
sudo docker run --name=CONTENEDOR REPOSITORIO
```

El problema de este comando es que dejamos de tener acceso a la shell y sólo se puede parar el proceso desde otro terminal.

Lo primero que hace es bajarse la imagen si no está en el sistema

Lo habitual es poner en marcha el contenedor en modo separado (detached), es decir, en segundo plano, y así podemos seguir utilizando la shell:

```
sudo docker run -d --name=CONTENEDOR REPOSITORIO
```

- Si queremos ver la secuencia de arranque del contenedor, podemos poner en marcha el contenedor en modo pseudo-tty, que trabaja en primer plano, pero del que podemos salir con Ctrl+C.

```
sudo docker run -t --name=CONTENEDOR REPOSITORIO
```



# Comandos con contenedores

- Al crear el contenedor se pueden añadir diversas opciones:

Para incluir el contenedor en una red privada virtual (y que se pueda comunicar con el resto de contenedores incluidos en esa red):

```
sudo docker run --name=CONTENEDOR --net=RED REPOSITORIO
```

Para que el contenedor atienda a un puerto determinado, aunque internamente atienda un puerto distinto:

```
sudo docker run --name=CONTENEDOR -p PUERTO_EXTERNO:PUERTO_INTERNO REPOSITORIO
```

Para establecer variables de configuración del contenedor:

```
sudo docker run --name=CONTENEDOR -e VARIABLE=VALOR REPOSITORIO
```

Las variables de configuración se pueden consultar en el repositorio del que obtenemos la imagen.



# Comandos con contenedores

- Para ver los contenedores en funcionamiento:  
`sudo docker ps`
- Para ver los contenedores en funcionamiento o detenidos:  
`sudo docker ps -a`
- Para detener un contenedor:  
`sudo docker stop CONTENEDOR`
- Para detener todos los contenedores:  
`sudo docker stop ($sudo docker ps -aq)`
- Para borrar un contenedor:  
`sudo docker rm CONTENEDOR`
- Para poner en marcha un contenedor detenido:  
`sudo docker start CONTENEDOR`



# Comandos con contenedores

- Para entrar en la shell de un contenedor:

```
sudo docker exec -it CONTENEDOR /bin/bash
```

- Para entrar en la shell del contenedor como root:

```
sudo docker exec -u 0 -it CONTENEDOR /bin/bash
```

- Para salir de la shell del contenedor:

```
exit
```

- Para copiar (o mover) archivos entre el contenedor y el sistema anfitrión o viceversa:

```
sudo docker cp CONTENEDOR:ORIGEN DESTINO
```

```
sudo docker cp ORIGEN CONTENEDOR:DESTINO
```



# Comandos de red

- Para gestionar las redes, se utiliza el comando:  
`sudo docker network OPCIONES`
- Para crear una red:  
`sudo docker network create RED`
- Para ver las redes existentes:  
`sudo docker network ls`
- Para ver información detallada de una red (entre ella, los contenedores incluidos y sus IP privadas):  
`sudo docker network inspect RED`
- Para borrar una red:  
`sudo docker network rm RED`



# Comandos de sistema

- Para ver el espacio ocupado por las imágenes, los contenedores y los volúmenes:

```
sudo docker system df
```

- Para eliminar los elementos que no están en marcha:

Contenedores:

```
sudo docker container prune
```

Imágenes:

```
sudo docker image prune
```

Volúmenes:

```
sudo docker volume prune
```

Redes:

```
sudo docker network prune
```

todo:

```
sudo docker system prune -a
```



# Repaso de comandos con Hello-Run

- Comprueba que inicialmente no hay ningún contenedor creado (la opción -a hace que se muestren también los contenedores detenidos, sin ella se muestran sólo los contenedor que estén en marcha):

```
sudo docker ps -a
```

o también

```
sudo docker container ls -a
```

\$ docker ps -a CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
---------------------------------	-------	---------	---------	--------	-------	-------

- Comprueba que inicialmente tampoco disponemos de ninguna imagen:

```
sudo docker image ls
```

docker image ls REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
-------------------------------	-----	----------	---------	------



# Repaso de comandos con Hello-Run

- Docker crea los contenedores a partir de imágenes locales (ya descargadas), pero si al crear el contenedor no se dispone de la imagen local, Docker descarga la imagen de su repositorio.
- La orden más simple para crear un contenedor es:  
`sudo docker run IMAGEN`
- Crea un contenedor con la aplicación de ejemplo hello-world. La imagen de este contenedor se llama hello-world:  
`sudo docker run hello-world`
- Comprueba que inicialmente tampoco disponemos de ninguna imagen:  
`sudo docker image ls`

```
$ docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
hello-world         latest   bf756fb1ae65  9 months ago  13.3kB
```

# Repaso de comandos con Hello-Run

- Si listamos ahora los contenedores existentes ...

```
sudo docker ps -a
```

- ... se mostrará información del contenedor creado:

```
sudo docker run hello-world
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
10a829dbd778	hello-world	"/hello"	About a minute ago	Exited (0) About a minute ago		strange_cerf

- Cada contenedor tiene un identificador (ID) y un nombre distinto. Docker "bautiza" los contenedores con un nombre peculiar, compuesto de un adjetivo y un apellido.

# Repaso de comandos con Hello-Run

- Podemos crear tantos contenedores como queramos a partir de una imagen. Una vez la imagen está disponible localmente, Docker no necesita descargarla y el proceso de creación del contenedor es inmediato (aunque en el caso de hello-world la descarga es rápida, con imágenes más grandes la descarga inicial puede tardar un rato)
- Normalmente se aconseja **usar siempre la opción -d, que arranca el contenedor en segundo plano (detached)** y permite seguir teniendo acceso a la shell (aunque con hello-world no es estrictamente necesario porque el contenedor hello-world se detiene automáticamente tras mostrar el mensaje).
- Al crear el contenedor hello-world con la opción -d no se muestra el mensaje, simplemente muestra el identificador completo del contenedor.

```
$ sudo docker run -d hello-world  
f3059d7adaf9b2bb28b749bba44785fa331f5334b6cf87c5c9b7c7ccc13e8d29
```

```
$ sudo docker ps -a  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES  
f3059d7adaf9        hello-world         "/hello"           13 seconds ago   Exited (0) 12 seconds ago  
affectionate_varahamihira  hello-world         "/hello"           4 minutes ago    Exited (0) 4 minutes ago  
10a829dbd778        hello-world         "/hello"
```

# Repaso de comandos con Hello-Run

- Los contenedores se pueden destruir mediante el comando rm, haciendo referencia a ellos mediante su nombre o su id. No es necesario indicar el id completo, basta con escribir los primeros caracteres (de manera que no haya ambigüedades).

- Borra los dos contenedores existentes:

```
sudo docker rm 1ae
```

```
sudo docker rm affectionate_varahamihira
```

- Comprueba que ya no quedan contenedores



# Repaso de comandos con Hello-Run

- Podemos dar nombre a los contenedores al crearlos:  
`sudo docker run -d --name=hola-1 hello-world`
- Al haber utilizado la opción `-d` únicamente se mostrará el ID completo del contenedor
- Si listamos los contenedores existentes ...  
`sudo docker ps -a`
- ... se mostrará el contenedor con el nombre que hemos indicado y podremos trabajar con él
- Si intentamos crear un segundo contenedor con un nombre ya utilizado ...  
`sudo docker run -d --name=hola-1 hello-world`
- Docker nos avisará de que no es posible
- Podemos ver las imágenes con:  
`sudo docker images`  
Y borrarlas con:  
`sudo images rm nombre_imagen`
- Prueba a borrar la imagen `hello_world` y compruébalo



# Jugando con contenedores

- Crea un contenedor que contenga un servidor Apache a partir de la imagen bitnami/apache
- La opción -P hace que Docker asigne de forma aleatoria un puerto de la máquina virtual al puerto asignado a Apache en el contenedor. La imagen bitnami/apache asigna a Apache el puerto 8080 del contenedor para conexiones http y el puerto 8443 para conexiones https.

```
sudo docker run -d -P --name=apache-1 bitnami/apache
```

- Consulta el puerto del host utilizado por el contenedor ...

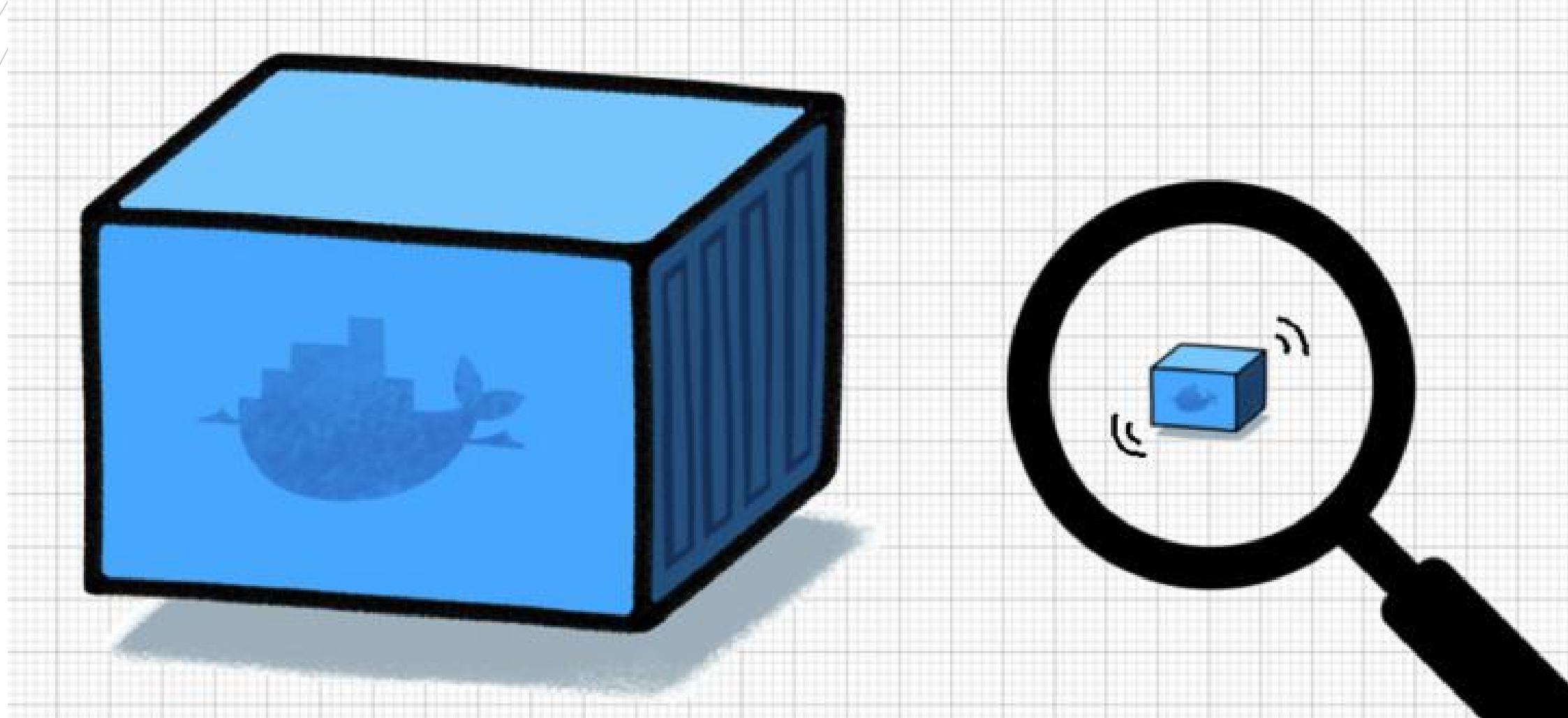
```
sudo docker ps -a
```

- ... se mostrará el contenedor con el nombre que hemos indicado:

- Abre en el navegador la página inicial del contenedor y compruebe que se muestra una página que dice "It works!".



# Reto Docker



# Jugando con contenedores

- En este apartado vamos a modificar la página web inicial de Apache del contenedor Docker.
- Debemos tener en cuenta que modificar el contenido de un contenedor tal y como vamos a hacer en este apartado sólo es aconsejable en un entorno de desarrollo, pero no es aconsejable en un entorno de producción porque va en contra de la "filosofía" de Docker.
- **Los contenedores de Docker están pensados como objetos de "usar y tirar", es decir, para ser creados, destruidos y creados de nuevo tantas veces como sea necesario y en la cantidad que sea necesaria.**
- En el apartado siguiente realizaremos la misma tarea de una forma más conveniente, modificando no el contenedor sino la imagen a partir de la cual se crean los contenedores.

# Jugando con contenedores

- Cree un segundo contenedor que contenga un servidor Apache a partir de la imagen httpd  
`sudo docker run -d -P --name=apache-daw httpd`
- Consulte el puerto del host utilizado por el contenedor ...  
`sudo docker ps -a`  
... se mostrarán los dos contenedores creados
- Crea la nueva página index.html ...  
`code index.html`
- Entre en la shell del contenedor para averiguar la ubicación de la página inicial:  
`sudo docker exec -it apache-daw /bin/bash`  
es: /usr/local/apache2/htdocs
- Salimos de la imagen  
`exit`
- Copiamos nuestra página web en esa ruta  
`docker cp index.html apache-daw:/usr/local/apache2/htdocs`
- Recargamos el ordenador... Ya tienes tu página ejecutada con Docker :)



# Jugando con contenedores

- Vamos a intentar ahorrarnos el copiar los ficheros siempre. Probamos lo siguiente desde donde tenemos la página web  
`docker run -dit --name miapache -p 5555:80 -v "$PWD":/usr/local/apache2/htdocs httpd`  
Una vez lanzado y puesto en marcha podemos ir a la dirección `http://localhost:5555` con nuestro navegador y visitar nuestro sitio web funcionando desde el contenedor.
- ¿Qué ha pasado?  
El contenedor tendrá asociado el nombre `miapache` que podrá ser utilizando a partir de su creación para realizar cualquier operación sobre él  
Se ha mapeado el puerto 80 del contenedor sobre el puerto 5555 de la máquina real. De esa manera, conectándonos al puerto 5555 de nuestro equipo podremos ver que ocurre en el 80 del contenedor, justamente donde estará escuchando el Apache del mismo  
Hemos mapeado la ruta actual de nuestro equipo con la ruta `/usr/local/apache2/htdocs` del contenedor, que es donde Apache va a ir a buscar la web que aloje. Suponemos que el directorio actual de mi máquina tengo el sitio web que quiero testear  
Indicamos que queremos crear y lanzar un contenedor utilizando la imagen del Hub de Docker, concretamente con la versión `latest`  
También hemos indicado con la opción `-d` que queremos que la máquina se lance en segundo plano  
con `it` dejamos que tengamos acceso a la consola por defecto.  
Puedes entrar a la maquina con  
`docker exec -it miapache /bin/bash`  
y ver en la carpeta `htdocs` que esta allí todo.  
Prueba a crear una página nueva a ver que pasa



# Dockerfile



# Dockerfile

- **Un Dockerfile es un archivo de texto plano que contiene una serie de instrucciones necesarias para crear una imagen que, posteriormente, se convertirá en el patrón de nuestros contenedores.** Para ello le añadimos:

- **FROM:** indica la imagen base sobre la que se construirá la aplicación dentro del contenedor.

```
FROM <imagen>
```

```
FROM <imagen>:<tag>
```

Por ejemplo la imagen puede ser un sistema operativo como Ubuntu, Centos, etc. O una imagen ya existente en la cual con base a esta queramos construir nuestra propia imagen.

- **RUN:** nos permite ejecutar comandos en el contenedor, por ejemplo, instalar paquetes o librerías (apt-get, yum install, etc.). Además, tenemos dos formas de colocarlo:

Opción 1 -> `RUN <comando>`

Esta instrucción ejecuta comandos Shell en el contenedor.

Opción 2 -> `["ejecutable", " parametro1", " parametro2"]`

Esta otra instrucción bastante útil, que permite ejecutar comandos en imágenes que no tengan /bin/sh.

- **ENV:** establece variables de entorno para nuestro contenedor, en este caso la variable de entorno es DEBIAN\_FRONTEND noninteractive, el cual nos permite instalar un montón de archivos .deb sin tener que interactuar con ellos.

```
ENV <key><valor>
```



# Dockerfile

- **ADD:** esta instrucción copia archivos a un destino específico dentro del contenedor, normalmente nos sirve para dejar ubicados ciertos archivos que queremos mover entre directorios, podemos usar como origen una URL o un .tar y descomprimirlo

```
ADD <fuente> <destino>
```

```
ADD ./script.sh /var/tmp/script.sh
```

- **COPY:** esta instrucción copia archivos o directorios a un destino específico dentro del contenedor.

```
COPY <fuente> <destino>
```

```
COPY ./script.sh /var/tmp/script.sh
```

- **MAINTAINER:** Este nos permite indicar el nombre del autor del dockerfile.

```
MAINTAINER <nombre> <" correo">
```

```
MAINTAINER JL Gonzalez "jlgs@cifpvirgendegracia.com"
```

- **CMD:** esta instrucción nos provee valores por defecto a nuestro contenedor, es decir, mediante esta podemos definir una serie de comandos que solo se ejecutarán una vez que el contenedor se ha inicializado, pueden ser comandos Shell con parámetros establecidos.

```
CMD ["ejecutable", "parámetro1", "parámetro2"], este es el formato de ejecución.
```

```
CMD ["parámetro1", "parámetro2"], parámetro por defecto para punto de entrada.
```

```
CMD comando parámetro1 parámetro2, modo shell
```

# Dockerfile

- **ENTRYPOINT:** la instrucción entrypoint define el comando y los parámetros que se ejecutan primero cuando se ejecuta el contenedor. En simples palabras, todos los comandos pasados en la instrucción docker run <image> serán agregados al comando entrypoint

Opción 1 -> la forma exec es donde especificamos comandos y argumentos, como la sintaxis de los formatos JSON.

```
ENTRYPOINT ["executable", "param1", "param2"]
```

Opción 2 -> la otra forma es ejecutar un script para ejecutar los comandos que queremos como entrada en el contenedor.

```
COPY ./script-entrypoint.sh /  
ENTRYPOINT ["/script-entrypoint.sh"]  
CMD ["postgres"]
```

- **VOLUME:** esta instrucción crea un volumen como punto de montaje dentro del contenedor y es visible desde el host anfitrión marcado con otro nombre.

```
VOLUME /var/tmp
```

- **USER:** determina el nombre de usuario a utilizar cuando se ejecuta un contenedor, y adicionalmente cuando se ejecutan comandos como RUN, CMD, ENTRYPOINT o WORKDIR.

```
WORKDIR ruta/de/Proyecto
```

# Dockerfile

- Ahora vamos a revisar otro dockerfile, en el cual crearemos una imagen con base Ubuntu 16:04 y haremos las siguientes sub-tareas:

Actualizar sus paquetes

Instalar el paquete Nginx

Añadir un archivo en una ruta específica dentro del contenedor

Exponer un puerto

```
# Descarga la imagen de Ubuntu 16.04
FROM ubuntu:16.04

# Actualiza la imagen base de Ubuntu 16.04
RUN apt-get update

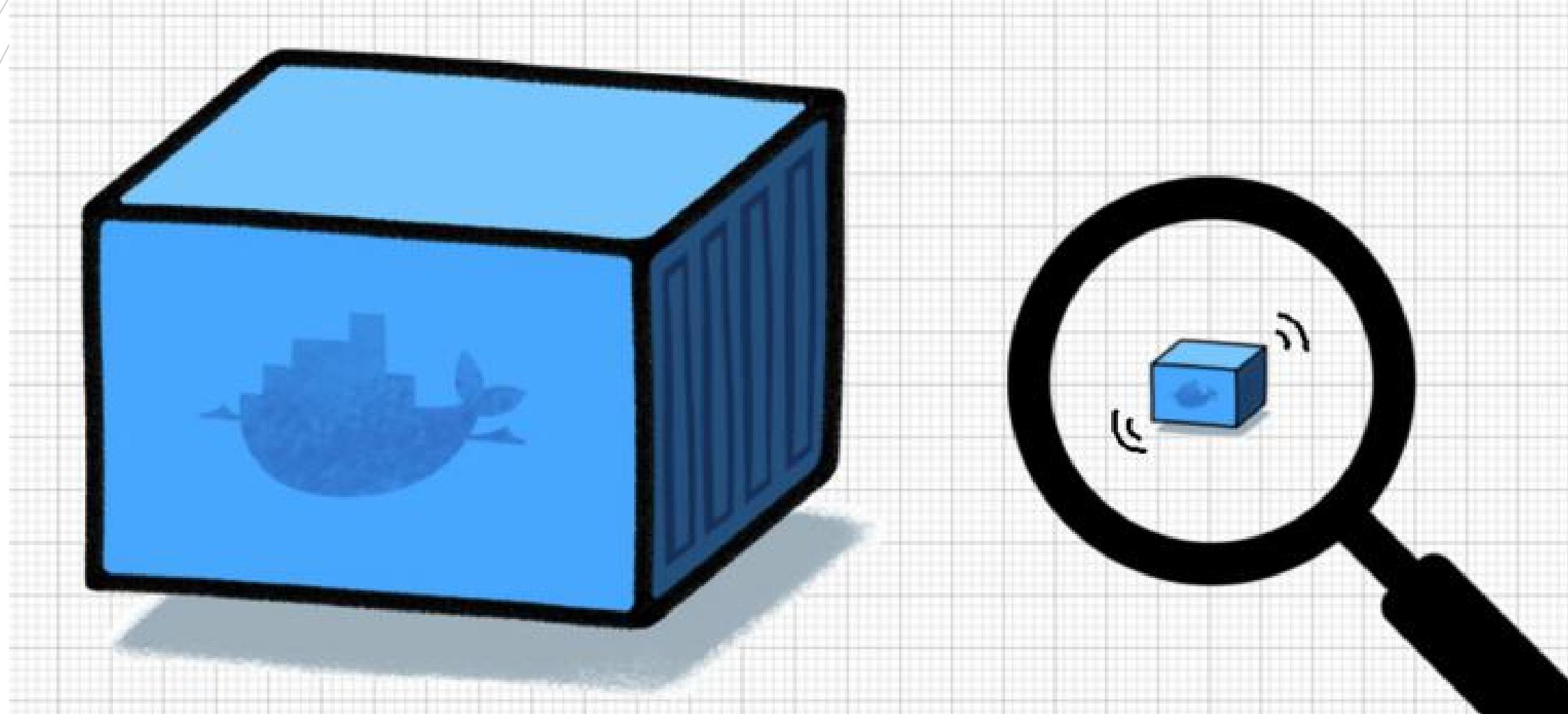
# Ejecuta el comando apt-get install y elimina determinados archivos y temporales
RUN apt-get install -y nginx \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# Indica los puertos TCP/IP los cuales se pueden acceder a los servicios del contenedor
EXPOSE 80

# Establece el comando del proceso de inicio del contenedor
CMD ["nginx"]
```



# Reto Dockerfile



# Personalizando imágenes

- Vamos a crear nuestra primera imagen de apache con php, para ello usaremos el fichero Dockerfile. Lo creamos teniendo en cuenta: [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
- El fichero Dockerfile define una plantilla que le dice como serán nuestros contenedores

```
# Imagen a usar
FROM php:7.0-apache
# copiamos todos los ficheros en el directorio en cuestion
COPY src/ /var/www/html
# Exponemos el puerto 80
EXPOSE 80
```

- Creamos la imagen con

```
docker build -t miapache-php .
```

(el punto es importante le dice donde esta el fichero Dockerfile, que es en este directorio)
- Comprobamos que existe

```
docker images
```
- Lanzamos la imagen

```
docker run -dit --name miapache-php -p 5555:80 miapache-php
```



# Personalizando imágenes

- A partir de aquí podremos parar y arrancar ese contenedor (o borrarlo).

```
docker stop miapache-php  
docker start miapache-php
```

- En este modelo de trabajo, cada vez que cambiemos el contenido de nuestra web deberemos regenerar la imagen local, podemos ( o no ) eliminar los contenedores viejos y volver a crear/correr el contenedor.

- Agregar contenido a la carpeta de trabajo y rearmar todo.

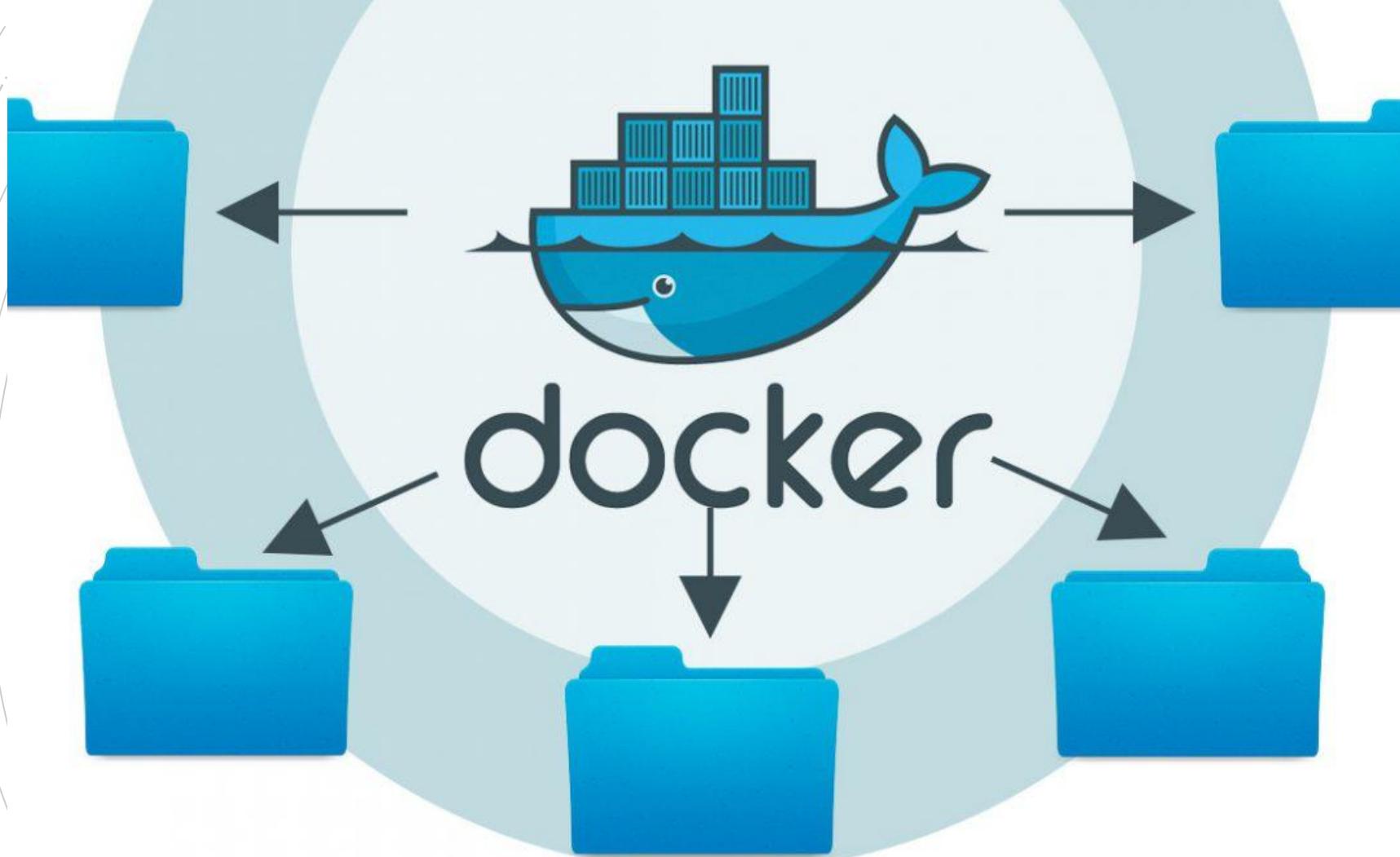
```
docker rmi -f miapache-php (borramos la imagen aunque esté siendo usada)  
docker rm -f miapache-php (borramos el contenedor aunque se esté ejecurando)  
docker build -t miapache-php . (construimos la imagen)  
docker run -dit --name miapache-php -p 5555:80 miapache-php (la ejecutamos)
```

- Podemos crearnos un script .sh para hacerlo todo, como run.sh

```
#!/bin/bash  
docker rmi -f miapache-php  
docker rm -f miapache-php  
docker build -t miapache-php .  
docker run -dit --name miapache-php -p 5555:80 miapache-php
```

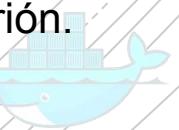


# Persistencia de datos



# Persistencia de datos

- Por defecto ya hemos indicado que un contenedor está aislado de todo. Hemos visto como podemos conectar el contenedor a un puerto de red para poder acceder a él. Eso incluye al sistema de archivos que contiene. De tal manera que si se elimina el contenedor, se eliminan también sus archivos.
- Si queremos almacenar datos (una web, una base de datos, etc.) dentro de un contenedor necesitamos una manera de almacenarlos sin perderlos.
- **Docker ofrece tres maneras:**
  - A través de volúmenes, que son objetos de Docker como las imágenes y los contenedores.**
  - Montando un directorio de la máquina anfitrión dentro del contenedor (enlazando directorios).**
  - Almacenándolo en la memoria del sistema (aunque también se perderían al reiniciar el servidor).**
- Lo normal es usar volúmenes, pero habrá ocasiones en que es preferible montar directamente un directorio de nuestro espacio de trabajo. Por ejemplo, para guardar los datos de una base de datos usaremos volúmenes, pero para guardar el código de una aplicación o de una página web montaremos el directorio.
- La razón para esto último es que tanto nuestro entorno de desarrollo como el contenedor tengan acceso a los archivos del código fuente. Los volúmenes, al contrario que los directorios montados, no deben accederse desde la máquina anfitrión.



# Persistencia de datos

- **Enlazar directorios**
- La opción --mount permite crear el enlace entre el directorio de la máquina virtual y el contenedor. La opción tiene tres argumentos separados por comas pero sin espacios:

type=bind,source=ORIGEN-EN-MÁQUINA-VIRTUAL,target=DESTINO-EN-CONTENEDOR. Ambos directorios deben existir previamente.

```
docker run -dit --name miapache-php -p 5555:80 --mount  
type=bind,source="$PWD/src",target=/var/www/html miapache-php
```

# Persistencia de datos

- **Volumenes**
- En vez de guardar los datos persistentes en la máquina host, Docker dispone de unos elementos llamados **volúmenes** que podemos asociar también a directorios del contenedor, de manera que cuando el contenedor lea o escriba en su directorio, donde leerá o escribirá será en el volumen.
- Los **volúmenes son independientes de los contenedores**, por lo que también podemos conservar los datos aunque se destruya el contenedor, reutilizarlos con otro contenedor, etc. La ventaja frente a los directorios enlazados es que pueden ser gestionados por Docker. Otro detalle importante es que el acceso al contenido de los volúmenes sólo se puede hacer a través de algún contenedor que utilice el volumen.
- Vamos a repetir un ejemplo similar al ejemplo anterior, pero utilizando un volumen en vez de un directorio enlazado. En este caso, enlazaremos el directorio /html con un volumen de Docker, es decir directorio de la máquina virtual, el contenedor servirá las páginas contenidas en el directorio de la máquina virtual.
- Podemos crear un volumen de la siguiente manera

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



# Persistencia de datos

- Crea un contenedor Apache.

La opción --mount permite crear el volumen o usar uno creado . La opción tiene tres argumentos separados por comas pero sin espacios: type=volume,source=NOMBRE-DEL-VOLUMEN,target=DESTINO-EN-CONTENEDOR. El directorio de destino debe existir previamente.

```
sudo docker run -dit -p 5555:80 --name miapache-php --mount type=volume,source=vol-miapache-php,target=/var/www/html miapache-php
```

- Compruebe que se ha creado un volumen con el nombre asignado al crear el contenedor

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	vol-miapache-php

- Los volúmenes son entidades independientes de los contenedores, pero para acceder al contenido del volumen hay que hacerlo a través contenedor, más exactamente a través del directorio indicado al crear el contenedor.

# Persistencia de datos

- Creamos ahora un nuevo contenedor que use el mismo volumen

```
sudo docker run -dit -p 6666:80 --name miapache-php-2 --mount type=volume,source=vol-miapache-php,target=/var/www/html miapache-php
```

- Vamos a modificar el html y meterlo en el contenedor primero

```
docker cp index.html miapache-php:/var/www/html
```

- Podemos ver que se ha modificado en los dos sitios (contenedores)



# Persistencia de datos

- También podemos crear volumens automáticos (no elegimos su nombre y se crean sobre la marcha) usando la opción -v y enlazarlos a un directorio nuestro, como ya hemos hecho antes en otros ejemplos.
- Ahora la información se guarda en un directorio local nuestro y todos los cambios se ven.

```
docker run -dit --name miapache-php -p 5555:80 -v "$PWD/src":/var/www/html  
miapache-php
```

# Persistencia de datos

- Los volúmenes son independientes de los contenedores, pero Docker tiene en cuenta qué volúmenes están siendo utilizados por un contenedor.
- Si intenta borrar el volumen del ejemplo anterior mientras los contenedores están en marcha, Docker muestra un mensaje de error que indica los contenedores afectados:

```
sudo docker volume rm vol-apache
```

Error response from daemon: remove vol-apache: volume is in use - [a6c8a30f7b1dc7a4ef165046daff226ee

1d6a69573269ca24d57b5b4b6802881, 3b1bcc5a67f38853810972b1da8a67148fad78c6cd6f22b2c823d141be59c81c]

Detenga los contenedores:

```
sudo docker stop apache-volume-1
```

```
sudo docker stop apache-volume.2
```



# Persistencia de datos

- Si intenta de nuevo borrar el volumen del ejemplo anterior ahora que los contenedores están detenidos, Docker sigue mostrando el mensaje de error que indica los contenedores afectados:

```
sudo docker volume rm vol-apache
```

```
Error response from daemon: remove vol-apache: volume is in use - [a6c8a30f7b1dc7a4ef165046daff226ee  
1d6a69573269ca24d57b5b4b6802881, 3b1bcc5a67f38853810972b1da8a67148fad78c6cd6f22b2c823d141be59c81c]
```

Borra los contenedores:

```
sudo docker rm apache-volume-1
```

```
sudo docker rm apache-volume.2
```



# Persistencia de datos

- Si intenta de nuevo borrar el volumen del ejemplo anterior ahora que no hay contenedores que utilicen el volumen, Docker ahora sí que borrará el volumen:

```
sudo docker volume rm vol-apache  
vol-apache
```

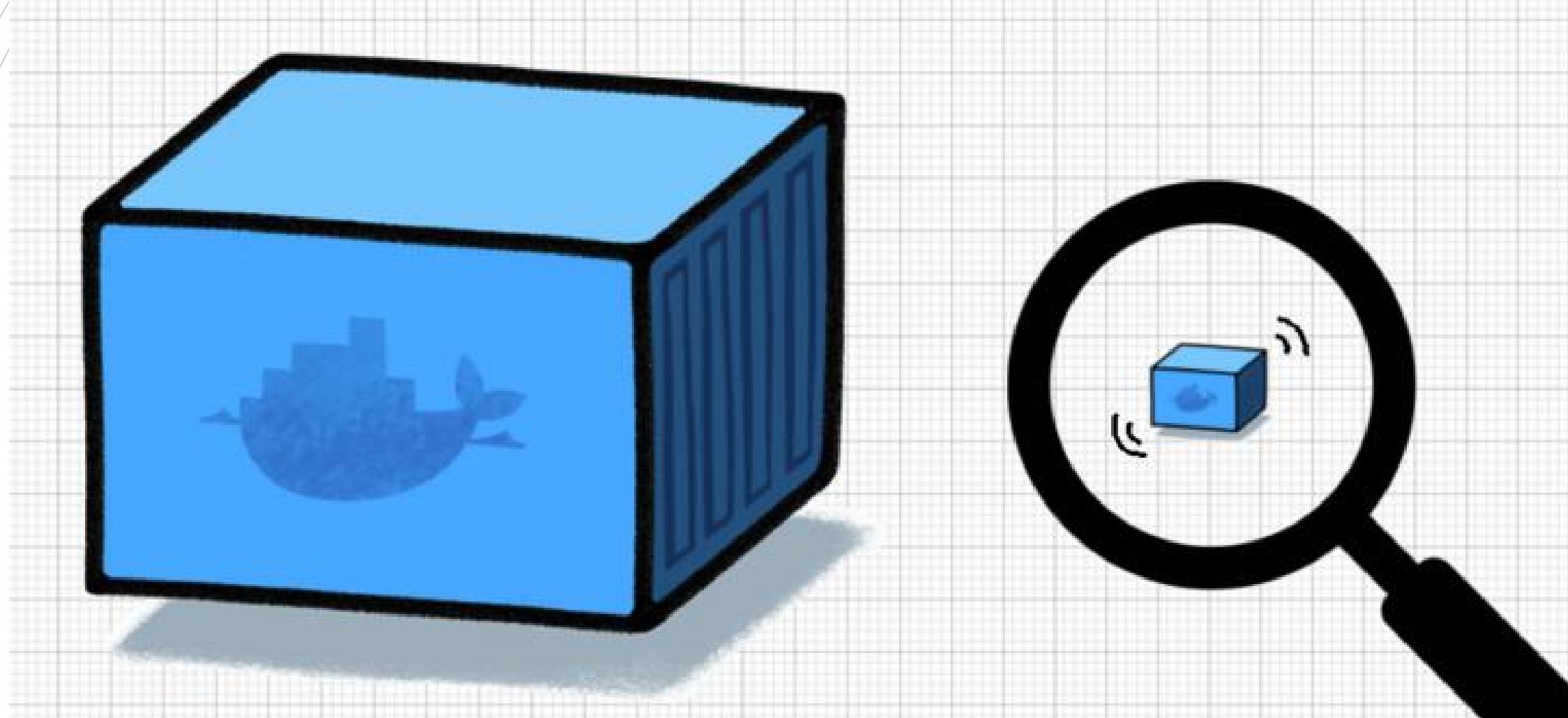
Comprueba que el volumen ya no existe:

```
docker volume ls
```

DRIVER	VOLUME NAME
--------	-------------

- Ten en cuenta que al borrar un volumen, los datos que contenía el volumen se pierden para siempre, salvo que hubiera realizado una copia de seguridad.

# Reto Wordpress



# Levantando un WordPress

Para crear un blog con WordPress necesitamos tener una base de datos dónde almacenar las entradas. Así que empezaremos creándola y después crearemos el contenedor de nuestro blog.

```
docker run -d --name wordpress-db \
  --mount source=wordpress-db,target=/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=wordpress \
  -e MYSQL_USER=manager \
  -e MYSQL_PASSWORD=secret \
  mariadb:10.5
```

- El principal cambio en docker run con respecto a la última vez es que no hemos usado -p (el parámetro para publicar puertos) y hemos añadido el parámetro -d.
- Lo primero que habremos notado es que el contenedor ya no se queda en primer plano. El parámetro -d indica que debe ejecutarse como un proceso en segundo plano. Así no podremos pararlo por accidente con Control+C.



# Levantando un Wordpress

- Lo segundo es que vemos que el contenedor usa un puerto, el 3306/tcp, pero no está linkado a la máquina anfitrión. No tenemos forma de acceder a la base de datos directamente. Nuestra intención es que solo el contenedor de WordPress pueda acceder.
- Luego una serie de parámetros -e que nos permite configurar nuestra base de datos.
- Por último, el parámetro --mount nos permite enlazar el volumen que creamos en el paso anterior con el directorio /var/lib/mysql del contenedor. Ese directorio es donde se guardan los datos de MariaDB. Eso significa que si borramos el contenedor, o actualizamos el contenedor a una nueva versión, no perderemos los datos porque ya no se encuentran en él, si no en el volumen. Solo lo perderíamos si borramos explícitamente el volumen.

# Levantando un Wordpress

- Creamos el directorio donde queremos almacenar nuestro WordPress

```
mkdir -p ~/Sites/wordpress && cd ~/Sites/wordpress
```

- Y dentro de este directorio arrancamos el contenedor:

```
docker run -d --name wordpress \
--link wordpress-db:mysql \
--mount type=bind,source="$(pwd)"/wordpress,target=/var/www/html \
-e WORDPRESS_DB_USER=manager \
-e WORDPRESS_DB_PASSWORD=secret \
-p 8080:80 \
wordpress:4.9.8
```

- Cuando termine la ejecución, si accedemos a la dirección <http://localhost:8080/>, ahora sí podremos acabar el proceso de instalación de nuestro WordPress. Si listamos el directorio `wordpress` comprobaremos que tenemos todos los archivos de instalación accesibles desde el directorio anfitrión.



# Levantando un WordPress

- Ejercicios:

Para los contenedores, tanto el de WordPress como el MariaDB.

Borra ambos.

Vuelve a crearlos y mira como ya no es necesario volver a instalar WordPress.

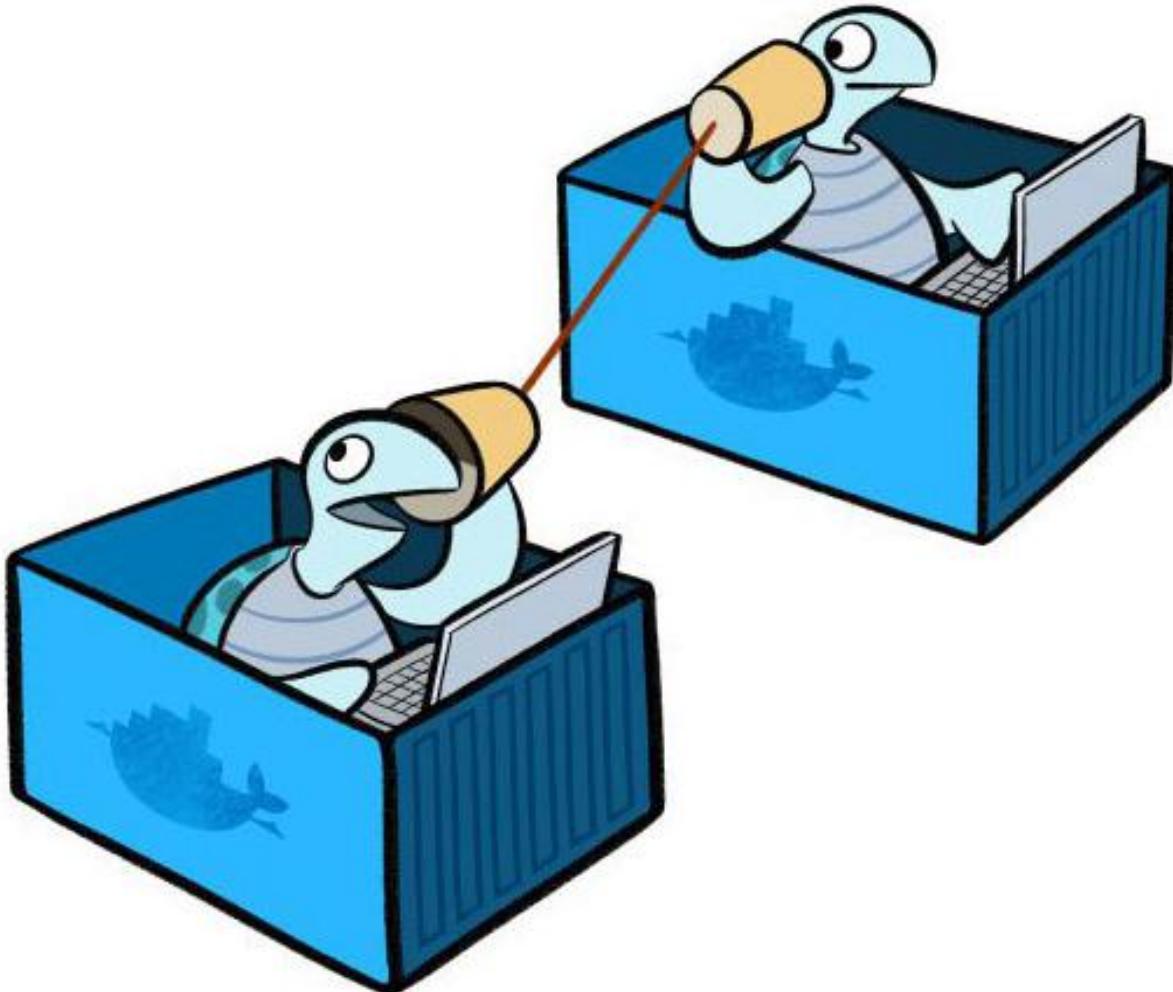
Vuelve a borrarlos y borra también el volumen.

Vuelve a crear el volumen y los contenedores y comprueba que ahora sí hay que volver a instalar WordPress.

Explica por qué



# Enlazando contenedores



# Enlazando contenedores

- Docker nos permite distintos mecanismos para enlazar contenedores, es decir que exista un mecanismo de resolución de nombres que permita acceder a los distintos contenedores por medio de su nombre.
- Por defecto los contenedores que creamos se conectan a la red de tipo bridge llamada bridge (por defecto el direccionamiento de esta red es 172.17.0.0/16). Los contenedores conectados a esta red que quieren exponer algún puerto al exterior tienen que usar la opción -p para mapear puertos.
- Si conecto un contenedor a la red host, el contenedor estaría en la misma red que el host (por lo tanto toma direccionamiento del servidor DHCP de nuestra red).
- Además los puertos son accesibles directamente desde el host.



# Enlazando contenedores

- Nosotros podemos crear nuevas redes (redes definidas por el usuario), por ejemplo para crear una red de tipo bridge:

```
docker network create mired
```

- Y para ver las características de esta nueva red, podemos ejecutar:

```
docker network inspect mired
```

- Para crear un contenedor en esta red, ejecutamos:

```
docker run -d --name mi_apache --network mired -p 80:80 httpd
```

- Dependiendo de la red que estemos usando (la red puente por defecto o una red definida por el usuario) el mecanismo de enlace entre contenedores será distinto.

# Enlazando contenedores

- Nosotros podemos crear nuevas redes (redes definidas por el usuario), por ejemplo para crear una red de tipo bridge:

```
docker network create mired
```

- Y para ver las características de esta nueva red, podemos ejecutar:

```
docker network inspect mired
```

- Para crear un contenedor en esta red, ejecutamos:

```
docker run -d --name mi_apache --network mired -p 80:80 httpd
```

- Dependiendo de la red que estemos usando (la red puente por defecto o una red definida por el usuario) el mecanismo de enlace entre contenedores será distinto.



# Enlazando contenedores

- Creamos los contenedores asignando la nueva red

```
docker run -d --name wordpress-db \
--network mired
--mount source=wordpress-db,target=/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=secret \
-e MYSQL_DATABASE=wordpress \
-e MYSQL_USER=manager \
-e MYSQL_PASSWORD=secret mariadb:10.3.9
```

```
docker run -d --name wordpress \
--network mired
--link wordpress-db:mysql \
--mount type=bind,source="$(pwd)"/wordpress,target=/var/www/html \
-e WORDPRESS_DB_USER=manager \
-e WORDPRESS_DB_PASSWORD=secret \
-p 8080:80 \
wordpress:4.9.8
```

# Enlazando contenedores

- **Enlazando contenedores conectados a la red bridge por defecto**
- Esta manera en enlazar contenedores no está recomendada y esta obsoleta. Además el uso de contenedores conectados a la red por defecto no está recomendado en entornos de producción. Para realizar este tipo de enlace vamos a usar el flag --link:
- Veamos un ejemplo, primero creamos un contenedor de mariadb:

```
docker run -d --name servidor_mariadb -e MYSQL_DATABASE=bd_wp -e MYSQL_USER=user_wp -e  
MYSQL_PASSWORD=asdasd -e MYSQL_ROOT_PASSWORD=asdasd mariadb
```

- A continuación vamos a crear un nuevo contenedor, enlazado con el contenedor anterior:  

```
docker run -d --name servidor --link servidor_mariadb:mariadb httpd
```
- Para realizar la asociación entre contenedores hemos utilizado el parámetro --link, donde se indica el nombre del contenedor enlazado y un alias por el que nos podemos referir a él.

# Enlazando contenedores

- En este tipo de enlace tenemos dos características:
  - Se comparten las variables de entorno. Las variables de entorno del primer contenedor son accesibles desde el segundo contenedor.
  - Los contenedores son conocido por resolución estática.
- Otro mecanismo que se realiza para permitir la comunicación entre contenedores asociados es modificar el fichero /etc/hosts para que tengamos resolución estática entre ellos.



# Enlazando contenedores

- **Enlazando contenedores conectados a la red bridge por defecto**

- Este caso vamos a definir una red de tipo bridge:

```
docker network create mired
```

- Y creamos los contenedores conectados a dicha red:

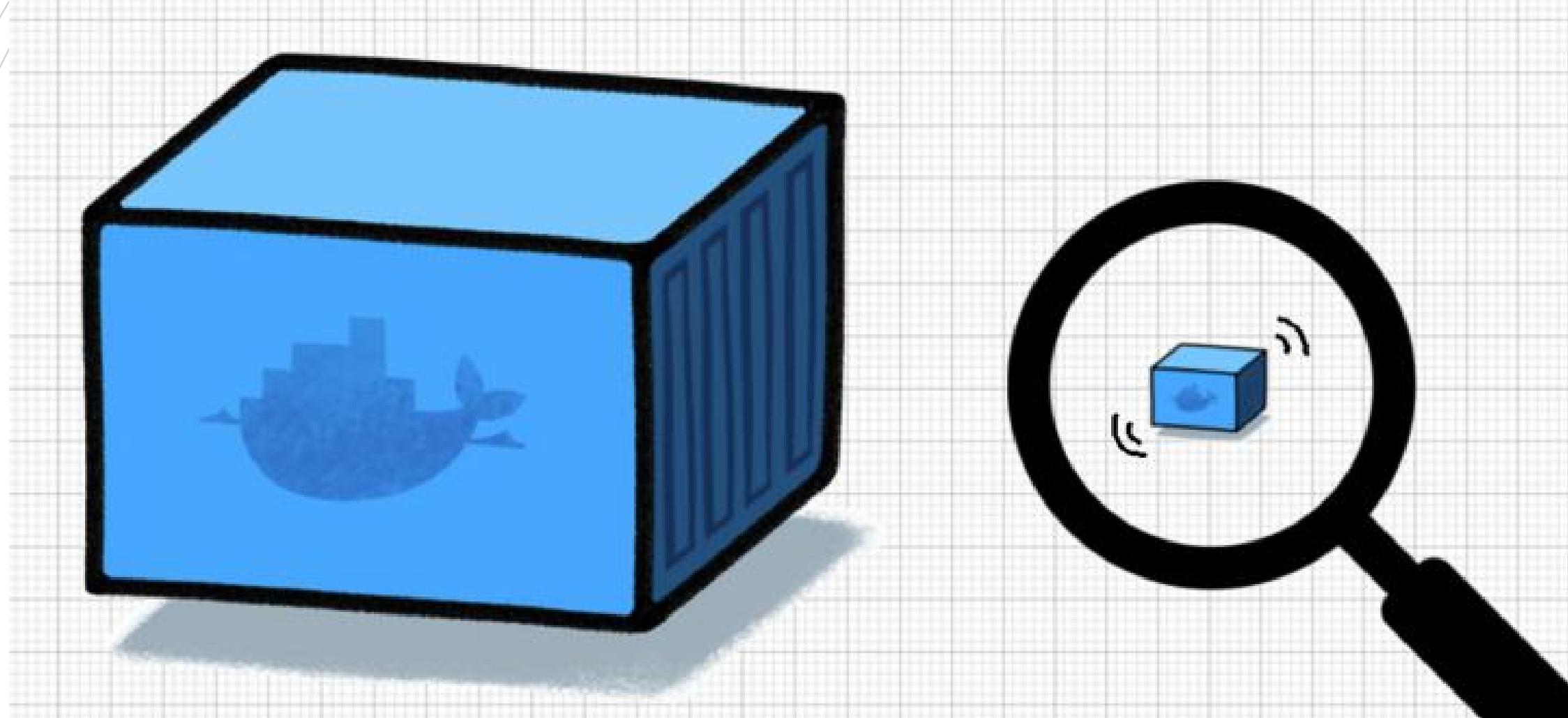
```
docker run -d --name servidor_mariadb --network mired -e MYSQL_DATABASE=bd_wp -e  
MYSQL_USER=user_wp -e MYSQL_PASSWORD=asdasd -e MYSQL_ROOT_PASSWORD=asdasd mariadb
```

```
docker run -d --name servidor --network mired httpd
```

En este caso no se comparten las variables de entorno, y la resolución de nombres de los contenedores se hace mediante un servidor dns que se ha creado en el gateway de la red que hemos creado:



# Reto Wordpress



# Reto Wordpress

- Creamos una red de tipo bridge:

```
docker network create red_wp
```

- Creamos un contenedor desde la imagen mariadb con el nombre servidor\_mysql, conectada a la red creada:

```
docker run -d --name servidor_mariadb --network red_wp -e MYSQL_DATABASE=bd_wp -e  
MYSQL_USER=user_wp -e MYSQL_PASSWORD=asdasd -e MYSQL_ROOT_PASSWORD=asdasd mariadb
```

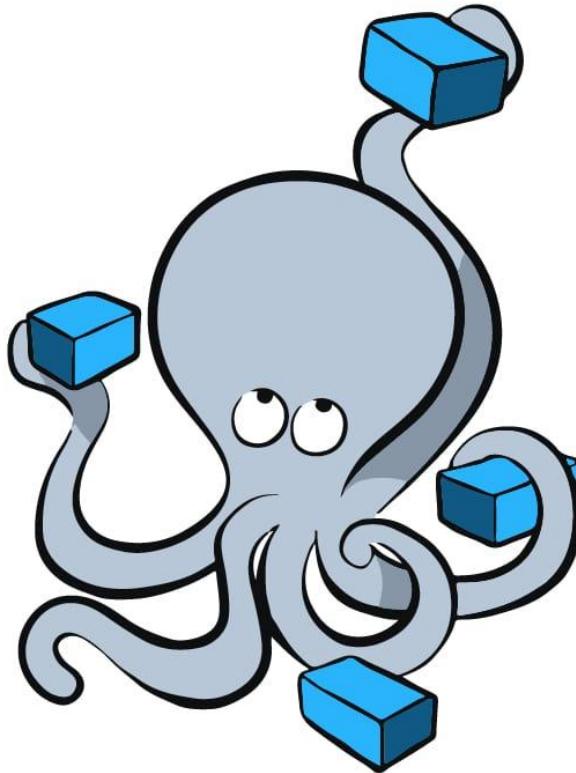
- A continuación vamos a crear un nuevo contenedor, con el nombre servidor\_wp, con el servidor web a partir de la imagen wordpress, conectada a la misma red y con las variables de entorno necesarias:

```
docker run -d --name servidor_wp --network red_wp -e WORDPRESS_DB_HOST=servidor_mariadb -e  
WORDPRESS_DB_USER=user_wp -e WORDPRESS_DB_PASSWORD=asdasd -e WORDPRESS_DB_NAME=bd_wp -p 80:80  
wordpress
```

- Accede a la ip del servidor docker y comprueba la instalación de wordpress.



# Docker Compose



docker  
Compose

# Docker Compose

- El cliente de Docker es engorroso para crear contenedores, así como para crear el resto de objetos y vincularlos entre sí a base de Dockerfiles.
- Para automatizar la creación, inicio y parada de un contenedor o un conjunto de ellos, Docker proporciona una herramienta llamada **Docker Compose**.
- Compose es una herramienta para definir y ejecutar aplicaciones multi-contenedor. Con un solo comando podremos crear e iniciar todos los servicios que necesitamos para nuestra aplicación.
- Los casos de uso más habituales para docker-compose son:
  - Entornos de desarrollo
  - Entornos de testeo automáticos (integración continua)
  - Despliegue en host individuales (no clusters)
- Compose tiene comandos para manejar todo el ciclo de vida de nuestra aplicación:
  - Iniciar, detener y rehacer servicios.
  - Ver el estado de los servicios.
  - Visualizar los logs.
  - Ejecutar un comando en un servicio.



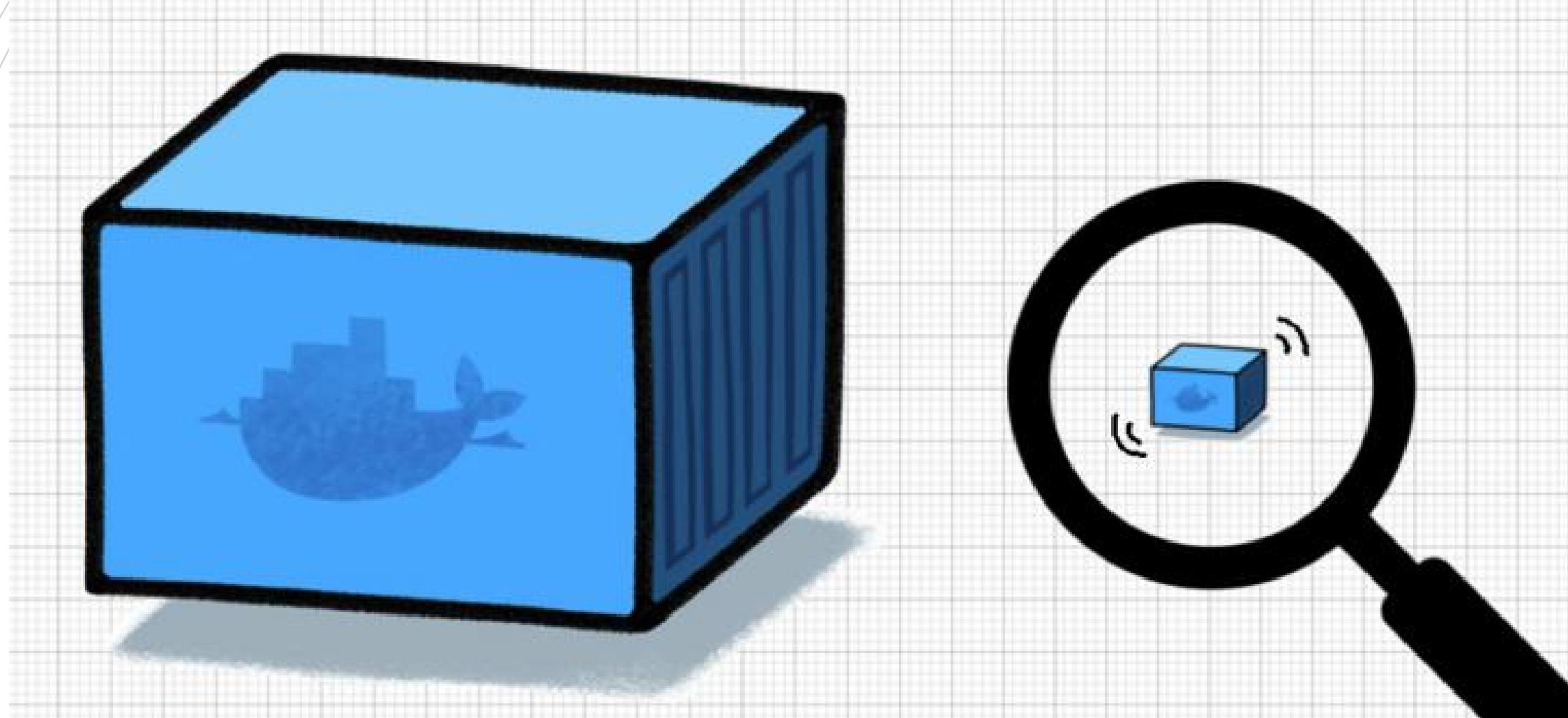
# Docker Compose

- Un docker-compose.yml muy básico tiene este aspecto:

```
version: '2'  
services:  
  hello_world:  
    image: ubuntu  
    command: [/bin/echo, 'Hola Docker Compose']
```



# Reto Wordpress



# Docker Compose

- Vamos a crear el fichero docker-compose.yaml para Wordpress y MariaDB

```
version: '3'

services:
  db:
    image: mariadb:10.5
    container_name: mariadb
    volumes:
      - data:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=secret
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=manager
      - MYSQL_PASSWORD=secret
  web:
    image: wordpress:4.9.8
    container_name: wordpress
    depends_on:
      - db
    volumes:
      - ./wordpress:/var/www/html
    environment:
      - WORDPRESS_DB_USER=manager
      - WORDPRESS_DB_PASSWORD=secret
      - WORDPRESS_DB_HOST=db
    ports:
      - 8080:80
  volumes:
    data:
```

# Docker Compose

- Iniciar servicios. Vamos a ejecutar esta aplicación y luego procederemos a explicarla:

```
docker-compose up -d
```

- `docker-compose ps` solo muestra información de los servicios que se define en `docker-compose.yaml`, mientras que `docker` muestra todos.

- El nombre de cada contenedor se define con `container_name: mariadb`, si no se pone será el nombre del directorio y de la imagen

- Detener servicios

```
docker-compose stop
```

- Borrar servicios

```
docker-compose down
```

- Esto borra los contenedores, pero no los volúmenes. Así que si hemos creado bien la aplicación nuestros datos están a salvo.

- Si queremos borrar también los volúmenes:

```
docker-compose down -v
```



# Docker Compose

- Estructura de la configuración. Veamos la configuración por partes:

`version: '3'`. Compose se actualiza a menudo, con lo que el archivo de configuración va adquiriendo nuevas funcionalidades. La versión '3' (es una cadena, importante poner comillas) es la última y para conocer todas sus características mira la página de referencia de la versión 3 de Compose.

- `volumes: data`:

Ya hemos indicado que es importante guardar los datos volátiles de las aplicaciones en volúmenes. En este caso hemos creado un volumen llamado `data`. Recordemos que Compose siempre añade como prefijo el nombre del directorio, con lo que el nombre real del volumen es `wordpress_data`. Podemos comprobarlo con el cliente de docker como hicimos en el capítulo de volúmenes:

- Nos saltamos la sección de redes (`networks`) y vamos a la sección de servicios, que son los contenedores que precisa o componen nuestra aplicación

# Docker Compose

- **Primero la base de datos:**

```
services:  
  db:  
    image: mariadb:10.5  
    volumes:  
      - data:/var/lib/mysql  
    environment:  
      - MYSQL_ROOT_PASSWORD=secret  
      - MYSQL_DATABASE=wordpress  
      - MYSQL_USER=manager  
      - MYSQL_PASSWORD=secret
```

- Despues de abrir la parte de servicios, el primer nivel indica el nombre del servicio db, que genera el contenedor wordpress\_db. Lo que vemos a continuación es lo mismo que hicimos en la sección anterior pero de forma parametrizada. Si recordamos, para levantar nuestra base de datos, indicamos la imagen (línea 3), luego montamos los volúmenes (línea 4), y después indicamos las variables de entorno que configuraban el contenedor (línea 6).
- Es decir, lo anterior es equivalente, excepto por el nombre, a:

```
$ docker run -d --name wordpress-db \  
  --mount source=wordpress-db,target=/var/lib/mysql \  
  -e MYSQL_ROOT_PASSWORD=secret \  
  -e MYSQL_DATABASE=wordpress \  
  -e MYSQL_USER=manager \  
  -e MYSQL_PASSWORD=secret mariadb:10.5
```



# Docker Compose

- Y después nuestro WordPress:

```
services:  
  web:  
    image: wordpress:4.9.8  
    depends_on:  
      - db  
    volumes:  
      - ./target:/var/www/html  
    environment:  
      - WORDPRESS_DB_USER=manager  
      - WORDPRESS_DB_PASSWORD=secret  
      - WORDPRESS_DB_HOST=db  
    ports:  
      - 8080:80
```

- En este caso la equivalencia es al comando:

```
$ docker run -d --name wordpress \  
  --link wordpress-db:mysql \  
  --mount type=bind,source="$(pwd)"/wordpress,target=/var/www/html \  
  -e WORDPRESS_DB_USER=manager \  
  -e WORDPRESS_DB_PASSWORD=secret \  
  -p 8080:80 \  
  wordpress:4.9.8
```



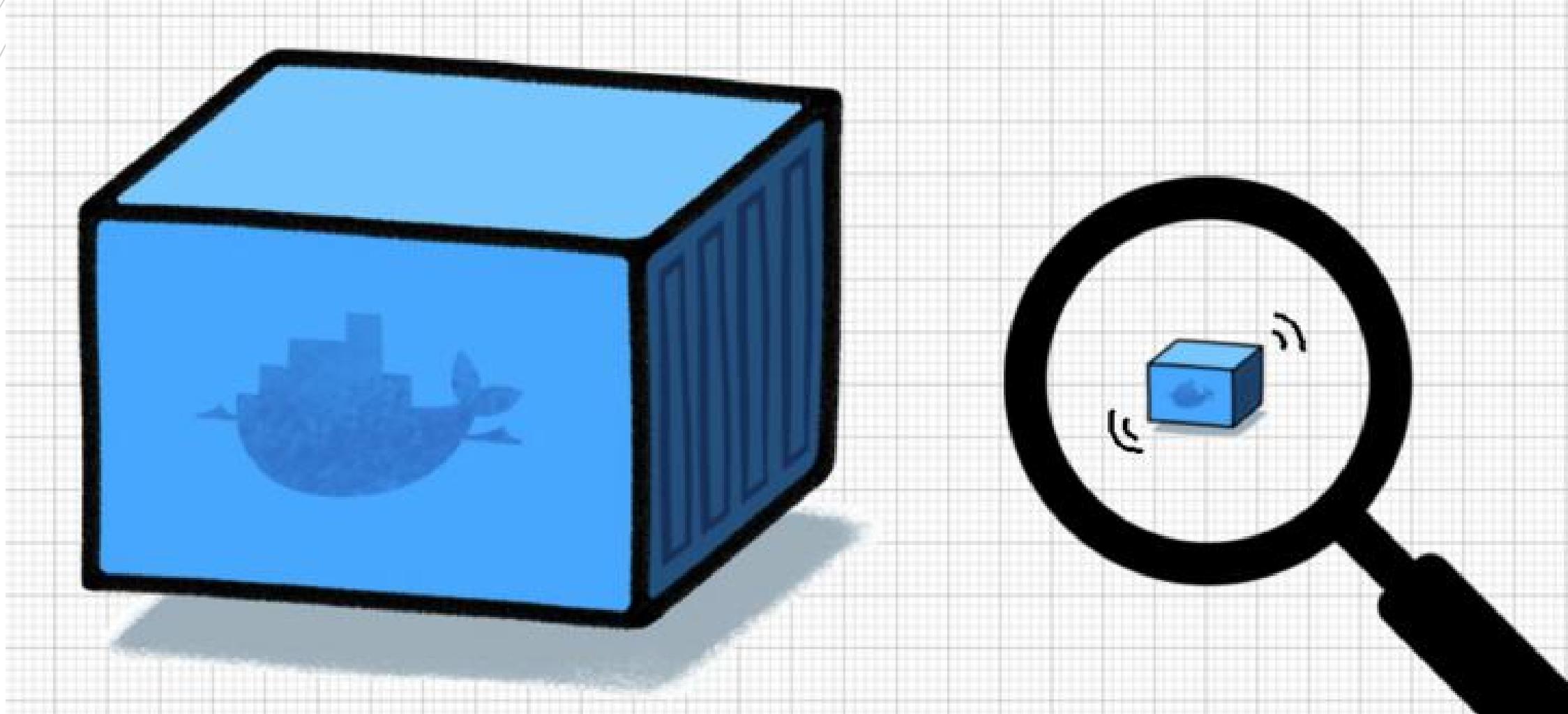
# Docker Compose

- La equivalencia de los parámetros es la siguiente:

<b>parámetro Docker</b>	<b>parámetro Composer</b>
--link	depends_on
--mount	volumes
-e	environment
-p, --publish	ports
image	From



# Reto LAMP



# Docker LAMP

- Apache con PHP

```
# Indicamos la versión
version: '3.7'

# Iniciamos los servicios
services:
  # Apache con PHP
  php-httpd:
    image: php:7.3-apache
    container_name: apache-php
    ports:
      - 80:80
    volumes:
      - './src:/var/www/html'
```



# Docker LAMP

## ■ MariaDB

```
mariadb:  
  image: mariadb:10.5.2  
  container_name: mariadb-lamp  
  volumes:  
    - mariadb-volume:/var/lib/mysql  
  environment:  
    TZ: 'Europe/Rome'  
    MYSQL_ALLOW_EMPTY_PASSWORD: 'no'  
    MYSQL_ROOT_PASSWORD: 'rootpwd'  
    MYSQL_USER: 'testuser'  
    MYSQL_PASSWORD: 'testpassword'  
    MYSQL_DATABASE: 'testdb'  
  
volumes:  
  mariadb-volume:
```



# Docker LAMP

- Variables a tener en cuenta

TZ: Zona para las fechas

MYSQL\_ALLOW\_EMPTY\_PASSWORD: Habilita el uso de password en blanco para root

MYSQL\_ROOT\_PASSWORD: Campo obligatorio, es password de root

MYSQL\_DATABASE: Opcional, es el nombre de la base de datos inicial

MYSQL\_USER: Opcional, es el nombre del usuario que nos crearemos para la base de datos

MYSQL\_PASSWORD: Es el password de MYSQL\_USER



# Docker LAMP

- PHP MyAdmin

```
phpmyadmin:  
  image: phpmyadmin/phpmyadmin  
  container_name: phpmyadmin-lamp  
  links:  
    - 'mariadb:db'  
  ports:  
    - 8081:80
```

Usamos links para segurarnos que se enlaza en la misma red con MariaDB y comparte las variables de entorno (usuarios y contraseña)

# Docker Compose y Docker Files

- Podemos personalizar aún mas usando dockerfiles con docker compose, usando build, en vez de image y nuestra red propia aislandolos del resto.
- Apache

```
# Indicamos la versión
version: '3.7'
# Iniciamos los servicios
services:
    # Apache con PHP
    apache-php:
        build: ./apache-php
        container_name: apache-php-lamp
        ports:
            - 80:80
        volumes:
            - './src:/var/www/html'
        networks:
            - lamp-network
```



# Docker Compose y Docker Files

- En el directorio apache-php tenemos nuestro dockfile

```
# Apache + PHP + complementos
FROM php:7.4-apache
# Complementos, es importante que no nos pasemos con el run!! (cuanto más lineas peor, más lento)
RUN docker-php-ext-install pdo pdo_mysql mysqli
```

# Docker Compose y Docker Files

## ■ MariaDB

```
# Maria DB
mariadb:
  build: ./mariadb
  container_name: mariadb-lamp
  volumes:
    # - ./mariadb_data:/var/lib/mysql podríamos usar un directorio local llamado mariadb_data y no volumen
    - mariadb-volume:/var/lib/mysql
  networks:
    - lamp-network
```



# Docker Compose y Docker Files

- En el directorio mariadb tenemos el Dockerfile y un fichero sql que se ejecuta al iniciar por primera vez la BD creando las tablas y los datos que queramos que tenga

```
## MariaDB
FROM mariadb:10.5
# Configuramos BBDD
ENV MYSQL_ROOT_PASSWORD 123
ENV MYSQL_USER joseluis
ENV MYSQL_PASSWORD 123
ENV MYSQL_DATABASE testdb
# Copiamos los ficheros sql para que se ejecuten
COPY ./sql /docker-entrypoint-initdb.d/
```



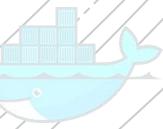
# Docker Compose y Docker Files

- En el directorio mariadb/sql, es el fichero para inicializar nuestra BD. Mantén el nombre a init-db-sql

```
USE testdb;

CREATE TABLE test (
    nombre varchar(30),
    email varchar(50)
);

INSERT INTO test (nombre, email)
VALUES
    ('Jose Luis', 'joseluis@docker.com'),
    ('Soraya', 'soraya@docker.com'),
    ('Victor', 'victor@docker.com');
```



# Docker Compose y Docker Files

- PHPMyAdmin, Volumen y Redes (para que estén aisladas del resto de contenedores o solo comparten ellas)

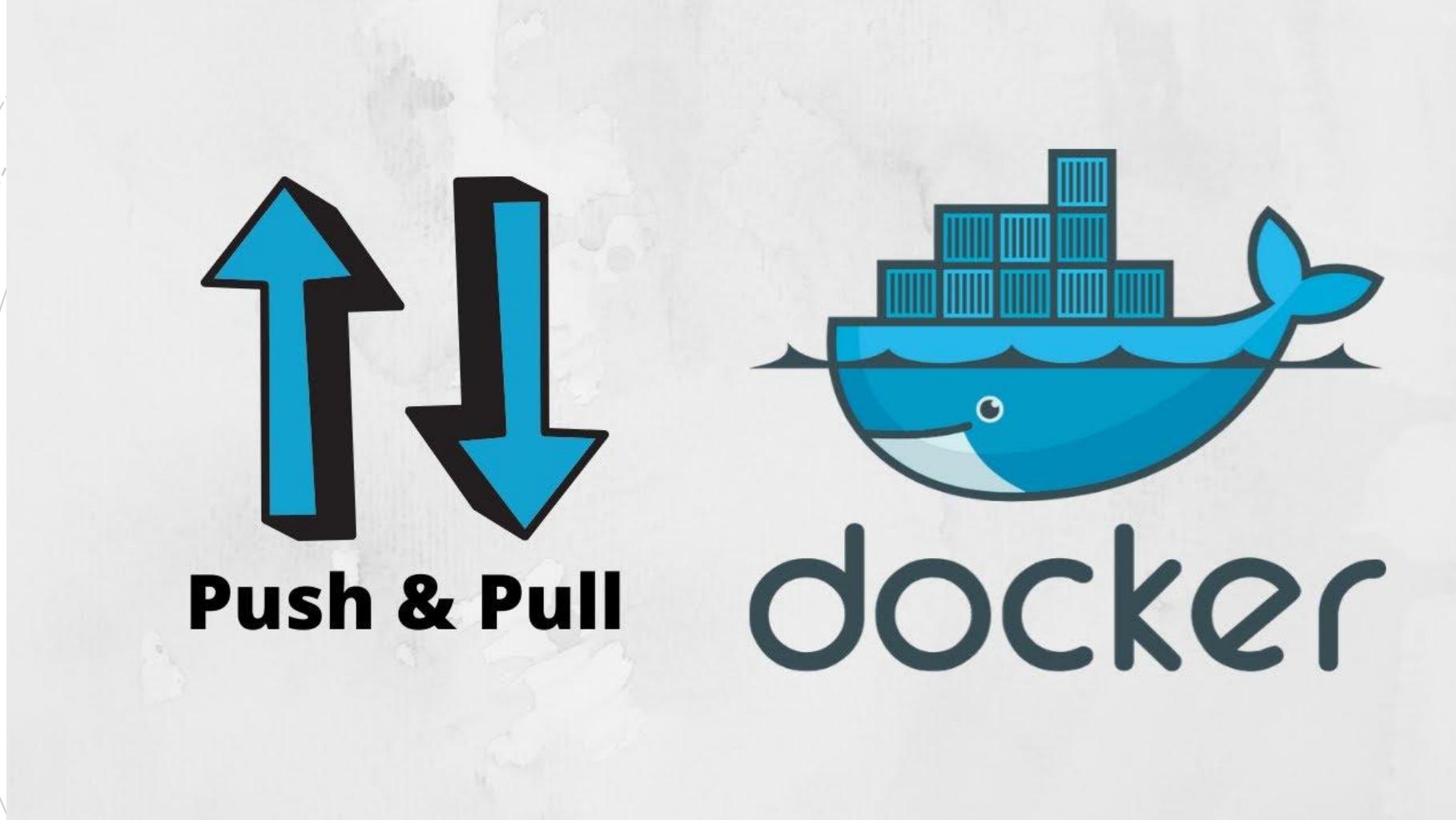
```
# PHPMyAdmin
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  container_name: phpmyadmin-lamp
  links:
    - 'mariadb:db'
  ports:
    - 8081:80
  networks:
    - lamp-network

volumes:
  mariadb-volume:

networks:
  lamp-network:
    driver: bridge
```

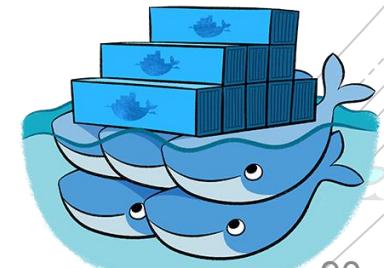


# Publicar imágenes a Docker Hub

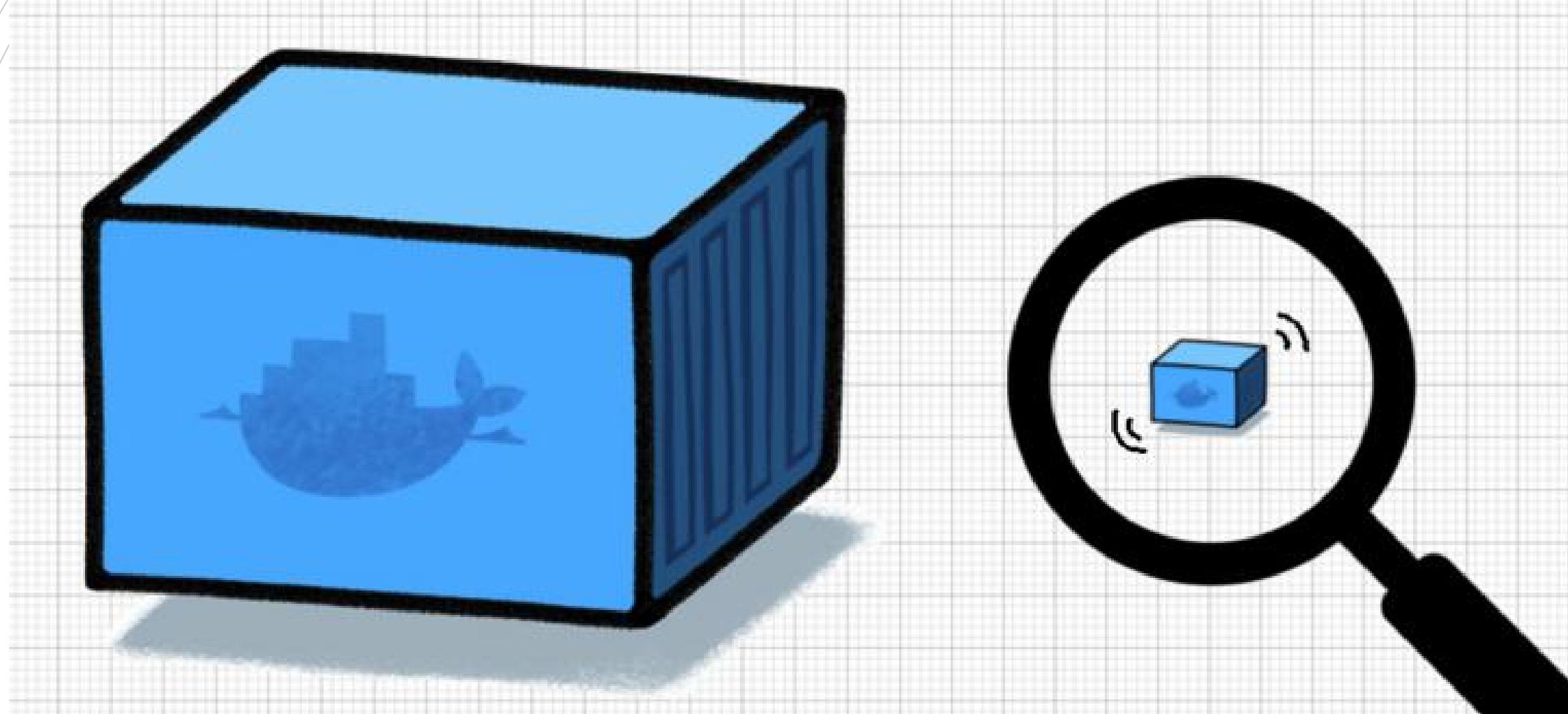


# Publicar imágenes a Docker Hub

- Docker Hub es el repositorio o registro donde podemos bajar una imagen o subir nuestras propias imágenes de contenedores basadas en nuestros Dockfiles. De esta manera podemos compartir nuestras imágenes y configuraciones de la misma manera que con GitHub podemos compartir nuestro proyecto.
- Bajarse imágenes es gratis, y podemos bajarnos una imagen para partir de ella para crear la nuestra con nuestro entorno de trabajo y compartirlo
- El primer paso es crearte una cuenta.
- Las imágenes que se ueden bajar usando `docker pull nombre_imagen`
- Las imágenes se deben subuir con `docker push nombre_imagen`
- Antes de subirla debes preparar tu imagen para que sea aceptada en este registro público. Todos los registros siguen una nomenclatura a la hora de almacenar los repositorios. En el caso de Docker Hub necesitamos que nuestra imagen se llame `nombre_de_usuario/nombre_del_repositorio:etiqueta`. Utiliza `docker tag` para generar una variante de tu imagen con ese nombre.



# Reto Docker Hub



# Apache en Docker Hub

- Vamos a subir un apache personalizado con una web que diga Hola Docker a Docker Hub.
- Partimos del siguiente Docker File

```
# Imagen a usar
FROM php:7.0-apache
# copiamos todos los ficheros en el directorio en cuestion
COPY src/ /var/www/html
# Exponemos el puerto 80
EXPOSE 80
# Quien lo ha realizado
MAINTAINER JL Gonzalez "jlgs@cifpvirgendegracia.com"
```

- Creamos nuestra imagen ya sea ejecutando `docker run` y echando andar el contenedor (crea la imagen antes) o si no usamos:  
`docker build .` o `docker build --tag=apache-php .`
- Comprobamos que tenemos nuestra imagen ya creada con  
`docker images`



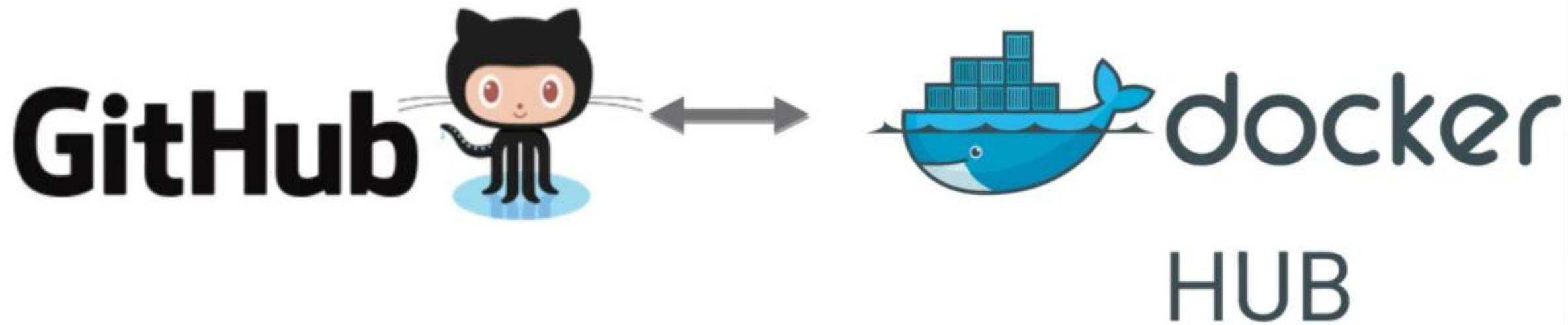
# Apache en Docker Hub

- Nos loguamos con nuestro nombre de usuario de Docker Hub en el terminal  
`docker login o docker login --username=yourhubusername --email=youremail@company.com`
- Las imágenes deben subirse con el formato nombre\_de\_usuario/nombre\_del\_repositorio:etiqueta. Etiquetamos si no lo hemos hecho antes nuestra imagen para que se suba así  
`docker tag apache-php joseluisgs/apache-php:v1`  
Podemos ver que se ha hecho con docker images  
`docker images`

REPOSITORY	TAG	IMAGE ID
joseluisgs/apache-php	v1	4172c72754a1
- Finalmente subimos la imagen  
`docker push joseluisgs/apache-php:v1`
- Nuestra imagen estará disponible en Docker Hub:  
<https://hub.docker.com/r/joseluisgs/apache-php>
- Podemos descargarla con:  
`docker pull joseluisgs/apache-php`

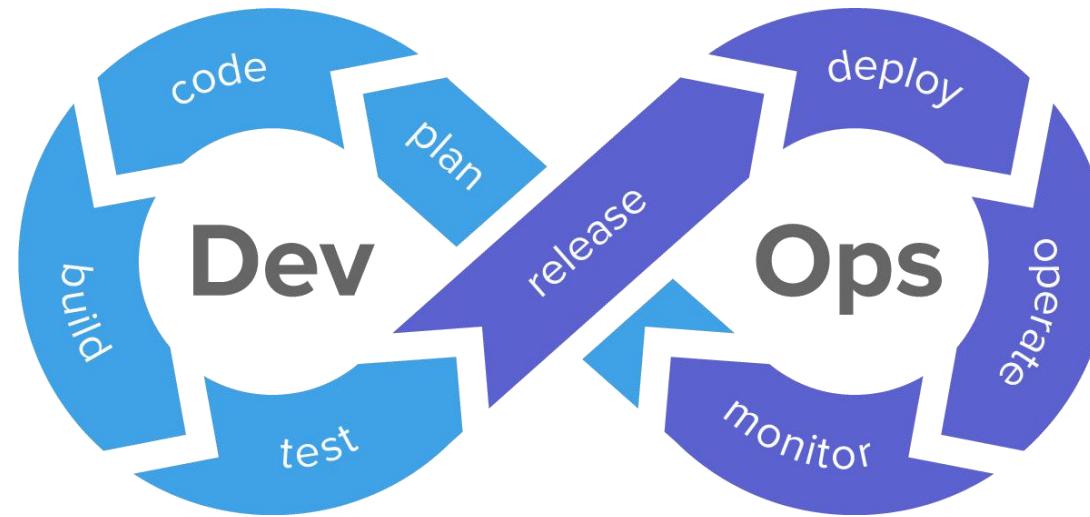


# Despliegue con Docker Hub y GitHub



# Despliegue con Docker Hub y GitHub

- Una de las grandes ventajas que tiene el uso de Docker es poder ser la base del DevOps, que permite que los desarrolladores puedan enfocarse sólo en desarrollar y puedan desplegar su código en segundos.



- Para ello vamos a ver cómo Docker/Docker Hub permite el despliegue continuo de nuestro proyecto basado en repositorios de GitHub



# Despliegue con Docker Hub y GitHub

- Imaginemos que tenemos un proyecto en GitHub, por ejemplo:  
<https://github.com/joseluisgs/docker-apache-php>
- Vamos a Docker Hub y pulsamos crear repositorio



- Creamos el repositorio y lo enlazamos a nuestro repositorio GitHub. Podemos elegir las reglas del despliegue, como que sea cuando haya cambios en una rama, o creamos una etiqueta
- Lo único que necesitamos es que en nuestro repositorio en el directorio raíz de la rama en cuestión tengamos nuestro Dockerfile
- En futuros tutoriales veremos como hacerlo automatizado sin salir de GitHub, con GitHub Actions

# Despliegue con Docker Hub y GitHub

The screenshot shows the Docker Hub interface for creating a new repository. The top navigation bar includes links for Explore, Repositories, Organizations, Get Help, and a user profile for 'joseluisgs'. A search bar at the top right says 'Search for great content (e.g., mysql)'. Below the search bar, there are buttons for 'Repositories' and 'Create'. The main section is titled 'Create Repository' and shows a dropdown for the owner 'joseluisgs' and a text input for the repository name 'docker-apache-php'. A 'Description' field is present but empty. A 'Pro tip' box contains the command: `docker tag local-image:tagname new-repo:tagname` and `docker push new-repo:tagname`. It also advises changing 'tagname' to a desired image repository tag. Under 'Visibility', the 'Public' option is selected, with a note that public repositories appear in Docker Hub search results. The 'Private' option is also shown. In the 'Build Settings (optional)' section, there is a GitHub icon labeled 'Connected' and a Docker icon labeled 'Disconnected'. Below this, there are dropdown menus for the repository owner ('joseluisgs') and name ('docker-apache-php'). At the bottom left, there is a 'BUILD RULES' button with a plus sign.

# Despliegue con Docker Hub y GitHub

- Ahora siempre que hagamos un commit a la rama indicada o creamos una tag nueva (si así le hemos puesto de regla) se genera automáticamente la imagen del docker en nuestro registro y la podemos pasar a todo el mundo que queramos. De esta manera tenemos un despliegue basado en Docker automatizado con los últimos cambios de nuestro proyecto.

The screenshot shows the Docker Hub interface for the repository 'joseluisgs/docker-apache-php'. At the top, there's a message about rate limits. Below it, the repository details are shown: 'Using 0 of 1 private repositories.' The 'Builds' tab is selected, showing one successful build:

- SUCCESS**
- NAME**: Build in 'main' (32155a05)
- TAG**: latest
- SOURCE**: joseluisgs/docker-apache-php
- CREATED**: 11 minutes ago
- DURATION**: 1 min
- USER**: joseluisgs

Below the build details, there are tabs for **BUILD LOGS**, **DOCKERFILE** (which is selected), and **README**. The **DOCKERFILE** tab displays the following Dockerfile content:

```
# Imagen a usar
FROM php:7.0-apache
# copiamos todos los ficheros en el directorio en cuestion
COPY src/ /var/www/html
# Exponemos el puerto 80
EXPOSE 80
# Quien lo ha realizado
MAINTAINER JL Gonzalez "jlgs@cifpvirgendegracia.com"
```

The screenshot shows the GitHub repository page for 'joseluisgs/docker-apache-php'. The repository name is 'joseluisgs / docker-apache-php'. The repository description is empty. It was last pushed 8 minutes ago. The Docker commands section contains the command 'docker push joseluisgs/docker-apache-php:tagname'. The repository has 1 tag: 'latest'. Vulnerability scanning is disabled. Recent builds show two entries: 'Build in 'main'' (24ab0149) and 'Build in 'main'' (32155a05). The 'DOCKER Apache-PHP' section contains a README file with a Dockerfile snippet, information about the tutorial, and author details for José Luis González Sánchez (@joseluisgsonsan).

# Despliegue con Docker Hub y GitHub

The screenshot shows the Docker Hub interface for a repository named 'joseluisgs/docker-apache-php'. The repository has a blue hexagonal icon and a star rating. It was created by 'joseluisgs' and updated 9 minutes ago. The 'Container' tab is selected. Below the main title, there are tabs for 'Overview', 'Tags', 'Dockerfile', and 'Builds'. The 'Overview' section contains a large image of a blue cube labeled 'DOCKER Apache-PHP', a brief description in Spanish, and a 'last commit' button. The 'Autor' section lists the author's name and links to their Twitter and GitHub profiles. To the right, there are sections for 'Docker Pull Command' (with a terminal command line), 'Owner' (with a profile picture of 'joseluisgs'), and 'Source Repository' (linking back to the GitHub repository). A decorative illustration of a blue whale carrying a stack of shipping containers is visible in the bottom right corner.

Search for great content (e.g., mysql)

Explore Repositories Organizations Get Help joseluisgs

Using 0 of 1 private repositories. [Get more](#)

**joseluisgs/docker-apache-php** ☆

By [joseluisgs](#) • Updated 9 minutes ago

Container

Manage Repository

Pulls 3

**Overview** Tags Dockerfile Builds

## DOCKER Apache-PHP

2DAW Docker Apache-PHP, un ejemplo práctico basado en el tutorial de docker

last commit today

### Acerca de

Tutorial de Docker de supervivencia a 2DAW. Docker Apache-PHP, un ejemplo práctico basado en el tutorial de Docker realizado.

### Autor

- José Luis González Sánchez [Follow @joseluisgsonsa](#) 553
- [GitHub](#)  Followers 45

### Docker Pull Command

```
docker pull joseluisgs/docker-apache
```

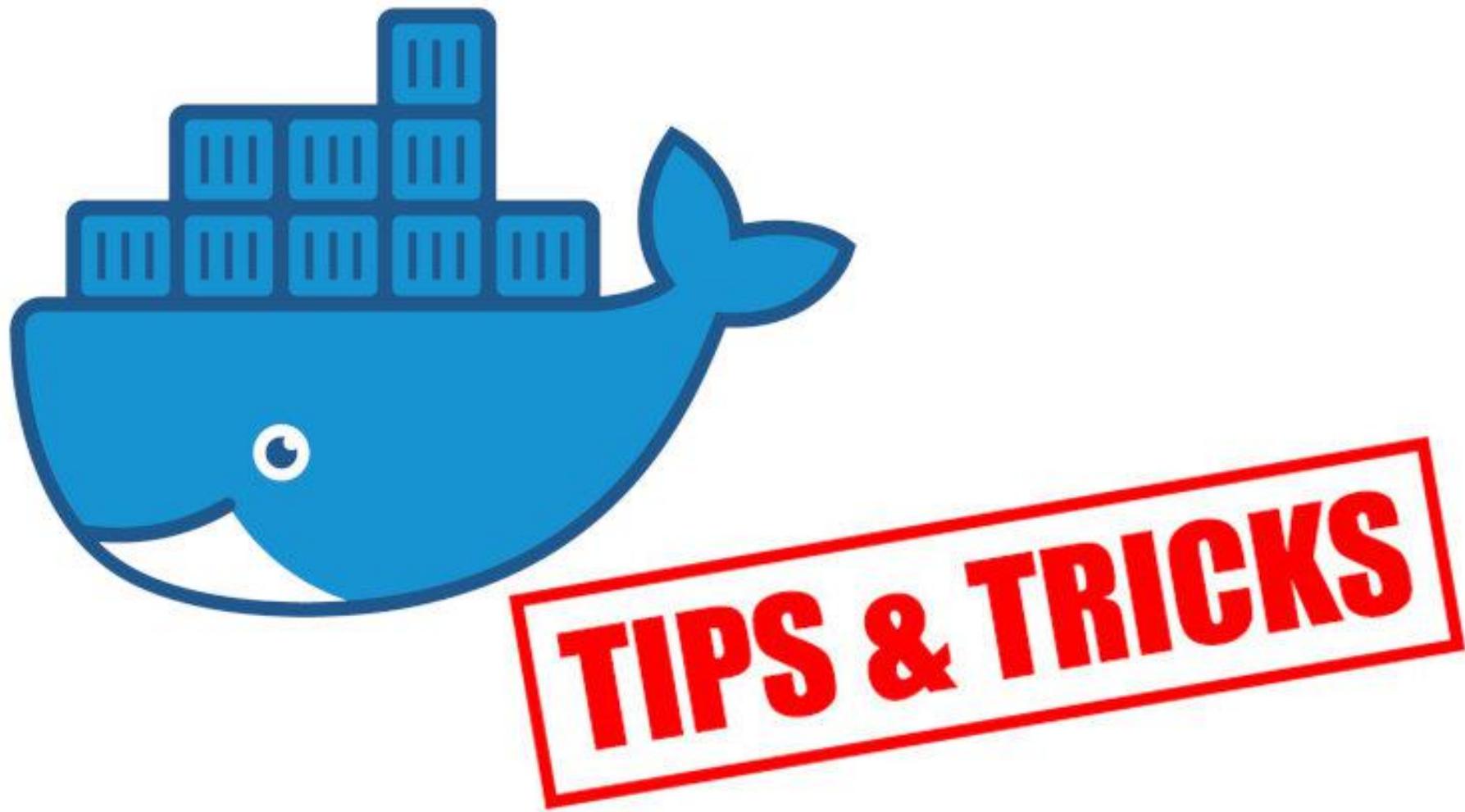
### Owner

 joseluisgs

### Source Repository

 [Github](#) [joseluisgs/docker-apache-php](#)

# Trucos y consejos



# Trucos y consejos

- Alias que te facilitarán la vida.
- Eliminar todos los contenedores de la máquina  

```
alias drma='docker rm $(docker ps -aq)'
```
- Eliminar todas las imágenes sin etiquetar que suelen crearse cuando falla una creación  

```
alias drm='docker rmi $(docker images -q -f "dangling=true")'
```
- Eliminar todas las imágenes  

```
alias drmai='docker rmi $(docker images -q)'
```
- Ejecutar un comando en una máquina  

```
alias drit='docker exec -ti'
drit debian bash
```



# Trucos y consejos

- Importante: Cada linea del dockerfile crea una imagen nueva si se modifica el estado de la imagen. Es importante buscar el balance entre la cantidad de capas creadas y la legibilidad del dockerfile

- No instalar paquetes innecesarios

- Utilizar ENTRYPOINT por dockerfile

- Combinar comandos similares usando "&&" y ""

- Utilizar el sistema de caché de manera inteligente

- **SIEMPRE INTENTAR:**

- Minimizar el tamaño de la imagen, el tiempo de compilación y el número de capas.

- Maximizar el uso de la caché de compilación y la legibilidad del Dockerfile.

- Queremos que trabajar con nuestro contenedor sea lo más agradable posible.

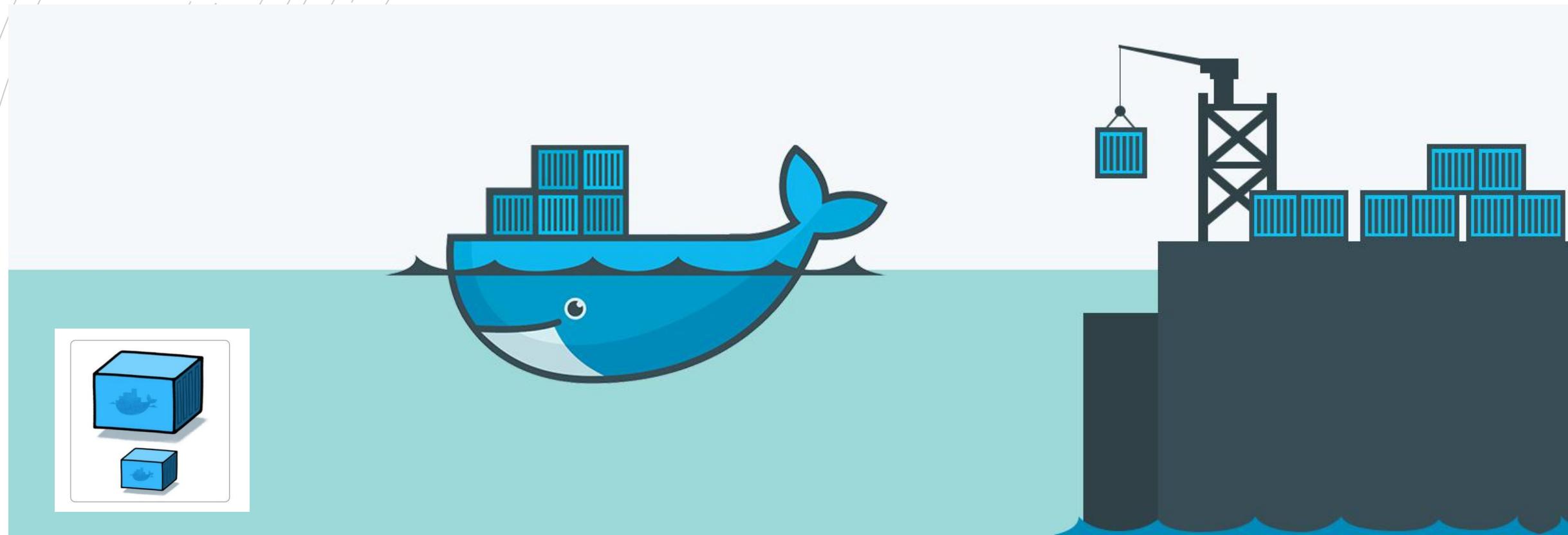


# Trucos y consejos

- Escribir archivo `.dockerignore`
- El contenedor debe hacer una sola cosa
- ¡Debemos entender el almacenamiento en caché de Docker! Usa los comandos COPY y RUN en el orden correcto a la hora de utilizarlo.
- Fusionar varios comandos RUN en uno solo.
- Eliminar archivos innecesarios después de cada paso.
- Usar una imagen base adecuada (las versiones Alpine deberían ser suficientes)
- Configurar WORKDIR y CMD
- Utilizar ENTRYPOINT cuando se tenga más de un comando y/o se necesite actualizar archivos usando datos en tiempo de ejecución.
- Usar exec dentro del script del punto de entrada.
- Mejor usar COPY que ADD
- Especificar variables de entorno, puertos y volúmenes predeterminados dentro de Dockerfile



# Optimizando Dockers



# Optimizando

- Comenzamos

```
FROM ubuntu
ADD . /app
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nodejs ssh mysql
RUN cd /app && npm install
# esto debería iniciar tres procesos, mysql y ssh
# en segundo plano y node app en primer plano
# no es tremadamente bonito? <3
CMD mysql & sshd & npm start
```



# Optimizando

- **1. Escribir .dockerignore al crear una imagen.** Docker debe preparar el contexto antes de nada: recopilar todos los archivos que se pueden utilizar en un proceso. El contexto predeterminado contiene todos los archivos de un directorio Dockerfile. Normalmente no queremos incluir ahí el directorio .git, las bibliotecas descargadas ni los archivos compilados. El archivo .dockerignore es exactamente igual a .gitignore, por ejemplo:

```
.git/  
node_modules/  
dist/
```

- **2. El contenedor debe hacer una sola cosa.** Técnicamente, PUEDES iniciar múltiples procesos dentro de un contenedor Docker. PUEDES poner aplicaciones de base de datos, frontend y backend, ssh, y supervisor en una imagen de docker. Pero muchas cosas no te irán bien:

Los tiempos de compilación serán largos (un cambio en, por ejemplo, el frontend te obligará a volver a compilar todo el backend)

Las imágenes serán muy grandes

Tendrás un registro duro de datos desde muchas aplicaciones (no un simple stdout)

Escalado horizontal innecesario

Problemas con los procesos "zombies" - Tendrás que acordarte de cuál es el proceso init adecuado

Mi consejo es que prepares una imagen Docker separada para cada componente y que utilices Docker Compose para iniciar fácilmente varios contenedores al mismo tiempo.



# Optimizando

- Mi consejo es que prepares una imagen Docker separada para cada componente y que utilices Docker Compose para iniciar fácilmente varios contenedores al mismo tiempo.
- Eliminemos los paquetes innecesarios de nuestro Dockerfile. SSH puede ser reemplazado por docker exec.

```
FROM ubuntu
ADD . /app
RUN apt-get update
RUN apt-get upgrade -y
# deberíamos quitar ssh y mysql, y usar
# contenedor separado para la base de datos
RUN apt-get install -y nodejs # ssh mysql
RUN cd /app && npm install
CMD npm start
```



# Optimizando

- **3. Fusionar varios comandos RUN en uno solo.** Docker está totalmente basado en capas. El conocimiento de cómo funcionan es esencial.

Cada comando en Dockerfile crea una capa.

Las capas se almacenan en caché y se reutilizan.

Invalidar la caché de una sola capa invalida todas las capas subsiguientes.

La invalidación ocurre después de un cambio de comando, si los archivos copiados son diferentes, o si la variable de compilación es diferente a la anterior.

Las capas son inmutables, así que, si añadimos un archivo a una capa y lo eliminamos en la siguiente, la imagen TODAVÍA contiene ese archivo (¡simplemente no estará disponible en el contenedor!)

Se debe tener en cuenta que se deben fusionar comandos que tienen una probabilidad similar de ser modificados o de sufrir cambios. Actualmente, cada vez que nuestro código fuente se modifique, necesitamos reinstalar Node.js por completo.

```
FROM ubuntu
RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install
CMD npm start
```



# Optimizando

- **4. No utilices la etiqueta de imagen base latest.** La etiqueta latest es la que se utiliza por defecto, cuando no se especifica ninguna otra etiqueta. Así que nuestra instrucción FROM ubuntu en realidad hace exactamente lo mismo que FROM ubuntu:latest. Pero la etiqueta latest apuntará a una imagen diferente cuando se publique una nueva versión, y tu build puede romperse. Por lo tanto, a menos que estés creando un Dockerfile genérico que tengas que estar actualizado con la imagen base, deberás proporcionar una etiqueta específica.
- En nuestro ejemplo, usemos la etiqueta 16.04:

```
FROM ubuntu:16.04 # ¡es así de fácil!
RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install
CMD npm start
```



# Optimizando

- **5. Eliminar archivos innecesarios después de cada paso de RUN.** Por lo tanto, supongamos que hemos actualizado las fuentes apt-get, instalado algunos paquetes necesarios para compilar otros, y hemos descargado y extraído los archivos. Obviamente no los necesitamos en nuestras imágenes finales, así que, mejor hagamos una limpieza. ¡El tamaño importa!
- En nuestro ejemplo podemos eliminar las listas apt-get (creadas por apt-get update):

```
FROM ubuntu:16.04
RUN apt-get update \
    && apt-get install -y nodejs \
    # líneas añadidas
    && rm -rf /var/lib/apt/lists/*
ADD . /app
RUN cd /app && npm install
CMD npm start
```



# Optimizando

- **6. Usar una imagen base adecuada.** En nuestro ejemplo estamos usando ubuntu. Pero, ¿por qué? ¿Realmente necesitamos una imagen de base de propósito general, cuando sólo queremos ejecutar una aplicación de Node.js? Una mejor opción es usar una imagen especializada con Node.js ya instalado o mejor aún, podemos elegir la versión Alpine (Alpine es una distribución Linux muy pequeña, de unos 4 MB de tamaño. Esto lo convierte en el candidato perfecto para una imagen base).

```
FROM node ADD . /app  
# ya no se necesita instalar  
# node ni usar apt-get  
RUN cd /app && npm install  
CMD npm start
```

---

```
-----  
FROM node:7-alpine  
ADD . /app  
RUN cd /app && npm install  
CMD npm start
```

Alpine tiene un gestor de paquetes, llamado apk. Es un poco diferente a apt-get, pero aún así es bastante fácil de aprender. Además, tiene algunas características realmente útiles, como las opciones --o-cache y --virtual. De esta manera, elegimos exactamente lo que queremos en nuestra imagen, nada más. Tu disco te va a amar :)



# Optimizando

- **7. Configurar WORKDIR y CMD.** El comando WORKDIR cambia el directorio por defecto, donde ejecutamos nuestros comandos RUN / CMD / ENTRYPOINT. CMD es una ejecución de comandos por defecto después de crear un contenedor sin otro comando especificado. Por lo general, es la acción que se realiza con más frecuencia. Añadámoslos a nuestro Dockerfile:

```
FROM node:7-alpine
WORKDIR /app
ADD . /app
RUN npm install
CMD [ "npm", "start" ]
```

# Optimizando

- **8. Usa ENTRYPOINT (opcional).** No siempre es necesario, es opcional, ya que Entrypoint, o punto de entrada, añade complejidad. ¿Cómo funciona este sistema? Entrypoint es un script, que se ejecutará en lugar de un determinado comando, y recibirá éste como un argumento. Es una excelente forma de crear imágenes ejecutables Docker
- `entrypoint.sh`

```
#!/usr/bin/env sh
# $0 is a script name,
# $1, $2, $3 etc are passed arguments # $1 is our command
CMD=$1
case "$CMD" in
  "dev" )
    npm install
    export NODE_ENV=development
    exec npm run dev
    ;;

  "start" )
  # we can modify files here, using ENV variables passed in
  # "docker create" command. It can't be done during build process.
  echo "db: $DATABASE_ADDRESS" >> /app/config.yml
  export NODE_ENV=production
  exec npm start
  ;;

  * )
  # Run custom command. Thanks to this line we can still use
  # "docker run our_image /bin/bash" and it will work
  exec $CMD ${@:2}
  ;;

esac
```



# Optimizando

- Guárdalo en tu directorio raíz, con el nombre entrypoint.sh. Así es su uso en Dockerfile:

```
FROM node:7-alpine
WORKDIR /app
ADD . /app
RUN npm install
ENTRYPOINT ["../entrypoint.sh"]
CMD ["start"]
```

- Ahora, valga la redundancia, podemos ejecutar esta imagen en un formato ejecutable:

```
docker run nuestra-app dev
docker run nuestra-app start
docker run -it nuestra-app /bin/bash` # Este también funcionará.
```

# Optimizando

- **9. Usar exec dentro del script entrypoint.** Como puedes ver en el ejemplo de entrypoint, estamos usando exec. Sin él, no podríamos detener nuestra aplicación de forma elegante (SIGTERM es engullido por el script bash). Exec básicamente reemplaza el proceso de script con uno nuevo, por lo que todas las señales y códigos de salida funcionan como se había previsto.
- **10. Mejor COPY que ADD.** COPY es más sencillo. ADD tiene cierta lógica para descargar archivos remotos y extraer archivos (más en la documentación oficial). Sólo tienes que quedarte con COPY. ADD puede ser útil si la compilación depende de recursos externos y quieres que la invalidación de la caché de compilación sea la adecuada en caso de cambio. No es una buena práctica, pero a veces es la única manera de hacerlo.

```
FROM node:7-alpine
WORKDIR /app
COPY . /app
RUN npm install
ENTRYPOINT [ "./entrypoint.sh" ]
CMD ["start"]
```



# Optimizando

- **11. Optimizar COPY y RUN.** Deberíamos poner los cambios que se producen con menor frecuencia en la parte superior de nuestros Dockerfiles para aprovechar el almacenamiento en caché. En nuestro ejemplo, el código cambiará a menudo, y no queremos reinstalar paquetes cada vez. Podemos copiar el package.json antes del resto del código, instalar dependencias y luego añadir otros archivos. Apliquemos esa mejora a nuestro Dockerfile:

```
FROM node:7-alpine
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

<https://www.campusvp.es/recursos/post/mejores-practicas-para-crear-dockerfiles-excelentes.aspx>



# Optimizando

- **12. Especificar variables de entorno, puertos y volúmenes predeterminados.** Probablemente necesitemos algunas variables de entorno para ejecutar nuestro contenedor. Es una buena práctica establecer valores predeterminados en Dockerfile. Además, debemos mostrar todos los puertos utilizados y definir los volúmenes.. Veamos la siguiente mejora aplicada a nuestro ejemplo:

```
FROM node:7-alpine
# env variables required during build
ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR
COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR
# env variables that can change
# volume and port settings
# and defaults for our application
ENV MEDIA_DIR=/media \
    NODE_ENV=production \
    APP_PORT=3000
VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
ENTRYPOINT [ "./entrypoint.sh" ]
CMD ["start"]
```

- Estas variables estarán disponibles en el contenedor. Si necesitas variables de compilación solamente, usa build args en su lugar.



# Optimizando

- **13. Añadir metadatos a la imagen usando LABEL.** Hay una opción para añadir metadatos a la imagen, como información sobre quién es el encargado de mantenerla o una descripción ampliada. Necesitamos la instrucción LABEL para ello (antes podíamos usar la opción MAINTAINER, pero ahora está obsoleta). Los metadatos son utilizados a veces por programas externos, por ejemplo nvidia-docker requiere la etiqueta com.nvidia.volumes.needed para funcionar correctamente. Ejemplo de un metadato en nuestro Dockerfile:

```
FROM node:7-alpine
LABEL maintainer "yo_soy_ese@example.com"
```

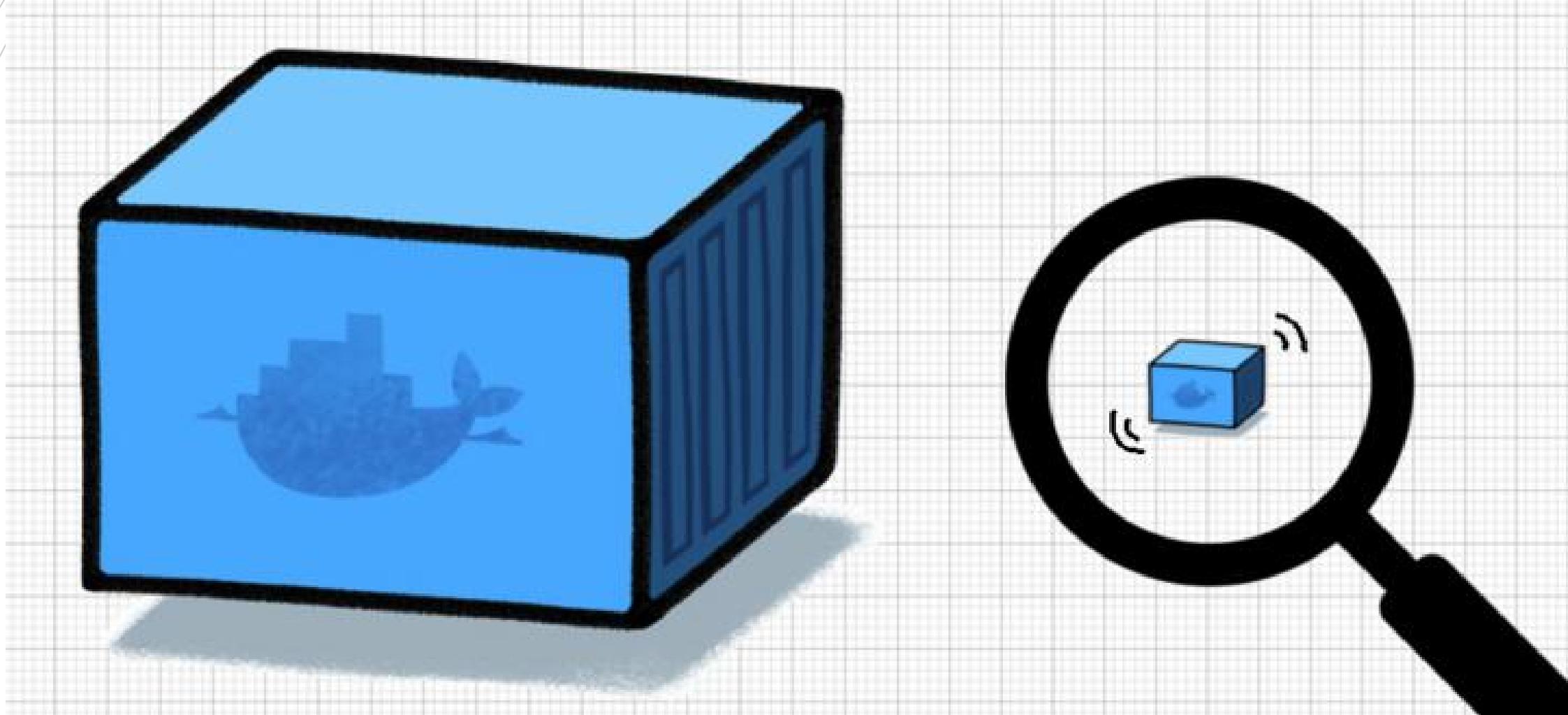
# Optimizando

- **14. Añadir HEALTHCHECK.** Podemos iniciar el contenedor docker con la opción --restart always (reiniciar siempre). Después de un fallo del contenedor, el "demonio" de Docker intentará reiniciarlo. Es muy útil si tu contenedor tiene que estar operativo todo el tiempo. Pero, ¿qué pasa si el contenedor se está ejecutando, pero no está disponible (bucle infinito, configuración no válida, etc.)? Con la instrucción HEALTHCHECK podemos decirle a Docker que compruebe periódicamente el estado de salud de nuestro contenedor. Puede ser cualquier comando, devolviendo 0 como código de salida si todo está bien, y 1 en el caso contrario. Último cambio a nuestro ejemplo:

```
FROM node:7-alpine
LABEL maintainer "yo_soy_ese@example.com"
ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR
COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR
ENV MEDIA_DIR=/media \
    NODE_ENV=production \
    APP_PORT=3000
VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit
ENTRYPOINT [ "./entrypoint.sh" ]
CMD ["start"]
curl --fail devuelve el código de salida que no es cero si la petición falla.
```



# Extra: Reto LEMP



# Extra: Entorno LEMP

- En este ejemplo vamos a ver cómo instalar un entorno LEMP, lo cual nos basaremos mucho en lo aprendido y teniendo en cuenta aspectos varios como configuración de servidores virtuales o proxy que igualmente podríamos haber usado en apartados anteriores. No está optimizando, así que eso os lo dejo a vosotros/as :)
- La idea es que poco a poco independientemente cuál sea el entorno lo adaptemos a nuestras necesidades ya sea creando volúmenes de datos específicos, redes, mapeando directorios como volúmenes o copiando dichos datos.
- Es decir, la libertad que tienes es tal que tú elegirás en todo momento cuál se adapta mejor a tus necesidades de desarrollo dependiendo el proyecto.
- Además, primero empezarás con configuraciones más sencillas y poco a poco irás avanzando y profundizando así como conociendo más esta tecnología para desarrollar las cosas de la manera que más te guste o que en ese momento necesites :)
- Yo lo resolveré como personalmente lo uso en mi caso ;)

# Extra: Entorno LEMP

- Partimos de la siguiente configuración en /path\_proyecto
  - code/myapp**: Dentro del directorio code estarán las aplicaciones, cada una en un subdirectorio. Para este ejemplo se creará el subdirectorio llamado myapp.
  - config/nginx**: Configuración de Nginx, tendremos un subdirectorio para cada configuración de proyecto, por ejemplo config/nginx/myapp, tambien podríamos tener para apache, o varios proyectos. Aquí podremos o copiar dicha configuración o mapear el directorio para que la coja.
  - config/php**: Configuración de PHP.
  - logs**: Directorio de logs, divididos por aplicación, por si me interesa leerlos, es un volumen mapeado
  - mariadb**: Directorio donde se almacenarán los datos de MariaDB, se debe tener en cuenta que tendremos una configuración por aplicación en data tendremos los datos en su docker si son necesarios
  - docker**: Directorio donde pondremos la configuración de las imágenes de docker a usar por aplicación, por si alguna es específica o requiere de configuraciones especiales.
- Host**: El último paso para terminar la configuración es añadir una entrada en el fichero hosts. En los sistemas Linux y OsX lo podemos encontrar en la ruta /etc/hosts y en sistemas Windows en c:\Windows\System32\drivers\etc\hosts. Hay que añadir al final la siguiente línea:  
127.0.0.1 myapp.dev
- Al guardar el fichero con el nuevo cambio se está indicando al sistema que cada vez que se solicite la url myapp.dev se resuelva al host local. podemos crear tantas entradas como proyectos tengamos.



# Extra: Entorno LEMP

- Configuración de MariaDB.
- Levantamos el contenedor

```
docker run -d --name mariadb -p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=password \
-e MYSQL_DATABASE=docker_sample \
-v "/home/informatica/Dropbox/Puertollano 2020-2021/DAW/Temario/docker-tutorial/ejemplos/ejem07/mariadb/data:/var/lib/mysql" \
mariadb:10.5
```

- Los parámetros que indicamos en el comando son:

- d: El contenedor se ejecuta en segundo plano
- name: El contenedor se llamará mariadb
- p 3306:3306, mapeamos los puertos con el objetivo de que podamos usar herramientas como Navicat, usando el público
- e son las variables de entorno para poner las contraseñas
- v /path\_proyecto/mariadb/data:/var/lib/mysql: Mapeamos el directorio local /path:proyecto/mariadb/data con el directorio del contenedor /var/lib/mysql que es donde mariadb guardará todos los datos sobre las base de datos y no perderlos por si borramos o si queremos sincronizarlos vía git..
- mariadb:10.5 La imagen que se arranca de MariaDB

# Extra: Entorno LEMP

- **Configuracion de PHPMyAdmin (Opcional)**
- Aunque ya hemos dejado los puertos abiertos para usar Navicat o similar...
- Levantamos el contenedor

```
docker run -d --name phmyadmin \
--link mariadb:db\
-p 8888:80 \
phpmyadmin
```

- Los parámetros que indicamos en el comando son:
  - d: El contenedor se ejecuta en segundo plano
  - name: El contenedor se llamará phpmyadmin
  - link, para que esté enlazado al conetenedor mariadb:db (el db es importante porque e la BD)
  - p 8888:80, mapeamos los puertos
  - phpmyadmin La imagen que usaremos de phpmyadmin

# Extra: Entorno LEMP

- **Configuración de PHP.**
- Lo primero es que al usar Nginx usamos un proxy inverso, por lo tanto usamos PHP-FPM. Además vamos a usar el Driver de PHP PDO y su conexión con MySQL/MariaDB, nuestro dockerfile sería:

```
# Imagen de PHP
FROM php:7.4-fpm
# Instalamos los paquetes que necesitamos
RUN apt-get update && docker-php-ext-install mysqli pdo pdo_mysql
```

- Probamos nuestra imagen, solo construyendo, no hace falta hacer el run. A parte he puesto mi nombre de usuario de DockerHub por si quisiera publicarla, como ya hemos visto anteriormente.

```
$ docker build -t joseluisgs/php-fpm .
```



# Extra: Entorno LEMP

- Para activar el driver pdo\_mysql que hemos comentado anteriormente, tenemos que hacerlo desde el fichero de configuración de php, php.ini. Para ello, vamos al directorio config/php y creamos el fichero php.ini con el siguiente contenido:

```
extension=pdo_mysql.so
```

- Para este ejemplo solo necesitamos poner esta configuración, pero si necesitamos ver un fichero php.ini completo para modificar algún parámetro, podemos tomar como referencia los ficheros del repositorio de php para [desarrollo](#) o [producción](#)
- Con el fichero php.ini actualizado, arranquemos ahora el contenedor con el siguiente comando:

```
docker run -d --name php7 \
--link mariadb \
-v "/path_proyecto/config/php":/usr/local/etc/php \
-v "/path_proyecto/code/myapp":/var/www/html/myapp \
joseluisgs/php-fpm
```



# Extra: Entorno LEMP

- Los parámetros que hemos utilizado en el comando significan lo siguiente:
  - d: El contenedor se va a ejecutar en segundo plano
  - link: se enlazará al contenedor de mariadb
  - name php7: El contenedor se va a llamar php7
  - v /path\_proyecto/config/php:/usr/local/etc/php: Mapeamos en la ruta de nuestro sistema local /path\_proyecto/config/php el directorio del contenedor /usr/local/etc/php. En este directorio es donde está el fichero php.ini para que el contenedor obtenga su configuración a partir de él. Si quisésemos podríamos haberlo copiado y ahorrarnos mapearlo.  
`docker cp /path_proyecto/config/php:/usr/local/etc/php/php.ini php7.4:/usr/local/etc/php`
- -v /path\_proyecto/code/myapp:/var/www/html/myapp: Mapeamos el código fuente de nuestra aplicación que se encuentra en /path\_proyecto/code/myapp al directorio del contenedor /var/www/html/myapp
- joseluisgs/php-fpm: La imagen que se va a arrancar es la que acabamos de crear

# Extra: Entorno LEMP

- **Configuracion de PHPMyAdmin (Opcional)**
- Aunque ya hemos dejado los puertos abiertos para usar Navicat o similar...
- Levantamos el contenedor

```
docker run -d --name phmyadmin \
--link mariadb:db\
-p 8888:80 \
phpmyadmin
```

- Los parámetros que indicamos en el comando son:
  - d: El contenedor se ejecuta en segundo plano
  - name: El contenedor se llamará phpmyadmin
  - link, para que esté enlazado al conetenedor mariadb:db
  - p 8888:80, mapeamos los puertos
  - phpmyadmin La imagen que usaremos de phpmyadmin

# Extra: Entorno LEMP

- **Configuración de Nginx.**
- Para el contenedor de Nginx también vamos a ir a Hub Docker y buscaremos la imagen oficial de Nginx. Pero antes hay que configurar un virtual host para que apunte al directorio de la aplicación. Para ello, dentro de la carpeta para la configuración de Nginx (/config/nginx en mi caso) se creará un fichero llamado myapp.conf con el siguiente contenido. Podemos tener uno para cada app que realicemos:

```
server {  
    index index.php index.html;  
    server_name myapp.dev;  
    error_log /var/log/nginx/myapp_error.log;  
    access_log /var/log/nginx/myapp_access.log;  
    root /var/www/html/myapp;  
    location ~ \.php$ {  
        try_files $uri =404;  
        fastcgi_split_path_info ^(.+\.php)(/.+)$;  
        fastcgi_pass php7:9000;  
        fastcgi_index index.php;  
        include fastcgi_params;  
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;  
        fastcgi_param PATH_INFO $fastcgi_path_info;  
    }  
}
```



# Extra: Entorno LEMP

- Este es un fichero típico de configuración de Nginx.
- Algunos de los campos más importantes son el server\_name donde se indica que las acciones descritas a continuación se ejecutarán cuando se solicite ese servidor.
- Las rutas de los fichero de acceso y de error (/var/log/nginx/myapp\_error.log y /var/log/nginx/myapp\_access.log) estas rutas pertenecen a la ruta de ficheros de dentro del contenedor Docker, no de nuestro sistemas host.
- También se indica donde está la ruta del código fuente de la aplicación (/var/www/html/myapp) de nuevo con la ruta dentro del sistema de ficheros del contenedor y finalmente una de las características más importantes, en la línea 14 se indica donde se está ejecutando el proceso php-fpm.
- Sabemos que este proceso se está ejecutando en el contenedor que hemos arrancado antes, pero una característica que tienen los contenedores Docker es que su ip puede variar cada vez que arranque el contenedor.
- Por ello, en vez de poner la ip, se indica el nombre que le hemos dado al contenedor al arrancar, si recordamos era php7.4. Con esto e indicando a la hora de arrancar este nuevo contenedor que cree un enlace con el contenedor de php automáticamente se resolverá ese nombre a la ip con la que haya arrancado el contenedor php.



# Extra: Entorno LEMP

- Levantamos el contenedor

```
docker run -d --name nginx \
-v "/path_proyecto/config/nginx":/etc/nginx/conf.d \
-v "/path_proyecto/code/myapp":/var/www/html/myapp \
-v "/path_proyecto//logs":/var/log/nginx \
-p 8080:80 \
--link php7.4 nginx
```

-itd: El contenedor se va a ejecutar en segundo plano y podremos tener acceso a su shell

--name nginx: El contenedor se va a llamar nginx

-v /path\_proyecto/config/nginx:/etc/nginx/conf.d: Mapeamos en nuestra ruta local /path\_proyecto/config/nginx el directorio del contenedor /etc/nginx/conf.d donde se encuentra la configuración de Nginx

-v /path\_proyecto/code/myapp:/var/www/html/myapp: Mapeamos también el directorio con el código fuente de la aplicación que se encuentra en /path\_proyecto/code/myapp al directorio del contenedor /var/www/html/myapp

-v //path\_proyecto/logs:/var/log/nginx: Mapeamos el directorio local /path\_proyecto/logs con el directorio del contenedor /var/log/nginx donde se guardarán los logs de Nginx

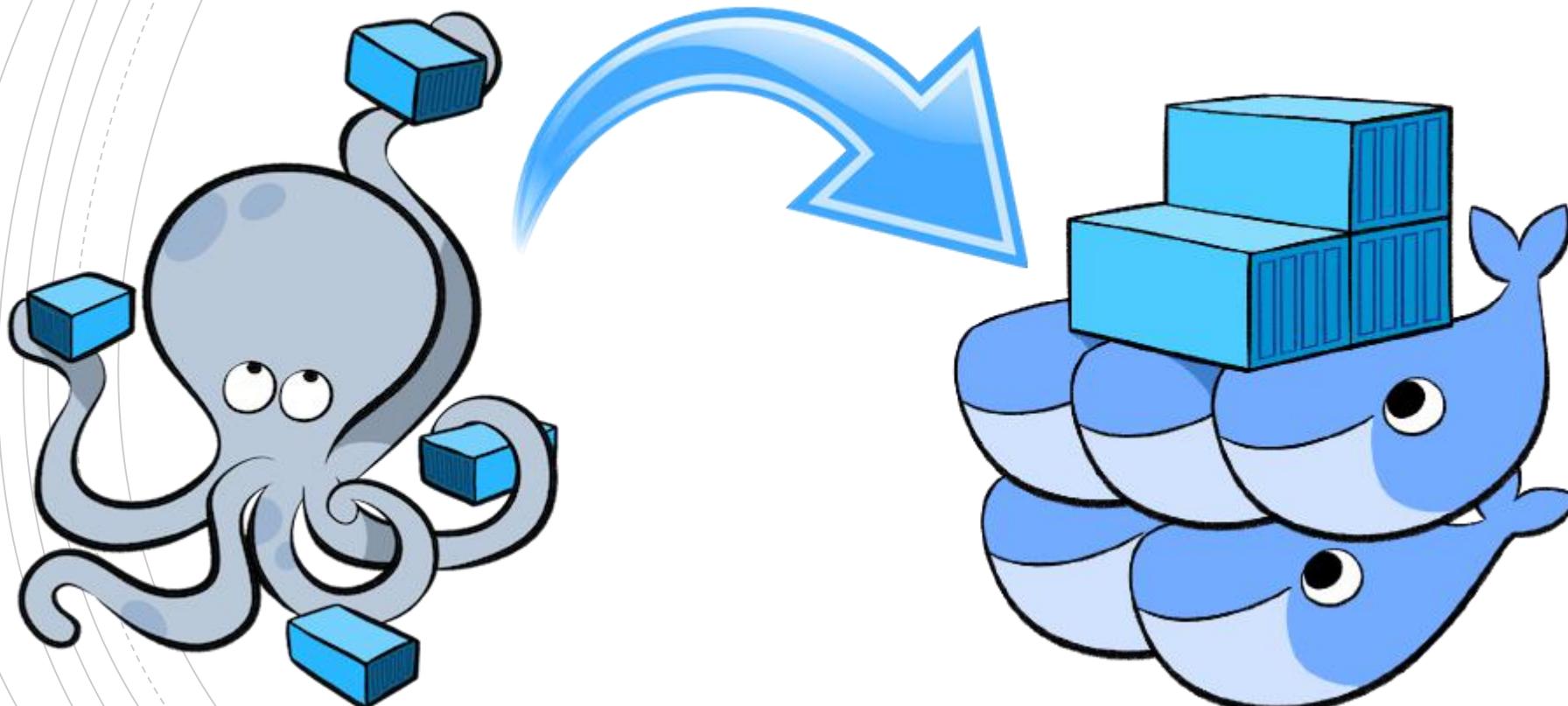
-p 8080:80: Hacemos que se redirija las conexiones que van al puerto 8080 de nuestro host al puerto 80 del contenedor

--link php7.4: Linkamos el contenedor con el contenedor antes creado llamado php7

nginx: La imagen que se arranca es la última versión de nginx

# Extra: Entorno LEMP

- Ahora vamos a ver la ventaja de Docker Compose para no estar tan ligado al orden de los ficheros dockerfile, los cuales están muy bien para tareas individuales, pero es un rollo cuando tenemos muchos servicios y están entrelazados. ¡Comenzamos!



# Extra: Entorno LEMP

```
# Indicamos la versión  
version: '3.7'  
  
# Iniciamos los servicios  
services:  
  # PHP  
  php7:  
    container_name: php7  
    build: ./php  
    volumes:  
      - /path_proyecto/config/php:/usr/local/etc/php  
      - /path_proyecto/code/myapp:/var/www/html/myapp  
    depends_on:  
      - mariadb  
  networks:  
    - lemp-network
```

# Extra: Entorno LEMP

```
# Indicamos la versión  
version: '3.7'  
  
# Iniciamos los servicios  
services:  
  # PHP  
  php7:  
    container_name: php7  
    build: ./php  
    volumes:  
      - /path_proyecto/config/php:/usr/local/etc/php  
      - /path_proyecto/code/myapp:/var/www/html/myapp  
    depends_on:  
      - mariadb  
  networks:  
    - lemp-network
```

# Extra: Entorno LEMP

```
# NGINX
nginx:
  container_name: nginx
  image: nginx
  ports:
    - 8080:80
  volumes:
    - /path_proyecto/config/nginx:/etc/nginx/conf.d
    - /path_proyecto/code/myapp:/var/www/html/myapp
    - /path_proyecto/logs:/var/log/nginx
  depends_on:
    - php7
  networks:
    - lemp-network
```

# Extra: Entorno LEMP

```
# MARIADB
mariadb:
  container_name: mariadb
  image: mariadb:latest
  volumes:
    - /path_proyecto/mariadb/data:/var/lib/mysql
  environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_DATABASE=docker_sample
  ports:
    - 3306:3306
  networks:
    - lemp-network
```



# Extra: Entorno LEMP

```
# PHPMYADMIN  
  
phpmyadmin:  
  image: phpmyadmin/phpmyadmin  
  container_name: phpmyadmin  
  links:  
    - 'mariadb:db'  
  depends_on:  
    - mariadb  
  ports:  
    - 8081:80  
  networks:  
    - lemp-network  
  
# Si queremos que tengan una red propia a otros contenedores
```

```
networks:  
  lemp-network:  
    driver: bridge
```



# Extra: Entorno LEMP

- Con Docker Compose, todo es más fácil y somos más felices :)

```
docker-compose up -d
```

Este comando creará y arrancará los tres contenedores que hemos definido. Si queremos que los contenedores se ejecuten en segundo plano como demonios añadiremos al comando el atributo -d.

- Con los contenedores arrancados ya podemos volver a acceder a la url de nuestra aplicación y ver el resultado.

Además del comando up para manejar docker compose tenemos los siguientes comandos:

- Para arrancar las contenedores una vez que están creados ejecutaremos:

```
docker-compose start
```

- Para parar contenedores:

```
docker-compose stop
```

- Para parar contenedores y eliminarlos:

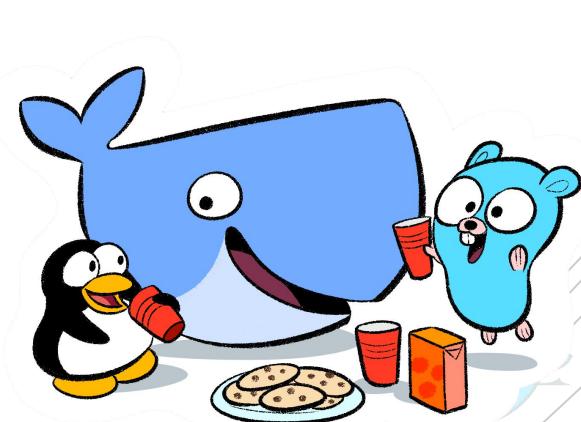
```
docker-compose down
```



# Conclusiones y reflexiones

## ■ ¿Por qué deberías poco a poco usar Dockers?

- Es open source.
- Facilita el testing y facilita la tarea.
- Ahorra tiempo: te evita instalar diferentes softwares para ejecutar una App.
- Ambiente comun para equipos de desarrollos, la misma imagen y contenedor puede por el equipo sin riesgo de fallar en paquetes, librerías, instalaciones de recursos, etc.
- Es muy sencillo crear y eliminar contenedores.
- Los contenedores se vuelven muy livianos: permite manejar varios dentro de una misma máquina.
- Es menos costoso: requiere menos espacio, menos menos máquinas y menos ordenadores.
- Da libertad de tener todo en un único lugar lo necesario para hacer correr una app
- Portabilidad. Al almacenar los contenedores en discos duros, se pueden transportar de un lugar a otro sin problemas.
- Repositorios Docker. “Banco de imágenes docker” creadas por usuarios a las cuales podemos tener acceso.
- Se acelera el proceso de mantenimiento y desarrollo gracias a las facilidades para generar copias.
- Las aplicaciones se ejecutan sin variaciones. Sin importar el equipo ni el ambiente.
- Facilita las visualizaciones al cliente gracias a que no tiene que instalar nada más que docker en su ordenador.
- Depliegue en a nube basado en dockers
- Facilidad en tener un ambiente de desarrollo y otro de producción basado en contenedores.
- Es un entorno seguro y no ofrece variaciones.
- Ideal para el uso de microservicios y para el CI/CD



# Referencias

- [Docker](#): Sitio oficial
- [DockerHub](#): Imágenes
- [Docker File](#): Manejo de Dockerfile
- [Docker Compose](#): Manejo de Compose
- [Docker Libro](#): Repositorio sobre Docker
- [Curso de Docker](#): Libro en PDF de José Juan Sánchez Hernández
- [Docker para desarrolladores](#). Curso de Openwebinars

