

1. Conceptos básicos

1.1. ¿Qué es Docker?

[Docker](#) es una plataforma para que desarrolladores y administradores puedan desarrollar, desplegar y ejecutar aplicaciones en un entorno aislado denominado **contenedor**.

[Docker](#) **empaqueta software en unidades estandarizadas llamadas contenedores** que incluyen todo lo necesario para que el software se ejecute (librerías, código, archivos de configuración, etc).

Antes de [Docker](#) ya existían implementaciones de aislamiento de recursos como:

- *Chroot*, en el año 1982.
- *FreeBSD Jails*, en el año 2000.
- *Linux Containers (LXC)*, en el año 2008.

[Docker](#) empezó a ganar popularidad en el año 2013 permitiendo a los desarrolladores crear, ejecutar y escalar rápidamente sus aplicaciones creando contenedores.

El uso de contenedores **es actualmente uno de los mecanismos más comunes para desplegar software**.

Empresas como **Google, Microsoft, Amazon, Oracle, VMware, IBM y RedHat** están apostando fuertemente por las tecnologías de contenerización.

El pasado 13 de noviembre de 2019, la empresa desarrolladora de Docker, Docker Inc, fue [adquirida por Mirantis por 35 millones de dólares](#).

Uno de los principales problemas que resuelve es el de **It works on my machine**.

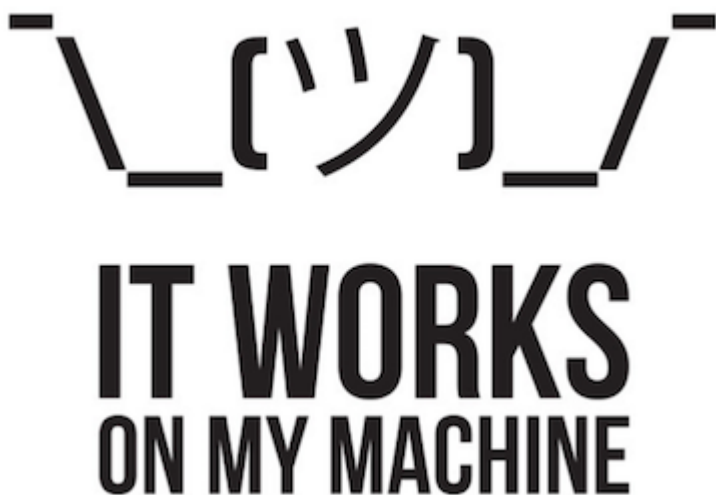


Figure 1. It works on my machine

Referencia:

- [De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo](#)

1.2. Analogía con el transporte marítimo de contenedores

Los contenedores de transporte marítimo:

- Cumplen un **estándar** para enviar mercancías.
- No nos importa el contenido sino que su forma sea estándar.
- Pueden ser transportados en cualquier **embarcación** que cumpla el estándar.

Los contenedores software:

- Cumplen un **estándar** para empaquetar software.
- No nos importa el contenido sino que su "forma" sea estándar.
- Pueden ser ejecutados en cualquier **servidor** que "cumpla el estándar".

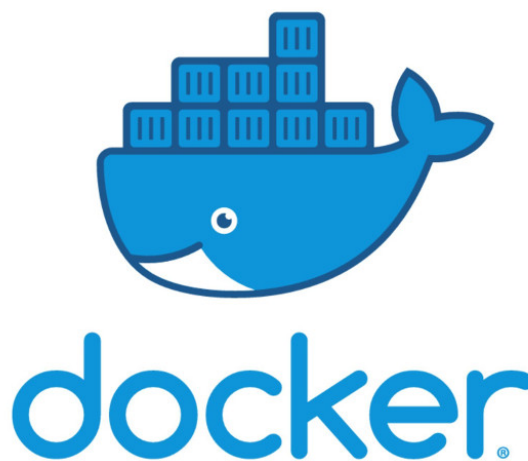


Figure 2. Transporte marítimo de contenedores. Imagen de [FreeCodeCamp](#)

1.3. ¿Qué es una Máquina Virtual?

Es un software que simula un sistema de computación y puede ejecutar **programas** como si fuese una computadora real.

Una característica esencial de las máquinas virtuales es que **los procesos** que ejecutan están limitados por los recursos y abstracciones proporcionados por ellas.

Referencia: [Wikipedia](#)

1.4. ¿Qué es un contenedor?

Un contenedor es **un proceso** que ha sido aislado de todos los demás procesos en la máquina anfitriona (máquina *host*). Ese aislamiento aprovecha características de Linux como los **namespaces del kernel y cgroups**.



Aunque es posible tener más de un proceso en un contenedor las buenas prácticas nos recomiendan ejecutar **sólo un proceso por contenedor (PID 1)**.

1.5. Contenedores vs Máquinas Virtuales

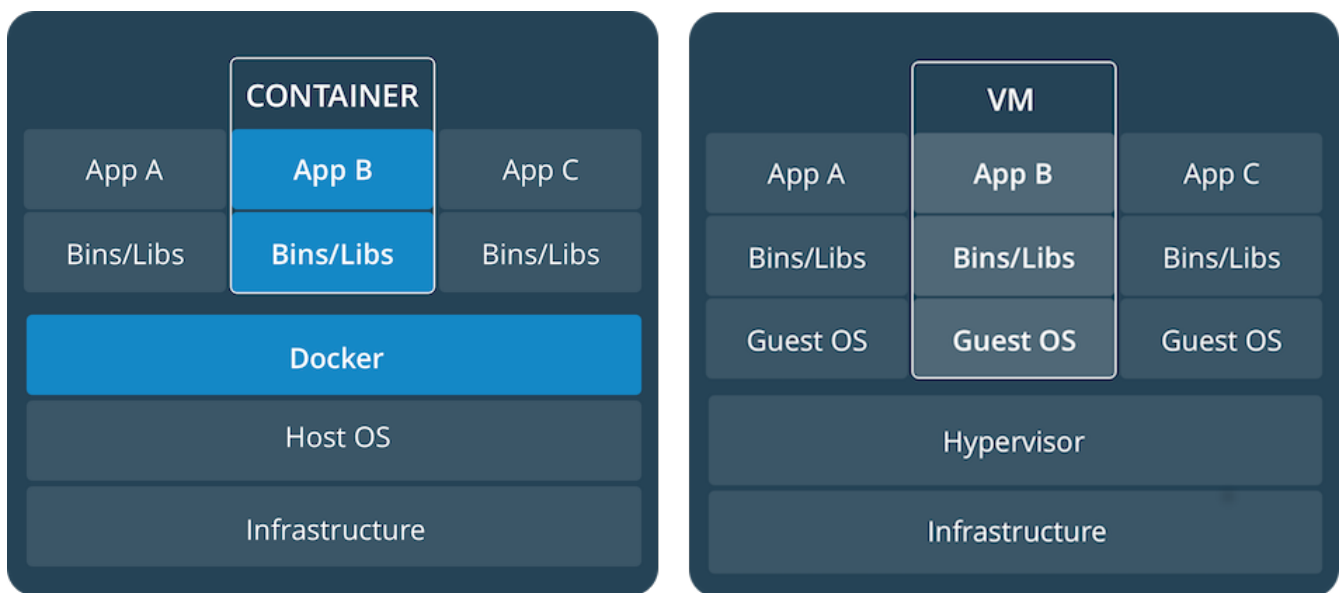


Figure 3. Contenedores vs Máquinas Virtuales. Imagen de [Docker.com](#)

- Los contenedores son más ligeros que las máquinas virtuales porque comparten el kernel del host.
- Con el mismo hardware, es posible tener un mayor número de contenedores que de máquinas virtuales.
- Los contenedores se pueden ejecutar en hosts que sean máquinas virtuales.

1.5.1. Ventajas para los desarrolladores

- Soluciona el problema *"It works on my machine"*.
- Permite tener un entorno de desarrollo **limpio, seguro y portátil**.
- Permite la **automatización** de pruebas, integración y empaquetado.
- Permite **empaquetar** una aplicación con todas las dependencias que necesita (código fuente, librerías, configuración, etc.) para ser ejecutada en cualquier plataforma.

1.5.2. Ventajas para administradores

- Se **eliminan inconsistencias** entre los entornos de desarrollo, pruebas y producción.

- El proceso de **despliegue es rápido y repetible**.

Referencia:

[What is the difference between a process, a container, and a VM?](#)

1.6. Arquitectura de Docker

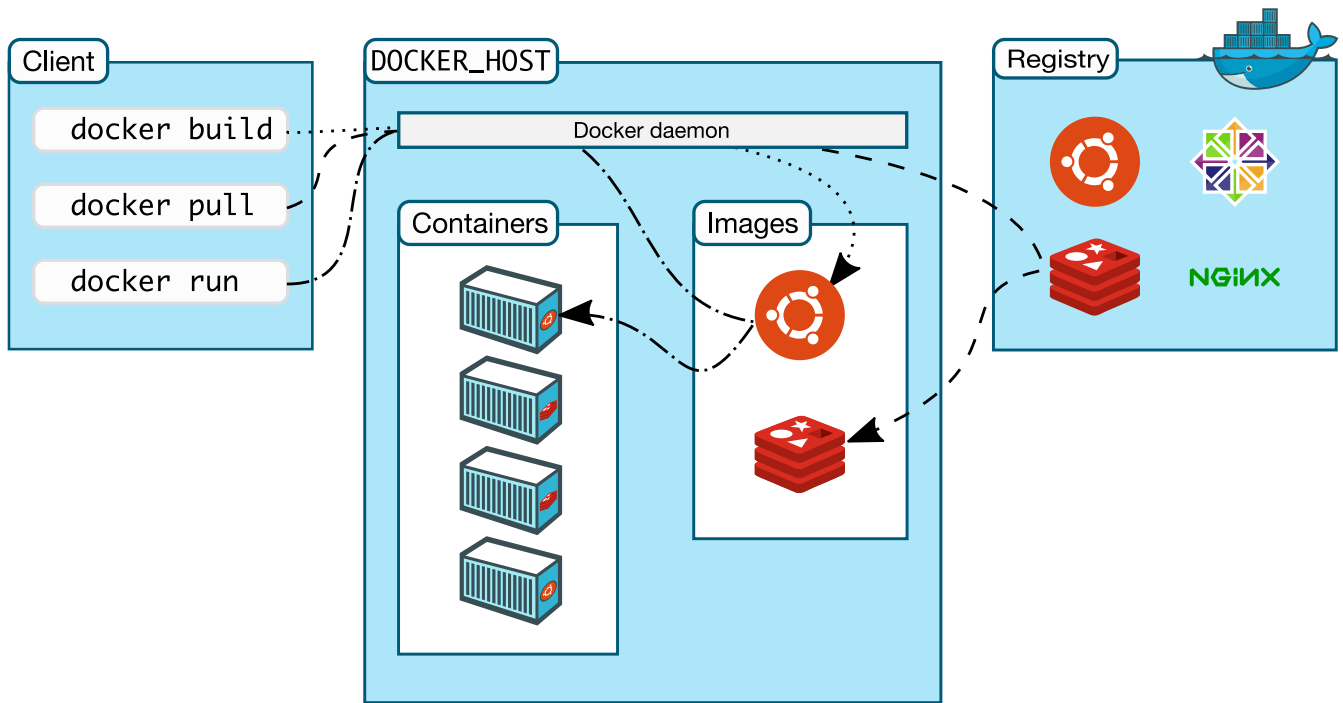


Figure 4. Arquitectura de Docker. Imagen de [Docker.com](#)

- Docker Daemon
- Docker Client
- Docker Registries
- Docker Objects
 - Images
 - Containers
 - Volumes
 - Networks

Referencias:

- <https://docs.docker.com/engine/docker-overview/>
- <https://docs.docker.com/glossary/>

1.7. Docker Engine

El *Docker Engine* es una aplicación cliente-servidor formada por los siguientes componentes:

- Docker daemon

- Docker REST API
- Docker CLI

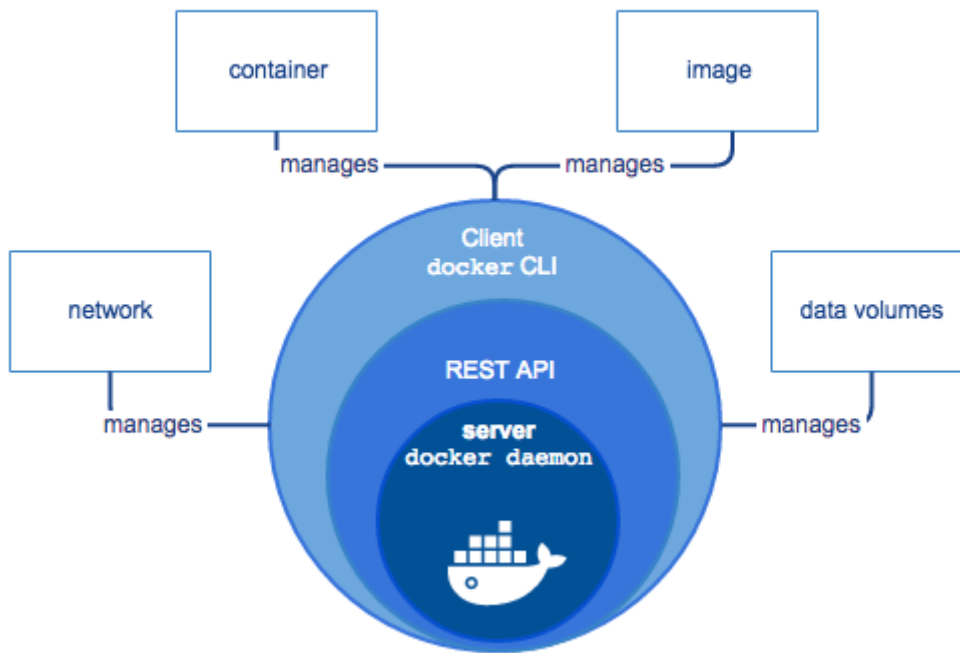


Figure 5. Docker Engine. Imagen de [Docker.com](https://docs.docker.com/engine/docker-overview/)

Referencia:

- <https://docs.docker.com/engine/docker-overview/>

1.8. Dockerfiles vs Imágenes vs Contenedores

Un **Dockerfile** es un archivo de texto que contiene los comandos necesarios para crear una imagen.

Una **imagen** se crea a partir de un archivo **Dockerfile**. Contienen la unión de sistemas de archivos apilados en capas, donde cada capa representa una modificación de la imagen y equivale a una instrucción en el archivo **Dockerfile**.

Un **contenedor** es una instancia en ejecución de una **imagen**.

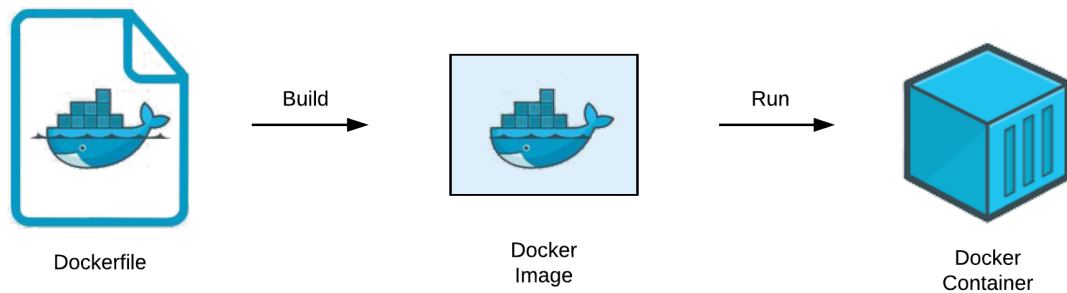


Figure 6. Dockerfiles vs Imágenes vs Contenedores. Imagen de [Ekaba Bisong](#)

1.9. ¿Qué tecnología hay detrás de Docker?

Docker está escrito en [Go](#).

1.9.1. Espacio de nombres

Es una característica de **aislamiento** de recursos del kernel de Linux. Nos permiten realizar visualizaciones restringidas de los recursos.

Cuando ejecutamos un contenedor, Docker crea un conjunto de *namespaces* para ese contenedor.

- Process trees (PID namespace)
- Mounts (MNT namespace)
- Network (NET namespace)
- Unix Timesharing System (UTS namespace)
- Inter Process Communication (IPC Namespace)

1.9.2. Cgroups (Control Groups)

Es una característica del kernel de Linux que permite **limitar y aislar** recursos (CPU, memoria, disco I/O, red, etc.) utilizados por un grupo de procesos.

1.9.3. Sistemas de archivos en capas (Union File Systems)

Estos sistemas de archivos que funcionan creando capas, haciéndolos muy ligeros y rápidos. Docker Engine utiliza **UnionFS** para proporcionar los bloques de construcción para contenedores.

Docker Engine puede usar múltiples variantes de UnionFS, como: AUFS, btrfs, vfs y DeviceMapper.

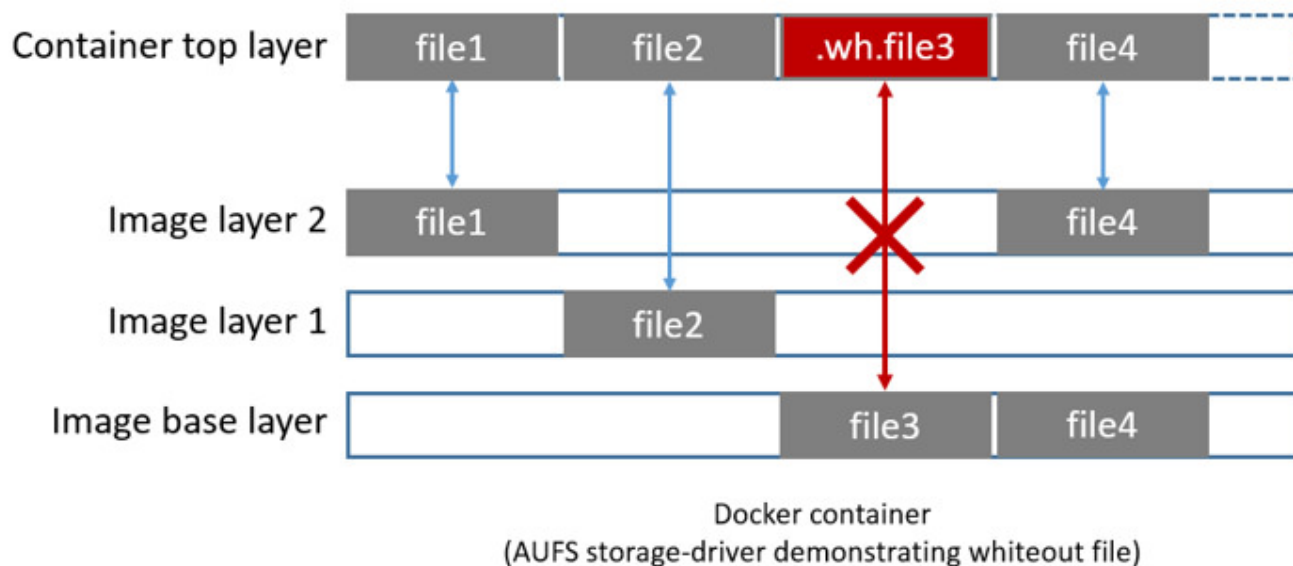


Figure 7. AUFS storage-driver demonstrating whiteout file. Imagen de [AUFS and Docker Deployment](#)

Referencias:

- <https://docs.docker.com/engine/docker-overview/>
- [Docker Internals. A Deep Dive Into Docker For Engineers Interested In The Gritty Details.](#)

1.10. Productos de Docker

- **Docker Enterprise Edition (EE):** Es la versión empresarial y es de pago.
- **Docker Community Edition (CE):** Es la versión de uso gratuito, *open source* y se puede usar en **Windows, Mac y Linux**.
 - Docker for Linux
 - Docker Desktop for MacOS
 - Docker Desktop for Windows



Docker Toolbox: Para versiones **previas a** Windows 10 Professional o Enterprise 64-bit

Cuando instalamos **Docker Desktop** en **Windows** o **Mac** viene acompañado de una máquina virtual llamada **MobyLinux** que nos permite ejecutar contenedores para Linux.

Docker CE puede funcionar en Windows de dos modos:

- **Windows Containers:** Permite ejecutar contenedores con imágenes de Windows Server Core o Windows Nano Server.
- **Linux Containers:** Crea y arranca automáticamente una máquina virtual en Hyper-V llamada MobyLinux donde se ejecutarán nuestros contenedores con imágenes basadas en Linux.

Referencia:

- [LCOW: Linux Containers on Windows](#)

1.11. El stack de contenerización

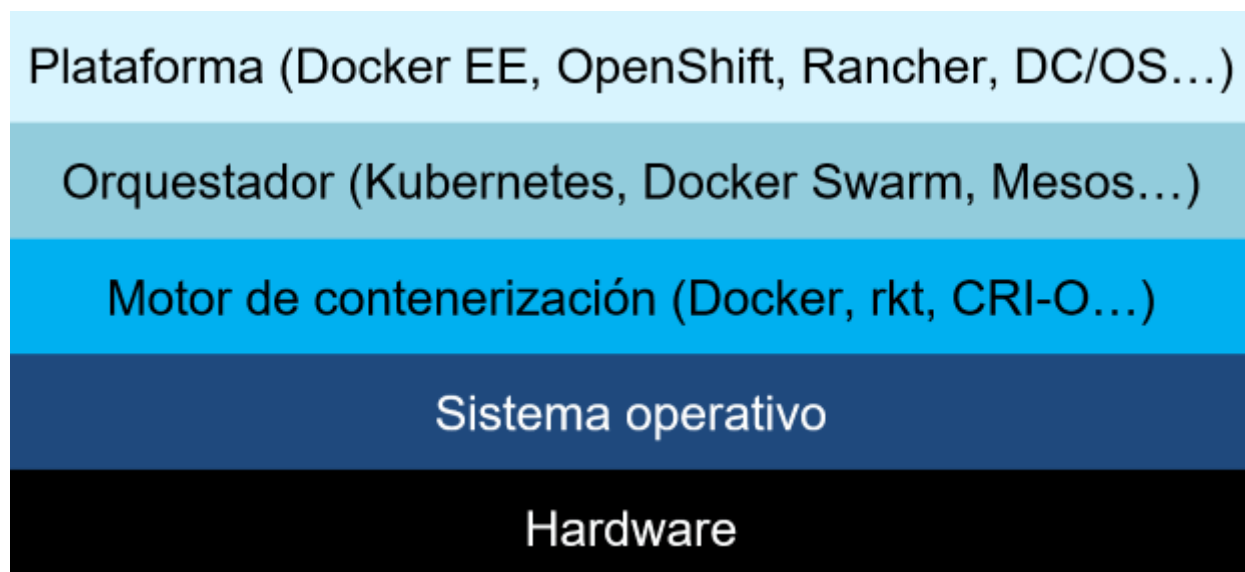


Figure 8. Stack de contenerización. Imagen de [Carlos Milán](#)

Referencia: <https://calnus.com/2018/08/14/rancher-2-x-iis-arr-kubernetes-on-premises/>

2. Instalación de Docker

1. [Instalación para Ubuntu \(Linux\).](#)
2. [Pasos posteriores a la instalación en Linux.](#)
 - a. Configurar nuestro usuario para trabajar con Docker.
 - b. Configurar el servicio de Docker para que se inicie automáticamente.



Errores comunes después de la instalación

Si nos aparece el siguiente error después de la instalación en Linux es porque el usuario con el que estamos ejecutando **docker** no tiene privilegios de **root** o no está en el grupo **docker**.

```
$ docker search ubuntu
```

```
Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock: Get
http://%2Fvar%2Frun%2Fdocker.sock/v1.39/images/search?limit=25&term=ubuntu: dial unix
/var/run/docker.sock: connect: permission denied
```

2.1. Configuración del usuario

El *daemon* de Docker utiliza un socket Unix y por defecto, el socket Unix es propiedad del usuario **root**, de modo que los demás usuarios solo pueden acceder a él usando **sudo**.

Para evitar tener que escribir **sudo** cada vez que vayamos a ejecutar un comando de **docker** tenemos que añadir el usuario con el que vamos a trabajar al grupo docker.

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

Para activar los cambios en los grupos sin tener que cerrar la sesión podemos ejecutar lo siguiente.

```
$ newgrp docker
```

2.2. Configurar el servicio de Docker para que se inicie automáticamente.

```
$ sudo systemctl enable docker
```

2.3. Comprobamos si docker está instalado correctamente

```
$ docker version
```

3. Administración básica de contenedores Docker

3.1. Ciclo de vida de un contenedor Docker

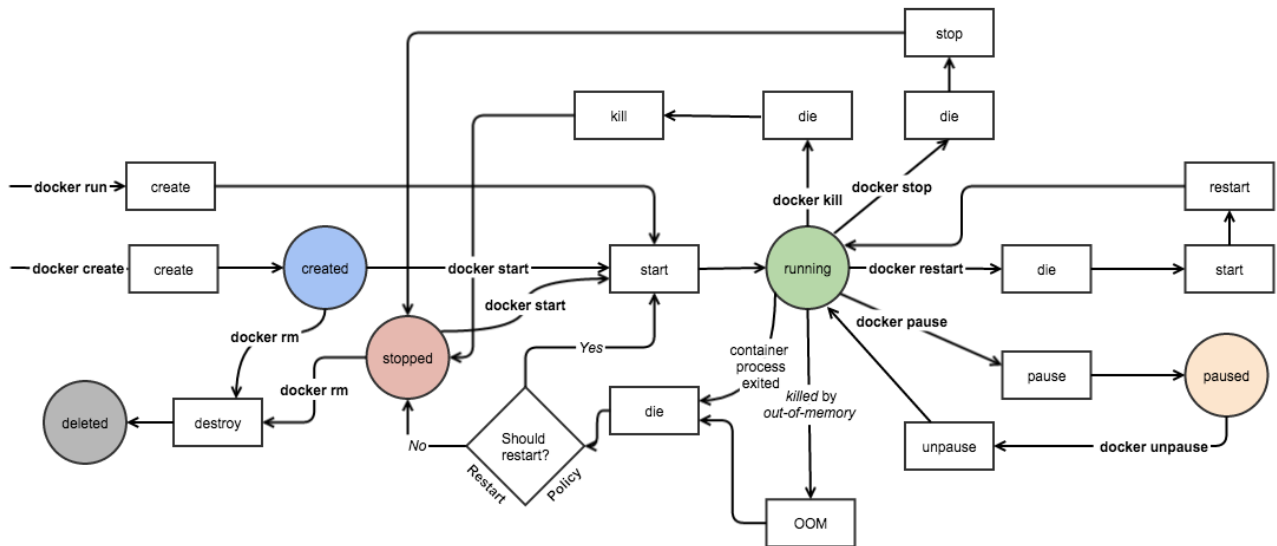


Figure 9. Ciclo de vida de un contenedor Docker. Imagen de [Nitin Agarwal](#)

3.2. Docker client query verbs

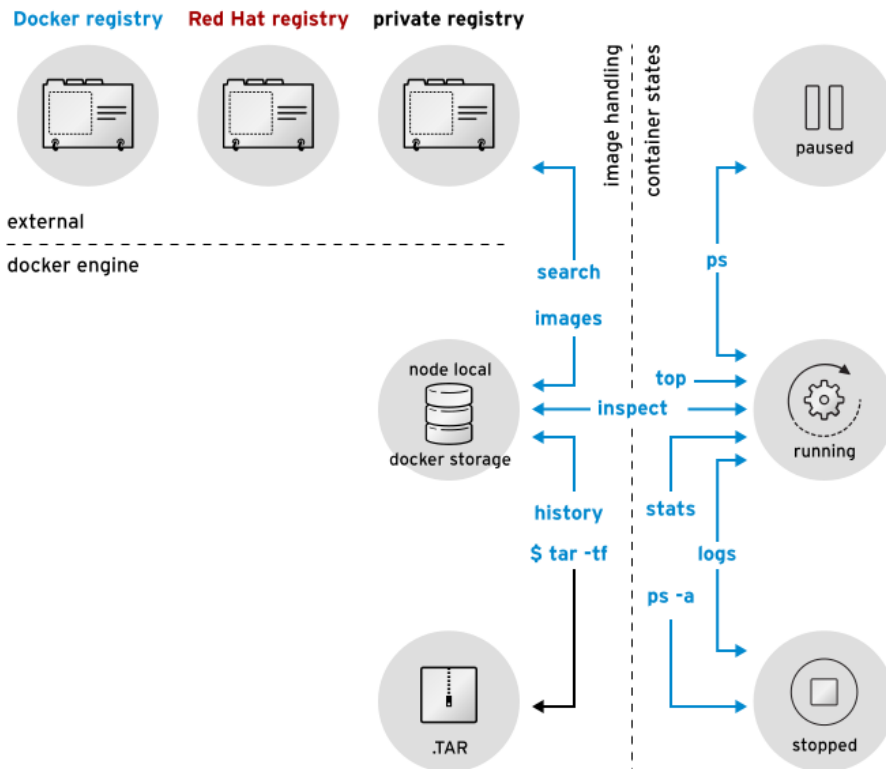


Figure 10. Docker client query verbs. Imagen de edX.

3.3. Docker client query actions

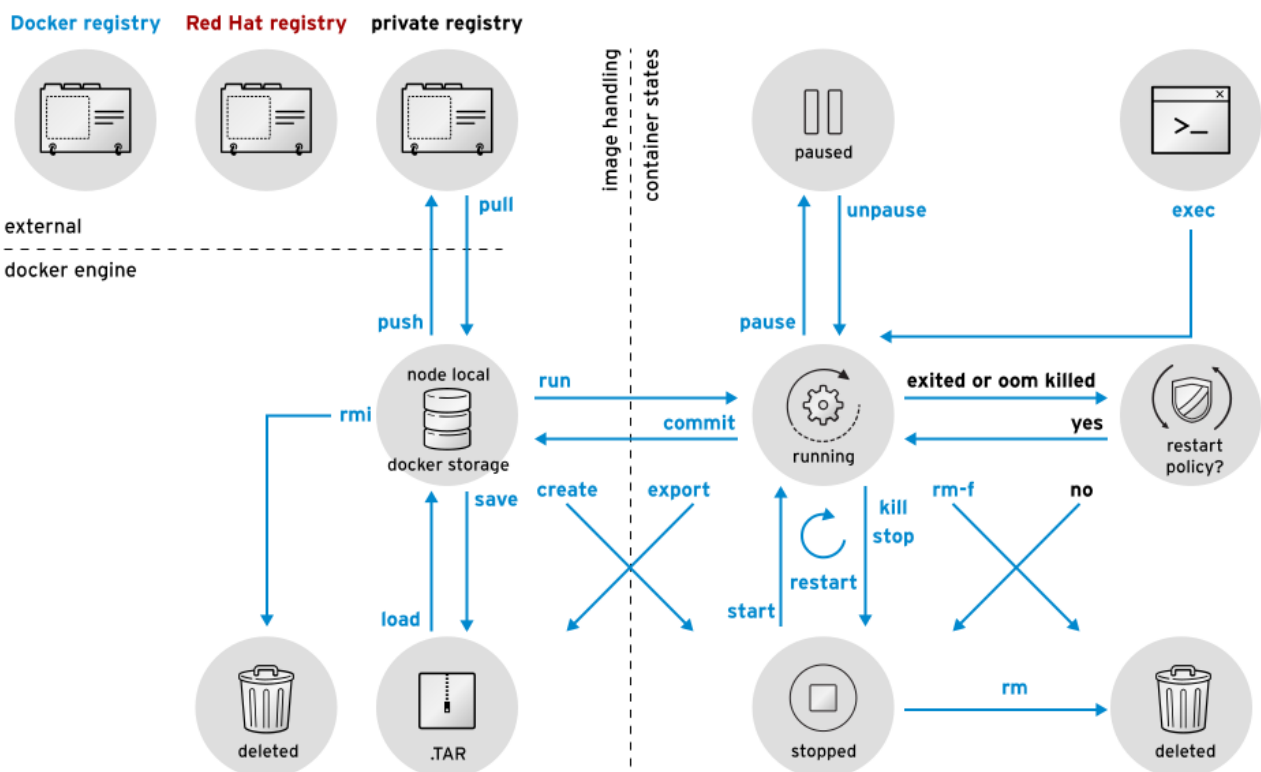


Figure 11. Docker client query actions. Imagen de edX.

3.4. Comandos de Docker CLI

```
$ docker --help
```

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

--config string	Location of client config files (default "/Users/josejuansanchez/.docker")
-c, --context string	Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var and default context set with "docker context use")
-D, --debug	Enable debug mode
-H, --host list	Daemon socket(s) to connect to
-l, --log-level string	Set the logging level ("debug" "info" "warn" "error" "fatal") (default "info")
--tls	Use TLS; implied by --tlsverify
--tlscacert string	Trust certs signed only by this CA (default "/Users/josejuansanchez/.docker/ca.pem")
--tlscert string	Path to TLS certificate file (default "/Users/josejuansanchez/.docker/cert.pem")
--tlskey string	Path to TLS key file (default "/Users/josejuansanchez/.docker/key.pem")
--tlsverify	Use TLS and verify the remote
-v, --version	Print version information and quit

Management Commands:

builder	Manage builds
config	Manage Docker configs
container	Manage containers
context	Manage contexts
image	Manage images
network	Manage networks
node	Manage Swarm nodes
plugin	Manage plugins
secret	Manage Docker secrets
service	Manage services
stack	Manage Docker stacks
swarm	Manage Swarm
system	Manage Docker
trust	Manage trust on Docker images
volume	Manage volumes

Commands:

attach	Attach local standard input, output, and error streams to a running container
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories on a container's filesystem
events	Get real time events from the server
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
info	Display system-wide information
inspect	Return low-level information on Docker objects
kill	Kill one or more running containers
load	Load an image from a tar archive or STDIN
login	Log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container

ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save one or more images to a tar archive (streamed to STDOUT by default)
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
version	Show the Docker version information
wait	Block until one or more containers stop, then print their exit codes

4. Imágenes Docker

4.1. Buscar imágenes en Docker Hub

```
$ docker search --help
```

```
Usage:  docker search [OPTIONS] TERM
```

```
Search the Docker Hub for images
```

```
Options:
```

```
-f, --filter filter    Filter output based on conditions provided
--format string        Pretty-print search using a Go template
--limit int            Max number of search results (default 25)
--no-trunc             Don't truncate output
```

Ejemplo:

Vamos a buscar la imagen de [Alpine Linux](#), que es una distribución Linux muy ligera. Esta imagen ocupa **menos de 6 MB**.

```
$ docker search alpine
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
alpine	A minimal Docker image based on ...	5797	[OK]	
mhart/alpine-node	Minimal Node.js built on...	444		
anapsix/alpine-java	Oracle Java 8 (and 7) with GLIBC ...	428		[OK]
frolvlad/alpine-glibc	Alpine Docker image with gl	218		[OK]
gliderlabs/alpine	Image based on Alpine Linux will ...	180		
...				



Docker Hub nos informa de cuales son las imágenes oficiales. Por seguridad, se recomienda hacer uso exclusivamente de las imágenes oficiales.

Puede encontrar más información sobre las imágenes oficiales de Docker en la [documentación de la web oficial](#).



Las imágenes que aparecen marcadas como **AUTOMATED** son imágenes que Docker Hub ha generado automáticamente a partir del código fuente de un repositorio externo y se han enviado a los repositorios de Docker.

Puede encontrar más información sobre los *builds* automáticos en la [documentación oficial de Docker](#).

En el listado del resultado de la búsqueda podemos ver que la descripción de las imágenes aparece truncada. Podemos hacer uso de la opción **--no-trunc** para poder visualizar la descripción completa

de cada una de las imágenes.

Ejemplo:

```
$ docker search alpine --no-trunc
```

Por defecto, cada búsqueda devolverá un **máximo de 25 resultados**. Pero es posible incrementar el número de resultados de la búsqueda utilizando el flag **--limit**. Este flag nos permite indicar un número entre 1 y 100.

Ejemplo:

```
$ docker search alpine --limit 100
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/search/>

4.2. Buscar imágenes en Docker Hub utilizando filtros

Los únicos filtros que podemos realizar desde están basados en las siguientes condiciones:

- **stars=(number)**
- **is-automated=(true|false)**
- **is-official=(true|false)**

A continuación se muestran algunos **ejemplos de uso**:

Por el número de estrellas

Buscar las imágenes de [Alpine Linux](#) que tengan al menos 10 estrellas.

```
$ docker search --filter stars=10 alpine
```

Sólo imágenes oficiales

Buscar las imágenes de [Alpine Linux](#) que sean oficiales.

```
$ docker search --filter is-official=true alpine
```

Sólo imágenes con *build* automático

Buscar las imágenes de [Alpine Linux](#) que tengan un *build* automático.


```
$ docker search --filter is-automated=true alpine
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/search/>

Ejercicios

1. Utiliza `docker search` para buscar imágenes de sistemas operativos que conozcas y de aplicaciones web que te puedan ser de utilidad en los módulos que impartas.
2. Hemos visto que `docker search` sólo nos permite realizar tres tipos de filtros, sin embargo en la web oficial de [Docker Hub](#) tenemos la posibilidad de utilizar otros filtros diferentes. Utiliza la web oficial para [conocer cuáles son las imágenes de Docker más populares](#) y realiza algunas búsquedas.
3. ¿Existe alguna imagen que permita ejecutar Docker dentro de un contenedor Docker?
4. Busca qué imágenes de sistemas operativos base existen para crear contenedores con Windows.

4.3. Imágenes interesantes de Docker

Algunas imágenes de Docker que podemos destacar por su popularidad son las siguientes:

- alpine: Linux reducido
- nginx: Servidor web Nginx
- httpd: Servidor web Apache
- ubuntu: Ubuntu
- redis: Base de datos Redis (clave-valor)
- mongo: Base de datos MongoDB (documentos)
- mysql: Base de datos MySQL (relacional)
- postgres: Base de datos PostgreSQL (relacional)
- node: Node.js
- registry: Registro de imágenes on-premise
- php, haproxy, wordpress, rabbitmq, python, openjdk, tomcat, jenkins, redmine, elasticsearch...



En la web de la documentación oficial puede encontrar un [listado de las imágenes Docker más populares](#).

4.4. Descargar imágenes desde Docker Hub

```
$ docker pull --help
```

```
Usage:  docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Pull an image or a repository from a registry

Options:

-a, --all-tags	Download all tagged images in the repository
--disable-content-trust	Skip image verification (default true)
-q, --quiet	Suppress verbose output

Ejemplo:

Vamos a descargar la última versión de la imagen **alpine**. Para descargarse la última versión (**latest**) no es necesario indicar ninguna etiqueta.

```
$ docker pull alpine
```

El comando anterior es equivalente a:

```
$ docker pull alpine:latest
```

Si quisiera descargar una versión específica, por ejemplo la versión 3.7 de **alpine** tendría que indicarlo de la siguiente manera:

```
$ docker pull alpine:3.7
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/pull/>

Ejercicios

1. Descarga las imágenes de **alpine**, **ubuntu**, **httpd**, **nginx** y **mysql** de Docker Hub.

4.5. Descargar imágenes desde un *Registry* diferente a Docker Hub

Por defecto, Docker descarga las imágenes desde Docker Hub, pero es posible descargar imágenes desde un *registry* diferente si indicamos el *path* donde está la imagen que queremos descargar. El *path* es similar a una URL, pero no es necesario indicar el protocolo (https://).

Ejemplo:

El siguiente comando descargará la imagen **curso-docker/test-image** de un *registry* local que estará escuchando en el puerto 5000 (**myregistry.local:5000**):

```
$ docker pull myregistry.local:5000/curso-docker/test-image
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/pull/>

4.6. Layers de una imagen

Una imagen está formada por distintas capas que contienen las modificaciones que se han ido realizando sobre una imagen base.

A las capas también se les puede llamar **imágenes intermedias**.

Cada capa representa **cada una de las instrucciones del archivo Dockerfile** con el que se ha creado la imagen.



Recuerda que un contenedor es una instancia de una imagen.

En un **contenedor** todas las **capas son de lectura**, excepto la **última capa que será de lectura/escritura**.

Ejemplo

Suponga que partimos del siguiente archivo Dockerfile:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

El archivo contiene cuatro instrucciones, por lo tanto se crearán cuatro capas.

La primera capa (**d3a1f33e8a5a**) será la que hace referencia a la instrucción **FROM ubuntu:18:04** y ocupa 188.1 MB. Sobre esta capa se crea la de la instrucción **COPY (c22013c84729)**, en tercer lugar se crea la capa de la instrucción **RUN (d7508fb6632)**. En último lugar se crea la capa de la instrucción **CMD (91e54dfb1179)** que ocupa 0 Bytes.

Las capas de la imagen son de **sólo lectura**. Cuando instanciamos un contenedor, se crea una nueva capa encima de la imagen que será de **lectura/escritura**.

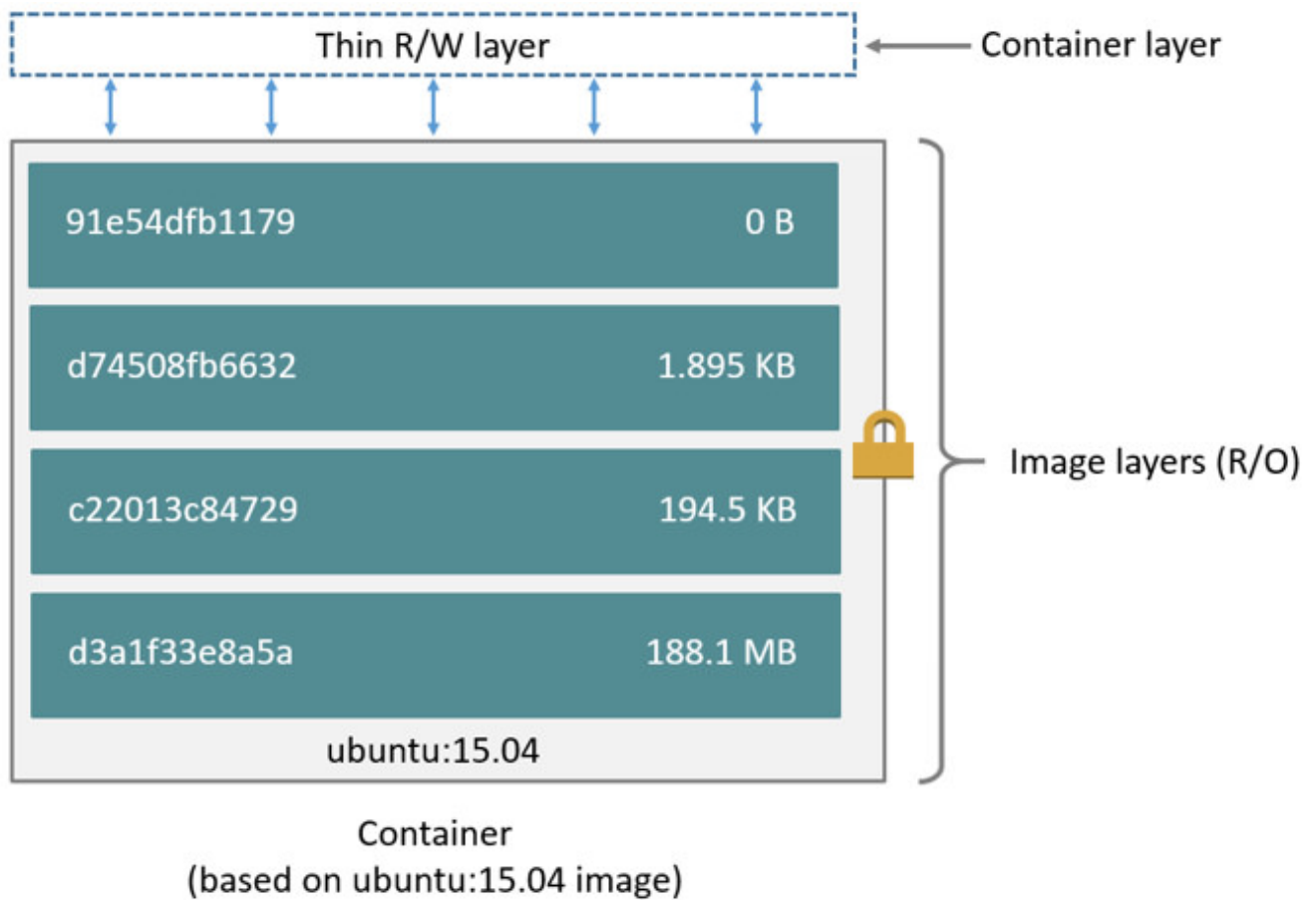


Figure 12. Imagen de las capas de lectura y escritura de un contenedor **ubuntu**. Imagen de [Docker](#)

La siguiente figura muestra como varios contenedores en ejecución comparten la misma imagen. Cada contenedor tiene su propia capa de lectura/escritura, donde almacenan todos los cambios. Cuando eliminamos un contenedor, sólo borramos su capa de lectura/escritura.

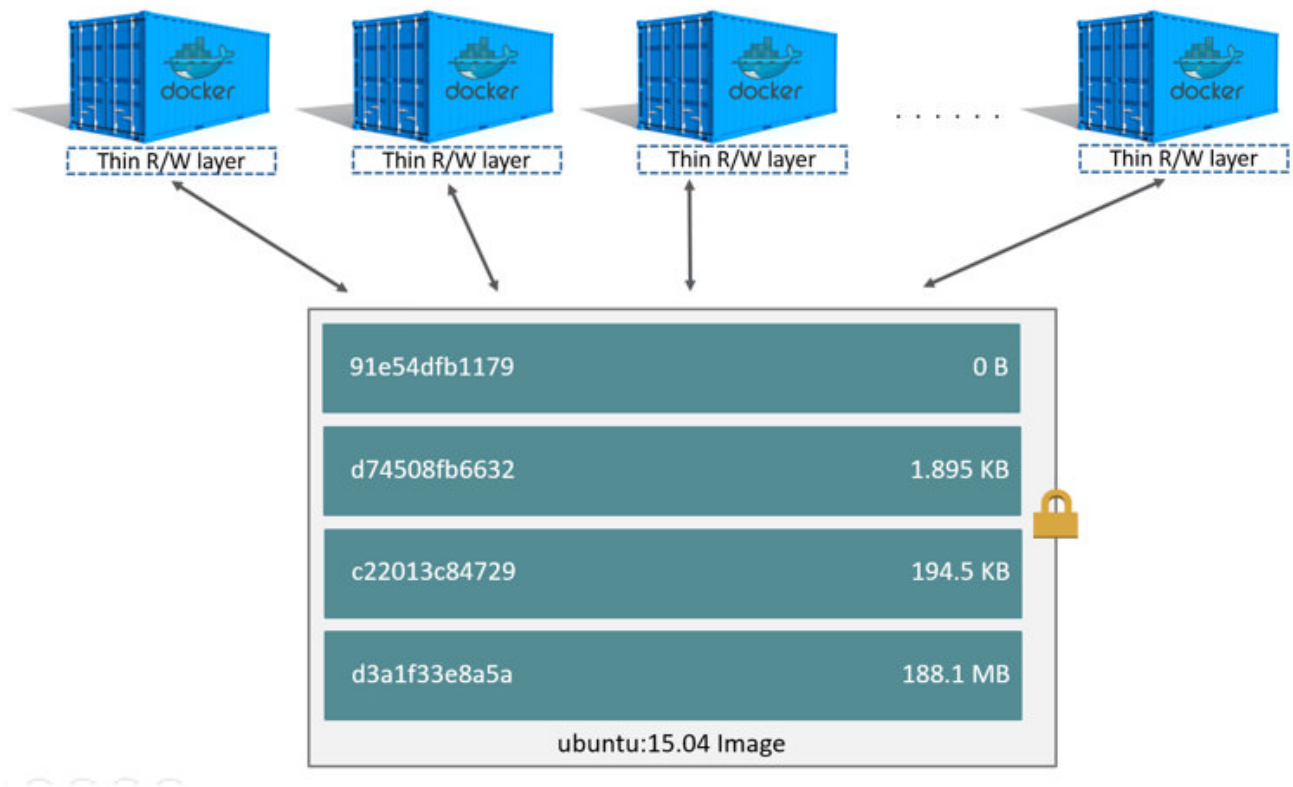


Figure 13. Varios contenedores comparten la misma imagen de **ubuntu**. Imagen de **Docker**

Ejemplo

Cuando descargamos una imagen obtenemos una salida similar a la del siguiente ejemplo, donde descargamos la imagen de **alpine**:

```
$ docker pull alpine

Using default tag: latest
latest: Pulling from library/alpine
89d9c30c1d48: Pull complete ①
Digest: sha256:c19173c5ada610a598915111163d28a67368362762534d8a8121ce95cf2bd5a
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

① Esta línea indica que hemos descargado una capa de la imagen.

En este caso la imagen de **alpine** está formada por una única capa.

Si observamos el archivo **Dockerfile** de la imagen de **alpine** encontramos lo siguiente:

```
FROM scratch
ADD alpine-minirootfs-3.10.3-x86_64.tar.gz /
CMD ["/bin/sh"]
```

Todas las instrucciones del archivo Dockerfile crean una capa, pero sólo las líneas que incluyen las instrucciones **RUN**, **ADD** y **COPY** generan una capa con contenido (el resto de capas ocupan 0

bytes). Por lo tanto en el Dockerfile del ejemplo anterior sólo existe una capa con la instrucción **ADD**, que es la que se ha descargado cuando hemos hecho `docker pull alpine`.

Más adelante profundizaremos un poco más sobre este tema, cuando veamos la sección sobre cómo crear nuestras propias imágenes con Dockerfiles.

Ejemplo

Vamos a observar la salida que obtenemos cuando descargamos la imagen de `ubuntu`:

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu
7ddbc47eeb70: Pull complete ①
c1bbdc448b72: Pull complete ②
8c3b70e39044: Pull complete ③
45d437916d57: Pull complete ④
Digest: sha256:6e9f67fa63b0323e9a1e587fd71c561ba48a034504fb804fd26fd8800039835d
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

- ① Se descarga la 1ª capa
- ② Se descarga la 2ª capa
- ③ Se descarga la 3ª capa
- ④ Se descarga la 4ª capa

En este ejemplo podemos ver como ha sido necesario descargar cuatro capas para la imagen de `ubuntu`.

Si observamos el archivo `Dockerfile` de la imagen de `ubuntu` encontramos lo siguiente:

```

FROM scratch
ADD ubuntu-bionic-core-cloudimg-amd64-root.tar.gz /
# verify that the APT lists files do not exist
RUN [ -z "$(apt-get indextargets)" ]
# (see https://bugs.launchpad.net/cloud-images/+bug/1699913)

# a few minor docker-specific tweaks
# see https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap
RUN set -xe \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L40-L48
&& echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
&& echo 'exit 101' >> /usr/sbin/policy-rc.d \
&& chmod +x /usr/sbin/policy-rc.d \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L54-L56
&& dpkg-divert --local --rename --add /sbin/initctl \
&& cp -a /usr/sbin/policy-rc.d /sbin/initctl \
&& sed -i 's/^exit.*/exit 0/' /sbin/initctl \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L71-L78
&& echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L85-L105
&& echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /etc/apt/apt.conf.d/docker-clean \
&& echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >> /etc/apt/apt.conf.d/docker-clean \
&& echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >> /etc/apt/apt.conf.d/docker-clean \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L109-L115
&& echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-languages \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L118-L130
&& echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes \
\
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec0740716226fadb3735c/contrib/mkimage/debootstrap#L134-L151
&& echo 'Apt::AutoRemove::SuggestsImportant "false";' > /etc/apt/apt.conf.d/docker-autoremove-suggests

# make systemd-detect-virt return "docker"
# See: https://github.com/systemd/systemd/blob/aa0c34279ee40bce2f9681b496922dedbadfca19/src/basic/virt.c#L434
RUN mkdir -p /run/systemd && echo 'docker' > /run/systemd/container

CMD ["/bin/bash"]

```

Podemos ver que hay una instrucción de tipo **ADD** y tres de tipo **RUN**, por lo tanto tendremos que descargar cuatro capas.

Ejemplo

Una capa puede ser reutilizada por otras imágenes para ahorrar espacio.

En el siguiente ejemplo podemos ver cómo una misma capa puede ser compartida por varias imágenes.

```
$ docker pull php
Using default tag: latest
latest: Pulling from library/php
8d691f585fa8: Pull complete
cba12d3fd8b1: Pull complete
cda54d6474c8: Pull complete
412447ed0729: Pull complete
be73f2c7a508: Pull complete
ccea27a56d46: Pull complete
e652349e8aa0: Pull complete
35d8aa4ba783: Pull complete
dd9d93e7999d: Pull complete
66f02e7e72bd: Pull complete
Digest: sha256:bcb652b60a7c3d9ca36d7bea93573fe052e18487b4ccd39dc9455222f03252eb
Status: Downloaded newer image for php:latest
docker.io/library/php:latest
```

```
$ docker pull php:apache
apache: Pulling from library/php
8d691f585fa8: Already exists ①
cba12d3fd8b1: Already exists ②
cda54d6474c8: Already exists ③
412447ed0729: Already exists ④
84de6fc539c3: Pull complete
d67567ed6145: Pull complete
22ca6c438da4: Pull complete
aaaf25e57dd6: Pull complete
fbccd385090a: Pull complete
15b403f621d7: Pull complete
1cae2d7071d0: Pull complete
5c0cbd6e0573: Pull complete
1b48a6c1e889: Pull complete
855d31502496: Pull complete
Digest: sha256:2dd2c5f682306c0738f2ac826fae0c98f467a447ef2a9d6bc4ee86eed97eefc6
Status: Downloaded newer image for php:apache
docker.io/library/php:apache
```

- ① Esta capa no es necesario descargarla porque ya se descargó previamente para otra imagen. Ocurre lo mismo para 2, 3 y 4.

Referencia:

- [Digging into Docker layers](#)
- [About storage drivers - Docker](#)

4.7. Consultar el historial de una imagen

Usage: `docker history [OPTIONS] IMAGE`

Show the history of an image

Options:

<code>--format string</code>	Pretty-print images using a Go template
<code>-H, --human</code>	Print sizes and dates in human readable format (default true)
<code>--no-trunc</code>	Don't truncate output
<code>-q, --quiet</code>	Only show numeric IDs

Ejemplo

Vamos a consultar el historial de la imagen **alpine**.

```
$ docker history alpine
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
965ea09ff2eb	3 weeks ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B	
<missing>	3 weeks ago	/bin/sh -c #(nop) ADD file:fe1f09249227e2da2...	5.55MB	

Ejemplo

Vamos a consultar el historial de la imagen **ubuntu**.

```
$ docker history ubuntu
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
775349758637	11 days ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0B	
<missing>	11 days ago	/bin/sh -c mkdir -p /run/systemd && echo 'do...	7B	
<missing>	11 days ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /...	745B	
<missing>	11 days ago	/bin/sh -c [-z "\$(apt-get indextargets)"]	987kB	
<missing>	11 days ago	/bin/sh -c #(nop) ADD file:a48a5dc1b9dbfc632...	63.2MB	

Referencia:

- <https://docs.docker.com/engine/reference/commandline/history/>

4.8. Mostrar las imágenes que tenemos descargadas

```
$ docker images --help
```

```
Usage:  docker images [OPTIONS] [REPOSITORY[:TAG]]
```

List images

Options:

-a, --all	Show all images (default hides intermediate images)
--digests	Show digests
-f, --filter filter	Filter output based on conditions provided
--format string	Pretty-print images using a Go template
--no-trunc	Don't truncate output
-q, --quiet	Only show numeric IDs

Ejemplo

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	775349758637	11 days ago	64.2MB
httpd	latest	d3017f59d5e2	12 days ago	165MB
alpine	latest	965ea09ff2eb	3 weeks ago	5.55MB
mariadb	latest	a9e108e8ee8a	3 weeks ago	356MB
mediawiki	latest	1d774b717f24	3 weeks ago	733MB
joomla	latest	37b651a98b60	3 weeks ago	457MB
mysql	latest	c8ee894bd2bd	3 weeks ago	456MB
hello-world	latest	fce289e99eb9	10 months ago	1.84kB

4.9. Mostrar las imágenes intermedias (ocultas por defecto)

También podemos mostrar las imágenes intermedias que se han ido generando en nuestro equipo cada vez que hemos creado una nueva imagen a partir de un archivo Dockerfile.

Estas imágenes intermedias están **ocultas por defecto**.

Para poder mostrarlas necesitaremos utilizar la opción **-a**.

```
$ docker images -a
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	775349758637	10 days ago	64.2MB
alpine	latest	965ea09ff2eb	2 weeks ago	5.55MB
<none>	<none>	2ca708c1c9cc	7 weeks ago	64.2MB
...				



- Las imágenes que aparecen como `<none>:<none>` con el comando `docker images` son *dangling images* (imágenes colgadas). Estas imágenes habrá que borrarlas para liberar espacio porque ya no se usarán.
- Las imágenes que aparecen como `<none>:<none>` con el comando `docker images -a` son imágenes intermedias. Estas imágenes no podemos borrarlas. Cuando se borre la imagen de la que dependen, estas imágenes también se borran.

Para conocer más detalles sobre este tema se recomienda la lectura del siguiente artículo: [What are Docker <none>:<none> images?](#)

4.10. Mostrar el ID de las imágenes

Para mostrar el ID de las imágenes utilizamos la opción `-q`.

```
$ docker images -q
```

El siguiente comando mostraría el ID de todas las imágenes que tenemos, incluyendo las **imágenes intermedias**. Este comando nos será útil más adelante para eliminar todas las imágenes que tenemos en nuestro *host*, incluyendo las *dangling images*.

```
$ docker images -aq
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/images/>

4.11. Eliminar imágenes

```
$ docker rmi --help
Usage:  docker rmi [OPTIONS] IMAGE [IMAGE...]

Remove one or more images

Options:
  -f, --force          Force removal of the image
  --no-prune           Do not delete untagged parents
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/rmi/>

4.12. Eliminar una imagen por su nombre (REPOSITORY)

Ejemplo:

En este ejemplo vamos a eliminar la imagen `ubuntu`.

```
$ docker rmi ubuntu

Untagged: ubuntu:latest
Untagged: ubuntu@sha256:6e9f67fa63b0323e9a1e587fd71c561ba48a034504fb804fd26fd8800039835d
Deleted: sha256:775349758637aff77bf85e2ff0597e86e3e859183ef0baba8b3e8fc8d3cba51c
Deleted: sha256:4fc26b0b0c6903db3b4fe96856034a1bd9411ed963a96c1bc8f03f18ee92ac2a
Deleted: sha256:b53837dafdd21f67e607ae642ce49d326b0c30b39734b6710c682a50a9f932bf
Deleted: sha256:565879c6effe6a013e0b2e492f182b40049f1c083fc582ef61e49a98dca23f7e
Deleted: sha256:cc967c529ced563b7746b663d98248bc571afdb3c012019d7f54d6c092793b8b
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/rmi/>

4.13. Eliminar una imagen por su ID

El `IMAGE ID` puede ser:

- el identificador largo del contenedor (64 caracteres).
- el identificador corto del contenedor (16 caracteres).
- también es posible utilizar solamente los primeros caracteres del identificador, siempre que no existan contenedores que empiecen con esos caracteres.

Ejemplo:

En este ejemplo estamos eliminando la imagen de `alpine:3.7` que tiene como `IMAGE ID` el valor `6d1ef012b567`.

```
$ docker rmi 6d1ef012b567
```

En este caso podría haber utilizado los primeros caracteres del identificador.

```
$ docker rmi 6d
```

También es posible eliminar varias imágenes de una vez indicando una lista con todos los ID de las imágenes que queremos eliminar.

```
$ docker rmi 6d1ef012b567 965ea09ff2eb 4c9d84aee9f
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/rmi/>

4.14. Eliminar todas las imágenes que tenemos descargadas

Para eliminar todas las imágenes podemos utilizar el siguiente comando:

```
$ docker rmi $(docker images -aq)
```

Recuerda que la opción `docker images -aq` muestra el ID de todas las imágenes del *host*, incluyendo las imágenes intermedias.



```
$ docker images -aq
```

```
4c9d84aee9f  
965ea09ff2eb  
6b7ac1689533  
a9e108e8ee8a  
1d774b717f24  
37b651a98b60
```

5. Creación y ejecución de contenedores Docker

```
docker run --help
```

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

Options:

--add-host list	Add a custom host-to-IP mapping (host:ip)
-a, --attach list	Attach to STDIN, STDOUT or STDERR
--blkio-weight uint16	Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)
--blkio-weight-device list	Block IO weight (relative device weight) (default [])
--cap-add list	Add Linux capabilities
--cap-drop list	Drop Linux capabilities
--cgroup-parent string	Optional parent cgroup for the container
--cidfile string	Write the container ID to the file
--cpu-period int	Limit CPU CFS (Completely Fair Scheduler) period
--cpu-quota int	Limit CPU CFS (Completely Fair Scheduler) quota
--cpu-rt-period int	Limit CPU real-time period in microseconds
--cpu-rt-runtime int	Limit CPU real-time runtime in microseconds
-c, --cpu-shares int	CPU shares (relative weight)
--cpus decimal	Number of CPUs
--cpuset-cpus string	CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems string	MEMs in which to allow execution (0-3, 0,1)
-d, --detach	Run container in background and print container ID
--detach-keys string	Override the key sequence for detaching a container
--device list	Add a host device to the container
--device-cgroup-rule list	Add a rule to the cgroup allowed devices list
--device-read-bps list	Limit read rate (bytes per second) from a device (default [])
--device-read-iops list	Limit read rate (IO per second) from a device (default [])
--device-write-bps list	Limit write rate (bytes per second) to a device (default [])
--device-write-iops list	Limit write rate (IO per second) to a device (default [])
--disable-content-trust	Skip image verification (default true)
--dns list	Set custom DNS servers
--dns-option list	Set DNS options
--dns-search list	Set custom DNS search domains
--domainname string	Container NIS domain name
--entrypoint string	Overwrite the default ENTRYPOINT of the image
-e, --env list	Set environment variables
--env-file list	Read in a file of environment variables
--expose list	Expose a port or a range of ports
--gpus gpu-request	GPU devices to add to the container ('all' to pass all GPUs)
--group-add list	Add additional groups to join
--health-cmd string	Command to run to check health
--health-interval duration	Time between running the check (ms s m h) (default 0s)
--health-retries int	Consecutive failures needed to report unhealthy
--health-start-period duration	Start period for the container to initialize before starting health-retries
countdown (ms s m h) (default 0s)	
--health-timeout duration	Maximum time to allow one check to run (ms s m h) (default 0s)
--help	Print usage
-h, --hostname string	Container host name
--init	Run an init inside the container that forwards signals and reaps processes
-i, --interactive	Keep STDIN open even if not attached
--ip string	IPv4 address (e.g., 172.30.100.104)
--ip6 string	IPv6 address (e.g., 2001:db8::33)
--ipc string	IPC mode to use
--isolation string	Container isolation technology
--kernel-memory bytes	Kernel memory limit
-l, --label list	Set meta data on a container
--label-file list	Read in a line delimited file of labels

--link list	Add link to another container
--link-local-ip list	Container IPv4/IPv6 link-local addresses
--log-driver string	Logging driver for the container
--log-opt list	Log driver options
--mac-address string	Container MAC address (e.g., 92:d0:c6:0a:29:33)
-m, --memory bytes	Memory limit
--memory-reservation bytes	Memory soft limit
--memory-swap bytes	Swap limit equal to memory plus swap: '-1' to enable unlimited swap
--memory-swappiness int	Tune container memory swappiness (0 to 100) (default -1)
--mount mount	Attach a filesystem mount to the container
--name string	Assign a name to the container
--network network	Connect a container to a network
--network-alias list	Add network-scoped alias for the container
--no-healthcheck	Disable any container-specified HEALTHCHECK
--oom-kill-disable	Disable OOM Killer
--oom-score-adj int	Tune host's OOM preferences (-1000 to 1000)
--pid string	PID namespace to use
--pids-limit int	Tune container pids limit (set -1 for unlimited)
--privileged	Give extended privileges to this container
-p, --publish list	Publish a container's port(s) to the host
-P, --publish-all	Publish all exposed ports to random ports
--read-only	Mount the container's root filesystem as read only
--restart string	Restart policy to apply when a container exits (default "no")
--rm	Automatically remove the container when it exits
--runtime string	Runtime to use for this container
--security-opt list	Security Options
--shm-size bytes	Size of /dev/shm
--sig-proxy	Proxy received signals to the process (default true)
--stop-signal string	Signal to stop a container (default "SIGTERM")
--stop-timeout int	Timeout (in seconds) to stop a container
--storage-opt list	Storage driver options for the container
--sysctl map	Sysctl options (default map[])
--tmpfs list	Mount a tmpfs directory
-t, --tty	Allocate a pseudo-TTY
--ulimit ulimit	Ulimit options (default [])
-u, --user string	Username or UID (format: <name uid>[:<group gid>])
--userns string	User namespace to use
--uts string	UTS namespace to use
-v, --volume list	Bind mount a volume
--volume-driver string	Optional volume driver for the container
--volumes-from list	Mount volumes from the specified container(s)
-w, --workdir string	Working directory inside the container

Referencia:

- [Docker run reference. Docker](#)
- <https://docs.docker.com/v17.12/engine/reference/commandline/run/>

5.1. Hello World!

En Docker Hub existe una [imagen oficial que contiene un ejemplo de "Hello World!"](#). Este contenedor lo único que hace es mostrar un mensaje de bienvenida.

El contenido del [archivo Dockerfile](#) de la imagen *hello-world* es el siguiente:

```
FROM scratch ①
COPY hello / ②
CMD ["/hello"] ③
```

- ① Esta instrucción indica que está utilizando la imagen *scratch* como imagen base. Esta imagen es una imagen especial que se corresponde con una **imagen vacía**.
- ② Copia el archivo *hello* al directorio raíz del sistema de archivos de la imagen. El archivo *hello* es un archivo binario que podemos ver en el mismo repositorio de GitHub donde está alojado el Dockerfile.
- ③ Indica que el contenedor ejecutará esta instrucción cuando se inicie.

Vamos a ejecutar nuestro primer contenedor:

```
$ docker run hello-world
```

Veamos con detalle qué es lo que ha ocurrido.

1. En primer lugar busca si la imagen *hello-world* existe en el repositorio local de imágenes de nuestro equipo. Como no la ha encontrado, la ha descargado automáticamente de Docker Hub. Por lo tanto, hemos visto que **no es necesario descargar la imagen previamente con *docker pull***.
2. En segundo lugar se crea un contenedor a partir de la imagen *hello-world* y se inicia.
3. Se ejecuta el archivo binario *hello* y cuando finaliza la ejecución el contenedor se detiene.

Listar los contenedores que están en ejecución

Para consultar los contenedores que están en ejecución actualmente utilizamos el siguiente comando:

```
$ docker ps
```

Podemos ver que actualmente no hay ningún contenedor en ejecución. El comando anterior nos devuelve la siguiente salida.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Listar todos los contenedores (los que están en ejecución y los que están detenidos)

Para poder ver los contenedores que están detenidos utilizamos la opción **-a**.

```
$ docker ps -a
```

Ahora obtendremos la siguiente salida.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c48138039ade	hello-world	"/hello"	12 seconds ago	Exited (0) 12 seconds ago		docha_can

- CONTAINER ID (**c48138039ade**): Es el identificador único del contenedor.
- IMAGE (**hello-world**): Es el nombre de la imagen utilizada para instanciar el contenedor.
- COMMAND (**/hello**): Instrucción que ejecuta el contenedor. En este caso es la que aparece en el archivo Dockerfile.
- CREATED (**12 seconds ago**): Cuando fue creado el contenedor.
- STATUS (**Exited (0) 12 seconds ago**): El estado actual del contenedor. En este caso el contenedor se ha ejecutado y se ha detenido, finalizando su ejecución con el **código 0** que indica que ha finalizado **sin errores**.
- PORTS: Indica los puertos expuestos por el contenedor. En este caso está vacío porque no exponen ningún puerto.
- NAMES (**docha_can**): Nombre del contenedor. Si no le asignamos un nombre durante la creación del contenedor, Docker creará un nombre de forma automática.

Referencia:

- <https://docs.docker.com/engine/reference/run/>
- <https://docs.docker.com/engine/reference/commandline/ps/>

5.2. Creación de un contenedor para ejecutar un comando

En este ejemplo vamos a utilizar [Alpine Linux](#), una distribución Linux muy ligera. La imagen para Docker ocupa **menos de 6 MB**.

```
$ docker pull alpine
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	965ea09ff2eb	2 weeks ago	5.55MB

El contenido del [archivo Dockerfile](#) de la imagen *alpine* es el siguiente:

```
FROM scratch ①
ADD alpine-minirootfs-3.10.3-x86_64.tar.gz / ②
CMD ["/bin/sh"] ③
```

- ① Esta instrucción indica que está utilizando la imagen *scratch* como imagen base. Esta imagen es una imagen especial que se corresponde con una **imagen vacía**.
- ② La instrucción **ADD** copia el archivo *alpine-minirootfs-3.10.3-x86_64.tar.gz* al directorio raíz del sistema de archivos de la imagen y lo descomprime. El archivo *alpine-minirootfs-3.10.3-x86_64.tar.gz* es un archivo que podemos ver en el mismo [repositorio de GitHub](#) donde está alojado el Dockerfile.
- ③ Indica que el contenedor ejecutará esta instrucción cuando se inicie.

Es posible ejecutar comandos dentro del contenedor indicando el comando después del nombre de la imagen. Veamos la sintaxis de `docker run`.

```
Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Por lo tanto, si queremos ejecutar el comando `cat /etc/os-release` dentro de un contenedor basado en la imagen `alpine` ejecutaríamos el siguiente comando.

```
$ docker run alpine cat /etc/os-release

NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.3
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```



Cuando indicamos un comando a la hora de ejecutar un comando con `docker run`, estamos reemplazando el comando que aparece definido en la instrucción `CMD` del Dockerfile, por el comando que le estamos indicando.

En este caso el contenedor ejecutará el comando que le hemos indicado (`cat /etc/os-release`) y cuando el comando termina su ejecución **el contenedor se detiene**.

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2d250ff407ef	alpine	"cat /etc/os-release"	7 seconds ago	Exited (0) 7 s ago		gifted_me

Mostrar la salida estándar (STDOUT) de un contenedor

Aunque el contenedor esté detenido podemos consultar sus registros de salida (STDOUT) con el comando `docker logs`.

La sintaxis es la siguiente:

```
$ docker logs [OPTIONS] CONTAINER
```

Donde `CONTAINER` puede ser:

- el identificador largo del contenedor (64 caracteres).
- el identificador corto del contenedor (16 caracteres).
- el nombre del contenedor.



También es posible utilizar solamente los primeros caracteres del identificador

Por ejemplo, para mostrar la salida estándar (STDOUT) del contenedor del ejemplo anterior podría haber utilizado:

```
$ docker logs 2d250ff407ef201f496e4e2f66d3e8fcdd8b91d828de60e70a29f6613161fc8a
```

```
$ docker logs 2d250ff407ef
```

```
$ docker logs gifted_me
```

```
$ docker logs 2d
```

Ejercicios

Ejecuta el siguiente comando:

```
$ docker run alpine
```

1. ¿Qué ha ocurrido en este caso?
2. ¿Qué devuelve el contenedor?
3. Comprueba el estado del contenedor (si está en ejecución o detenido).

5.3. Creación de un contenedor en modo interactivo

Para que un contenedor no se detenga al ejecutarse debemos indicarle que queremos iniciarlo en modo interactivo.

Ejemplo de creación de un contenedor con Alpine Linux

```
$ docker run -it --name alpinec alpine  
/ #
```

- **docker run** es el comando que nos permite crear un contenedor a partir de una imagen Docker.
- El parámetro **-i** nos permite interactuar con el contenedor a través de la entrada estándar STDIN.
- El parámetro **-t** nos asigna un terminal dentro del contenedor.
- Los dos parámetros **-it** nos permiten usar un contenedor como si fuese una máquina virtual tradicional.
- El parámetro **--name** nos permite asignarle un nombre a nuestro contenedor. Si no le asignamos

un nombre Docker nos asignará un nombre automáticamente.

- **alpine** es el nombre de la imagen. En primer lugar buscará la imagen en local y si no está disponible la buscará en el repositorio oficial Docker Hub.

Una vez ejecutado el comando anterior **nos aparece un *prompt* para poder interaccionar con el contenedor** que acabamos de crear.

```
/ #
```

Podemos probar a escribir algunos comandos.

```
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
```

```
/ # cat /etc/os-release

NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.3
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

Gestión de paquetes en Alpine Linux

El gestor de paquetes de [Alpine Linux](#) es **apk**. En la [documentación oficial](#) podemos encontrar más detalles sobre cómo usarlo.

Si quisiéramos instalar **nano** en el contenedor tendríamos que hacer lo siguiente.

1) Actualizar el índice de paquetes disponibles

```
apk update
```

2) Añadir el nuevo paquete al sistema.

```
apk add nano
```

Para salir del contenedor escribimos el comando **exit**.

```
exit
```

Como **no hemos iniciado el contenedor con el parámetro `--rm`**, cuando el contenedor se detiene **no se elimina y ocupa espacio en nuestro disco**. Podemos comprobarlo con el siguiente comando.

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e9af6c8dbffb	alpine	"/bin/sh"	9 seconds ago	Exited (0) 9 s ago	dazz_bra	

Eliminar un contenedor

Para eliminar el contenedor que está detenido y está ocupando espacio en nuestro disco ejecutaremos el comando **docker rm**. Este comando nos permite eliminar un contenedor indicando su ID su nombre.

Para el ejemplo anterior podríamos utilizar cualquiera de estas dos opciones.

```
$ docker rm alpinec
```

o

```
$ docker rm e9af6c8dbffb
```



A partir de este momento **siempre** que vayamos a crear un nuevo contenedor **añadiremos el parámetro `--rm`** para que cuando se detenga se elimine automáticamente.



Para **salir del contenedor y detenerlo**:

- Escribir **exit**
- Pulsar **CTRL + D**

Para **salir del contenedor SIN detenerlo**:

- Pulsar **CTRL + P + Q**

Ejercicios

1. Crea un contenedor con **Ubuntu** en modo interactivo, asígnale un nombre e incluye el parámetro `--rm`.
2. Una vez que has creado el contenedor finaliza la sesión con `exit` o `CTRL + D` para salir y detenerlo.
3. Comprueba con `docker ps -a` que el contenedor se ha eliminado automáticamente.
4. Crea un contenedor con **Ubuntu** en modo interactivo, asígnale un nombre e incluye el parámetro `--rm`.
5. Una vez que has creado el contenedor sal del contenedor sin detenerlo con `CTRL + P + Q`.
6. Comprueba con `docker ps -a` que el contenedor está en ejecución.
7. ¿Cómo puedo volver a conectarme al terminal del contenedor que está en ejecución?

5.4. attach y exec

5.4.1. attach

```
$ docker attach --help
```

```
Usage:  docker attach [OPTIONS] CONTAINER
```

```
Attach local standard input, output, and error streams to a running container
```

```
Options:
```

<code>--detach-keys string</code>	Override the key sequence for detaching a container
<code>--no-stdin</code>	Do not attach STDIN
<code>--sig-proxy</code>	Proxy all received signals to the process (default true)

Nos permite acceder al terminal de un **contenedor que está en ejecución** indicando su nombre o su ID. Tenga en cuenta que **no crea un nuevo terminal (tty)**, sino que usa el terminal original que está en ejecución de modo que **si salimos del terminal con `exit` el contenedor se detendrá**.

Ejemplo

Ejecutamos un contenedor en modo *detached* (`-d`) y le añadimos la opción (`-it`) para poder interactuar con él a través de un terminal.

Vamos a ejecutar en el contenedor el comando `/usr/bin/top` en modo *batch* (`-b`) para que siga ejecutándose en segundo plano. Si no utilizásemos el modo *batch* el contenedor se detendría una vez que finaliza la ejecución del comando.

```
$ docker run -dit \  
--rm \  
--name topdemo \  
ubuntu /usr/bin/top -b
```

Comprobamos que el contenedor está ejecutándose en segundo plano.

```
$ docker ps
```

Ahora podríamos acceder al terminal el contenedor que está en ejecución con **attach**.

```
$ docker attach topdemo
```



- Si salimos pulsando **CTRL + C** el contenedor se detendrá y finalizará su ejecución.
- Si salimos pulsando **CTRL + P + Q** el contenedor seguirá ejecutándose en background.

Ejercicios

Crea los siguientes contenedores:

```
$ docker run -dit \  
--rm \  
--name topdemo1 \  
ubuntu /usr/bin/top -b  
  
$ docker run -dit \  
--rm \  
--name topdemo2 \  
ubuntu /usr/bin/top -b
```

1. Accede al terminal del contenedor **topdemo1** con **attach** y una vez dentro finaliza la sesión pulsando **CTRL + C**.
2. Comprueba los contenedores que están en ejecución.
3. Accede al terminal del contenedor **topdemo2** con **attach** y una vez dentro finaliza la sesión pulsando **CTRL + P + Q**.
4. Comprueba los contenedores que están en ejecución.

5.4.2. `exec`

```
docker exec --help
```

```
Usage:  docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Run a command in a running container

Options:

<code>-d, --detach</code>	Detached mode: run command in the background
<code>--detach-keys string</code>	Override the key sequence for detaching a container
<code>-e, --env list</code>	Set environment variables
<code>-i, --interactive</code>	Keep STDIN open even if not attached
<code>--privileged</code>	Give extended privileges to the command
<code>-t, --tty</code>	Allocate a pseudo-TTY
<code>-u, --user string</code>	Username or UID (format: <name uid>[:<group gid>])
<code>-w, --workdir string</code>	Working directory inside the container

Nos permite **ejecutar un comando en un contenedor que está en ejecución** indicando su nombre o su ID. `exec` ejecuta el comando en un proceso nuevo, asignándonos un nuevo terminal.

Esto significa que **si salimos del contenedor con `exit`, el contenedor no detendrá su ejecución.**

Cuando queramos conectarnos a un contenedor que está en ejecución podemos abrir un nuevo terminal con `exec`.

```
$ docker exec -it container_name /bin/sh
```

Ejemplo

Vamos a utilizar el ejemplo anterior.

Ejecutaremos el comando `/usr/bin/top` en modo *batch* (`-b`) para que siga ejecutándose en segundo plano. Si no utilizásemos el modo *batch* el contenedor se detendría una vez que finaliza la ejecución del comando.

```
$ docker run -dit \
--rm \
--name topdemo \
ubuntu /usr/bin/top -b
```

Comprobamos que el contenedor está ejecutándose en segundo plano.

```
$ docker ps
```

Ahora vamos a ejecutar el comando `/bin/bash` para crear un nuevo terminal sobre el contenedor.


```
$ docker exec -it topdemo /bin/sh
```

Una vez ejecutado el comando anterior **nos aparece un *prompt* para poder interaccionar con el contenedor** que acabamos de crear.

```
/ #
```

Si ejecutamos **ps aux** para ver los procesos que están en ejecución dentro del contenedor veremos lo siguiente.

```
# ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	36480	3032	pts/0	Ss+	10:25	0:00	/usr/bin/top -b
root	16	0.6	0.0	4624	816	pts/1	Ss	10:28	0:00	/bin/sh
root	23	0.0	0.1	34396	2816	pts/1	R+	10:28	0:00	ps aux

Vemos como el proceso con **PID 1** es el que está ejecutando el comando **/usr/bin/top -b** y el proceso con **PID 16** es el del comando **/bin/sh** que es el que hemos ejecutado con **docker exec**.

Referencia:

- <https://docs.docker.com/engine/reference/commandline/attach/>
- <https://docs.docker.com/engine/reference/commandline/exec/>
- [Docker — attach vs exec commands](#)

5.5. Eliminar contenedores

5.5.1. **rm**

```
$ docker rm --help
```

```
Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Remove one or more containers

Options:

- | | |
|---------------|---|
| -f, --force | Force the removal of a running container (uses SIGKILL) |
| -l, --link | Remove the specified link |
| -v, --volumes | Remove the volumes associated with the container |

Eliminar un contenedor que está detenido

```
$ docker rm httpdc
```

Eliminar un contenedor que está en ejecución

```
$ docker rm -f httpdc
```

Al utilizar el parámetro **-f** puedo eliminar un contenedor que está en ejecución. De otro modo, tendría que detener el contenedor y luego eliminarlo.

Eliminar todos los contenedores que están detenidos

```
$ docker rm $(docker ps -aq)
```

Eliminar todos los contenedores (en ejecución y detenidos)

```
$ docker rm -f $(docker ps -aq)
```

Referencia:

- <https://docs.docker.com/engine/reference/commandline/rm/>

Ejercicios

1. Elimina todos los contenedores que tengas en ejecución y los que estén detenidos.

5.6. stop y start

5.6.1. stop

```
docker start --help
```

Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]

Start one or more stopped containers

Options:

-a, --attach	Attach STDOUT/STDERR and forward signals
--detach-keys string	Override the key sequence for detaching a container
-i, --interactive	Attach container's STDIN

5.6.2. start

```
docker stop --help
```

```
Usage:  docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

Stop one or more running containers

Options:

```
-t, --time int    Seconds to wait for stop before killing it (default 10)
```

Ejercicios

Crea los siguientes contenedores:

```
$ docker run -dit \  
--name topdemo1 \  
ubuntu /usr/bin/top -b
```

```
$ docker run -dit \  
--rm \  
--name topdemo2 \  
ubuntu /usr/bin/top -b
```

1. Detenga la ejecución de los dos contenedores con **docker stop**.
2. Comprueba los contenedores que están detenidos. ¿Existe alguno? ¿Por qué?
3. Inicia los contenedores que estén detenidos con **docker start**.
4. Comprueba los contenedores que están en ejecución.
5. Elimina todos los contenedores que tengas en ejecución y los que estén detenidos.

5.7. Creación de un contenedor en segundo plano

Hasta este momento hemos visto dos formas usar los contenedores:

1. Creamos un contenedor para ejecutar un comando dentro de él, esperamos a que finalice el comando y cuando el comando finaliza el contenedor se detiene.
2. Creamos un contenedor en modo interactivo, donde podemos acceder a un terminal y ejecutar comandos en él.

Existe otra posibilidad, que además es la más utilizada, consiste en la **ejecución de un contenedor en segundo plano mientras que éste ejecuta una aplicación en primer plano**.

Para ejecutar un contenedor en segundo plano se utiliza la opción **-d** (*detach*).

Ejemplo

Vamos a ejecutar un contenedor que ejecuta un servidor web en primer plano. Mientras que el servidor web esté en ejecución el contenedor también lo estará.

Utilizaremos la imagen oficial [httpd](#), que es de Apache HTTP Server.

```
$ docker run -d \
--rm \
--name httpdc \
httpd
```

Comprobamos que el contenedor está en ejecución:

```
$ docker ps
```

Para ver los registros de salida (STDOUT) del contenedor podemos utilizar el comando `docker logs`. Si utilizamos la opción `-f` la salida se irá actualizando automáticamente.

```
$ docker logs -f httpdc
```

Con el comando `docker inspect` podemos obtener información sobre el contenedor.

```
$ docker inspect httpdc
```



El comando `docker inspect` nos permite inspeccionar objetos Docker como: contenedores, imágenes y redes.

Si buscamos la dirección IP del contenedor vemos que es una dirección privada del tipo 172.17.0.x.

```
$ docker inspect httpdc | grep IPAddress

"IPAddress": "172.17.0.2"
```

El contenedor está en la red *bridge* y por lo tanto sólo es accesible desde el servidor que está ejecutando el servicio de Docker o desde otro contenedor de la misma red (**En Linux, en macOS tiene otro comportamiento**).

```
# curl http://172.17.0.2

<html><body><h1>It works!</h1></body></html>
```

Por lo tanto, para acceder al servidor web desde nuestra red y no sólo desde el servidor que está

ejecutando el servicio de Docker tendremos que **exponer los puertos**.

Eliminamos el contenedor.

```
$ docker rm -f httpdc
```

5.8. Exponer los puertos

Consiste en reservar un puerto del servidor de Docker con el objetivo de redirigir las peticiones a un puerto específico de un contenedor.

Existen dos opciones:

- **-p**

Con esta opción tenemos que indicar qué puerto local de nuestra máquina vamos a redireccionar con el puerto del contenedor. Si el puerto local que indicamos ya está en uso, obtendremos un error y el contenedor no se creará.

La sintaxis para indicar los puertos será **puerto_local:puerto_contenedor**

En el siguiente ejemplo vamos a redireccionar el puerto 81 de nuestra máquina con el puerto 80 del contenedor.

```
$ docker run -d \
--rm \
--name httpdc \
-p 81:80 \
httpd
```

Comprobamos que el contenedor está ejecutándose.

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3ac8d440ab2	httpd	"httpd-foreground"	6 seconds ago	Up 5 seconds	0.0.0.0:81->80/tcp	httpdc

Comprobamos que podemos acceder al contenido del contenedor desde un navegador web accediendo a la URL <http://localhost:81>, o desde el terminal con el comando **curl**.

```
$ curl http://localhost:81
```

```
<html><body><h1>It works!</h1></body></html>
```

- **-P**

Con esta opción no tenemos que indicar el puerto local, será **seleccionado aleatoriamente entre los puertos que estén libres**. El puerto del contenedor con el que hacemos la redirección estará definido en el archivo Dockerfile con el que se ha creado la imagen del contenedor. Tenga en cuenta que en el archivo Dockerfile se pueden exponer varios puertos a la vez.

En el siguiente ejemplo se seleccionará un puerto aleatorio de nuestra máquina y se redirigirá al puerto 80 del contenedor.

Podemos consultar el [archivo Dockerfile de la imagen httpd en Docker Hub](#) y comprobar que el puerto que expone esta imagen es el 80. Los puertos aparecen definidos con la instrucción **EXPOSE**.

```
$ docker run -d \
--rm \
--name httpdc \
-P \
httpd
```

Para conocer cuál es el puerto aleatorio de nuestra máquina que se ha utilizado ejecutamos el siguiente comando.

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
96922862a483	httpd	"httpd-foreground"	3 seconds ago	Up 1 second	0.0.0.0:32770->80/tcp	httpdc

Aquí podemos ver que el puerto aleatorio que se ha seleccionado es el puerto 32770.

Comprobamos que podemos acceder al contenido del contenedor desde un navegador web accediendo a la URL <http://localhost:32770>, o desde el terminal con el comando **curl**.

```
$ curl http://localhost:32770
```

```
<html><body><h1>It works!</h1></body></html>
```

Ejercicios

1. Crea un contenedor con la imagen oficial de [nginx](#) y haz una redirección del puerto 8080 de tu máquina local con el puerto 80 del contenedor.
2. ¿Cómo podría sustituir la página de bienvenida de Nginx por una página que está en nuestro equipo local?

5.9. Copiar archivos/carpetas

5.9.1. cp

```
$ docker cp --help
```

```
Usage:  docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-  
        docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH
```

Copy files/folders between a container and the local filesystem

Use '-' as the source to read a tar archive from stdin and extract it to a directory destination in a container.
Use '-' as the destination to stream a tar archive of a container source to stdout.

Options:

-a, --archive Archive mode (copy all uid/gid information)
-L, --follow-link Always follow symbol link in SRC_PATH

Ejercicios

1. Copia el archivo `/usr/share/nginx/html/index.html` que está en el contenedor nginx del ejercicio anterior, a tu equipo local.
2. Modifica el contenido del archivo `index.html`.
3. Copia el archivo `index.html` que acabas de modificar a la ruta `/usr/share/nginx/html/` del contenedor nginx del ejercicio anterior.

5.10. Crear un contenedor con un volumen (de tipo *bind mount*)

Para conocer más detalles sobre la gestión de volúmenes se recomienda ver la sección [almacenamiento en Docker](#).

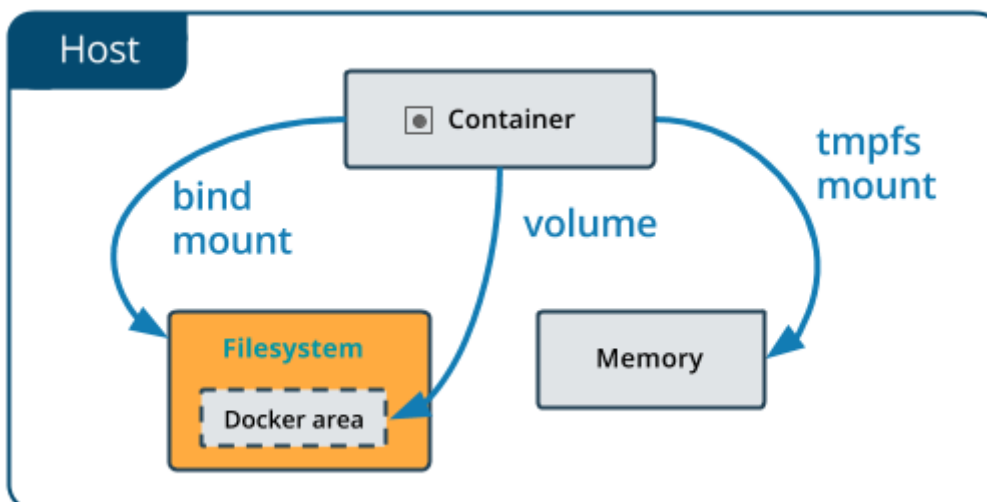


Figure 14. Use bind mounts. Imagen de [Docker.com](#)

En este ejemplo vamos a utilizar un volumen de tipo *bind mount*, que será **un directorio de nuestra máquina *host* que vamos a montar en un directorio dentro del contenedor**. El directorio que vamos a indicar dentro del contenedor no tiene por qué existir previamente.

Con *bind mount* podemos montar archivos o directorios.

¿Cuándo sería apropiado utilizar un volumen de tipo *bind mount*?

- Para compartir archivos de configuración entre la máquina *host* y el contenedor.
- Para compartir el código de las aplicaciones entre la máquina *host* y el contenedor en un entorno de desarrollo.

Para montar un directorio podemos utilizar los *flags* `-v` o `--mount`. En nuestros ejemplos utilizaremos `-v`.

En el caso de los *bind mounts* tendremos tres campos separados por dos puntos (`:`) y tendrán el siguiente orden:

- En primer lugar se indica el archivo o directorio de la máquina *host*.
- En segundo lugar se indica en qué archivo o directorio lo vamos a montar dentro del contenedor.
- El tercer parámetro es **opcional**, y puede ser una lista separada por comas con las siguientes opciones: `ro`, `consistent`, `delegated`, `cached`, `z` y `Z`.

Por lo tanto, la sintaxis para las dos formas de crear un *bind mount* serán:

- `path_directorio_host:path_directorio_contenedor`
- `path_directorio_host:path_directorio_contenedor:ro`

Ejemplos de cómo ejecutar un contenedor con un *bind mount*

En los siguientes ejemplos vamos a montar el directorio `/home/josejuan/target` de nuestra máquina local en el directorio `/app` del contenedor.

Podemos hacerlo de tres formas:

1) Indicar el **path completo** del directorio que queremos montar en el contenedor.

```
$ docker run -d \
--rm \
--name devtest \
-v /home/josejuan/target:/app \
nginx
```

2) Utilizar la salida del **comando** `pwd` para construir el *path*.


```
$ docker run -d \
--rm \
--name devtest \
-v "$(pwd)/target:/app \
nginx
```

3) Utilizar la **variable** **\$PWD** para construir el *path*.

```
$ docker run -d \
--rm \
--name devtest \
-v "$PWD"/target:/app \
nginx
```

Podemos inspeccionar el contenido del contenedor para verificar que el directorio se ha creado correctamente.

```
$ docker inspect devtest

"Mounts": [
{
  "Type": "bind",
  "Source": "/home/josejuan/target",
  "Destination": "/app",
  "Mode": "",
  "RW": true,
  "Propagation": "rprivate"
}
],
```

Cuando realizamos un *bind mount* sobre un directorio del contenedor que no está vacío, **el contenido de este directorio será reemplazado** por el contenido del directorio del *host*.

Hay que tener cuidado con esto porque puede provocar comportamientos no esperados.



En el siguiente **ejemplo** estamos reemplazando el directorio */usr* del contenedor por el */tmp* del *host*, y por lo tanto el contenedor no podrá iniciarse.

```
$ docker run -d \
--rm \
--name broken-container \
-v /tmp:/usr \
nginx
```

```
docker: Error response from daemon: OCI runtime create failed: container_linux.go:346: starting container process caused "exec: \"nginx\": executable file not found in $PATH": unknown.
```

Crear *bind_mounts* de sólo lectura

Es posible indicar que el contenido del directorio que estamos montando en el contenedor sea de sólo lectura, añadiendo el *flag* *ro* a la lista de parámetros de creación del volumen.

```
$ docker run -d \
--rm \
--name devtest \
-v "$PWD"/target:/app:ro \
nginx
```

Inspeccionamos el contenido del contenedor para verificar que el directorio se ha creado correctamente.

```
$ docker inspect devtest

"Mounts": [
{
  "Type": "bind",
  "Source": "/home/josejuan/target",
  "Destination": "/app",
  "Mode": "ro",
  "RW": false,
  "Propagation": "rprivate"
}
],
```

Referencias:

- <https://docs.docker.com/storage/>

- <https://docs.docker.com/storage/volumes/>
- <https://docs.docker.com/storage/bind-mounts/>

Ejercicio

1. Crea un directorio llamado **webapp** en tu directorio actual de trabajo.
2. Crea un archivo **index.html** dentro del directorio **webapp**.
3. Edita el archivo **index.html** y añade algún contenido de prueba.
4. Crea un contenedor con el servidor web **nginx** que cumpla los siguientes requisitos:
 - Se ejecuta en modo *detached* (background).
 - El contenedor se debe eliminar cuando se detiene.
 - Redirige el puerto **80** de tu máquina *host* con el puerto **80** del contenedor.
 - Crea un volumen de tipo *bind mount* con el directorio **webapp** de tu *host* y el directorio `/usr/share/nginx/html`.

Ejercicio

Clona el siguiente repositorio en tu directorio actual de trabajo.

```
git clone https://github.com/josejuansanchez/lab-cep-awesome-docker.git
```

También puedes descargar el archivo .zip desde la siguiente URL:

<https://github.com/josejuansanchez/lab-cep-awesome-docker/archive/master.zip>

El repositorio contiene una **web estática en HTML** en el directorio **site**.

Crea un contenedor con el servidor web **nginx** que cumpla los siguientes requisitos:

- Se ejecuta en modo *detached* (background).
- El contenedor se debe eliminar cuando se detiene.
- Redirige el puerto **81** de tu máquina *host* con el puerto **80** del contenedor.
- Crea un volumen de tipo *bind mount* con el directorio **site** de tu *host* y el directorio `/usr/share/nginx/html`.

5.11. Creación de un contenedor con Apache y PHP 7.2 (en segundo plano)

En este caso vamos a utilizar la imagen oficial [php:7.2-apache](#).

Esta imagen está configurada para servir el contenido que se encuentre dentro del directorio

`/var/www/html.`

```
$ docker run -d \
--rm \
--name apache_php \
-p 80:80 \
-v "$PWD":/var/www/html \
php:7.2-apache
```

Comprobamos que el contenedor está en ejecución:

```
$ docker ps
```

Creamos el archivo `info.php` en nuestro directorio de trabajo actual con el siguiente contenido:

```
<?php
phpinfo();
?>
```

Ahora vamos a conectarnos a un terminal del contenedor para comprobar que el volumen se ha montado correctamente.

```
$ docker exec -it apache_php /bin/bash
```

Abrimos un navegador y accedemos a la URL <http://localhost/info.php> para comprobar que la página `info.php` se sirve correctamente.

También podemos hacer la comprobación desde la línea de comandos con `curl`.

```
$ curl http://localhost/info.php
```

Ejercicio propuesto (*)

1. Busca una **imagen oficial** que te permita servir una web **PHP** con el servidor **Nginx**. En caso de no encontrar ninguna, ¿qué podríamos hacer?

5.12. Creación de un contenedor con MySQL sin persistencia de datos (en segundo plano)

Vamos a utilizar la imagen oficial [mysql](#).

```
$ docker run -d \
--rm \
--name mysqlc \
-e MYSQL_ROOT_PASSWORD=root \
-p 3306:3306 \
mysql:5.7.28
```

Abrimos un terminal en el contenedor para interactuar con él.

```
$ docker exec -it mysqlc /bin/bash
```

Una vez que estamos dentro del contenedor nos conectamos desde la consola de MySQL.

```
# mysql -u root -p
```

Creamos una nueva base de datos con los siguientes datos.

```

DROP DATABASE IF EXISTS tienda;
CREATE DATABASE tienda CHARSET utf8mb4;
USE tienda;

CREATE TABLE fabricante (
    codigo INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL
);

CREATE TABLE producto (
    codigo INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    precio DOUBLE NOT NULL,
    codigo_fabricante INT UNSIGNED NOT NULL,
    FOREIGN KEY (codigo_fabricante) REFERENCES fabricante(codigo)
);

INSERT INTO fabricante VALUES(1, 'Asus');
INSERT INTO fabricante VALUES(2, 'Lenovo');
INSERT INTO fabricante VALUES(3, 'Hewlett-Packard');
INSERT INTO fabricante VALUES(4, 'Samsung');
INSERT INTO fabricante VALUES(5, 'Seagate');
INSERT INTO fabricante VALUES(6, 'Crucial');
INSERT INTO fabricante VALUES(7, 'Gigabyte');
INSERT INTO fabricante VALUES(8, 'Huawei');
INSERT INTO fabricante VALUES(9, 'Xiaomi');

INSERT INTO producto VALUES(1, 'Disco duro SATA3 1TB', 86.99, 5);
INSERT INTO producto VALUES(2, 'Memoria RAM DDR4 8GB', 120, 6);
INSERT INTO producto VALUES(3, 'Disco SSD 1 TB', 150.99, 4);
INSERT INTO producto VALUES(4, 'GeForce GTX 1050Ti', 185, 7);
INSERT INTO producto VALUES(5, 'GeForce GTX 1080 Xtreme', 755, 6);
INSERT INTO producto VALUES(6, 'Monitor 24 LED Full HD', 202, 1);
INSERT INTO producto VALUES(7, 'Monitor 27 LED Full HD', 245.99, 1);
INSERT INTO producto VALUES(8, 'Portátil Yoga 520', 559, 2);
INSERT INTO producto VALUES(9, 'Portátil Ideapd 320', 444, 2);
INSERT INTO producto VALUES(10, 'Impresora HP Deskjet 3720', 59.99, 3);
INSERT INTO producto VALUES(11, 'Impresora HP Laserjet Pro M26nw', 180, 3);

```

Comprobamos que la base de datos se ha creado correctamente.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_tienda |
+-----+
| fabricante        |
| producto          |
+-----+
2 rows in set (0.00 sec)
```

Comprobamos que las tablas tienen datos.

```
mysql> SELECT * FROM fabricante;
+-----+-----+
| codigo | nombre          |
+-----+-----+
| 1      | Asus            |
| 2      | Lenovo          |
| 3      | Hewlett-Packard |
| 4      | Samsung         |
| 5      | Seagate         |
| 6      | Crucial         |
| 7      | Gigabyte        |
| 8      | Huawei          |
| 9      | Xiaomi          |
+-----+-----+
9 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM producto;
+-----+-----+-----+-----+
| codigo | nombre                                | precio | codigo_fabricante |
+-----+-----+-----+-----+
| 1      | Disco duro SATA3 1TB                | 86.99  | 5                  |
| 2      | Memoria RAM DDR4 8GB                 | 120    | 6                  |
| 3      | Disco SSD 1 TB                       | 150.99 | 4                  |
| 4      | GeForce GTX 1050Ti                   | 185    | 7                  |
| 5      | GeForce GTX 1080 Xtreme               | 755    | 6                  |
| 6      | Monitor 24 LED Full HD               | 202    | 1                  |
| 7      | Monitor 27 LED Full HD               | 245.99 | 1                  |
| 8      | Porttil Yoga 520                     | 559    | 2                  |
| 9      | Porttil Ideapd 320                   | 444    | 2                  |
| 10     | Impresora HP Deskjet 3720            | 59.99  | 3                  |
| 11     | Impresora HP Laserjet Pro M26nw      | 180    | 3                  |
+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

Una vez que hemos llegado a este punto tendríamos **la base de datos almacenada dentro del sistema de archivos del contenedor**, de modo que si eliminamos el contenedor y volvemos a crear

uno nuevo con el mismo comando, **no tendríamos acceso a la base de datos**. Vamos a comprobarlo.

Comprobamos que el contenedor está en ejecución.

```
$ docker ps
```

Detenemos el contenedor. Como hemos iniciado el contenedor con la opción `--rm` al detenerlo se eliminará automáticamente.

```
$ docker stop mysqlc
```

Comprobamos que el contenedor se ha detenido y se ha eliminado correctamente.

```
$ docker ps -a
```

Ejercicios

1. Instancia un nuevo contenedor que tenga el mismo nombre y los mismos parámetros que el contenedor que hemos eliminado (utiliza el mismo comando).
2. Abre un terminal en el contenedor que acabas de crear para interactuar con él.
3. Una vez dentro del contenedor inicia una conexión a la consola de MySQL.
4. Comprueba si existe la base de datos `tienda`.

5.13. Creación de un contenedor con MySQL con persistencia de datos (en segundo plano)

Vamos a utilizar la imagen oficial [mysql](#).

Existen dos formas de añadir persistencia de datos:

1. Crear un volumen interno gestionado por Docker.
2. Crear un volumen de tipo *bind mount* donde montamos un directorio de nuestra máquina local en un directorio dentro del contenedor.

Solución 1. Crear un volumen interno gestionado por Docker

```
$ docker volume create mysql_data
```



```
$ docker run -d \
--rm \
--name mysqlc \
-e MYSQL_ROOT_PASSWORD=root \
-p 3306:3306 \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Abrimos un terminal en el contenedor para interactuar con él.

```
$ docker exec -it mysqlc /bin/bash
```

Una vez que estamos dentro del contenedor nos conectamos desde la consola de MySQL.

```
# mysql -u root -p
```

Creamos una nueva base de datos con los siguientes datos.

```

DROP DATABASE IF EXISTS tienda;
CREATE DATABASE tienda CHARSET utf8mb4;
USE tienda;

CREATE TABLE fabricante (
    codigo INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL
);

CREATE TABLE producto (
    codigo INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    precio DOUBLE NOT NULL,
    codigo_fabricante INT UNSIGNED NOT NULL,
    FOREIGN KEY (codigo_fabricante) REFERENCES fabricante(codigo)
);

INSERT INTO fabricante VALUES(1, 'Asus');
INSERT INTO fabricante VALUES(2, 'Lenovo');
INSERT INTO fabricante VALUES(3, 'Hewlett-Packard');
INSERT INTO fabricante VALUES(4, 'Samsung');
INSERT INTO fabricante VALUES(5, 'Seagate');
INSERT INTO fabricante VALUES(6, 'Crucial');
INSERT INTO fabricante VALUES(7, 'Gigabyte');
INSERT INTO fabricante VALUES(8, 'Huawei');
INSERT INTO fabricante VALUES(9, 'Xiaomi');

INSERT INTO producto VALUES(1, 'Disco duro SATA3 1TB', 86.99, 5);
INSERT INTO producto VALUES(2, 'Memoria RAM DDR4 8GB', 120, 6);
INSERT INTO producto VALUES(3, 'Disco SSD 1 TB', 150.99, 4);
INSERT INTO producto VALUES(4, 'GeForce GTX 1050Ti', 185, 7);
INSERT INTO producto VALUES(5, 'GeForce GTX 1080 Xtreme', 755, 6);
INSERT INTO producto VALUES(6, 'Monitor 24 LED Full HD', 202, 1);
INSERT INTO producto VALUES(7, 'Monitor 27 LED Full HD', 245.99, 1);
INSERT INTO producto VALUES(8, 'Portátil Yoga 520', 559, 2);
INSERT INTO producto VALUES(9, 'Portátil Ideapd 320', 444, 2);
INSERT INTO producto VALUES(10, 'Impresora HP Deskjet 3720', 59.99, 3);
INSERT INTO producto VALUES(11, 'Impresora HP Laserjet Pro M26nw', 180, 3);

```

Comprobamos que la base de datos se ha creado correctamente.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_tienda |
+-----+
| fabricante        |
| producto          |
+-----+
2 rows in set (0.00 sec)
```

Comprobamos que las tablas tienen datos.

```
mysql> SELECT * FROM fabricante;
+-----+-----+
| codigo | nombre          |
+-----+-----+
| 1      | Asus            |
| 2      | Lenovo          |
| 3      | Hewlett-Packard |
| 4      | Samsung         |
| 5      | Seagate         |
| 6      | Crucial         |
| 7      | Gigabyte        |
| 8      | Huawei          |
| 9      | Xiaomi          |
+-----+-----+
9 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM producto;
+-----+-----+-----+-----+
| codigo | nombre                                | precio | codigo_fabricante |
+-----+-----+-----+-----+
| 1      | Disco duro SATA3 1TB                | 86.99  | 5                  |
| 2      | Memoria RAM DDR4 8GB                 | 120    | 6                  |
| 3      | Disco SSD 1 TB                       | 150.99 | 4                  |
| 4      | GeForce GTX 1050Ti                   | 185    | 7                  |
| 5      | GeForce GTX 1080 Xtreme               | 755    | 6                  |
| 6      | Monitor 24 LED Full HD               | 202    | 1                  |
| 7      | Monitor 27 LED Full HD               | 245.99 | 1                  |
| 8      | Porttil Yoga 520                     | 559    | 2                  |
| 9      | Porttil Ideapd 320                   | 444    | 2                  |
| 10     | Impresora HP Deskjet 3720            | 59.99  | 3                  |
| 11     | Impresora HP Laserjet Pro M26nw      | 180    | 3                  |
+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

Una vez que hemos llegado a este punto tendríamos **la base de datos almacenada en el volumen `mysql_data`**, de modo que si eliminamos el contenedor y volvemos a crear uno nuevo que haga uso

del mismo volumen, tendríamos acceso a la misma base de datos. Vamos a comprobarlo.

Comprobamos que el contenedor está en ejecución.

```
$ docker ps
```

Detenemos el contenedor. Como hemos iniciado el contenedor con la opción `--rm` al detenerlo se eliminará automáticamente.

```
$ docker stop mysqlc
```

Comprobamos que el contenedor se ha detenido y se ha eliminado correctamente.

```
$ docker ps -a
```

Ejercicios

1. Instancia un nuevo contenedor con el nombre `mysql_container_2`, que haga uso del volumen `mysql_data` donde está almacenada la base de datos `tienda`.
2. Abre un terminal en el contenedor que acabas de crear (`mysql_container_2`) para interactuar con él.
3. Una vez dentro del contenedor inicia una conexión a la consola de MySQL.
4. Comprueba que la base de datos `tienda` existe y tiene datos.
5. ¿Qué comando tendríamos que ejecutar si quisiéramos eliminar el volumen `mysql_data`?

5.14. Inicializar un contenedor de MySQL con una Base de Datos

La imagen oficial de `mysql`, ejecuta los archivos con extensión `.sh`, `.sql` y `.sql.gz` que se encuentren en el directorio `/docker-entrypoint-initdb.d`. Estos archivos serán ejecutados por orden alfabético.

Gracias a esta funcionalidad es muy sencillo importar una base de datos en nuestro contenedor de forma automática con un solo comando.

Lo único que necesitamos es crear un nuevo directorio en nuestro directorio de trabajo que contenga los scripts SQL que queremos importar en el contenedor. Este directorio local con los scripts SQL tendremos que montarlo sobre el directorio `/docker-entrypoint-initdb.d` del sistema de ficheros del contenedor. Los scripts SQL se importarán por defecto en la base de datos que se haya indicado en la variable de entorno `MYSQL_DATABASE`.

Ejemplo

El siguiente ejemplo ejecutaría en el contenedor todos los scripts SQL que se encuentren en el

directorio **sql** de nuestro directorio local de trabajo.

```
$ docker run -d \
--rm \
--name mysqlc \
-e MYSQL_ROOT_PASSWORD=root \
-p 3306:3306 \
-v mysql_data:/var/lib/mysql \
-v "$PWD/sql":/docker-entrypoint-initdb.d \
mysql:5.7.28
```

Ejercicio

1. Crear un nuevo directorio con el nombre **sql** en tu directorio de trabajo.
2. Crear un archivo con el nombre **tienda.sql** que contenga todas las sentencias SQL del ejercicio anterior y guárdalo en el directorio **sql**.
3. Instancia un nuevo contenedor montando un volumen entre el directorio **sql** de tu directorio local del trabajo con el directorio **/docker-entrypoint-initdb.d** del sistema de ficheros del contenedor.
4. Abre un terminal en el contenedor que acabas de crear para interactuar con él.
5. Una vez dentro del contenedor inicia una conexión a la consola de MySQL.
6. Comprueba que la base de datos **tienda** existe y tiene datos.

5.15. Conectar un contenedor con Adminer con MySQL

Adminer es una herramienta que permite administrar contenido de bases de datos MySQL desde un sitio web. Se distribuye en **un solo archivo PHP**.

Para este ejemplo usaremos la imagen oficial de **adminer**.

Para conectar dos contenedores podemos hacerlo de dos formas:

1. Utilizando **legacy container links** con el flag **--link**, en la **bridge network**.
2. Utilizando una **user-defined bridge network**.

Solución 1. Legacy container links con el flag --link, en la bridge network

Los enlaces permiten que los contenedores se descubran entre sí y transfieran de manera segura información sobre un contenedor a otro contenedor. Para crear un enlace se utiliza el flag **--link**.

En primer lugar debe existir un contenedor con MySQL Server.

```
$ docker run -d \
--rm \
--name mysqlc \
-p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=root \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Una vez que la instancia de MySQL está en ejecución podemos crear el contenedor con Adminer.

```
$ docker run -d \
--rm \
--link mysqlc \
-p 8080:8080 \
adminer
```

Con el flag **--link mysqlc** hemos creado un enlace entre el contenedor **mysql** y **adminer**.

En el archivo **/etc/hosts** del contenedor **adminer** se ha añadido una nueva línea que permite resolver la dirección IP del contenedor de MySQL a partir de su nombre (**mysqlc**) o su ID (**8411f6064e44**).

```
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3 mysqlc 8411f6064e44
172.17.0.4 c25ca9a48fb3
```

Comprobamos que el contenedor **adminer** puede conectar con el contenedor **mysql** abriendo un navegador web y accediendo a la URL: <http://localhost:8080>.



Tenga en cuenta que el nombre del servidor al que queremos conectarnos no es **db**, sino que es **mysqlc**, que es el nombre del contenedor de MySQL.

Solución 2. Utilizando una **user-defined bridge network**

En primer lugar creamos una **user-defined bridge network**.

```
$ docker network create my-net
```

Creamos un contenedor con MySQL indicando que queremos que esté en la red **--network my-net**.

```
$ docker run -d \
--rm \
--name mysqlc \
--network my-net \
-p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=root \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Creamos un contenedor con Adminer indicando que queremos que esté en la red `--network my-net`.

```
$ docker run -d \
--rm \
--network my-net \
-p 8080:8080 \
adminer
```

Comprobamos que el contenedor `adminer` puede conectar con el contenedor `mysql` abriendo un navegador web y accediendo a la URL: <http://localhost:8080>.



Tenga en cuenta que el nombre del servidor al que queremos conectarnos no es **db**, sino que es **mysqlc**, que es el nombre del contenedor de MySQL.

Para eliminar la red que hemos creado ejecutamos lo siguiente.

```
$ docker network rm my-net
```

Referencias:

- [Legacy container links. Docker](#)
- [Use bridge networks. Docker](#)

5.16. Conectar un contenedor phpMyAdmin con MySQL

Para este ejemplo usaremos la imagen oficial de [phpmyadmin](#).

Para conectar dos contenedores podemos hacerlo de dos formas:

1. Utilizando `legacy container links` con el flag `--link`, en la `bridge network`.
2. Utilizando una `user-defined bridge network`.

Solución 1. Legacy container links con el flag `--link`, en la bridge network

```
$ docker run -d \
--rm \
--name mysqlc \
-p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=root \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Le pasamos la variable de entorno `-e PMA_ARBITRARY=1` para que en la página principal de phpMyAdmin me aparezca un campo en el formulario donde pueda indicar el servidor al que quiero conectarme.

```
$ docker run -d \
--rm \
--link mysqlc \
-e PMA_ARBITRARY=1 \
-p 8080:80 \
phpmyadmin
```

Solución 2. Utilizando una `user-defined bridge network`

En primer lugar creamos una `user-defined bridge network`.

```
$ docker network create my-net
```

Creamos un contenedor con MySQL indicando que queremos que esté en la red `--network my-net`.

```
$ docker run -d \
--rm \
--name mysqlc \
--network my-net \
-p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=root \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Creamos un contenedor con phpMyAdmin indicando que queremos que esté en la red `--network my-net`.

```
$ docker run -d \
--rm \
--network my-net \
-e PMA_ARBITRARY=1 \
-p 8080:80 \
phpmyadmin
```


Comprobamos que el contenedor **phpMyAdmin** puede conectar con el contenedor **mysql** abriendo un navegador web y accediendo a la URL: <http://localhost:8080>.

Para eliminar la red que hemos creado ejecutamos lo siguiente.

```
$ docker network rm my-net
```

Ejercicio

1. Busca en Docker Hub las imágenes del sistema gestor de bases de datos **PostgreSQL** y **phpPgAdmin**.
2. Crea una instancia de PostgreSQL y phpPgAdmin, de modo que desde phpPgAdmin pueda conectarme a PostgreSQL.



Necesitarás consultar en Docker Hub cuáles son las variables de entorno que necesitas para ambos casos.

Ejercicio con múltiples contenedores: WordPress + MySQL + phpMyAdmin

Utiliza las imágenes oficiales que hay en Docker Hub para **WordPress**, **MySQL** y **phpMyAdmin**.

A continuación se detallan los pasos que tendrá que seguir:

1) Crea una **user-defined bridge network** para todos los contenedores. Por ejemplo, esta red se puede llamar **wordpress-net**.

```
$ docker network create wordpress-net
```

2) Crea un volumen **nuevo** para almacenar los datos de MySQL. Por ejemplo, este volumen se puede llamar **wordpress_mysql_data**.



MUY IMPORTANTE: Para que las variables de entorno de MySQL tengan efecto, debemos trabajar sobre un **volumen que no tenga datos y que no haya sido usado previamente** porque si el volumen ya tiene datos, la base de datos y el usuario que le estamos indicando en las variables de entorno no se crearán.

```
$ docker volume rm wordpress_mysql_data  
$ docker volume create wordpress_mysql_data
```

3) Crea una instancia de un contenedor con **MySQL** con las siguientes características:

- Se ejecuta en modo *detached* (background).
- Está en la red **wordpress-net**.
- Redirige el **puerto 3306 del host** al **puerto 3306 del contenedor**.
- Crea la variable de entorno **MYSQL_ROOT_PASSWORD** y asígnale un valor.
- Crea la variable de entorno **MYSQL_DATABASE** y asígnale un valor.
- Crea la variable de entorno **MYSQL_USER** y asígnale un valor.
- Crea la variable de entorno **MYSQL_PASSWORD** y asígnale un valor.
- Usa el volumen que creaste en el paso 2 (**wordpress_mysql_data**) para montarlo en el directorio **/var/lib/mysql** del contenedor.

```
$ docker run -d \
--rm \
--name mysqlc \
--network wordpress-net \
-p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=root \
-e MYSQL_DATABASE=wp_database \
-e MYSQL_USER=wp_user \
-e MYSQL_PASSWORD=wp_password \
-v wordpress_mysql_data:/var/lib/mysql \
mysql:5.7.28
```

4) Crea una instancia de un contenedor con **phpMyAdmin** con las siguientes características:

- Se ejecuta en modo *detached* (background).
- Está en la red **wordpress-net**.
- Redirige el **puerto 8080 del host** al **puerto 80 del contenedor**.
- Crea la variable de entorno **PMA_ARBITRARY** y asígnale el valor **1**, para que nos permita indicar el nombre del servidor de base de datos al que queremos conectarnos.

```
$ docker run -d \
--rm \
--network wordpress-net \
-e PMA_ARBITRARY=1 \
-p 8080:80 \
phpmyadmin
```

5) Crea una instancia de un contenedor con **WordPress** con las siguientes características:

- Se ejecuta en modo *detached* (background).
- Está en la red **wordpress-net**.
- Redirige el **puerto 80 del host** al **puerto 80 del contenedor**.
- Crea la variable de entorno **WORDPRESS_DB_HOST** y asígnale el nombre del contenedor que está ejecutando MySQL.
- Crea la variable de entorno **WORDPRESS_DB_NAME** y asígnale el valor de la base de datos que has creado en la instancia de MySQL.
- Crea la variable de entorno **WORDPRESS_DB_USER** y asígnale el valor del usuario de la base de datos que has creado en la instancia de MySQL.
- Crea la variable de entorno **WORDPRESS_DB_PASSWORD** y asígnale el valor de la contraseña del usuario de la base de datos que has creado en la instancia de MySQL.
- Necesitaremos crear un volumen (**wordpress_data**) para montarlo en el directorio **/var/www/html** del contenedor.

```
$ docker run -d \
--rm \
--name wordpressc \
--network wordpress-net \
-p 80:80 \
-e WORDPRESS_DB_HOST=mysqlc \
-e WORDPRESS_DB_NAME=wp_database \
-e WORDPRESS_DB_USER=wp_user \
-e WORDPRESS_DB_PASSWORD=wp_password \
-v wordpress_data:/var/www/html \
wordpress
```

Ejercicio con múltiples contenedores: Moodle + MySQL + phpMyAdmin

Utiliza las imágenes oficiales que hay en Docker Hub para **Moodle**, **MySQL** y **phpMyAdmin**, para crear un sitio web con Moodle.

5.17. Docker restart policies (--restart)

Docker nos permite establecer políticas de reinicio para controlar si los contenedores deben reiniciarse automáticamente cuando finalizan por algún motivo o cuando el servicio de Docker se reinicia.

Para configurar una política de reinicio en un contenedor se utiliza el *flag* **--restart**.

Las diferentes políticas de reinicio que podemos configurar son:

Flag	Descripción
no	El contenedor no se reinicia. Es la opción por defecto.
on-failure[:max-retries]	El contenedor se reinicia si finaliza por un error, es decir, cuando el <i>exit code</i> es distinto de 0.
always	El contenedor se reinicia cada vez que se detiene. Si se detiene manualmente, sólo se reinicia cuando el servicio de Docker se reinicia o cuando se reinicia manualmente.
unless-stopped	Es similar a always , excepto que cuando el contenedor se detiene (manualmente o de otro modo), éste no se reiniciará cuando se reinicia el servicio de Docker.



Si iniciamos un contenedor con alguna política de reinicio no podremos utilizar el *flag* **--rm**.

Ejemplo

En este ejemplo **vamos a intentar ejecutar** un contenedor con el servicio de MySQL con una política de reinicio de tipo **always**, **pero no vamos a poder hacerlo porque la opción `--rm` es incompatible con las políticas de reinicio.**

```
$ docker run -d \  
--rm \  
--restart always \  
--name mysqlc \  
-p 3306:3306 \  
-e MYSQL_ROOT_PASSWORD=root \  
-v mysql_data:/var/lib/mysql \  
mysql:5.7.28
```

Para poder aplicar la política de reinicio **always** deberemos eliminar el *flag* `--rm`.

```
$ docker run -d \  
--restart always \  
--name mysqlc \  
-p 3306:3306 \  
-e MYSQL_ROOT_PASSWORD=root \  
-v mysql_data:/var/lib/mysql \  
mysql:5.7.28
```

Referencias:

- <https://docs.docker.com/config/containers/start-containers-automatically/>
- [Ensuring Containers Are Always Running with Docker's Restart Policy.](#)

6. Portainer

Portainer es una herramienta web open-source que permite gestionar contenedores Docker de forma local o remota.

Portainer nos permite realizar las siguientes tareas:

- Gestionar contenedores de Docker
- Acceder a la consola del contenedor
- Gestionar imágenes de Docker
- Etiquetar y subir imágenes Docker
- Gestionar redes de Docker
- Gestionar volúmenes de Docker
- Navegar por los eventos de Docker
- Preconfigurar templates de contenedores
- Vista de clúster con Docker Swarm

Fuente: [Wikipedia](#)

6.1. Gestión de un servidor local

Portainer está compuesto por dos elementos, un **servidor** y un **agente**. En nuestro caso sólo vamos a gestionar un servidor local, por lo tanto **no será necesario utilizar el agente**.

Aquí vamos a diferenciar dos escenarios:

1. Gestión de un servidor local Linux, Mac, o Windows 10 ejecutándose en modo "Linux containers".

Los comandos Docker para ejecutar Portainer son los siguientes:

```
$ docker volume create portainer_data
```

```
$ docker run -d \
--rm \
-p 9000:9000 \
--name portainerc \
-v /var/run/docker.sock:/var/run/docker.sock \
-v portainer-data:/data \
portainer/portainer
```

- **-d** nos permite ejecutar el contenedor en modo *detached*, es decir, ejecutándose en segundo plano.

- `--rm` nos permite eliminar el contenedor cuando finaliza su ejecución.
- `-p 9000:9000` nos permite para mapear el puerto 9000 del host (nuestra máquina) con el puerto 9000 expuesto en el contenedor.
- `--name portainer` nos permite asignarle un nombre al contenedor
- `-v /var/run/docker.sock:/var/run/docker.sock` nos permite montar el sock UNIX del Docker *daemon* en el contenedor de portainer.
- `-v portainer-data:/data` nos permite crear un volumen para persistir la configuración de Portainer fuera del contenedor.

Para gestionar un servidor local de Docker donde se está ejecutando un contenedor de Portainer es necesario montar el socket UNIX del Docker *daemon* (`/var/run/docker.sock`).



El Docker *daemon* puede recibir peticiones de la API del Docker Engine utilizando tres tipos de sockets: `unix`, `tcp` y `fd`.

Puede encontrar más información sobre los sockets del *daemon* de Docker en la [documentación oficial](#).

Una vez hecho esto podemos acceder con un navegador web al puerto `9000` de [nuestra máquina](#).

Al acceder tendremos que configurar una contraseña para el usuario *admin*.

En el siguiente paso tendremos que indicar que queremos administrar un **Servidor Local**.

2. Gestión de un servidor local Windows ejecutándose en modo "Windows containers".

Los comandos Docker para ejecutar Portainer son los siguientes:

```
$ docker volume create portainer_data
```

```
$ docker run -d \
--rm \
-p 9000:9000 \
--name portainerc \
-v \\.\pipe\docker_engine:\\.\pipe\docker_engine \
-v portainer_data:C:\data \
portainer/portainer portainer/portainer
```

Referencias:

- [How simple is it to deploy Portainer?](#)
- <https://onthedock.github.io/post/180317-portainer/>
- <https://onthedock.github.io/post/170429-portainer-para-gestionar-tus-contenedores-en-docker/>

6.2. Gestión de un servidor remoto

Para configurar el acceso remoto a través del API de Docker, hay que modificar cómo arranca el Docker *daemon*.

Tenga en cuenta que habilitar el acceso remoto a través de la API de Docker puede suponer un riesgo de seguridad si no se realiza correctamente, por lo que se recomienda revisar la documentación oficial: [Protect the Docker daemon socket](#).

Los puertos que se utilizan para el acceso remoto son el 2375 para comunicaciones no encriptadas y el 2376 para comunicaciones encriptadas.

Habilitar el acceso remoto a la API de Docker

En primer lugar habrá que habilitar el acceso remoto a la API de Docker. Puede encontrar información de cómo hacerlo en la siguiente referencia de la documentación oficial: [How do I enable the remote API for dockerd](#).

Ejecutar Portainer indicando la IP del servidor remoto

El comando Docker para ejecutar Portainer es el siguiente:

```
$ docker run -d \
-p 9000:9000 \
--name portainerc \
-v /var/run/docker.sock:/var/run/docker.sock \
-v portainer-data:/data \
portainer/portainer -H tcp://192.168.21.100:2375
```

Donde **192.168.21.100** se corresponde con la IP del servidor remoto de Docker y **2375** el puerto donde el Docker *daemon* está escuchando las peticiones.



Tenga en cuenta que tendrá que reemplazar la dirección IP **192.168.21.100** por la dirección IP del servidor remoto que quiera gestionar.

Referencia:

- [Portainer: gestión de servidores Docker](#)

7. Redes en Docker

Cuando instalamos Docker Engine se crean tres redes:

- **bridge:** Es la red que usarán por defecto todos los contenedores que se ejecutan en el mismo *host*. También se les conoce como las **default bridge network**. En Linux durante la instalación se crea una nueva interfaz de red virtual llamada **docker0**. Cuando ejecutamos un contenedor, esta es la red que utilizará por defecto a no ser que indiquemos lo contrario.
- **none:** En esta red el contenedor no tendrá asociada ninguna interfaz de red, sólo tendrá la de **loopback** (lo).
- **host:** En esta red el contenedor tendrá la misma configuración que el servidor Docker Engine donde se esté ejecutando.

Además de estas redes, **también es posible crear las user-defined bridge network**.



En la documentación oficial de Docker aseguran que las redes **user-defined bridge** ofrecen mejores prestaciones que las **default bridge**.

7.1. Diferencias entre las redes **default bridge** y **user-defined bridge**

- Los contenedores que pertenecen a la misma red **user-defined bridge** **exponen todos los puertos entre ellos y ninguno al exterior**.
- Las redes **user-defined bridge** ofrecen un **DNS automático** entre los contenedores de la misma red, mientras que los contenedores de una red **default bridge** sólo pueden acceder a los otros contenedores a través de su IP o haciendo uso de la opción **--link** que está considerada **legacy**.
- Las redes **user-defined bridge** **permiten que un contenedor en ejecución pueda ser añadido o eliminado de la red**, mientras que la redes **default bridge** no lo permiten.

Puede encontrar [más diferencias en la documentación oficial](#).

Ejemplo

La siguiente imagen muestra un ejemplo de tres contenedores que están dentro del mismo *host*. El contenedor **c1** está conectado a la red **default bridge** y los contenedores **c2** y **c3** están conectados a una red de tipo **user-defined bridge** que se llama **my_bridge**.

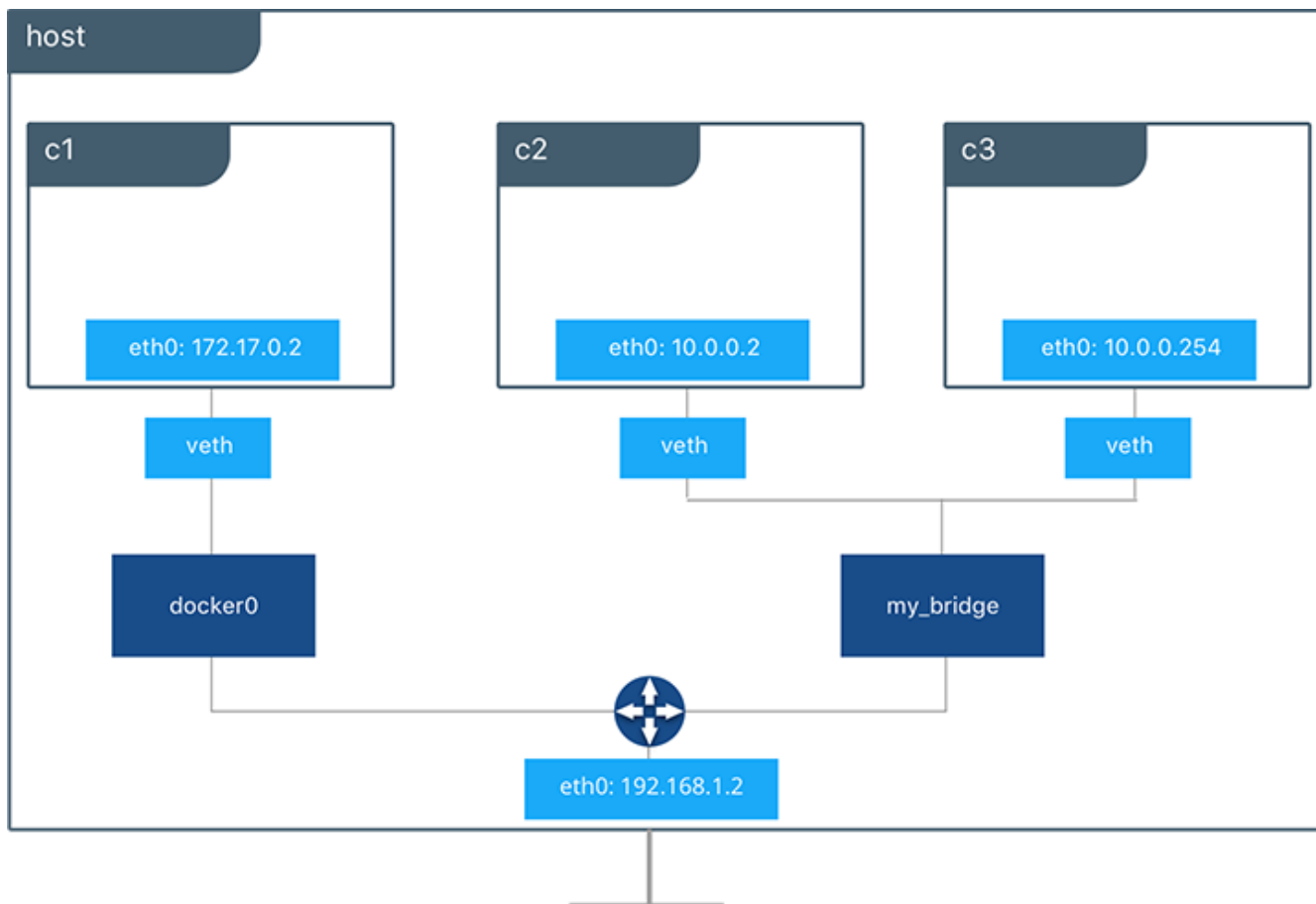


Figure 15. Ejemplo de un host con tres contenedores y dos redes. Imagen de [Docker Education Team](#)

El comando para gestionar las redes es `docker network`.

```
$ docker network --help
```

```
Usage: docker network COMMAND
```

```
Manage networks
```

```
Commands:
```

```
connect    Connect a container to a network
create     Create a network
disconnect Disconnect a container from a network
inspect    Display detailed information on one or more networks
ls         List networks
prune      Remove all unused networks
rm         Remove one or more networks
```

```
Run 'docker network COMMAND --help' for more information on a command.
```

Ver la lista de redes disponibles

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
f3ecd102593d	bridge	bridge	local
851c1a03b334	host	host	local
878629111af9	none	null	local

Inspeccionar una red

```
$ docker inspect bridge
```

Referencias:

- <https://docs.docker.com/network/>
- <https://docs.docker.com/network/bridge/>
- <https://github.com/docker/labs/blob/master/slides/docker-networking.pdf>

Tutorial

[Networking with standalone containers](#)

Lab: Docker Networking

<https://github.com/docker/labs/tree/master/dockercon-us-2017/docker-networking>

8. Almacenamiento en Docker

Por defecto, todos los archivos que se crean dentro de un contenedor se almacenan en la última capa del sistema de archivos (la capa de lectura/escritura), esto quiere decir que **los datos que tenemos en esta capa se perderán cuando el contenedor se elimine y no podremos compartirlos con otros contenedores**.

Docker nos ofrece dos posibilidades para implementar persistencia de datos en los contenedores:

- *Bind mounts*
- *Volumes*

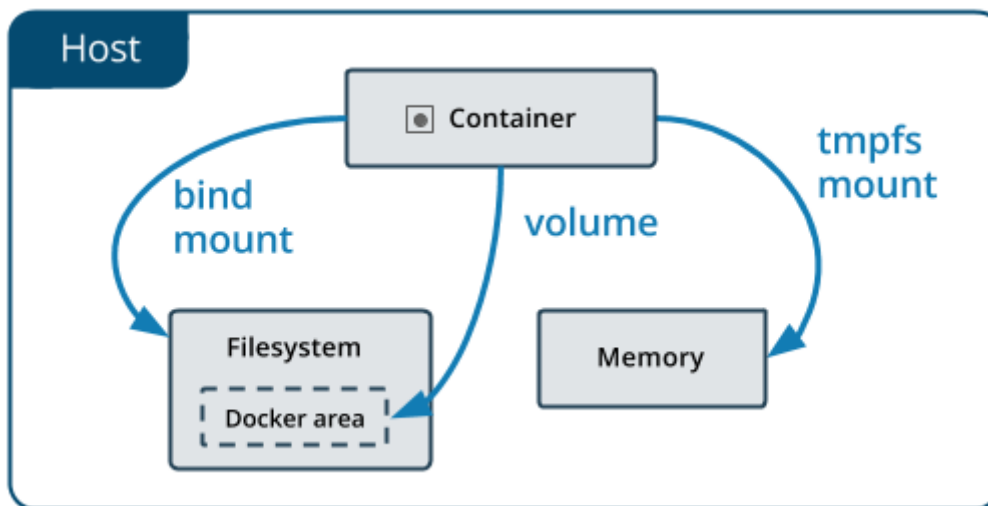


Figure 16. Manage data in Docker. Imagen de [Docker.com](https://docs.docker.com/storage/)

8.1. Bind mounts

Los *bind mounts* pueden estar almacenados en **cualquier directorio del sistema de archivos de la máquina host**. Estos archivos pueden ser consultados o modificados por otros procesos de la máquina *host* o incluso por otros contenedores Docker.

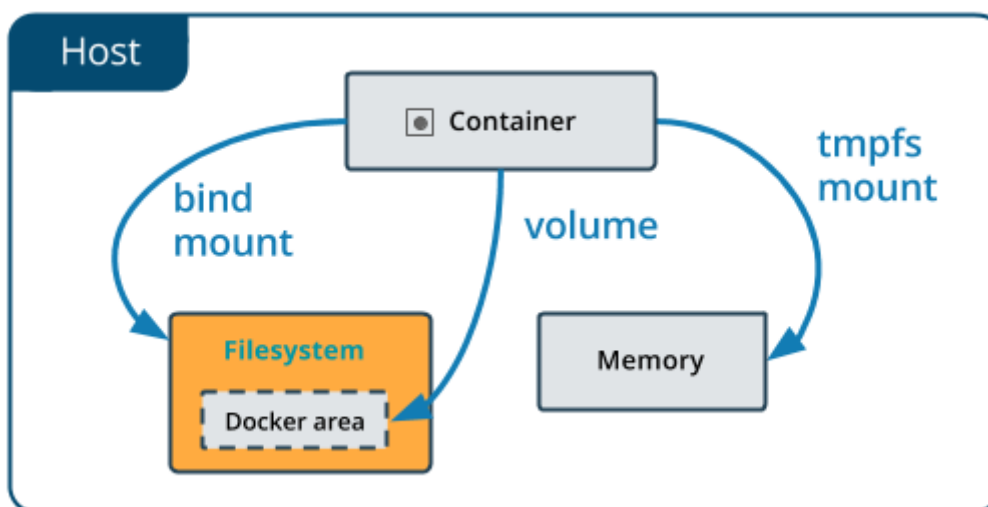


Figure 17. Use bind mounts. Imagen de [Docker.com](https://docs.docker.com/storage/)

8.2. Volumes

Los *volumes* se almacenan en la máquina *host* dentro del área del sistema de archivos que gestiona Docker. Por ejemplo, en Linux será el directorio `/var/lib/docker/volumes`.

Otros procesos de la máquina *host* no deberían modificar estos archivos, **sólo deberían ser modificados por contenedores Docker**.

Desde la documentación oficial de Docker nos aseguran que esta es la **mejor forma de implementar persistencia de datos** en los contenedores Docker.

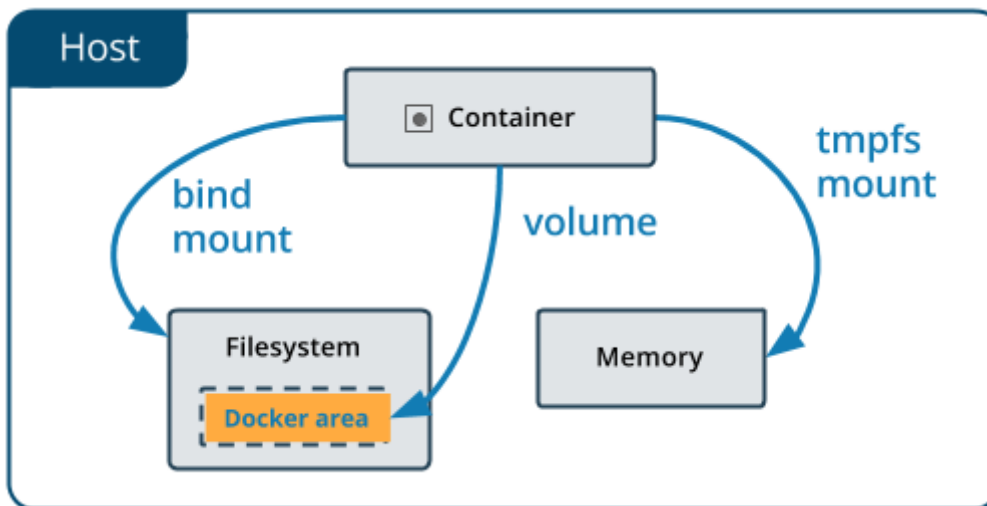


Figure 18. Use volumes. Imagen de [Docker.com](https://docs.docker.com/storage/volumes/)

El comando para gestionar volúmenes en Docker es `docker volume`.

```
$ docker volume --help

Usage:  docker volume COMMAND

Manage volumes

Commands:
  create      Create a volume
  inspect     Display detailed information on one or more volumes
  ls          List volumes
  prune       Remove all unused local volumes
  rm          Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a command.
```

Referencias:

- <https://docs.docker.com/storage/>
- <https://docs.docker.com/storage/volumes/>
- <https://docs.docker.com/storage/bind-mounts/>

9. Docker system

```
$ docker system --help
```

Usage: docker system COMMAND

Manage Docker

Commands:

df	Show docker disk usage
events	Get real time events from the server
info	Display system-wide information
prune	Remove unused data

Mostrar el espacio de disco utilizado por Docker

```
$ docker system df
```

Mostrar información detallada del sistema

```
$ docker system info
```

Eliminar datos que no están siendo utilizados

```
$ docker system prune
```

Con el *flag* **-a** también se eliminarán todas las imágenes que no están siendo utilizadas.

```
$ docker system prune -a
```

10. Limpieza del equipo

Después de ejecutar los siguientes comandos sólo tendremos en nuestra máquina los contenedores que estén en ejecución y sus imágenes correspondientes.

```
$ docker system df  
$ docker system prune -a  
$ docker volume prune
```

11. Plugin de Docker y Docker Compose para Visual Studio

Vamos a instalar el plugin de Docker que ha desarrollado Microsoft para Visual Studio.

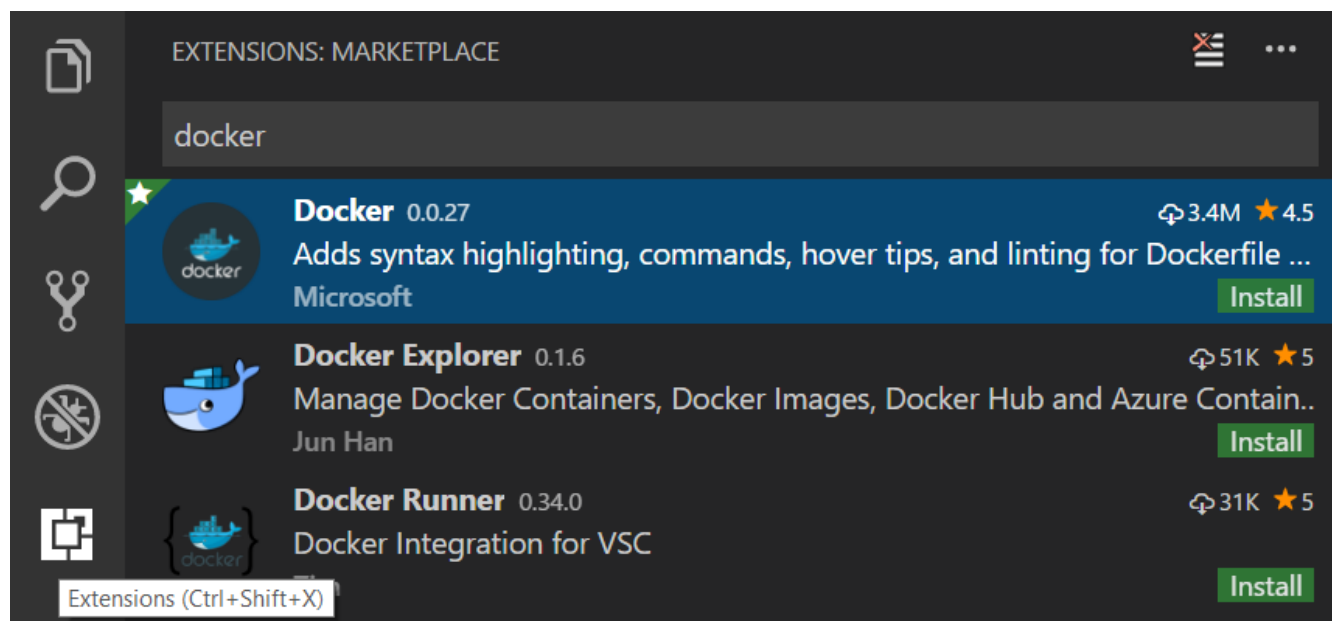


Figure 19. Working wit Docker. Imagen de [Microsoft](#)

En la [documentación oficial del plugin](#) podemos encontrar todas las funcionalidades que ofrece.

Referencia:

- [Working with Docker](#)

12. Creación de imágenes a partir de un archivo Dockerfile

12.1. Dockerfiles

Algunas de las **instrucciones** que podemos incluir en un **Dockerfile** son las siguientes:

- **FROM:**
 - Indica el nombre de la imagen base de la que partimos.
- **ARG:**
 - Podemos recibir argumentos en el proceso de creación de la imagen.
- **LABEL:**
 - Nos permite incluir metainformación en la imagen, como el nombre del autor, versión, etc.
- **RUN:**
 - Son los comandos que quiero ejecutar en la imagen.
- **VOLUME:**
 - Indica el volumen donde vamos a guardar datos persistentes fuera del contenedor. Para que sigan estando disponibles cuando el contenedor no esté en ejecución.
- **COPY:**
 - Para poder copiar archivos de nuestra máquina a la imagen.
- **EXPOSE:**
 - Para indicar los puertos que se quieren exponer cuando se ejecute un contenedor con el parámetro **-P**. Es informativo.
- **ENTRYPOINT:**
 - Una vez que se crea la instancia del contenedor e instanciado, es lo primero que ejecuta.
 - El objetivo es **dejar el contenedor en el estado inicial deseado**.
 - Cada vez que iniciamos un contenedor debe estar en este estado.
 - Lo que hagamos en el entrypoint se ejecuta en cada una de las instancias que se realicen.
- **CMD:**
 - Puede tener los parámetros que le vamos a pasar al **ENTRYPOINT** para que se ejecuten después del **ENTRYPOINT**.
 - Si no hay **ENTRYPOINT**, ejecuta los comandos que tengamos aquí.
 - Si **ENTRYPOINT** y **CMD** están presentes en el Dockerfile, primero se ejecuta el **ENTRYPOINT** y el **CMD** se le pasa como parámetro.
 - Si sólo está el **ENTRYPOINT** sólo se ejecuta éste.
 - Si sólo está el **CMD** sólo se ejecuta este.



Si tenemos varios CMD sólo se ejecuta el último.

Referencia:

- [Dockerfile reference. Docker](#)

12.2. build

Para crear imágenes a partir de un archivo Dockerfile utilizamos el comando `docker build`.

```
docker build --help

Usage:  docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

Options:
  --add-host list          Add a custom host-to-IP mapping (host:ip)
  --build-arg list         Set build-time variables
  --cache-from strings     Images to consider as cache sources
  --cgroup-parent string   Optional parent cgroup for the container
  --compress               Compress the build context using gzip
  --cpu-period int         Limit the CPU CFS (Completely Fair Scheduler) period
  --cpu-quota int          Limit the CPU CFS (Completely Fair Scheduler) quota
  -c, --cpu-shares int     CPU shares (relative weight)
  --cpuset-cpus string     CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems string     MEMs in which to allow execution (0-3, 0,1)
  --disable-content-trust Skip image verification (default true)
  -f, --file string        Name of the Dockerfile (Default is 'PATH/Dockerfile')
  --force-rm              Always remove intermediate containers
  --iidfile string        Write the image ID to the file
  --isolation string       Container isolation technology
  --label list            Set metadata for an image
  -m, --memory bytes       Memory limit
  --memory-swap bytes      Swap limit equal to memory plus swap: '-1' to enable unlimited swap
  --network string         Set the networking mode for the RUN instructions during build (default "default")
  --no-cache              Do not use cache when building the image
  --pull                  Always attempt to pull a newer version of the image
  -q, --quiet             Suppress the build output and print image ID on success
  --rm                   Remove intermediate containers after a successful build (default true)
  --security-opt strings   Security options
  --shm-size bytes        Size of /dev/shm
  -t, --tag list          Name and optionally a tag in the 'name:tag' format
  --target string          Set the target build stage to build.
  --ulimit ulimit         Ulimit options (default [])
```

Referencias:

- [Dockerfile reference. Docker](#)
- [Best practices for writing Dockerfiles](#)
- [Digging into Docker layers](#)

Ejercicio 1

Crea una imagen que muestre el mensaje *"Hola Mundo!"*.

- Utiliza **Alpine** como imagen base.
- Publícala en **Docker Hub**

Ejercicio 2

Crea una imagen que ejecute la aplicación `cmatrix`.

- Utiliza **Ubuntu** como imagen base.
- Publícala en **Docker Hub**

Ejercicio 3

Crea una imagen que sirva la web estática que está en el directorio `site` del siguiente repositorio:

<https://github.com/josejuansanchez/lab-cep-awesome-docker>

Utiliza la **imagen base que prefieras** (httpd, nginx, ubuntu...).

Ejercicio 4

Crea una imagen que ejecute la siguiente aplicación web desarrollada con **Python** y **Flask**.

<https://github.com/josejuansanchez/lab-cep-flask-app>

Una vez que hayas creado la imagen publícala en **Docker Hub**.

Ejercicio 5

Crea una imagen que ejecute la siguiente aplicación web desarrollada en **Node.js**.

<https://github.com/docker-training/node-bulletin-board>

Ejercicio 6

Crea una imagen que utilice como base la imagen de **nginx** e instala/configura los paquetes necesarios para que pueda servir páginas PHP.

Incluya un archivo **index.php** dentro de la imagen con el siguiente contenido:

```
<?php  
  
phpinfo();  
  
?>
```

En la siguiente referencia puede encontrar los pasos necesarios para instalar y configurar **PHP-FPM** (PHP FastCGI Process Manager) en el servidor web **Nginx**.

<https://josejuansanchez.org/iaw/practica-06-teoria/index.html>

13. Docker Hub

13.1. Cómo publicar una imagen en Docker Hub

En primer lugar debemos de tener un usuario en [Docker Hub](#).

Una vez que nos hemos creado un usuario, hacemos *login* desde nuestro terminal.

```
$ docker login
```

Ahora vamos a crear la imagen a partir de un archivo Dockerfile. Para poder subir una imagen a [Docker Hub](#) el nombre de la imagen tiene que incluir nuestro **nombre de usuario** y el **nombre del repositorio** donde se almacenará en Docker Hub.

Opcionalmente podemos indicar un *tag* con la versión de la imagen.

Ejemplo

Vamos a crear una imagen donde el nombre de usuario es **josejuansanchez**, el nombre del repositorio es **hola-mundo** y el *tag* es **1.0**.

```
$ docker build -t josejuansanchez/hola-mundo:1.0 .
```

Una vez hecho esto podemos hacer un *push* de la imagen para subirla a [Docker Hub](#).

```
$ docker push josejuansanchez/hola-mundo:1.0
```

Después del paso anterior [Docker Hub](#) creará un repositorio en nuestra cuenta con la imagen que acabamos de subir.



Es posible cambiar el nombre y el tag de una imagen que tenemos ya creada en nuestro equipo local sin necesidad de volver a crearla.

```
$ docker tag <local-image>[:tag] <new-repo>[:tag]
```

Referencia:

- <https://docs.docker.com/docker-hub/>
- <https://docs.docker.com/docker-hub/repos/>

14. Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones multi-contenedor con Docker. Utiliza un archivo [YAML](#) para definir y configurar los **servicios**, los **volúmenes** y las **redes** que utilizará nuestra aplicación. El nombre del archivo que se utiliza por defecto es **docker-compose.yml**, aunque es posible asignarle otro nombre. Una vez definidos todos los servicios de nuestra aplicación, podemos crearlos e iniciarlos **con un solo comando** (**docker-compose**).

Los casos de uso más habituales donde se utiliza son:

- Entornos de desarrollo.
- Entornos de prueba automatizados (Continuous Deployment / Continuous Integration).
- Para realizar despliegues en un único servidor y no en alta disponibilidad.

14.1. Instalación de Docker Compose

Para utilizar la herramienta **Docker Compose** es necesario tener instalado previamente **Docker Engine**.

Docker Compose está incluida en la instalación de:

- Docker Desktop for Mac
- Docker Desktop for Windows
- Docker Toolbox (*Legacy desktop solution*)

Para instalarla en **Linux** se recomienda seguir los pasos que se describen en la [documentación oficial](#).

Referencia:

- <https://docs.docker.com/compose/install/#install-compose>

14.2. Comandos básicos de **docker-compose**

```
$ docker-compose
```

Define and run multi-container applications with Docker.

Usage:

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
docker-compose -h|--help
```

Options:

-f, --file FILE	Specify an alternate compose file (default: docker-compose.yml)
-p, --project-name NAME	Specify an alternate project name (default: directory name)

<code>--verbose</code>	Show more output
<code>--log-level LEVEL</code>	Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
<code>--no-ansi</code>	Do not print ANSI control characters
<code>-v, --version</code>	Print version and exit
<code>-H, --host HOST</code>	Daemon socket to connect to
<code>--tls</code>	Use TLS; implied by <code>--tlsverify</code>
<code>--tlscacert CA_PATH</code>	Trust certs signed only by this CA
<code>--tlscert CLIENT_CERT_PATH</code>	Path to TLS certificate file
<code>--tlskey TLS_KEY_PATH</code>	Path to TLS key file
<code>--tlsverify</code>	Use TLS and verify the remote
<code>--skip-hostname-check</code>	Don't check the daemon's hostname against the name specified in the client certificate
<code>--project-directory PATH</code>	Specify an alternate working directory (default: the path of the Compose file)
<code>--compatibility</code>	If set, Compose will attempt to convert keys in v3 files to their non-Swarm equivalent

Commands:

<code>build</code>	Build or rebuild services
<code>bundle</code>	Generate a Docker bundle from the Compose file
<code>config</code>	Validate and view the Compose file
<code>create</code>	Create services
<code>down</code>	Stop and remove containers, networks, images, and volumes
<code>events</code>	Receive real time events from containers
<code>exec</code>	Execute a command in a running container
<code>help</code>	Get help on a command
<code>images</code>	List images
<code>kill</code>	Kill containers
<code>logs</code>	View output from containers
<code>pause</code>	Pause services
<code>port</code>	Print the public port for a port binding
<code>ps</code>	List containers
<code>pull</code>	Pull service images
<code>push</code>	Push service images
<code>restart</code>	Restart services
<code>rm</code>	Remove stopped containers
<code>run</code>	Run a one-off command
<code>scale</code>	Set number of containers for a service
<code>start</code>	Start services
<code>stop</code>	Stop services
<code>top</code>	Display the running processes
<code>unpause</code>	Unpause services
<code>up</code>	Create and start containers
<code>version</code>	Show the Docker-Compose version information

14.3. El archivo de configuración `docker-compose.yml`

El archivo `docker-compose.yml` está escrito en formato `YAML` y está formado por las siguientes secciones:

- `version` (**Obligatorio**. Si no se indica una version se trataría de la versión 1 que está en desuso)
- `services` (**Obligatorio**. Debe incluir al menos un servicio)
- `volumes` (Opcional)
- `networks` (Opcional)

Ejemplo:

Plantilla para un archivo `docker-compose.yml`.

```
version: '3'

services:
  ...

volumes:
  ...

networks:
  ...
```



La estructura de un documento YAML se denota **indentando con espacios en blanco**.

YAML no permite utilizar tabulaciones.

Si utilizamos [Visual Studio Code](#) no tendremos problemas porque reemplazará las tabulaciones por espacios en blanco.



Si no está familiarizado con el formato YAML se recomienda la lectura del tutorial [Aprende YML en Y minutos](#)

Lo habitual es que el archivo `docker-compose.yml` esté ubicado en el directorio raíz de nuestro proyecto.

Ejemplo

El siguiente ejemplo muestra el contenido del directorio raíz de un proyecto llamado `proyecto-lamp`.


```
proyecto-lamp
|
|—— apache
|     |—— Dockerfile
|—— docker-compose.yml ①
|—— sql
|     |—— database.sql
|—— src
|     |—— add.html
|     |—— add.php
|     |—— config.php
|     |—— delete.php
|     |—— edit.php
|     |—— index.php
```

① Archivo `docker-compose.yml` con la definición de todos los servicios del proyecto.

A continuación vamos a estudiar cada una de las secciones que pueden aparecer en un archivo `docker-compose.yml`.

14.4. version

La etiqueta `version` debe estar **definida al inicio del documento YAML**. El número de versión es una cadena y por lo tanto debe ir encerrado entre **comillas simples o dobles**.

Ejemplo:

```
version: '3'
```

Actualmente existen tres versiones para el formato de archivo de `docker-compose.yml`.

- **Versión 1:** Está obsoleta.
- **Versión 2.x**
- **Versión 3.x:** Es la última versión y la recomendada por Docker. Ha sido diseñada para ser compatible con **Docker Compose** y **Docker Swarm** (Cluster de Docker Engine).

La versión actual de **Docker Compose** es la **3.7**.

Dependiendo de la versión utilizada se podrá hacer uso o no de determinadas opciones. En la [documentación oficial](#) puede encontrar más información sobre las opciones que están disponibles en cada una de las versiones.

Referencia:

- <https://docs.docker.com/compose/compose-file/compose-versioning/>

14.5. services

Esta sección del archivo hace referencia a **la configuración que tendrá cada uno de los contenedores** de nuestra aplicación.

Ejemplo:

Suponga que tenemos una aplicación PHP que hace uso de la pila LAMP. En este caso podríamos tener dos contenedores, que se corresponderían con **dos servicios** dentro de nuestro archivo `docker-compose.yml`

- Servicio 1: `apache-php`
- Servicio 2: `mysql`

En el archivo `docker-compose.yml` se podrían definir así:

```
services:
  apache-php:
    image: php:7.2-apache
    ...

  mysql:
    image: mysql:5.7.28
    ...
```

Nombre de servicio vs Nombre del contenedor

Un contenedor dentro del archivo `docker-compose.yml` puede ser referenciado por el **nombre del servicio** y el **nombre del contenedor**. Aunque tengan diferentes valores, **los dos hacen referencia al mismo contenedor**.



```
services:
  apache-php: ①
    image: php:7.2-apache
    container_name: apachec ②
    ...

  mysql: ①
    image: mysql:5.7.28
    container_name: mysqlc ②
    ...
```

① Nombre del servicio

② Nombre del contenedor

Cuando trabajamos con **Docker Compose** es habitual **utilizar únicamente el nombre del servicio**, ya que si especificamos un nombre personalizado no podremos escalar los servicios con `scale`.

14.6. volumes

En la sección global de `volumes` **debemos incluir todos los volúmenes** que hayamos definido en los servicios, **excepto los volúmenes de tipo `bind_mount`**.

Si hemos declarado `volumes` en los servicios pero no los hemos declarado en la sección global del archivo, obtendremos un mensaje de error al intentar crear y ejecutar los contenedores con `docker-compose up`.

Los volúmenes que aparezcan en la sección global **serán visibles por todos los contenedores**.

Ejemplo

```

version: '3.7'

services:
  web:
    image: nginx
    ...
    volumes:
      - ./src:/usr/share/nginx/html ❶

  db:
    image: mysql:5.7.28
    ...
    volumes:
      - mysql_data:/var/lib/mysql ❷

volumes: ❸
  mysql_data:

```

- ❶ Declara un volumen de tipo `bind_mount` dentro del servicio `web`. Este volumen no se declarará en la sección global `volumes`.
- ❷ Declara un volume dentro del servicio `db`. Este volumen hay que declararlo en la sección global `volumes`.
- ❸ Esta es la sección global donde se declaran todos los volúmenes a nivel global. Este volumen será visible por todos los contenedores

14.7. networks

En la sección global de `networks` **debemos incluir todas las redes** que hayamos definido en los servicios.

Si hemos declarado `volumes` en los servicios pero no los hemos declarado en la sección global del archivo, obtendremos un mensaje de error al intentar crear y ejecutar los contenedores con `docker-compose up`.

Si no definimos ninguna `network` entonces todos los servicios que se definen en el archivo `docker-compose.yml` se ejecutarán por defecto en una red de tipo `user-defined bridge network`, que se llamará igual que el directorio que contiene el archivo `docker-compose.yml`, con el sufijo `_default`.

Ejemplo

```

version: '3'

services:
  apache:
    image: php:7.2-apache
    ports:
      - 80:80
    volumes:
      - ./src:/var/www/html
    networks: ①
      - frontend-network
      - backend-network

  mysql:
    image: mysql:5.7.28
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=database
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
    volumes:
      - mysql_data:/var/lib/mysql
    networks: ②
      - backend-network

volumes:
  mysql_data:

networks: ③
  frontend-network:
  backend-network:

```

- ① Declaramos que el servicio **apache** está incluido en dos redes: **frontend-network** y **backend-network**.
- ② Declaramos que el servicio **mysql** está incluido en la red **backend-network**.
- ③ Esta es la sección global donde se declaran todas las **networks** que existen en nuestro archivo a nivel global.

14.8. Ejemplo con un servicio **httpd**

Suponga que queremos ejecutar un contenedor Docker con las siguientes características:

- Imagen: **httpd**.
- Puertos: **80** de nuestra máquina con el puerto **80** del contenedor.
- Volumen: **"\$PWD"/src** de nuestra máquina con el **/usr/local/apache2/htdocs/** del contenedor.

El comando que tendríamos que ejecutar sería el siguiente:

```
$ docker run -d \
--rm \
-p 80:80 \
-v "$PWD"/src:/usr/local/apache2/htdocs/ \
httpd
```

Con **Docker Compose** es posible definir las características del contenedor en un archivo **docker-compose.yml**, de modo que para el ejemplo anterior quedaría así:

```
version: '3'

services:
  httpd:
    image: httpd
    ports:
      - 80:80
    volumes:
      - ./src:/usr/local/apache2/htdocs/
```

Como el volumen que estamos utilizando en este caso es de tipo **bind mount**, no es necesario declararlo en la sección **volumes** del archivo.

ports



La documentación oficial nos avisa que cuando mapeamos los puertos utilizando el formato **HOST:CONTAINER** podemos obtener **resultados erróneos cuando utilizamos un número de puerto del contenedor menor de 60**, debido a que YAML parsea los números con formato xx:yy como un valor en base-60.

Por este motivo, recomienda siempre indicar los puertos como una cadena encerrada entre comillas simples o dobles.

Referencia:

- <https://docs.docker.com/compose/compose-file/>

Ejecutar los servicios en segundo plano

Para crear y ejecutar los servicios en segundo plano (**-d detach**) con **Docker Compose** usamos la opción **up -d**:

```
$ docker-compose up -d

Creating network "httpd_default" with the default driver ①
Creating httpd_httpd_1 ... done ②
```

① Crea una red por defecto de tipo **user-defined bridge** con el nombre del directorio donde

estamos ejecutando `docker-compose` y el sufijo `_default`.

- ② Crea un contenedor para el servicio `httpd` y le asigna el nombre `httpd_httpd_1`, que está formado por el nombre del directorio, el nombre del servicio y un número.

Consultar la lista de contenedores que están en ejecución

Para consultar la lista de contenedores que están definidos en el archivo `docker-compose.yml` que están en ejecución utilizamos la opción `ps`.

```
$ docker-compose ps
```

Name	Command	State	Ports
httpd_httpd_1	httpd-foreground	Up	0.0.0.0:80->80/tcp



`docker ps` vs `docker-compose ps`

El comando `docker ps` mostrará todos los contenedores que están en ejecución dentro del *host* y `docker-compose ps` sólo mostrará los contenedores del archivo `docker-compose.yml` que están en ejecución.

Consultar la salida estándar (STDOUT)

Para consultar la salida estándar (STDOUT) de los contenedores que están definidos en el archivo `docker-compose.yml` utilizamos la opción `logs`.

Esta opción sólo nos mostrará las últimas líneas de la salida estándar.

```
$ docker-compose logs
```

Si utilizamos la opción `logs -f` el comando se queda mostrando la salida estándar hasta que pulsemos `CTRL + C`.

```
$ docker-compose logs -f
```

Detener y eliminar todos los servicios

Para detener y eliminar los servicios con utilizamos la opción `down`.

```
$ docker-compose down
```

```
Stopping httpd_httpd_1 ... done ①  
Removing httpd_httpd_1 ... done ②  
Removing network httpd_default ③
```

- ① Detiene el contenedor

- ② Elimina el contenedor
- ③ Elimina la red

14.9. Ejemplo con un servicio `mysql`

Suponga que queremos ejecutar un contenedor Docker con las siguientes características:

- Imagen: `mysql:5.7.28`.
- Puertos: `3306` de nuestra máquina con el puerto `3306` del contenedor.
- Variables de entorno:
 - `MYSQL_ROOT_PASSWORD=root`
 - `MYSQL_DATABASE=database`
 - `MYSQL_USER=user`
 - `MYSQL_PASSWORD=password`
- Volumen: `mysql_data` con el `/var/lib/mysql` del contenedor.

El comando que tendríamos que ejecutar sería el siguiente:

```
$ docker run -d \
--rm \
-e MYSQL_ROOT_PASSWORD=root \
-e MYSQL_DATABASE=database \
-e MYSQL_USER=user \
-e MYSQL_PASSWORD=password \
-p 3306:3306 \
-v mysql_data:/var/lib/mysql \
mysql:5.7.28
```

Con **Docker Compose** es posible definir las características del contenedor en un archivo `docker-compose.yml`, de modo que para el ejemplo anterior quedaría así:


```
version: '3'

services:
  mysql:
    image: mysql:5.7.28
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=database
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```



Como el servicio **mysql** está haciendo uso del volumen **mysql_data**, es necesario incluirlo en la **sección global de volumes**.

environment

Las variables de entorno se pueden declarar de dos formas:

1) Como un array

```
environment:
  - MYSQL_ROOT_PASSWORD=root
  - MYSQL_DATABASE=database
  - MYSQL_USER=user
  - MYSQL_PASSWORD=password
```

2) Como un diccionario

```
environment:
  MYSQL_ROOT_PASSWORD: root
  MYSQL_DATABASE: database
  MYSQL_USER: user
  MYSQL_PASSWORD: password
```



Después de los dos puntos (:) tiene que existir un espacio en blanco.

Ejecutar los servicios en segundo plano

Para crear y ejecutar los servicios en segundo plano (**-d detach**) con **Docker Compose** usamos la opción **up -d**:

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

```
Creating network "mysql_default" with the default driver ①  
Creating volume "mysql_mysql_data" with default driver ②  
Creating mysql_mysql_1 ... done ③
```

- ① Crea una red por defecto de tipo **user-defined bridge** con el nombre del directorio donde estamos ejecutando **docker-compose** y el sufijo **_default**.
- ② Crea el volumen que hemos declarado añadiéndole como prefijo el nombre del directorio donde estamos ejecutando **docker-compose**.
- ③ Crea un contenedor para el servicio **mysql** y le asigna el nombre **mysql_mysql_1**, que está formado por el nombre del directorio, el nombre del servicio y un número.

Consultar la lista de contenedores que están en ejecución

Para consultar la lista de contenedores que están definidos en el archivo **docker-compose.yml** que están en ejecución utilizamos la opción **ps**.

```
$ docker-compose ps
```

Name	Command	State	Ports
mysql_mysql_1	docker-entrypoint.sh mysqld	Up	0.0.0.0:3306->3306/tcp, 33060/tcp

Consultar la salida estándar (STDOUT)

Para consultar la salida estándar (STDOUT) de los contenedores que están definidos en el archivo **docker-compose.yml** utilizamos la opción **logs**.

Esta opción sólo nos mostrará las últimas líneas de la salida estándar.

```
$ docker-compose logs
```

Si utilizamos la opción **logs -f** el comando se queda mostrando la salida estándar hasta que pulsemos **CTRL + C**.

```
$ docker-compose logs -f
```

Detener y eliminar todos los servicios

Para detener y eliminar los servicios con utilizamos la opción **down**.

```
$ docker-compose down
```

```
Stopping mysql_mysql_1 ... done ①  
Removing mysql_mysql_1 ... done ②  
Removing network mysql_default ③
```

① Detiene el contenedor

② Elimina el contenedor

③ Elimina la red



Con la opción `docker-compose down` no se eliminan los volúmenes.

Si ejecutamos el comando `docker volume ls` podemos ver que el volumen `mysql_mysql_data` todavía existe.

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	mysql_mysql_data

Para detener y **eliminar los servicios, con las redes y los volúmenes incluidos** tenemos que indicar la opción `down -v`.

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

```
Stopping mysql_mysql_1 ... done ①  
Removing mysql_mysql_1 ... done ②  
Removing network mysql_default ③  
Removing volume mysql_mysql_data ④
```

① Detiene el contenedor

② Elimina el contenedor

③ Elimina la red

④ Elimina el volumen

14.10. Ejemplo con dos servicios: `mysql` y `phpmyadmin`

En este caso vamos a crear un archivo `docker-compose.yml` para definir dos servicios: `mysql` y `phpmyadmin`.

```

version: '3'

services:
  mysql:
    image: mysql:5.7.28
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=database
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
    volumes:
      - mysql_data:/var/lib/mysql

  phpmyadmin:
    image: phpmyadmin
    ports:
      - 8080:80
    environment:
      - PMA_HOST=mysql ①

volumes:
  mysql_data:


```

① La variable de entorno **PMA_HOST** nos permite indicar el nombre del servicio con el que quiero conectar el servicio de **phpmyadmin**.



En lugar de la variable de entorno **PMA_HOST** podía haber utilizado la variable **PMA_ARBITRARY=1**.

La única diferencia es que con **PMA_ARBITRARY=1** me aparece un campo de texto en la página de login de **phpmyadmin** donde tengo que indicar el nombre del servidor al que quiero conectarme y con **PMA_HOST** no hay que escribir nada porque se configura automáticamente.



Bienvenido a phpMyAdmin

Idioma - Language

Español - Spanish ▼

Iniciar sesión ⓘ

Usuario:

Contraseña:

Continuar

Figure 20. Así se vería `phpmyadmin` cuando configuramos la variable `PMA_HOST`



Figure 21. Así se vería `phpmyadmin` cuando configuramos la variable `PMA_ARBITRARY=1`

14.10.1. `depends on`

El archivo `docker-compose.yml` anterior tiene un pequeño inconveniente, y es que **no estamos controlando en qué orden se están iniciando los servicios**. Por lo tanto, puede ocurrir que el servicio de `phpmyadmin` se inicie antes que el servicio de `mysql` y que intentemos conectarnos al servicio de `mysql` sin que éste se haya iniciado todavía. En este caso obtendremos un error de conexión.

Para solucionar este problema y controlar el orden en el que se deben iniciar los servicios utilizamos la opción `depends on`.



La opción `depends on` sólo nos asegura que un servicio se inicia antes que otro, pero no nos asegura que el servicio esté "preparado" para conectarnos a él.

Por ejemplo, para indicar que el servicio de `phpmyadmin` tiene que iniciarse después del servicio de `mysql` pondríamos la opción `depends on` en la declaración de `phpmyadmin` de la siguiente manera.

```
version: '3'

services:
  mysql:
    image: mysql:5.7.28
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=database
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
    volumes:
      - mysql_data:/var/lib/mysql

  phpmyadmin:
    image: phpmyadmin
    ports:
      - 8080:80
    environment:
      - PMA_HOST=mysql
    depends_on: ①
      - mysql

volumes:
  mysql_data:
```

① Indicamos que este servicio depende del servicio `mysql` y que no podrá iniciarse hasta que el servicio de `mysql` se haya iniciado.

Referencia:

- [Control startup and shutdown order in Compose](#)

Ejercicio

Crea un archivo `docker-compose.yml` para desplegar los servicios de `mysql` y `phpmyadmin`, pero deberá utilizar **la última versión de la imagen de `mysql`**.

Nota: Es posible que cuando quiera conectarse desde `phpmyadmin` a `mysql` obtenga errores similares a estos

- `mysqli_real_connect(): The server requested authentication method unknown to the client [caching_sha2_password]`
- `mysqli_real_connect(): (HY000/2054): The server requested authentication method unknown to the client`

Para solucionar este problema **deberá sobrescribir el comando** con el que se inicia el servicio de `mysql` para indicar que el plugin de autenticación por defecto es de tipo `mysql_native_password`.

En la sección de definición del servicio de `mysql` deberá añadir la opción `command` con el siguiente valor.

```
...
mysql:
  image: mysql
  command: --default-authentication-plugin=mysql_native_password
...
```

14.10.2. `restart`

Con la opción `restart` podemos definir una política de reinicio para cada uno de los servicios de nuestro archivo `docker-compose.yml`. Estas políticas de reinicio nos permiten controlar si los contenedores deben reiniciarse automáticamente cuando finalizan por algún motivo o cuando el servicio de Docker se reinicia.

Las diferentes políticas de reinicio que podemos configurar son:

Flag	Descripción
<code>restart: no</code>	El contenedor no se reinicia. Es la opción por defecto.
<code>restart: on-failure</code>	El contenedor se reinicia si finaliza por un error, es decir, cuando el <i>exit code</i> es distinto de 0.
<code>restart: always</code>	El contenedor se reinicia cada vez que se detiene. Si se detiene manualmente, sólo se reinicia cuando el servicio de Docker se reinicia o cuando se reinicia manualmente.

Flag	Descripción
<code>restart: unless-stopped</code>	Es similar a <code>always</code> , excepto que cuando el contenedor se detiene (manualmente o de otro modo), éste no se reiniciará cuando se reinicia el servicio de Docker.

Si quisiéramos utilizar Docker Compose para realizar despliegues en un único servidor, deberíamos utilizar la opción `restart: always` para evitar los tiempos de inactividad de los servicios.

Ejemplo

```
version: '3'

services:
  mysql:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=database
      - MYSQL_USER=user
      - MYSQL_PASSWORD=password
    volumes:
      - mysql_data:/var/lib/mysql
    restart: always ①

  phpmyadmin:
    image: phpmyadmin
    ports:
      - 8080:80
    environment:
      - PMA_HOST=mysql
    depends_on:
      - mysql
    restart: always ①

volumes:
  mysql_data:
```

① Indicamos que la política de renicio de los dos servicios es de tipo `restart: always`.

Referencia:

- [Use Compose in production](#)
- [Compose file version 3 reference](#)

14.11. Ejemplo de una pila LAMP

En este ejemplo vamos a utilizar un repositorio con el siguiente contenido:

```
|— apache
|   |— Dockerfile
|— docker-compose.yml
|— sql
|   |— database.sql
|— src
```

El archivo `docker-compose.yml` tiene definidos tres servicios:

- `apache`
- `mysql`
- `phpmyadmin`

Servicio: `apache`

Para el servicio `apache` vamos a crear nuestra propia imagen a partir de un archivo **Dockerfile** con el siguiente contenido.

```
FROM ubuntu

LABEL title="apache-lamp" \
      author="José Juan Sánchez"

ENV DEBIAN_FRONTEND=noninteractive
ENV TZ=Europe/Madrid

RUN apt-get update \
    && apt-get install -y apache2 \
    && apt-get install -y php \
    && apt-get install -y libapache2-mod-php \
    && apt-get install -y php-mysql

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Se trata una imagen que parte de la imagen base de la última versión de `ubuntu` donde instalamos el **servidor web apache** y los paquetes necesarios para servir código **PHP** y poder conectarnos con un sistema gestor de bases de datos **MySQL**.

Servicio: mysql

Este servicio utilizará la última versión de la imagen oficial de `mysql` de **Docker Hub**.

Este servicio usará dos volúmenes:

- Un volumen para tener persistencia de datos en el servicio de mysql.
- Un volumen de tipo *bind mount* sobre el directorio `/docker-entrypoint-initdb.d` del contenedor para ejecutar los scripts de SQL que tenemos en el directorio local `sql` y así inicializar las bases de datos.

Servicio: phpmyadmin

Este servicio utilizará la última versión de la imagen oficial de `phpmyadmin` de **Docker Hub**.

El servicio de `phpmyadmin` se ejecutará sobre el puerto 8080 de nuestro host.

docker-compose.yml

El contenido del archivo `docker-compose.yml` será el siguiente.

```

version: '3'

services:
  apache:
    build: ./apache ①
    ports:
      - 80:80 ②
    volumes:
      - ./src:/var/www/html ③
    depends_on:
      - mysql ④

  mysql:
    image: mysql ⑤
    command: --default-authentication-plugin=mysql_native_password ⑥
    ports:
      - 3306:3306 ⑦
    environment: ⑧
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=lamp_db
      - MYSQL_USER=lamp_user
      - MYSQL_PASSWORD=lamp_password
    volumes:
      - mysql_data:/var/lib/mysql ⑨
      - ./sql:/docker-entrypoint-initdb.d ⑩

  phpmyadmin:
    image: phpmyadmin ⑪
    ports:
      - 8080:80 ⑫
    environment:
      - PMA_HOST=mysql ⑬
    depends_on:
      - mysql ⑭

volumes: ⑮
  mysql_data:

```

- ① Con la opción **build** indicamos el directorio donde está el archivo **Dockerfile** que vamos a utilizar para crear la imagen.
- ② Indicamos los puertos de nuestro *host* donde se ejecutará el servicio de **apache**.
- ③ Creamos un volumen de tipo *bind mount* donde estará el código de la aplicación que queremos servir.
- ④ Indicamos que este servicio depende del servicio **mysql** y se iniciará después de **mysql**.
- ⑤ La imagen del servicio de **mysql** será la etiquetada como **latest**.
- ⑥ Sobrescribimos el comando que inicia el servicio para indicar que queremos usar el plugin de autenticación **mysql_native_password**.

- ⑦ Indicamos los puertos del servicio de `mysql`.
- ⑧ Indicamos variables de entorno para el servicio de `mysql`.
- ⑨ Creamos un volumen para tener persistencia de datos en el servicio de `mysql`.
- ⑩ Creamos un volumen de tipo `bind mount` sobre el directorio `/docker-entrypoint-initdb.d` del contenedor para ejecutar los scripts de SQL que tenemos en local y así inicializar las bases de datos.
- ⑪ La imagen del servicio de `phpmyadmin` será la etiquetada como `latest`.
- ⑫ El servicio de `phpmyadmin` se ejecutará sobre el puerto `8080` de nuestro `host`.
- ⑬ La variable de entorno `PMA_HOST=mysql` nos permite indicar el nombre del `host` de la base de datos.
- ⑭ Indicamos que este servicio depende del servicio `mysql` y se iniciará después de `mysql`.
- ⑮ En la sección global de `volumes` indicamos los volúmenes que utilizarán los servicios.

Ejercicio 1

Clona el siguiente repositorio en tu máquina y despliega la aplicación con `docker-compose`.

```
git clone https://github.com/josejuansanchez/lab-cep-lamp.git
```

También puedes descargar el archivo `.zip` desde la siguiente URL:

<https://github.com/josejuansanchez/lab-cep-lamp/archive/master.zip>

- El directorio `apache` contiene un archivo **Dockerfile**.
- El script con la base de datos está en `sql/database.sql`.
- El directorio `src` contiene el código de la aplicación.
- El archivo de configuración de la aplicación está en `src/config.php`.

Ejercicio 2

Modifica el archivo `docker-compose.yml` del ejercicio anterior para que los servicios usen dos redes:

- `frontend-network`
- `backend-network`

En la red `frontend-network` estarán los servicios:

- `apache`
- `phpmyadmin`

Y en la red `backend-network` sólo estará el servicio:

- `mysql`

Sólo los servicios que están en la red `frontend-network` expondrán sus puertos en nuestro `host`. Por lo tanto, el servicio de `mysql` no deberá estar accesible desde nuestro `host`.

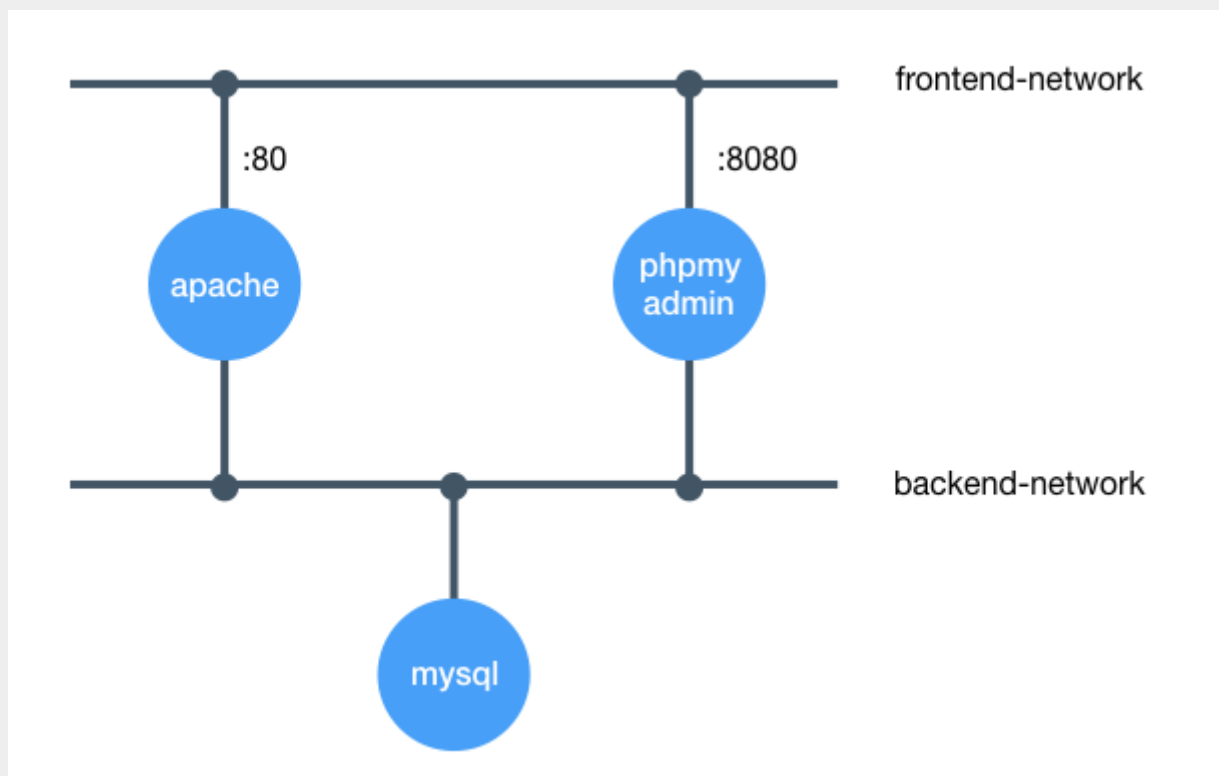


Figure 22. Diagrama de redes y servicios

Referencia:

- [Networking in Compose](#)

Ejercicio 3

Crea un archivo `docker-compose.yml` para desplegar los servicios de **WordPress**, **MySQL** y **phpMyAdmin**.

Referencia:

- <https://docs.docker.com/compose/wordpress/>

14.12. Variables de entorno en archivos `.env`

Docker Compose nos permite utilizar archivos `.env` para crear las variables de entorno.

Las reglas de sintaxis para los archivos `.env` son:

- Cada línea del archivo tiene que tener una variable de entorno con el formato `VARIABLE=valor`.
- Las líneas que empiezan con el carácter `#` son interpretadas como un comentario y son ignoradas.
- Los espacios en blanco también son ignorados.
- No hay un tratamiento especial para las comillas. Esto significa que son parte del valor de la variable.

Ejemplo

Podemos crear un archivo `.env` con el siguiente contenido.

```
MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=database
MYSQL_USER=user
MYSQL_PASSWORD=password
```

El archivo `docker-compose.yml` ahora quedaría así.

```

version: '3'

services:
  mysql:
    image: mysql:5.7.28
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD={MYSQL_PASSWORD}
    volumes:
      - mysql_data:/var/lib/mysql

  phpmyadmin:
    image: phpmyadmin
    ports:
      - 8080:80
    environment:
      - PMA_HOST=mysql
    depends_on:
      - mysql

volumes:
  mysql_data:

```

Es posible utilizar archivos con variables de entorno que no se llamen `.env`, en este caso tendremos que utilizar la opción `env_file` para indicar el nombre del archivo.



Ejemplo:

```

web:
  env_file:
    - web-variables.env

```

Referencias:

- [Declare default environment variables in file](#)
- <https://docs.docker.com/compose/environment-variables/>

Ejercicio

Modifica el archivo `docker-compose.yml` del ejercicio anterior para poder desplegar los servicios de **WordPress**, **MySQL** y **phpMyAdmin** haciendo uso de variables de entorno en un archivo `.env`.

14.13. Ejemplo de una pila LEMP

En este ejemplo vamos a utilizar un repositorio con el siguiente contenido:

```
|— docker-compose.yml
|— nginx
|   |— default.conf
|— php-fpm
|   |— Dockerfile
|— sql
|   |— database.sql
|— src
```

El archivo `docker-compose.yml` tiene definidos cuatro servicios:

- `nginx`
- `php-fpm`
- `mysql`
- `phpmyadmin`

Servicio: `nginx`

El servicio `nginx` utiliza la última versión de la imagen oficial de `nginx` de Docker Hub. En este servicio vamos a crear dos volúmenes:

- Un volumen de tipo *bind mount* para el **código fuente de la aplicación**
- Un volumen para **personalizar el archivo de configuración** que queremos utilizar en el contenedor de `nginx`.

El archivo `nginx/default.conf` tiene el siguiente contenido:

```
server {
    listen      80;
    server_name localhost;
    root        /usr/share/nginx/html;

    location / {
        index index.php index.html index.htm;
    }

    location ~ \.php$ {
        try_files $uri $uri/ /index.php$is_args$query_string;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass php-fpm:9000; ①
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

① En esta línea estamos indicando que todas las peticiones que reciba el servidor con extensión **.php** serán enviadas al puerto **9000** del servicio **php-fpm**.

Servicio: php-fpm

Este servicio será el encargado de interpretar el código **PHP** de las peticiones que recibe del servicio **nginx**. En este servicio tendremos que crear un volumen de tipo *bind mount* con el **código fuente de la aplicación**.

Para el servicio **php-fpm** vamos a crear nuestra propia imagen a partir de un archivo **Dockerfile** con el siguiente contenido.

```
FROM php:7.2-fpm

RUN docker-php-ext-install mysqli
```

Utilizamos como imagen base la imagen **php:7.2-fpm** y le añadimos una nueva capa con las librerías que me permite conectar con MySQL.



La imagen base **php:7.2-fpm** incluye los scripts **docker-php-ext-configure**, **docker-php-ext-install**, y **docker-php-ext-enable** para instalar extensiones PHP de una manera sencilla.

Servicio: mysql

Este servicio usará dos volúmenes:

- Un volumen para tener persistencia de datos en el servicio de mysql.

- Un volumen de tipo *bind mount* sobre el directorio `/docker-entrypoint-initdb.d` del contenedor para ejecutar los scripts de SQL que tenemos en el directorio local `sql` y así inicializar las bases de datos.

Servicio: phpmyadmin

El servicio de `phpmyadmin` se ejecutará sobre el puerto 8080 de nuestro host.

docker-compose.yml

El contenido del archivo `docker-compose.yml` será el siguiente.

```

version: '3'

services:
  nginx:
    image: nginx ①
    ports:
      - 80:80 ②
    volumes:
      - ./nginx:/etc/nginx/conf.d ③
      - ./src:/usr/share/nginx/html ④
    depends_on:
      - php-fpm ⑤

  php-fpm:
    build: ./php-fpm ⑥
    volumes:
      - ./src:/usr/share/nginx/html ⑦
    depends_on:
      - mysql ⑧

  mysql:
    image: mysql ⑨
    command: --default-authentication-plugin=mysql_native_password ⑩
    ports:
      - 3306:3306 ⑪
    environment: ⑫
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=lamp_db
      - MYSQL_USER=lamp_user
      - MYSQL_PASSWORD=lamp_password
    volumes:
      - mysql_data:/var/lib/mysql ⑬
      - ./sql:/docker-entrypoint-initdb.d ⑭

  phpmyadmin:
    image: phpmyadmin ⑮
    ports:
      - 8080:80 ⑯
    environment:
      - PMA_HOST=mysql ⑰
    depends_on:
      - mysql ⑱

volumes: ⑲
  mysql_data:

```

- ① Utilizamos la última versión de la imagen de **nginx**
- ② Indicamos los puertos de nuestro *host* donde se ejecutará el servicio de **nginx**.
- ③ Volumen de tipo *bind mount* para **personalizar el archivo de configuración** de **nginx**.

- ④ Volumen de tipo *bind mount* para el **código fuente de la aplicación**
- ⑤ Indicamos que este servicio depende del servicio `php-fpm` y se iniciará después de éste.
- ⑥ Con la opción `build` indicamos el directorio donde está el archivo **Dockerfile** que vamos a utilizar para crear la imagen.
- ⑦ Volumen de tipo *bind mount* para el **código fuente de la aplicación**.
- ⑧ Indicamos que este servicio depende del servicio `mysql` y se iniciará después de éste.
- ⑨ La imagen del servicio de `mysql` será la etiquetada como `latest`.
- ⑩ Sobrescribimos el comando que inicia el servicio para indicar que queremos usar el plugin de autenticación `mysql_native_password`.
- ⑪ Indicamos los puertos del servicio de `mysql`.
- ⑫ Indicamos variables de entorno para el servicio de `mysql`.
- ⑬ Creamos un volumen para tener persistencia de datos en el servicio de `mysql`.
- ⑭ Creamos un volumen de tipo *bind mount* sobre el directorio `/docker-entrypoint-initdb.d` del contenedor para ejecutar los scripts de SQL que tenemos en local y así inicializar las bases de datos.
- ⑮ La imagen del servicio de `phpmyadmin` será la etiquetada como `latest`.
- ⑯ El servicio de `phpmyadmin` se ejecutará sobre el puerto `8080` de nuestro *host*.
- ⑰ La variable de entorno `PMA_HOST=mysql` nos permite indicar el nombre del *host* de la base de datos.
- ⑱ Indicamos que este servicio depende del servicio `mysql` y se iniciará después de `mysql`.
- ⑲ En la sección global de `volumes` indicamos los volúmenes que utilizarán los servicios.

Ejercicio 1

Clona el siguiente repositorio en tu máquina y despliega la aplicación con `docker-compose`.

```
git clone https://github.com/josejuansanchez/lab-cep-lemp.git
```

También puedes descargar el archivo `.zip` desde la siguiente URL:

<https://github.com/josejuansanchez/lab-cep-lemp/archive/master.zip>

- El directorio `apache` contiene un archivo **Dockerfile**.
- El script con la base de datos está en `sql/database.sql`.
- El directorio `src` contiene el código de la aplicación.
- El archivo de configuración de la aplicación está en `src/config.php`.

Autor

Este material ha sido desarrollado por [José Juan Sánchez](#).

Licencia

El contenido de esta web está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).