

Docker básico

Índice

1. Introducción.....	3
1.1. Finalidad.....	3
1.2. Virtualización: Hipervisor.....	4
1.3. Arquitectura Docker.....	6
1.4. Instalación.....	8
2. Contenedores.....	8
2.1. Concepto de contenedor.....	9
2.2. Contenedores de Docker.....	10
2.3. Imágenes vs Contenedores.....	10
2.4. Operaciones con contenedores.....	11
2.4.1. Crear contenedores.....	11
2.4.2. Parar contenedores.....	15
2.4.3. Nombrar contenedores.....	17
2.4.4. Logs de contenedores.....	18
2.4.5. Eliminar contenedores.....	20
2.4.6. EJERCICIOS.....	22
2.5. Resolviendo el misterio: un contenedor es un proceso.....	23
2.6. Inspección de contenedores.....	24



GENERALITAT
VALENCIANA
Conselleria de Educació,
Cultura y Esports

TOTS
A UNA
VEU



Carrer d'Illueca, 28
03206 Elx, Espanya
03013224.secret@gva.es
(+34) 966.912.260



Fondo Social Europeo
El FSE invierte en tu futuro
UNIÓN EUROPEA

2.7. Interacción con contenedores.....	26
3. Redes.....	31
3.1. Por qué crear nuevas redes virtuales.....	32
3.2. Configuración IP en host y en contenedores.....	33
3.3. Operaciones con redes.....	34
3.4. DNS entre contenedores.....	37
3.5. EJERCICIOS.....	40
4. Imágenes.....	41
4.1. Docker Hub.....	42
4.2. Capas de imágenes.....	44
4.3. Etiquetas de imágenes.....	47
4.4. Etiquetado y pull a Docker Hub.....	49
4.5. Construcción de imágenes.....	53
4.5.1. Dockerfile.....	53
4.5.2. Docker build.....	54
4.5.3. Extensión de imágenes.....	56
4.5.4. Borrar imágenes.....	59
4.6. EJERCICIOS.....	61
5. Volúmenes.....	62
5.1. Operaciones con volúmenes.....	64
5.2. Operaciones con bind mounts.....	68
6. System prune.....	70

1. Introducción

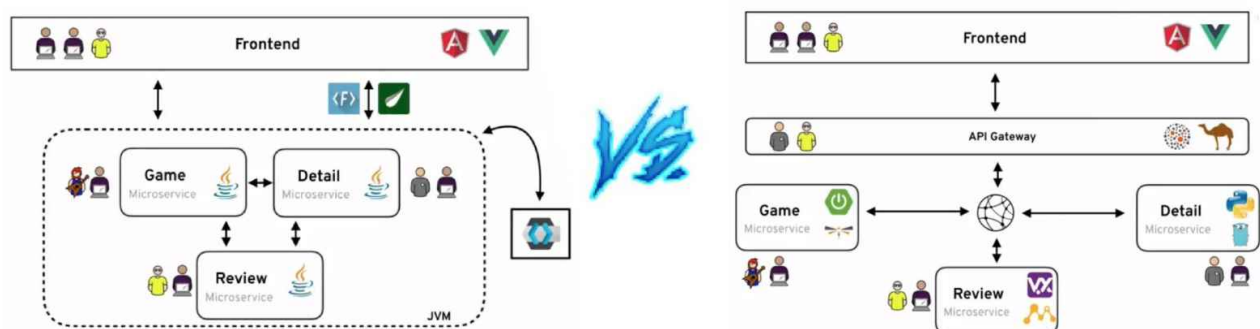
Docker (estibador en inglés) es un Sistema de Virtualización de Aplicaciones mediante contenedores, creado por Solomon Hykes y su equipo de ingenieros.

- En 2013 se convirtió en un proyecto de software libre (licencia Apache) en el que participan cada vez más empresas.
- La versión 1.0 se publicó en junio de 2014 y ha tenido un desarrollo muy rápido.
- En marzo de 2017, Docker anunció un desarrollo todavía más rápido, pasando a publicar una nueva versión cada mes. La numeración de las versiones adoptó al formato AA.MM (la primera fue Docker 17.03).
- En julio de 2018, Docker anunció que volvían a un desarrollo más pausado. A partir de Docker 18.09 habría una versión "estable" cada seis meses.

1.1. Finalidad

El objetivo principal de la configuración e implantación con Docker es solucionar los problemas de:

- Errores de dependencias entre diferentes Sistemas Operativos de los desarrolladores y máquinas de puesta en marcha.
- Evitar la elevada carga y consumo de recursos de las Máquinas Virtuales.
- Caída de todos los servicios instalados de forma monolítica en los servidores.
- Cubrir diferentes tipos de despliegue:
 - Monolítico → Todos los servicios en la misma máquina.
 - SOA (Service Oriented Architecture) → Diferentes Máquinas una con cada Servicio conectadas.
 - MicroServicios → División más pequeña de los servicios.



1.2. Virtualización: Hipervisor

En general, el objetivo de la virtualización es poder utilizar simultáneamente en un mismo dispositivo dos o más sistemas operativos. Por lo tanto, para poder hablar de virtualización tienen que estar funcionando a la vez varios sistemas operativos.

- Los sistemas operativos son los encargados de la gestión del hardware y requieren un control completo del mismo, por lo que dos sistemas operativos no pueden en principio estar funcionando a la vez sobre el mismo hardware.
- La solución para la virtualización es la existencia de un hipervisor (en inglés, hypervisor).

NOTA:

- De acuerdo con esta definición, instalar dos sistemas operativos en un ordenador (Windows y Linux, por ejemplo) y poder elegir uno u otro mediante un arranque dual no se considera virtualización, puesto que mediante un arranque dual no podemos ejecutar a la vez ambos sistemas.
- Y tampoco sería virtualización la simulación, que consiste en imitar el aspecto visual del sistema imitado. Por ejemplo, podríamos instalar un tema de escritorio en GNOME o KDE que imitara el escritorio de Windows. El problema de esta simulación sería que realmente no estaríamos utilizando Windows sino simplemente algo que parece Windows. Así que, por ejemplo, no podríamos instalar una aplicación de Windows puesto que el sistema operativo sería Linux, que no acepta instaladores de Windows.

Ventajas Virtualización

La virtualización tiene muchas aplicaciones interesantes:

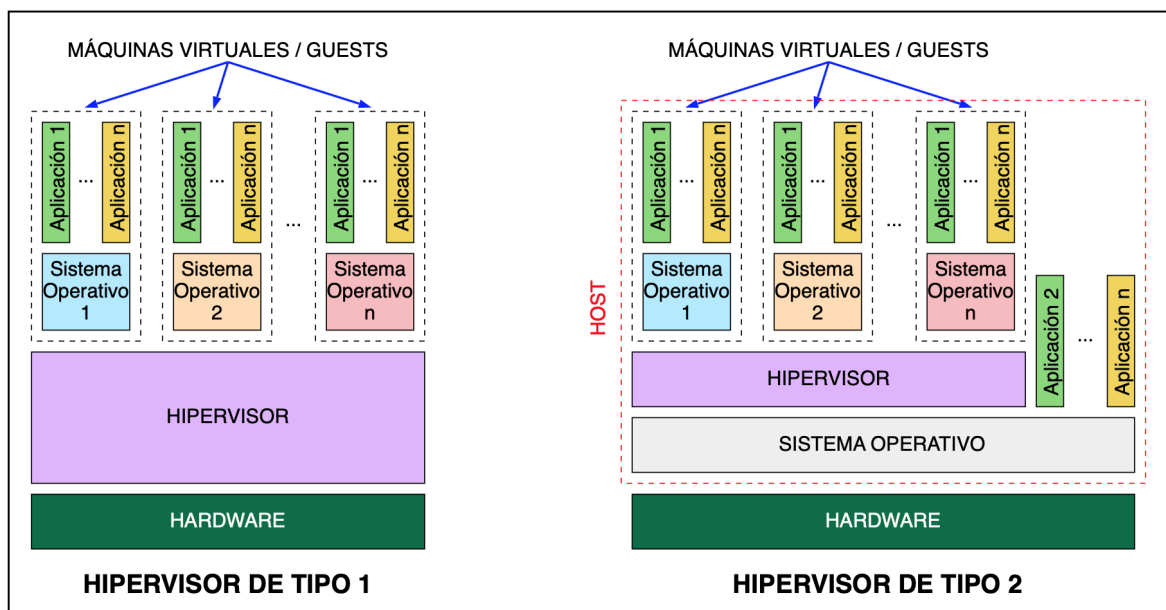
- La más habitual es poder ejecutar aplicaciones que no están disponibles para el sistema operativo host, pero sí para otro sistema operativo que se instalaría como guest.
- La virtualización permite la conservación del software. Debido al progreso del hardware, los procesadores antiguos dejan de fabricarse y los sistemas operativos y aplicaciones antiguos dejan de desarrollarse y dejan de funcionar en el hardware moderno. Pero si el sistema operativo puede instalarse como guest, puede seguir utilizándose.
- La virtualización permite depurar y comprobar el funcionamiento de los programas y los sistemas operativos. Si un programa provoca un fallo de funcionamiento total, si se está ejecutando como guest, el sistema host puede recoger información sobre el motivo del fallo.
- La virtualización permite un mejor aprovechamiento del hardware, ya que un mismo ordenador puede contener muchos sistemas guests utilizados por usuarios diferentes, aislados unos de otros.

Hipervisor

El hipervisor es la pieza fundamental de la virtualización. Tradicionalmente, se distinguen dos tipos de hipervisores:

- Los hipervisores de tipo 1, denominados «hipervisores bare metal», dichos hipervisores están en contacto directo con el hardware de la máquina, sin necesidad de ningún sistema operativo previo.
- Los hipervisores de tipo 2, denominados «alojados», son una aplicación más del

sistema operativo instalado en la máquina. El hipervisor accede al hardware de la máquina a través de ese sistema operativo.



NOTA:

- En el caso de los hipervisores de tipo 2, el sistema operativo que tiene el control del hardware recibe el nombre de host (anfitrión). Es el sistema operativo que se instaló primero en el ordenador y el que se pone en marcha al arrancar el ordenador. Los demás sistemas operativos reciben el nombre de guests (huéspedes) y pueden ejecutarse o no a voluntad del usuario.
- En el caso de los hipervisores de tipo 1 no hay un sistema operativo host, todos los sistemas operativos son guests del hipervisor.

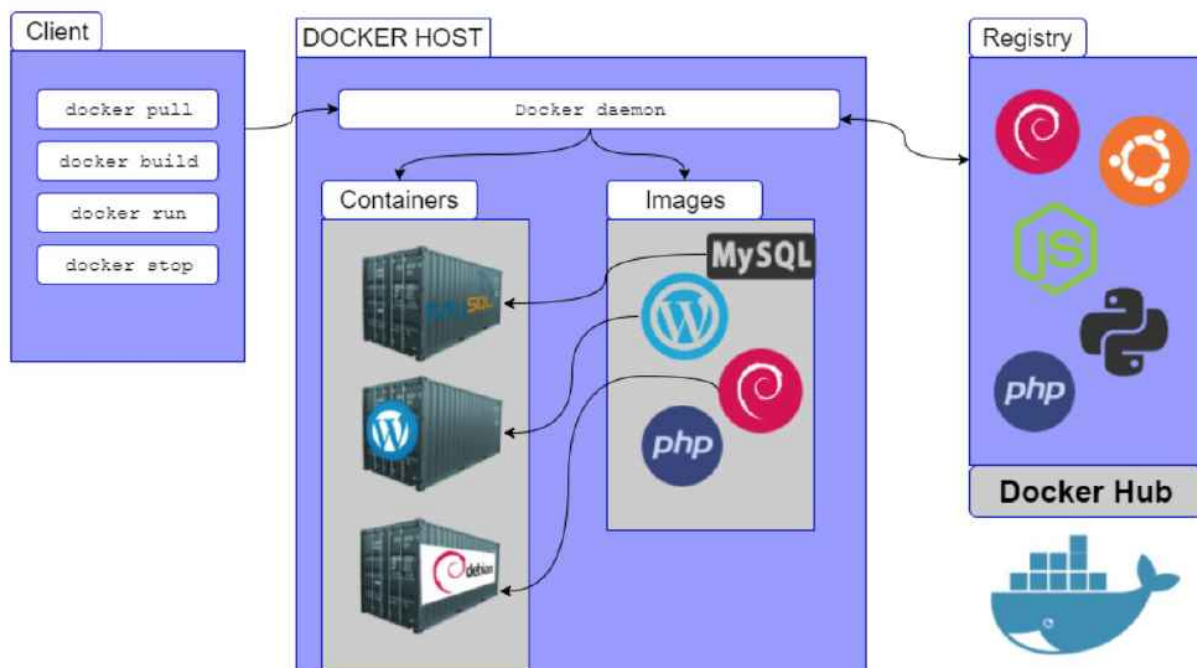
1.3. Arquitectura Docker

La arquitectura Docker consiste en los siguientes elementos:

- Docker Engine ("Motor" del Gestor Docker): está basado en la arquitectura de Cliente-Servidor (que pueden estar en la misma máquina, o en distintas), y realizada mediante una API de REST que utiliza HTTP.
- API REST: interfaz de programación con un estilo de arquitectura software para

sistemas hipermedia distribuidos como la World Wide Web.

2. "Daemon Docker" (Servicio): lleva a cabo Gestión y enlace de los componentes del gestor.
3. Imágenes: Las imágenes son una especie de plantillas que contienen como mínimo todo el software que necesita la aplicación para ponerse en marcha. están formadas por una colección ordenada de: Sistemas Archivos; Repositorios; Comandos; Parámetros; Aplicaciones.
4. Contenedores: son el conjunto de procesos que encapsulan e identifican a una Imagen. Pueden ser:
 - Creado, inicializado, parado, vuelto a ejecutar y destruido.
5. Registros son imágenes son guardadas en registros para: Almacenar o Distribuir.
 - Se realiza en Docker Hub y pueden ser Públicos y Privados.



1.4. Instalación

Para la Instalación de Docker es recomendable seguir la documentación oficial:

[Docker for Mac](#)

[Docker for Windows](#)

[Docker for ubuntu](#)

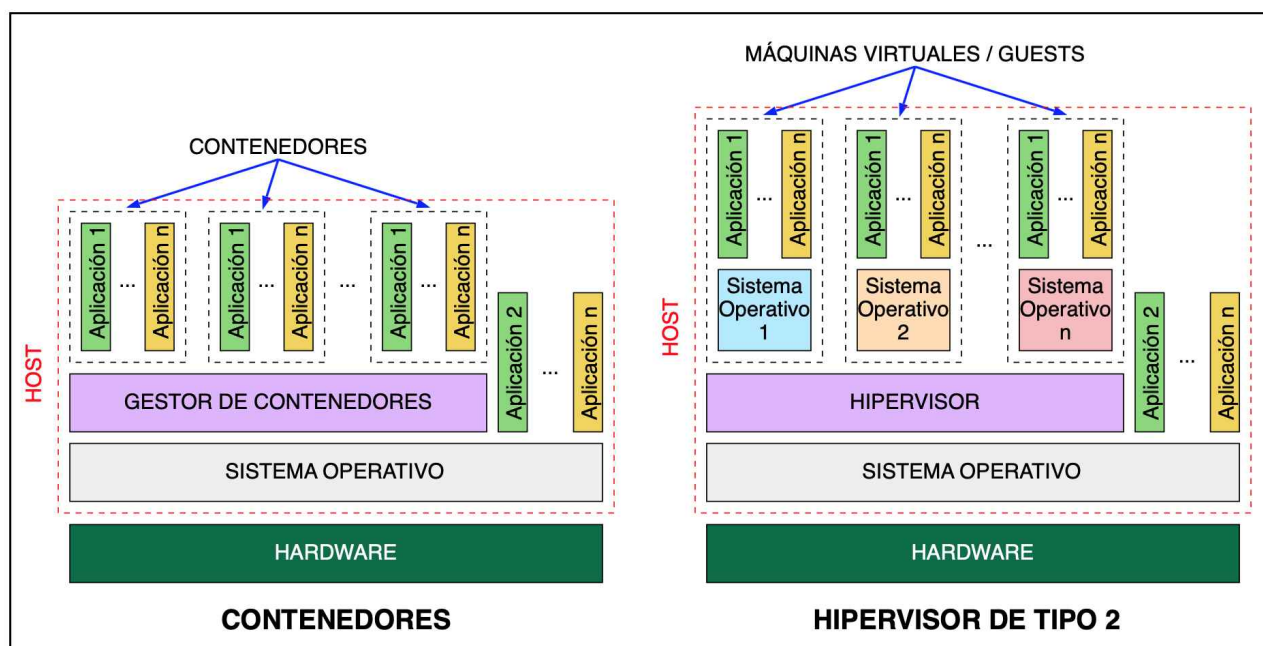
Si aún no tienes instalado Docker en tu máquina, **¡es el momento de hacerlo!**

2. Contenedores

El problema de los hipervisores y las máquinas virtuales es que cada máquina virtual es independiente de las demás. Al no reutilizarse ningún componente, se ocupa mucho espacio tanto en disco como en memoria y el tiempo de ejecución siempre será mayor que si solo hubiera un sistema operativo (sobre todo en el caso de hipervisores de tipo 2).

Para resolver este problema se crearon los contenedores en los que se utilizan mecanismos existentes en el sistema operativo para aislar las aplicaciones, pero compartiendo el mayor número posible de componentes del sistema operativo o incluso de las aplicaciones.

- Como definición, un contenedor es el equivalente a una máquina virtual de la virtualización clásica, pero mucho más ligera porque utiliza recursos del sistema operativo del host.
- Las aplicaciones de cada contenedor "ven" un sistema operativo, que puede ser diferente en cada contenedor, pero quien realiza el trabajo es el sistema operativo común que hay por debajo.



NOTA:

- Los contenedores tienen carácter efímero. La facilidad con la que pueden crearse y ponerse en marcha hace más fácil crear un nuevo contenedor que modificar uno ya existente. Por ello, los datos generados por las aplicaciones no se suelen guardar en los contenedores, sino fuera de ellos (como buena práctica).
- Su ligereza hace más fácil tener varios contenedores con una aplicación en cada uno de ellos que tener un único contenedor con varias aplicaciones en él. Por ello, un aspecto importante de los contenedores es su orquestación, es decir, la administración simultánea de muchos contenedores. Una de las herramientas más utilizadas es Kubernetes, aunque también existe la posibilidad de utilizar Docker Swarm.

2.1. Concepto de contenedor

En resumen los contenedores:

- Consisten en agrupar y aislar aplicaciones o grupos de aplicaciones que se ejecutan sobre un mismo núcleo del sistema operativo.
- Su característica principal se basa en su propio sistema de archivos, ejecutable en cualquier Sistema Operativo.

- No es necesario emular el HW y SW completo como en las máquinas virtuales, por lo tanto son mucho más ligeros, comparten el máximo de componentes con el sistema operativo host, y su rapidez, ya que gracias a que apenas añaden capas adicionales consiguen casi velocidades nativas.
- Soluciona problemas de espacio y compatibilidades a la hora de puesta en marcha en servidores de producción.
- Los contenedores tienen carácter efímero.

2.2. Contenedores de Docker

- Docker es una API amigable del tipo Open Source.
- Genera un proceso aislado del resto de los procesos de la máquina gracias a: Ejecutar sobre su propio sistema de ficheros, con su propio espacio de usuarios y procesos, y sus propias interfaces de red...
- Es modular, ya que esta dividido en varios componentes.
- Es portable e inmutable, utilizando la plataforma DockerHub.
- Su es lema “Build, Ship and Run, any app”.

2.3. Imágenes vs Contenedores

Una de las primeras confusiones que aparecen cuando se empieza a utilizar los términos imagen y contenedor en Docker es no discernir exactamente en qué se diferencian. Conforme se progrese con las actividades de este documento, esta primera distinción se irá concretando, pero por el momento, y para establecer un punto de partida, diremos que:

- Una imagen consiste en definir qué aplicación queremos ejecutar.
- Un contenedor es una instancia de una imagen, ejecutándose como un proceso en

el sistema operativo host.

- Se puede lanzar uno o más contenedores basados en la misma imagen.

2.4. Operaciones con contenedores

2.4.1. Crear contenedores

Empezamos con un ejemplo práctico para ilustrar en qué consiste un contenedor.

Antes que nada, vamos a ver si disponemos de alguna imagen en el sistema. Suponiendo que es la primera vez que ejecutamos un contenedor, el siguiente comando no debería devolver ningún registro:

`docker image ls`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
manu@manu-HP-Laptop-15s-fq1xxx:~$
```

Recordemos que los contenedores se crean a partir de imágenes. Para que exista un contenedor, antes se ha de crear una imagen.

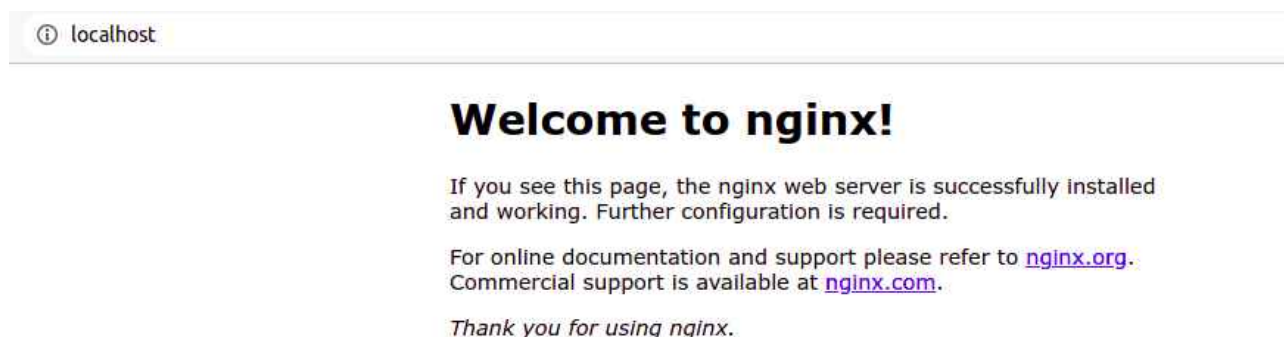
Ahora queremos lanzar nuestro primer contenedor, en este caso el servidor web Nginx.

Ejecutamos el siguiente comando:

`docker container run --publish 80:80 nginx`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run --publish 80:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
5eb5b503b376: Pull complete
1ae07ab881bd: Pull complete
78091884b7be: Pull complete
091c283c6a66: Pull complete
55de5851019b: Pull complete
b559bad762be: Pull complete
Digest: sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767
Status: Downloaded newer image for nginx:latest
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/02/08 15:30:47 [notice] 1#1: using the "epoll" event method
2022/02/08 15:30:47 [notice] 1#1: nginx/1.21.6
2022/02/08 15:30:47 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2022/02/08 15:30:47 [notice] 1#1: OS: Linux 5.4.0-99-generic
2022/02/08 15:30:47 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2022/02/08 15:30:47 [notice] 1#1: start worker processes
2022/02/08 15:30:47 [notice] 1#1: start worker process 30
2022/02/08 15:30:47 [notice] 1#1: start worker process 31
2022/02/08 15:30:47 [notice] 1#1: start worker process 32
2022/02/08 15:30:47 [notice] 1#1: start worker process 33
2022/02/08 15:30:47 [notice] 1#1: start worker process 34
2022/02/08 15:30:47 [notice] 1#1: start worker process 35
2022/02/08 15:30:47 [notice] 1#1: start worker process 36
2022/02/08 15:30:47 [notice] 1#1: start worker process 37
172.17.0.1 - - [08/Feb/2022:15:31:17 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:97.0.4692.99 Safari/537.36) "
```

Aún no sabemos muy bien qué hemos hecho, pero consultamos el resultado abriendo el navegador en la dirección localhost, y vemos la página de bienvenida de nginx:



¿Qué ha sucedido? Varias cosas han ocurrido tras ejecutar este comando (muchos de los conceptos siguientes los veremos a lo largo del documento):

- Docker Engine busca en la caché local si se dispone ya de la imagen, que es

"nginx".

- Como la imagen no se encuentra en la caché local, se descarga de Docker Hub, lo cual se indica en las siguientes líneas del log:

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
5eb5b503b376: Pull complete
1ae07ab881bd: Pull complete
78091884b7be: Pull complete
091c283c6a66: Pull complete
55de5851019b: Pull complete
b559bad762be: Pull complete
Digest: sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767
Status: Downloaded newer image for nginx:latest
```

- Se asigna al contenedor una IP virtual dentro de la red de Docker Engine.
- Abre el puerto 80 en el dispositivo host, y dentro del contenedor redirecciona las peticiones al puerto 80 del contenedor (si el puerto del host, el 80, se está utilizando por otro proceso, se podría utilizar otro puerto, por ejemplo 8080:80, y probar con localhost:8080).
- Se lanza el contenedor utilizando el comando CMD del Dockerfile de la imagen.
- Se muestran los logs del contenedor en la terminal de forma interactiva, de forma que si hacemos una visita a localhost desde el navegador, veremos dichos logs:

```
172.17.0.1 - - [08/Feb/2022:15:31:17 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (X11; Linu
e/97.0.4692.99 Safari/537.36" "-"
172.17.0.1 - - [08/Feb/2022:15:31:17 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "http://localhos
6 (KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36" "-"
2022/02/08 15:31:17 [error] 31#31: *2 open() "/usr/share/nginx/html/favicon.ico" failed (2: No s
calhost, request: "GET /favicon.ico HTTP/1.1", host: "localhost", referer: "http://localhost/"
```

Abrimos un terminal nuevo en el que vamos a ver la nueva imagen y el nuevo contenedor, con los siguientes comandos:

`docker image ls`

`docker container ls`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
nginx                latest          c316d5a335a5   13 days ago    142MB
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID        IMAGE           COMMAND                  CREATED         STATUS         PORTS                    NAMES
ca82cd54f981        nginx           "/docker-entrypoint..." 14 seconds ago Up 13 seconds    0.0.0.0:80->80/tcp      loving_cray
```

Ahora volvemos a la primera terminal y paramos el proceso con `ctrl + c` y se para el contenedor. Volvemos a ejecutar los comandos para ver las imágenes y contenedores de que disponemos, y vemos que conservamos la imagen, pero no el contenedor:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
nginx                latest          c316d5a335a5   13 days ago    142MB
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID        IMAGE           COMMAND                  CREATED         STATUS         PORTS                    NAMES
```

Hasta ahora hemos ejecutado el contenedor de nginx de forma interactiva, por eso éramos capaces de ver los logs del contenedor al visitar localhost, e interrumpir el proceso con `ctrl + c`.

Si quisiésemos ejecutar el contenedor de nginx como un proceso en background, deberíamos hacer una pequeña modificación al comando:

```
docker container run --publish 80:80 --detach nginx
```

Mediante el parámetro `--detach` (o `"-d"`, de forma abreviada) conseguimos desacoplar la ejecución del contenedor de la terminal:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run --publish 80:80 --detach nginx
5773ba46a00f50f9a00c9efa21c4ba7242f3695b0a4387f3731ff63a82d9a8cc
manu@manu-HP-Laptop-15s-fq1xxx:~$
```

Tras ejecutar el comando, docker nos devuelve el identificador del contenedor.

Si quisiésemos ejecutar otro contenedor de nginx en el mismo puerto, obtendríamos un error. Vamos a utilizar la forma abreviada del comando anterior para probarlo:

```
docker container run -p 80:80 -d nginx
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -p 80:80 -d nginx
a4a46e4ca99576d9a1fbabf78d36660cbd87a324a77ab06e814cd6134a00a215
docker: Error response from daemon: driver failed programming external connectivity on endpoint modest_herschel (67975574c74788c818206723f0652ccd29c563fd4e2d57ae6655d4db2f9f2622): Bind for 0.0.0.0:80 failed: port is already allocated.
```

Esto se debe a que no hemos parado el contenedor anterior, que está utilizando el puerto

80 del host. Si quisiésemos levantar un nuevo contenedor nginx, deberíamos hacerlo en un puerto diferente, por ejemplo el 8080. Para ello, podemos utilizar el siguiente comando:

`docker container run -p 8080:80 -d nginx`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -p 8080:80 -d nginx
2256c6abcf64f91e798d96d3345df54d5fd412e7e141c4d82c2cac662fc3c7ae
```

Ahora vemos que no devuelve error.

Y para comprobar el resultado, consultamos localhost:8080 en el navegador:

localhost:8080

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Finalmente, consultamos las imágenes y los contenedores de que disponemos en el sistema, y vemos que tenemos una imagen de nginx y dos contenedores basados en esta imagen:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
nginx         latest    c316d5a335a5   13 days ago   142MB
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
2256c6abcf64   nginx    "/docker-entrypoint..." 16 minutes ago Up 16 minutes   0.0.0.0:8080->80/tcp    epic_newton
5773ba46a00f   nginx    "/docker-entrypoint..." 30 minutes ago Up 30 minutes   0.0.0.0:80->80/tcp      priceless_stonebraker
```

2.4.2. Parar contenedores

Partimos de la imagen y los dos contenedores del apartado anterior. El propósito es eliminar los contenedores del sistema, pero ahora no nos sirve con un simple `ctrl + c`, ya que los contenedores se están ejecutando como procesos en segundo plano.

En primer lugar hemos de identificar los contenedores, y para ello listamos los que tenemos en marcha con el comando que hemos visto anteriormente:

docker container ls

También existe el siguiente comando, aunque en desuso, para conseguir lo mismo (se recomienda el anterior):

docker ps

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
2256c6abcf64   nginx    "/docker-entrypoint..." 16 minutes ago Up 16 minutes  0.0.0.0:8080->80/tcp      epic_newton
5773ba46a00f   nginx    "/docker-entrypoint..." 30 minutes ago Up 30 minutes  0.0.0.0:80->80/tcp        priceless_stonebraker
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
2256c6abcf64   nginx    "/docker-entrypoint..." 43 minutes ago Up 43 minutes  0.0.0.0:8080->80/tcp      epic_newton
5773ba46a00f   nginx    "/docker-entrypoint..." 57 minutes ago Up 57 minutes  0.0.0.0:80->80/tcp        priceless_stonebraker
```

Para parar los contenedores, nos hemos de fijar en las columnas CONTAINER ID o NAMES. Con cualquiera de estos valores podemos parar un contenedor con el siguiente comando:

docker container stop [ID|NAME]

Si queremos para el primer contenedor utilizando su ID (tendrás que consultar el ID concreto de tu máquina):

docker container stop 2256c6abcf64

Y nos devuelve el ID del contenedor:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container stop 2256c6abcf64
2256c6abcf64
```

Ahora hacemos la prueba de listar todos los contenedores:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
5773ba46a00f   nginx    "/docker-entrypoint..." About an hour ago Up About an hour  0.0.0.0:80->80/tcp        priceless_stonebraker
```

Pero, ¡si solo hemos parado el contenedor y no nos aparece en la lista! El problema es que el parámetro ls lista todos los contenedores que están ejecutándose. Si queremos ver todos los contenedores, incluso los que están parados, hemos de ejecutar el comando:

docker container ls -a


```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2256c6abcf64	nginx	"/docker-entrypoint..."	51 minutes ago	Exited (0) 4 minutes ago		epic_newton
5773ba46a00f	nginx	"/docker-entrypoint..."	About an hour ago	Up About an hour	0.0.0.0:80->80/tcp	priceless_stonebraker

En la columna STATUS podemos ver el estado de cada contenedor, y se puede apreciar que el primero dice Exited, y no está utilizando ningún puerto.

Ahora vamos a parar el segundo contenedor utilizando su nombre:

`docker container stop priceless_stonebraker`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2256c6abcf64	nginx	"/docker-entrypoint..."	16 hours ago	Exited (0) 15 hours ago		epic_newton
5773ba46a00f	nginx	"/docker-entrypoint..."	16 hours ago	Exited (0) 2 seconds ago		priceless_stonebraker

(La diferencia en la columna CREATED de esta captura y la anterior se debe al salto en la elaboración de esta documentación)

Ahora tenemos los dos contenedores parados, pero no eliminados. Los podríamos reiniciar en cualquier momento. Vamos a reiniciar el primero de ellos con el siguiente comando:

`docker container start [ID|NAME]`

El resultado:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container start 2256c6abcf64
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2256c6abcf64	nginx	"/docker-entrypoint..."	16 hours ago	Up 2 seconds	0.0.0.0:8080->80/tcp	epic_newton
5773ba46a00f	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) 11 minutes ago		priceless_stonebraker

2.4.3. Nombrar contenedores

Hasta el momento ha sido Docker quien ha dado un nombre a cada contenedor que hemos creado, de forma automática (columna NAMES).

CURIOSIDAD: Docker asigna los nombres de forma automática anteponiendo un adjetivo al nombre de un renombrado científico o hacker.

Pero, ¿y si quisiésemos dar un nombre personalizado a un nuevo contenedor o renombrar uno existente? Vamos a ello.

Primero vamos a crear un nuevo contenedor, con un nombre de nuestra elección, con el siguiente comando:

```
docker container run --publish 80:80 --detach --name webhost nginx
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7c8da25c28b1	nginx	"/docker-entrypoint..."	10 seconds ago	Up 9 seconds	0.0.0.0:80->80/tcp	webhost
2256c6abcf64	nginx	"/docker-entrypoint..."	17 hours ago	Up 18 minutes	0.0.0.0:8080->80/tcp	epic_newton
5773ba46a00f	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) 29 minutes ago		priceless_stonebraker

Ahora vamos a renombrar el contenedor con nombre "epic_newton", mediante el siguiente comando:

```
docker container rename [ID|NAME] [NEW NAME]
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rename epic_newton my_webhost
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7c8da25c28b1	nginx	"/docker-entrypoint..."	6 minutes ago	Up 6 minutes	0.0.0.0:80->80/tcp	webhost
2256c6abcf64	nginx	"/docker-entrypoint..."	17 hours ago	Up 24 minutes	0.0.0.0:8080->80/tcp	my_webhost
5773ba46a00f	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) 35 minutes ago		priceless_stonebraker

Primero se le pasa el nombre o identificador del contenedor, y a continuación el nuevo nombre que va a tener el contenedor.

Si intentamos renombrar el contenedor y asignarle el mismo nombre, Docker nos devuelve un error. Vamos a probar con el mismo contenedor, pero esta vez con su ID:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rename 2256c6abcf64 my_webhost
```

```
Error response from daemon: Renaming a container with the same name as its current name
```

```
Error: failed to rename container named 2256c6abcf64
```

2.4.4. Logs de contenedores

Cuando ejecutábamos un contenedor en modo interactivo (sin el parámetro --detach o -d), se podían ver sus logs desde el propio terminal, pero al ejecutarlo en segundo plano esto ya no es posible. Por tanto, hemos de poder consultar dichos logs de otra forma. Para ello, utilizamos el siguiente comando:

```
docker container logs [ID|NAME]
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container logs 7c8da25c28b1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/02/09 10:48:18 [notice] 1#1: using the "epoll" event method
2022/02/09 10:48:18 [notice] 1#1: nginx/1.21.6
2022/02/09 10:48:18 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2022/02/09 10:48:18 [notice] 1#1: OS: Linux 5.4.0-99-generic
2022/02/09 10:48:18 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2022/02/09 10:48:18 [notice] 1#1: start worker processes
2022/02/09 10:48:18 [notice] 1#1: start worker process 31
2022/02/09 10:48:18 [notice] 1#1: start worker process 32
2022/02/09 10:48:18 [notice] 1#1: start worker process 33
2022/02/09 10:48:18 [notice] 1#1: start worker process 34
2022/02/09 10:48:18 [notice] 1#1: start worker process 35
2022/02/09 10:48:18 [notice] 1#1: start worker process 36
2022/02/09 10:48:18 [notice] 1#1: start worker process 37
2022/02/09 10:48:18 [notice] 1#1: start worker process 38
```

Si vamos a localhost con el navegador y consultamos de nuevo los logs, veremos que se registra la petición HTTP:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container logs 7c8da25c28b1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/02/09 10:48:18 [notice] 1#1: using the "epoll" event method
2022/02/09 10:48:18 [notice] 1#1: nginx/1.21.6
2022/02/09 10:48:18 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2022/02/09 10:48:18 [notice] 1#1: OS: Linux 5.4.0-99-generic
2022/02/09 10:48:18 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2022/02/09 10:48:18 [notice] 1#1: start worker processes
2022/02/09 10:48:18 [notice] 1#1: start worker process 31
2022/02/09 10:48:18 [notice] 1#1: start worker process 32
2022/02/09 10:48:18 [notice] 1#1: start worker process 33
2022/02/09 10:48:18 [notice] 1#1: start worker process 34
2022/02/09 10:48:18 [notice] 1#1: start worker process 35
2022/02/09 10:48:18 [notice] 1#1: start worker process 36
2022/02/09 10:48:18 [notice] 1#1: start worker process 37
2022/02/09 10:48:18 [notice] 1#1: start worker process 38
2022/02/09 11:02:48 [error] 32#32: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "localhost", referrer: "http://localhost/"
172.17.0.1 - - [09/Feb/2022:11:02:48 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "http://localhost/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36" "-"
```

2.4.5. Eliminar contenedores

Finalmente, vamos a eliminar los contenedores que hemos creado a partir de la imagen

de Nginx. Para ello, primero hemos de pararlos. Hasta el momento tenemos dos contenedores siendo ejecutados, y uno parado:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7c8da25c28b1	nginx	"/docker-entrypoint..."	29 minutes ago	Up 29 minutes	0.0.0.0:80->80/tcp	webhost
2256c6abc6f4	nginx	"/docker-entrypoint..."	17 hours ago	Up 48 minutes	0.0.0.0:8080->80/tcp	my_webhost
5773ba46a00f	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) 59 minutes ago		priceless_stonebraker

Vamos a parar los dos primeros con el siguiente comando (podemos parar uno o más contenedores, con su ID o NAME):

`docker container stop [ID|NAME] [ID|NAME] ...`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container stop 7c8da25c28b1 my_webhost
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7c8da25c28b1	nginx	"/docker-entrypoint..."	32 minutes ago	Exited (0) 7 seconds ago		webhost
2256c6abc6f4	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) 7 seconds ago		my_webhost
5773ba46a00f	nginx	"/docker-entrypoint..."	17 hours ago	Exited (0) About an hour ago		priceless_stonebraker

Como vemos, todos tienen estado Exited.

Ahora, vamos a borrar los dos primeros (vamos a reservar el tercero para un experimento final). El comando para eliminar contenedores es el siguiente:

`docker container rm [ID|NAME] [ID|NAME] ...`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rm 7c8da25c28b1 my_webhost
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5773ba46a00f	nginx	"/docker-entrypoint..."	18 hours ago	Exited (0) About an hour ago		priceless_stonebraker

Ahora solo nos queda un contenedor, en estado de parado, con el que vamos a hacer un experimento: vamos a intentar eliminarlo cuando está siendo ejecutado. Para ello primero lo reiniciamos, y a continuación intentamos borrarlo:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container start 5773ba46a00f
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rm 5773ba46a00f
```

```
Error response from daemon: You cannot remove a running container 5773ba46a00f50f9a00c9efa21c4ba7242f3695b0a4387f3731ff63a82d9a8cc. Stop the container before attempting removal or force remove
```

Docker nos devuelve un error indicando que hemos de parar el contenedor primero, lo cual es coherente. Pero, ¿qué ocurriría si necesitásemos eliminar un contenedor que ha quedado inconsistente y no podemos parar? Para esto, podemos utilizar el siguiente comando, que utiliza el parámetro `--force` (`-f`, abreviado):

`docker container rm -f [ID|NAME] [ID|NAME] ...`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rm -f 5773ba46a00f
5773ba46a00f
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
manu@manu-HP-Laptop-15s-fq1xxx:~\$						

En este caso, sí hemos podido eliminar el contenedor sin necesidad de pararlo primero.

Como apunte final, existe otra vía para eliminar contenedores (y muchos otros elementos en Docker), mediante el comando siguiente:

`docker system prune`

Pero este comando lo veremos con más detalle más tarde, en este documento, ya que tiene más implicaciones (no solo la de eliminar contenedores).

2.4.6. EJERCICIOS

TEST

- Si quisieses ver los contenedores que se están ejecutando y los que se han parado (sin eliminarlos), ¿qué comando utilizarías?
 docker container ls
 docker container ps
 docker container ls -a
 docker list containers
- ¿Qué hace el flag -d en un comando docker?
 Hace que el contenedor se ejecute en background, y retorna a la línea de comandos
 Publica un puerto que elijas
 -d significa detach, por tanto el contenedor se ejecutará en otro servidor
 -d significa delete, con lo cual eliminamos el contenedor
- Los siguientes comandos, ¿crearán un conflicto en los puertos?
 docker container run -p 80:80 -d nginx
 docker container run -p 8080:80 -d nginx
 Sí
 No
- Si lanzo el siguiente comando, me desaparece la línea de comandos y no responde, ¿por qué?
 docker container run -p 80:80 nginx
 Porque el puerto donde se está intentando lanzar el comando está en uso por otro contenedor
 Porque hay demasiados contenedores ejecutándose y el sistema se ha quedado sin memoria
 Porque no se ha especificado el flag -d, y aún no hay logs
 Porque la imagen de nginx se ha vuelto de pago

EJERCICIO PRÁCTICO

En este ejercicio vas a crear 3 contenedores: nginx, mysql y httpd (apache), con los siguientes requisitos:

- Ejecución en segundo plano
- Nginx ha de escuchar en 80:80, httpd en 8080:80, y mysql en 3306:3306
- Al ejecutar mysql, introducir al contenedor una variable de entorno (mediante el parámetro --env o -e), para pasarle al contenedor la variable MYSQL_RANDOM_ROOT_PASSWORD=yes. Apóyate en la [documentación](#) oficial en Docker Hub de este contenedor.
- Has de consultar los logs de mysql para encontrar el password aleatorio que se crea, gracias a la variable de entorno del paso anterior.
- Parar todos los contenedores, y borrarlos. Pero no utilices el método mostrado en el apartado anterior, sino en el método del siguiente [enlace](#). ¿Cuál es la diferencia? ¿Qué significan los diferentes flags de los comandos utilizados? ¿Cómo los reescribirías para que siguiesen la sintaxis recomendada?

2.5. Resolviendo el misterio: un contenedor es un proceso

Ya se ha mencionado anteriormente que los contenedores no son exactamente máquinas virtuales reducidas, sino que en realidad se trata de procesos que se ejecutan en el sistema host. Vamos a ver una demostración práctica para comprobarlo.

Ejecutemos un contenedor de la base de datos de MongoDB:

```
docker run --name mongo -d mongo
```

Comprobamos que el contenedor se ha lanzado:

```
docker container ls
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker run --name mongo -d mongo
Unable to find image 'mongo:latest' locally
latest: Pulling from library/mongo
08c01a0ec47e: Pull complete
ceb608a7cda7: Pull complete
a160d3e3934a: Pull complete
544b72923120: Pull complete
812461eda79e: Pull complete
3e1ac5db1dae: Pull complete
801c92a93fab: Pull complete
34e6068e2f4c: Pull complete
2513dc6d2ec7: Pull complete
d1ac55eb6bf: Pull complete
Digest: sha256:9ae745b709512a09a8c105959f75bde3d8a25c3215842a8251c073e14cd2a04d
Status: Downloaded newer image for mongo:latest
832cfca7a24565947ca51947b0d6bebf0b5c3b9adbf2ad31dc0cc1bf539a7
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
832cfca7a	mongo	"docker-entrypoint.s..."	5 minutes ago	Up 5 minutes	27017/tcp	mongo

Veamos ahora los procesos que se están ejecutando dentro del contenedor, con el siguiente comando:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker top mongo
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
999	6289	6272	0	16:01	?	00:00:03	mongod --bind_ip_all

Ahora vemos el proceso correspondiente al contenedor que se está ejecutando en el sistema host, mediante el siguiente comando:

```
ps aux | grep mongo
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ ps aux | grep mongo
```

999	6289	0.6	0.7	1539884	114368	?	Ssl	16:01	0:07	mongod --bind_ip_all
manu	7461	0.0	0.0	11568	736	pts/0	S+	16:22	0:00	grep --color=auto mongo

Por tanto, vemos que el proceso correspondiente al contenedor no se encuentra detrás de una máquina virtual de ningún tipo, sino que los podemos ver como un proceso más en el

sistema host.

Ahora vamos a parar el contenedor:

`docker stop mongo`

Ahora volvamos a consultar el proceso del sistema host:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ ps aux | grep mongo
manu      7843  0.0  0.0 11568  724 pts/0    R+   16:27   0:00 grep --color=auto mongo
```

El proceso que se lista corresponde precisamente al comando que acabamos de ejecutar (ps aux | grep mongo).

2.6. Inspección de contenedores

Los contenedores se comportan como cajas negras, tienen carácter efímero e inmutable (este concepto se abordará más en el apartado de volúmenes). Pero no por ello perdemos la posibilidad de poder analizar qué está sucediendo dentro de ellos. Vamos a verlo con un ejemplo.

Lanzamos dos contenedores con los siguientes comandos:

`docker container run -d --name nginx nginx`

`docker container run -d --name mysql -e MYSQL_RANDOM_ROOT_PASSWORD=true mysql`

Con el siguiente comando vamos a poder listar los procesos que se están ejecutando dentro de cada contenedor:

`docker container top mysql`

`docker container top nginx`


```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container top mysql
UID          PID          PPID         C           STIME        TTY          TIME         CMD
999          9792         9773         0           16:55        ?            00:00:00    /bin/bash
/usr/local/bin/docker-entrypoint.sh mysqld
999          9931         9792         35          16:55        ?            00:00:00    mysqld --d
aemonize --skip-networking --default-time-zone=SYSTEM --socket=/var/run/mysqld/mysqld.sock
999          9979         9792         1           16:55        ?            00:00:00    mysql_tzin
fo_to_sql /usr/share/zoneinfo
999          9980         9792         0           16:55        ?            00:00:00    sed s/Loca
l time zone must be set--see zic manual page/ECTY/
999          9981         9792         0           16:55        ?            00:00:00    /bin/bash
/usr/local/bin/docker-entrypoint.sh mysqld
999          9983         9981         6           16:55        ?            00:00:00    mysql --de
faults-extra-file=/dev/fd/63 --protocol=socket -uroot -hlocalhost --socket=/var/run/mysqld/mysqld.sock --comments --database=mysql
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container top nginx
UID          PID          PPID         C           STIME        TTY          TIME         CMD
root         9445         9426         0           16:54        ?            00:00:00    nginx: mas
ter process nginx -g daemon off;
systemd+    9507         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9508         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9509         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9510         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9511         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9512         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9513         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
systemd+    9514         9445         0           16:54        ?            00:00:00    nginx: wor
ker process
```

Y con el siguiente comando comando podemos comprobar los parámetros de configuración de un contenedor:

`docker container inspect [ID|NAME]`

Si inspeccionamos el contenedor de mysql:

`docker container inspect mysql`

podemos ver, por ejemplo, el valor de la variable MYSQL_RANDOM_ROOT_PASSWORD que hemos configurado al iniciar el contenedor:

```
"Env": [
  "MYSQL_RANDOM_ROOT_PASSWORD=true",
  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  "GOSU_VERSION=1.14",
  "MYSQL_MAJOR=8.0",
  "MYSQL_VERSION=8.0.28-1debian10"
],
```

Además de todo ello, disponemos de otro comando para averiguar datos en tiempo real sobre el rendimiento de los contenedores:

`docker container stats`

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
0530f5863c30	mysql	0.14%	356.9MiB / 15.4GiB	2.26%	10.1kB / 0B	352kB / 246MB	37
0b828a9f2f9d	nginx	0.00%	9.137MiB / 15.4GiB	0.06%	11.7kB / 0B	0B / 8.19kB	9

2.7. Interacción con contenedores

Otra cuestión que nos puede surgir en el uso de contenedores es la siguiente: ¿podemos alterar un contenedor que está en curso o interactuar con él? La respuesta es sí, y lo podemos realizar con los siguientes comandos:

```
docker container run -it
```

```
docker container exec -it
```

Vamos a utilizar varios ejemplos de estos comandos para conocer su uso.

En primer lugar, para ser autosuficientes, vamos a indagar qué significan los parámetros `-it` (en realidad son dos: `-i` y `-t`). Para ello ejecutamos:

```
docker container run --help
```

Este comando nos devuelve una lista extensa de parámetros, pero podemos localizar fácilmente los que estamos buscando:

```
--health-timeout duration    Maximum time to allow one check to run (ms|s|m|...  
--help                      Print usage  
-h, --hostname string       Container host name  
--init                      Run an init inside the container that forwards  
-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)
```

```
--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>)
```

Resumiendo la información, podríamos decir que:

`-i`: deja la sesión abierta para poder ejecutar más comandos

`-t`: pseudo-tty, abre una terminal (similar a SSH)

Estos parámetros se pueden especificar con varias combinaciones:

--interactive --tty, -i -t, -t -i, -it, -ti

Vamos a lanzar un contenedor de nginx, llamado proxy, de forma que podamos ejecutar comandos dentro de él:

`docker container run -it --name proxy nginx bash`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -it --name proxy nginx bash
root@c54776ca41cf:/#
```

Podemos ver que se abre un terminal dentro del contenedor que se está ejecutando, y la línea de comandos cambia, porque ahora se está ejecutando dentro del contenedor (usuario root del contenedor). Si ejecutamos "ls -al" podremos visualizar el contenido del directorio root del contenedor:

```
root@c54776ca41cf:/# ls -al
total 80
drwxr-xr-x  1 root root 4096 Feb  9 16:48 .
drwxr-xr-x  1 root root 4096 Feb  9 16:48 ..
-rwxr-xr-x  1 root root    0 Feb  9 16:48 .dockerenv
drwxr-xr-x  2 root root 4096 Jan 25 00:00 bin
drwxr-xr-x  2 root root 4096 Dec 11 17:25 boot
drwxr-xr-x  5 root root  360 Feb  9 16:48 dev
drwxr-xr-x  1 root root 4096 Jan 26 08:58 docker-entrypoint.d
-rwxrwxr-x  1 root root 1202 Jan 26 08:58 docker-entrypoint.sh
drwxr-xr-x  1 root root 4096 Feb  9 16:48 etc
drwxr-xr-x  2 root root 4096 Dec 11 17:25 home
drwxr-xr-x  1 root root 4096 Jan 25 00:00 lib
drwxr-xr-x  2 root root 4096 Jan 25 00:00 lib64
drwxr-xr-x  2 root root 4096 Jan 25 00:00 media
drwxr-xr-x  2 root root 4096 Jan 25 00:00 mnt
drwxr-xr-x  2 root root 4096 Jan 25 00:00 opt
dr-xr-xr-x 405 root root    0 Feb  9 16:48 proc
drwx-----  2 root root 4096 Jan 25 00:00 root
drwxr-xr-x  3 root root 4096 Jan 25 00:00 run
drwxr-xr-x  2 root root 4096 Jan 25 00:00 sbin
drwxr-xr-x  2 root root 4096 Jan 25 00:00 srv
dr-xr-xr-x 13 root root    0 Feb  9 16:48 sys
drwxrwxrwt  1 root root 4096 Jan 26 08:58 tmp
drwxr-xr-x  1 root root 4096 Jan 25 00:00 usr
drwxr-xr-x  1 root root 4096 Jan 25 00:00 var
root@c54776ca41cf:/#
```

Si queremos salir del terminal del contenedor, introducimos exit, y volvemos a la línea de comandos del host:

```
root@c54776ca41cf:/# exit
exit
manu@manu-HP-Laptop-15s-fq1xxx:~$
```

Al hacer exit, hacemos docker container ls -a (para ver todos los contenedores, incluso los parados). Queremos ver exactamente el contenido de la columna COMMAND:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c54776ca41cf	nginx	"/docker-entrypoint..."	7 minutes ago	Exited (0) About a minute ago		proxy
0530f5863c30	mysql	"docker-entrypoint.s..."	About an hour ago	Up About an hour	3306/tcp, 33060/tcp	mysql
0b828a9f2f9d	nginx	"/docker-entrypoint..."	About an hour ago	Up About an hour	80/tcp	nginx
832cfcfcaa7a	mongo	"docker-entrypoint.s..."	2 hours ago	Exited (0) About an hour ago		mongo

Vemos que el contenido de COMMAND se trunca y no podemos ver todo el texto. Para que no se trunque, ejecutamos el mismo comando, pero con el parámetro --no-trunc:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls -a --no-trunc
```

CONTAINER ID	STATUS	PORTS	NAMES	IMAGE	COMMAND	CREATED
c54776ca41cf	Exited (0) 3 minutes ago		proxy	nginx	"/docker-entrypoint.sh bash"	9 minutes ago
0530f5863c30	Up About an hour	3306/tcp, 33060/tcp	mysql	mysql	"docker-entrypoint.sh mysqld"	About an hour
0b828a9f2f9d	Up About an hour	80/tcp	nginx	nginx	"/docker-entrypoint.sh nginx -g 'daemon off;'"	About an hour
832cfcfcaa7a	Exited (0) 2 hours ago		mongo	mongo	"docker-entrypoint.sh mongod"	2 hours ago

Podemos apreciar que existe un contenedor con COMMAND bash, que es el que abrimos anteriormente. Poner bash al final implica que el contenedor se ejecute de forma diferente a si no lo hubiésemos puesto. En este caso, abre el shell del contenedor. Esto está relacionado con la directiva CMD de la imagen, que es el comando de entrada por el que se ejecuta el contenedor, y que se puede sobrescribir en el momento de lanzar el contenedor.

A continuación vamos a ejecutar un contenedor de ubuntu de la misma forma:

```
docker container run -it --name ubuntu ubuntu
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -it --name ubuntu ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
08c01a0ec47e: Already exists
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f474be
Status: Downloaded newer image for ubuntu:latest
root@03315f4eb461:/#
```


La imagen de ubuntu, en este caso, tiene como CMD el bash (como se puede ver en el Dockerfile de su [imagen oficial](#)), con lo cual no es necesario especificarlo como punto de entrada. Al levantarlo se muestra el bash y se puede hacer apt-get update, por ejemplo.

Un contenedor de ubuntu es mucho más ligero que una máquina virtual de ubuntu. Si instalamos algo mediante el bash, se mantendrá solo para ese contenedor. Si eliminamos el contenedor y levantamos otro, no va a tener las modificaciones que hayamos hecho al contenedor anterior.

Hasta ahora lo que hemos hecho es lanzar un nuevo contenedor y abrir el bash como punto de entrada, pero: ¿podríamos interactuar de la misma forma con un contenedor que y está siendo ejecutado aunque su punto de entrada no sea bash? La respuesta es sí.

Vamos a interactuar con el contenedor existente "mysql". Comprobamos que está activo:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
0530f5863c30	mysql	"docker-entrypoint.s..."	2 hours ago	Up 2 hours	3306/tcp, 33060/tcp	mysql
0b828a9f2f9d	nginx	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	nginx

Ahora abrimos la línea de comandos dentro del contenedor activo, con el siguiente comando:

```
docker container exec -it mysql bash
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container exec -it mysql bash
root@0530f5863c30:/#
```

Ahora podríamos ejecutar cualquier comando que necesitemos lanzar dentro del contenedor.

Para seguir ilustrando la utilización de estos comandos, vamos a utilizar la imagen "alpine", que se trata de una distribución de linux muy ligera, se puede utilizar para construir imágenes a partir de ella. Para descargarla ejecutamos:

docker pull alpine

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
59bf1c3509f3: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

Para ver el tamaño de esta imagen podemos ejecutar el siguiente comando:

docker image ls

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	5285cb69ea55	7 days ago	698MB
ubuntu	latest	54c9d81cbb44	7 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	13 days ago	519MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
alpine	latest	c059bfaa849c	2 months ago	5.59MB

Vamos a intentar acceder al bash de alpine con el comando:

docker container run -it alpine bash

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -it alpine bash
docker: Error response from daemon: OCI runtime create failed: container_linux.go:349: starting container process caused "exec: \"bash\": executable f
ile not found in $PATH": unknown.
[0000] error waiting for container: context canceled
```

Para nuestra sorpresa, docker nos devuelve un error. Y es debido al hecho de que alpine es una versión tan ligera, que no tiene ni siquiera bash instalado.

Si consultamos el [Dockerfile](#) de alpine, vemos que la directiva CMD es el ejecutable sh, y por ello vamos a ejecutar el siguiente comando:

docker container run -it alpine sh

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -it alpine sh
/ #
```

Ahora podemos instalar bash mediante el gestor de paquetes apk, que es el que tiene instalado alpine, con las instrucciones:

apk update

apk upgrade

apk add bash

Y tras finalizar el proceso de instalación de bash, podemos abrir dicha terminal:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -it alpine sh
/ # apk update
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86_64/APKINDEX.tar.gz
v3.15.0-258-gd5d32e35cf [https://dl-cdn.alpinelinux.org/alpine/v3.15/main]
v3.15.0-268-gaf6454ec89 [https://dl-cdn.alpinelinux.org/alpine/v3.15/community]
OK: 15857 distinct packages available
/ # apk upgrade
(1/5) Upgrading busybox (1.34.1-r3 -> 1.34.1-r4)
Executing busybox-1.34.1-r4.post-upgrade
(2/5) Upgrading ca-certificates-bundle (20191127-r7 -> 20211220-r0)
(3/5) Upgrading libcrypto1.1 (1.1.1l-r7 -> 1.1.1l-r8)
(4/5) Upgrading libssl1.1 (1.1.1l-r7 -> 1.1.1l-r8)
(5/5) Upgrading ssl_client (1.34.1-r3 -> 1.34.1-r4)
Executing busybox-1.34.1-r4.trigger
OK: 6 MiB in 14 packages
/ # apk add bash
(1/4) Installing ncurses-terminfo-base (6.3_p20211120-r0)
(2/4) Installing ncurses-libs (6.3_p20211120-r0)
(3/4) Installing readline (8.1.1-r0)
(4/4) Installing bash (5.1.8-r0)
Executing bash-5.1.8-r0.post-install
Executing busybox-1.34.1-r4.trigger
OK: 8 MiB in 18 packages
/ # bash
bash-5.1#
```

3. Redes

Cuando se crean diferentes servicios o aplicaciones en contenedores distintos (siguiendo la premisa de microservicios), estos no están conectados entre sí, en el caso que necesitarán conexión se utilizarían "redes Docker". Este mecanismo funciona de manera distinta según la red docker donde estén conectados los contenedores.

A continuación se introducen los distintos tipos de redes que nos ofrece Docker:

- **Bridge:** red por defecto. Todos los contenedores están en la misma red, separada del Host. Hablaremos de esta red más abajo.

- **Host:** cuando quiero que los contenedores estén en la misma red del host. Evita la utilización de las redes privadas virtuales de Docker pero hace que aumente la eficiencia en la comunicación entre contenedores, a costa de sacrificar seguridad entre contenedores.
- **None:** contenedores aislados, equivalente a tener una interfaz de red no conectada.

3.1. Por qué crear nuevas redes virtuales

Cuando un contenedor se crea y no se indica una red, se incluye en la red virtual por defecto bridge. Dentro de una misma red virtual, se realiza un enrutamiento del tráfico a través de un firewall NAT sobre la IP del host. O dicho en otras palabras: el firewall NAT es en realidad el demonio de Docker que configura el acceso a Internet por parte de los contenedores, usando la IP del host).

En general, todos los contenedores en una red virtual pueden hablar entre sí, sin especificar el parámetro -p. Pero este no es el caso de la red por defecto bridge.

La práctica recomendada es establecer una red privada virtual para cada app, de forma que los contenedores que pertenecen a la app pertenezcan a la misma red y se puedan comunicar sin exponer sus puertos al exterior.

Siguiendo estas premisas, podríamos encontrarnos un escenario como el siguiente:

- Red virtual 1: conecta un contenedor MySQL y un contenedor Apache, pertenecientes a la aplicación 1.
- Red virtual 2: conecta un contenedor MongoDB con un contenedor Nodejs, pertenecientes a la aplicación 2.

En este escenario, los contenedores de la aplicación 1 pueden hablar entre ellos sin

exponer puertos, pero no con los de la aplicación 2.

Aunque el comportamiento de las redes de Docker es adaptable, existen ventajas en la configuración por defecto (red bridge), para las tareas de desarrollo:

- Frontend/backend siempre están en la misma red
- La comunicación entre contenedores no sale del host
- Los puertos externos se cierran por defecto
- Se han de exponer los puertos explícitamente con -p

Todo esto se mejora mucho más con Docker Swarm y las redes Overlay.

3.2. Configuración IP en host y en contenedores

En este apartado vamos a comparar la IP de nuestro host con la IP de uno de los contenedores Docker.

Levantamos un contenedor mediante el comando:

```
docker container run -p 80:80 --name webhost -d nginx
```

A continuación ejecutamos el siguiente comando, que muestra el puerto del host que está reenviando tráfico al puerto del contenedor:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -p 80:80 --name webhost -d nginx
05d350b3abb23ad98deeffb13e8628084dba5ff077af77ad88ec2dd006f207d2
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container port webhost
80/tcp -> 0.0.0.0:80
```

Ahora vamos a averiguar la IP del contenedor con el siguiente comando, y nos devuelve su IP:

```
docker container inspect webhost --format "{{ .NetworkSettings.IPAddress }}"
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container inspect webhost --format "{{ .NetworkSettings.IPAddress }}"
172.17.0.2
```

Si ahora ejecutamos el comando ifconfig (o el equivalente al sistema operativo en el que estemos trabajando), podremos comprobar, analizando las interfaces de red de nuestro

sistema host, que no se trata de la misma dirección IP ni la misma subred.

3.3. Operaciones con redes

En este apartado vamos a realizar las operaciones más usuales con redes Docker.

En primer lugar listamos todas las redes virtuales con el siguiente comando:

`docker network ls`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
589c2c676efe        bridge             bridge             local
6750ede529e5        host               host               local
6c0762cea506        none              null               local
```

A continuación vamos a crear una nueva red llamada my_app_net:

`docker network create my_app_net`

Vamos a ver con qué driver se ha creado esta nueva red con docker network ls:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker network create my_app_net
8544700148bdf07c77d5954021cad743dc6ea7e6891e48aa9feb94cc87a4d4fc
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
589c2c676efe        bridge             bridge             local
6750ede529e5        host               host               local
8544700148bd        my_app_net         bridge             local
6c0762cea506        none              null               local
```

Dependiendo del driver de una red virtual, se le puede dotar de características adicionales (aunque no vamos a indagar más en este apartado).

A continuación vamos a lanzar un nuevo contenedor y lo vamos a incluir en la red que acabamos de crear:

`docker container run -d --name new_nginx --network my_app_net nginx`

Con el siguiente comando comprobamos qué contenedores están incluidos en una determinada red:

`docker network inspect my_app_net`



GENERALITAT
VALENCIANA
Conselleria de Educació,
Cultura y Esports

TOTS
A UNA
VEU



ies
severo ochoa

Carrer d'illueca, 28
03206 Elx, Espanya
03013224.secret@gva.es
(+34) 966.912.260



Fondo Social Europeo

El FSE invierte en tu futuro

UNIÓN EUROPEA

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -d --name new_nginx --network my_app_net nginx
430ad64471a8cd1ad885d710aafccc6e62b7c01bd895a87672bcc1d97926af12
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker network inspect my_app_net
[
  {
    "Name": "my_app_net",
    "Id": "8544700148bdf07c77d5954021cad743dc6ea7e6891e48aa9feb94cc87a4d4fc",
    "Created": "2022-02-11T11:12:28.764155278+01:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "430ad64471a8cd1ad885d710aafccc6e62b7c01bd895a87672bcc1d97926af12": {
        "Name": "new_nginx",
        "EndpointID": "a132343c80ef50f180650cb771bb775032f54acf5b142666133ab148f02052c6",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
      }
    }
  }
]
```

Tras esto vamos a conectar el contenedor webhost que ya habíamos lanzado, a nuestra nueva red:

```
docker network connect [ID|NOMBRE red] [ID|NOMBRE contenedor]
```

Si introducimos valores concretos:

```
docker network connect my_app_net webhost
```

Y si inspeccionamos el contenedor webhost, podremos ver las redes a las que está conectado, que serán la bridge y my_app_net:

```
docker container inspect webhost
```

```
"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "589c2c676efe8da3123de86768be569a12fa0f99e862327fc8d1fa1598d6d5b2",
    "EndpointID": "b8c34954a2a1fb425ec3e7fac8c871e23f437d39a97585a2288536d3c6d90117",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02",
    "DriverOpts": null
  },
  "my_app_net": {
    "IPAMConfig": {},
    "Links": null,
    "Aliases": [
      "05d350b3abb2"
    ],
    "NetworkID": "8544700148bdf07c77d5954021cad743dc6ea7e6891e48aa9feb94cc87a4d4fc",
    "EndpointID": "b5dbc545a91355730431c509fe21c4faeea6a76640ad412998bef8eb2c288626",
    "Gateway": "172.18.0.1",
    "IPAddress": "172.18.0.3",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:12:00:03",
    "DriverOpts": {}
  }
}
```

Por último, vamos a desconectar webhost de my_app_net con el comando:

`docker network disconnect my_app_net webhost`

Y si inspeccionamos el contenedor veremos que ahora solo está conectado a una red:

```
"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "589c2c676efe8da3123de86768be569a12fa0f99e862327fc8d1fa1598d6d5b2",
    "EndpointID": "b8c34954a2a1fb425ec3e7fac8c871e23f437d39a97585a2288536d3c6d90117",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02",
    "DriverOpts": null
  }
}
```

3.4. DNS entre contenedores

Aunque los contenedores, como se ha visto anteriormente, tienen asignada una dirección IP en su configuración de red, ésta no se utiliza para la conexión entre ellos mismos, por

su carácter volátil o efímero. El demonio de Docker tiene un servidor DNS por defecto que los contenedores usan por defecto y es el que se encarga de permitir su interconexión.

Al crear una nueva red, ésta va a tener asociado un servidor DNS que va a permitir que todos los contenedores de esa red se puedan comunicar. El nombre de host utilizado para la resolución DNS es el nombre del contenedor, pero se pueden crear alias para utilizar nombres diferentes.

Vamos a hacer una prueba con ello. Primero creamos un nuevo contenedor en la red `my_app_net`:

```
docker container run -d --name my_nginx --network my_app_net nginx
```

Desde el contenedor `my_nginx` vamos a hacer ping al contenedor `new_nginx`, creado en el apartado anterior. Para ello, debido a que no tenemos la utilidad ping dentro del contenedor `nginx`, hemos de instalarla. Entramos en el bash de `my_nginx`

```
docker container exec -it my_nginx bash
```

Una vez dentro, instalamos ping:

```
apt update
```

```
apt install iputils-ping
```

Y finalmente hacemos ping con el nombre DNS de `new_nginx`:

```
root@fffb935c1369d:/# ping new_nginx
PING new_nginx (172.18.0.2) 56(84) bytes of data.
64 bytes from new_nginx.my_app_net (172.18.0.2): icmp_seq=1 ttl=64 time=0.130 ms
64 bytes from new_nginx.my_app_net (172.18.0.2): icmp_seq=2 ttl=64 time=0.077 ms
64 bytes from new_nginx.my_app_net (172.18.0.2): icmp_seq=3 ttl=64 time=0.098 ms
```

Iniciamos de igual modo el bash de `new_nginx`, instalamos las utilidades de red como para `my_nginx`, y vemos que también tiene conectividad:

```
root@430ad64471a8:/# ping my_nginx
PING my_nginx (172.18.0.3) 56(84) bytes of data.
64 bytes from my_nginx.my_app_net (172.18.0.3): icmp_seq=1 ttl=64 time=0.097 ms
64 bytes from my_nginx.my_app_net (172.18.0.3): icmp_seq=2 ttl=64 time=0.077 ms
64 bytes from my_nginx.my_app_net (172.18.0.3): icmp_seq=3 ttl=64 time=0.076 ms
```




GENERALITAT
VALENCIANA
Conselleria de Educació,
Cultura i Esport

TOTS
A UNA
VEU



Carrer d'illueca, 28
03206 Elx, Espanya
03013224.secret@gva.es
(+34) 966.912.260



Formació Professional
Comunitat Valenciana



Fondo Social Europeo

El FSE invierte en tu futuro

Esta resolución DNS será fundamental para Docker Swarm.

En la red por defecto bridge, no existe la opción de resolución DNS entre los contenedores. Se pueden ver entre sí por IP, pero no por nombre de host.

Para poder conectar contenedores conectados a la red bridge, ésta dispone de la la opción --link (docker container create --help), que permite enlazar contenedores en una red, pero es más fácil crear una nueva red para enlazar contenedores.

3.5. EJERCICIOS

TEST

1. ¿Qué comando se utilizaría para ver información general sobre un contenedor concreto, como sus redes, dirección IP, estado actual,...?
 - a. docker container top
 - b. docker container stats
 - c. docker container inspect
 - d. docker container logs
2. Para conectar a un shell dentro de un contenedor, se ha de utilizar docker ssh:
 - a. Verdadero
 - b. Falso
3. Si quisiésemos que múltiples contenedores se pudiesen conectar en el mismo host, qué driver de red utilizaríamos:
 1. Container
 2. Overlay
 3. Bridge
 4. Swarm

PRÁCTICA

El objetivo de esta práctica consiste en utilizar la herramienta de curl en varias distribuciones de Linux. Para ello:

- Utilizarás dos terminales para arrancar el bash en centos y ubuntu, usando -it.
- Es necesario asegurarse de que curl está instalado en las dos distribuciones, e instalarlo si no lo está:
 - ubuntu: apt-get update && apt-get install curl
 - centos: yum update curl
 - curl --version
- Limpia el sistema de los contenedores creados. Para ello investiga el uso del siguiente comando, y utilízalo: docker container --rm

INVESTIGACIÓN

Investiga el uso de --alias en docker network connect y pon un ejemplo.

4. Imágenes

En este apartado vamos a profundizar más en las imágenes de Docker. Sabemos ya que un contenedor se crea a partir de una imagen, y por tanto la imagen es anterior al contenedor, pero gracias a que ya hemos creado múltiples contenedores a partir de una imagen, tenemos una idea intuitiva de lo que puede ser una imagen a groso modo. Hasta ahora imaginamos una imagen como una plantilla que nos permite crear contenedores, que dicta el software que se ha de ejecutar en estos contenedores, además de otras características de su comportamiento.

Como definición formal, podríamos decir que una imagen es una colección ordenada de los cambios del sistema de archivos root y los correspondientes parámetros de ejecución para utilizar durante la ejecución de un contenedor determinado.

Esta definición en abstracto se podría desglosar en varios puntos:

- Una imagen contiene el conjunto de ficheros binarios, con sus dependencias, que forman una aplicación o pieza software.
- Una imagen tiene asociados metadatos e instrucciones sobre cómo ejecutar los contenedores que se creen a partir de ella.
- Una imagen no es un sistema operativo completo, no contiene un kernel o módulos de kernel (drivers, etc.). Son solo los binarios que la aplicación requiere, ya que le host es quien provee el kernel.
- Una imagen puede llegar a ser tan pequeña como un fichero (por ejemplo, un binario de golang), o de mayor tamaño (como una distribución Ubuntu + Apache + PHP ...)

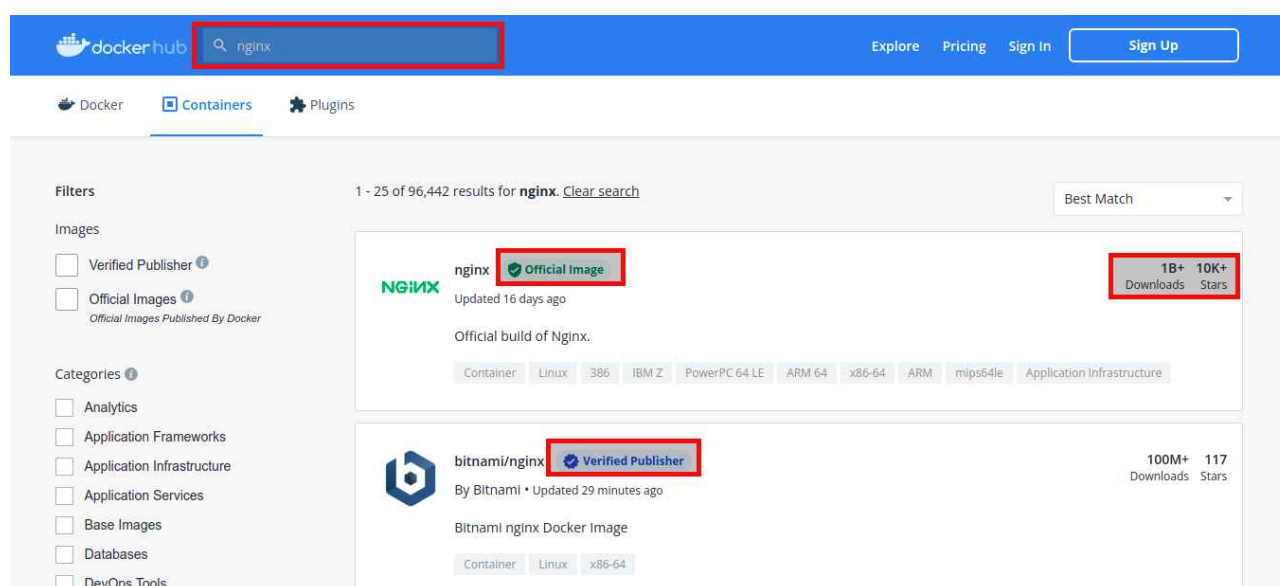
En los siguientes subapartados se desglosa mucho más todo lo concerniente a las

imágenes de Docker.

4.1. Docker Hub

El repositorio oficial de imágenes de Docker es [Docker Hub](#). Si aún no te has registrado en su versión gratuita, ¡hazlo ya!

Vamos a empezar buscando la imagen oficial de nginx. Introducimos 'nginx' en el buscador y vemos los resultados:



En primer lugar de la lista vemos la imagen oficial de nginx, marcado como "Official image" en verde. Además se muestra el número de veces que se ha descargado la imagen (o pulls), y las estrellas otorgadas por los usuarios.

En segundo lugar de la lista, en azul, tenemos una imagen catalogada como "Verified Publisher", también con gran cantidad de descargas y estrellas.

Estos parámetros (catalogación, estrellas, pulls...) dan una idea de la calidad de la imagen, de si el autor/es es más o menos fiable, etc.

Pulsamos sobre la primera imagen y consultamos la sección que empieza por "Supported tags":

Supported tags and respective Dockerfile links

- 1.21.6, mainline, 1, 1.21, latest
- 1.21.6-perl, mainline-perl, 1-perl, 1.21-perl, perl
- 1.21.6-alpine, mainline-alpine, 1-alpine, 1.21-alpine, alpine
- 1.21.6-alpine-perl, mainline-alpine-perl, 1-alpine-perl, 1.21-alpine-perl, alpine-perl
- 1.20.2, stable, 1.20
- 1.20.2-perl, stable-perl, 1.20-perl
- 1.20.2-alpine, stable-alpine, 1.20-alpine
- 1.20.2-alpine-perl, stable-alpine-perl, 1.20-alpine-perl

Cada uno de los bullets que se enumeran corresponden a una versión diferente de la imagen de nginx. Vemos, por ejemplo, que la última versión se identifica por "1.21.6", "mainline", "1", "1.21" o "latest". Estas identificaciones se denominan tags, o etiquetas, y cada grupo de etiquetas identifican una sola imagen, con un ID determinado.

Si quisiésemos descargar una imagen concreta, deberíamos hacerlo especificando el tag correspondiente. Por ejemplo, queremos descargar la versión 1.20.2 de Nginx, lo podemos hacer con el siguiente comando:

`docker pull nginx:1.20.2`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker pull nginx:1.20.2
1.20.2: Pulling from library/nginx
5eb5b503b376: Already exists
cdfcb356c029: Pull complete
d86da7454448: Pull complete
7976249980ef: Pull complete
8f66aa6726b2: Pull complete
c004cabebe76: Pull complete
Digest: sha256:02923d65cde08a49380ab3f3dd2f8f90aa51fa2bd358bd85f89345848f6e6623
Status: Downloaded newer image for nginx:1.20.2
docker.io/library/nginx:1.20.2
```

Las líneas que empiezan por un identificador (la primera dice "Already exists" y las demás

"Pull complete") son especialmente interesantes, y hablaremos de ello más adelante.

Si ahora intentásemos descargar la versión 1.20, que es la misma que la anterior, obtenemos el siguiente resultado:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker pull nginx:1.20
1.20: Pulling from library/nginx
Digest: sha256:02923d65cde08a49380ab3f3dd2f8f90aa51fa2bd358bd85f89345848f6e6623
Status: Downloaded newer image for nginx:1.20
docker.io/library/nginx:1.20
```

Es decir, Docker reconoce que esta imagen ya se ha descargado, incluso la línea que empieza por Digest tiene el mismo valor. Lo mismo sucede si lo intentamos con el tag "stable":

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker pull nginx:stable
stable: Pulling from library/nginx
Digest: sha256:02923d65cde08a49380ab3f3dd2f8f90aa51fa2bd358bd85f89345848f6e6623
Status: Downloaded newer image for nginx:stable
docker.io/library/nginx:stable
```

Pero si descargamos la etiqueta 1.20.2-alpine, realizará la descarga:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker pull nginx:1.20.2-alpine
1.20.2-alpine: Pulling from library/nginx
97518928ae5f: Pull complete
a15dfa83ed30: Pull complete
acae0b19bbc1: Pull complete
fd4282442678: Pull complete
b521ea0d9e3f: Pull complete
b3282d03aa58: Pull complete
Digest: sha256:74694f2de64c44787a81f0554aa45b281e468c0c58b8665fafceda624d31e556
Status: Downloaded newer image for nginx:1.20.2-alpine
docker.io/library/nginx:1.20.2-alpine
```

En este caso, ninguna de las líneas dice "Already exists".

4.2. Capas de imágenes

En primer lugar diremos que las imágenes y los contenedores se implementan mediante capas. Aunque las capas son transparentes cuando se utilizan los contenedores y las imágenes, son muy importantes para el funcionamiento interno de Docker.

Empecemos a indagar sobre las capas con un ejemplo práctico. Vamos a trabajar con la

última versión de nginx (la que tenga la etiqueta "latest"), por lo que lo comprobamos:

docker image ls

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	5285cb69ea55	11 days ago	698MB
ubuntu	latest	54c9d81cbb44	11 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	2 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	2 weeks ago	141MB
nginx	stable	d6c9558ba445	2 weeks ago	141MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
alpine	latest	c059d1a8849c	2 months ago	5.59MB
nginx	1.20.2-alpine	373f8d4d4c60	2 months ago	23.2MB

A continuación vamos a indagar sobre el historial de esta imagen:

docker history nginx:latest

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker history nginx:latest
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
c316d5a335a5	2 weeks ago	/bin/sh -c #(nop)	0B	CMD ["nginx" "-g" "daemon...
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	STOPSIGNAL SIGQUIT
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	EXPOSE 80
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	ENTRYPOINT ["/docker-entr...
<missing>	2 weeks ago	/bin/sh -c #(nop)	4.61kB	COPY file:09a214a3e07c919a...
<missing>	2 weeks ago	/bin/sh -c #(nop)	1.04kB	COPY file:0fd5fca330dcd6a7...
<missing>	2 weeks ago	/bin/sh -c #(nop)	1.96kB	COPY file:0b866ff3fc1ef5b0...
<missing>	2 weeks ago	/bin/sh -c #(nop)	1.2kB	COPY file:65504f71f5855ca0...
<missing>	2 weeks ago	/bin/sh -c set -x	61.1MB	&& addgroup --system -...
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	ENV PKG_RELEASE=1~bullseye
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	ENV NJS_VERSION=0.7.2
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	ENV NGINX_VERSION=1.21.6
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	LABEL maintainer=NGINX Do...
<missing>	2 weeks ago	/bin/sh -c #(nop)	0B	CMD ["bash"]
<missing>	2 weeks ago	/bin/sh -c #(nop)	80.4MB	ADD file:90495c24c897ec479...

Este comando muestra lo que se denomina Union file system, que consiste en los cambios realizados en una imagen, traducido como el historial de las capas de la imagen nginx para la captura anterior. En este [enlace](#) puedes encontrar información más detallada sobre el Union file system de Docker.

Imágenes diferentes, dan diferentes resultados para el comando anterior. Por ejemplo, para mysql:


```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker history mysql
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
d1dc36cf8d9e	2 weeks ago	/bin/sh -c #(nop) CMD ["mysqld"]	0B	
<missing>	2 weeks ago	/bin/sh -c #(nop) EXPOSE 3306 33060	0B	
<missing>	2 weeks ago	/bin/sh -c #(nop) ENTRYPOINT ["docker-entryp...	0B	
<missing>	2 weeks ago	/bin/sh -c ln -s usr/local/bin/docker-entryp...	34B	
<missing>	2 weeks ago	/bin/sh -c #(nop) COPY file:c112ec3a02a7b818...	13.2kB	
<missing>	2 weeks ago	/bin/sh -c #(nop) COPY dir:2e040acc386ebd23b...	1.12kB	
<missing>	2 weeks ago	/bin/sh -c #(nop) VOLUME [/var/lib/mysql]	0B	
<missing>	2 weeks ago	/bin/sh -c { echo mysql-community-server m...	384MB	
<missing>	2 weeks ago	/bin/sh -c echo 'deb http://repo.mysql.com/a...	55B	
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV MYSQL_VERSION=8.0.28-...	0B	
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV MYSQL_MAJOR=8.0	0B	
<missing>	2 weeks ago	/bin/sh -c set -ex; key='859BE8D7C586F53843...	2.29kB	
<missing>	2 weeks ago	/bin/sh -c apt-get update && apt-get install...	52.2MB	
<missing>	2 weeks ago	/bin/sh -c mkdir /docker-entrypoint-initdb.d	0B	
<missing>	2 weeks ago	/bin/sh -c set -eux; savedAptMark="\$(apt-ma...	4.06MB	
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV GOSU_VERSION=1.14	0B	
<missing>	2 weeks ago	/bin/sh -c apt-get update && apt-get install...	9.34MB	
<missing>	2 weeks ago	/bin/sh -c groupadd -r mysql && useradd -r -...	329kB	
<missing>	2 weeks ago	/bin/sh -c #(nop) CMD ["bash"]	0B	
<missing>	2 weeks ago	/bin/sh -c #(nop) ADD file:c51141702f568a28a...	69.3MB	

Cabe mencionar las siguientes características:

- Los cambios se leen de abajo a arriba, por eso el primero suele ocupar más espacio.
- La única capa que tiene un IMAGE ID es la última.
- Las capas que son comunes a diferentes imágenes (por ejemplo, dos imágenes que se basan en una imagen de ubuntu), se guardan como una sola en la caché de docker. Si ya se dispone de una capa en la caché local, no se descarga para una nueva imagen al hacer pull. Esto mejora la eficiencia en el almacenamiento (en el host) y la descarga de capas.
- Cuando se crea un contenedor, Docker crea una capa read/write encima de las capas de la imagen. La imagen representaría un fichero de solo lectura.
- Copy on write: cuando se quiere modificar un fichero de la imagen base, se copia ese fichero a la capa del contenedor. Por tanto en este caso el contenedor es una capa (de lectura/escritura) encima de las capas de la imagen, más los ficheros modificados de la imagen base.

Finalmente, para conocer los metadatos de una imagen, podemos utilizar el siguiente

comando:

`docker image inspect [IMAGEN]`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image inspect nginx
[
  {
    "Id": "sha256:c316d5a335a5cf324b0dc83b3da82d7608724769f6454f6d9a621f3ec2534a5a",
    "RepoTags": [
      "nginx:latest"
    ],
    "RepoDigests": [
      "nginx@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767"
    ],
    "Parent": "",
    "Comment": "",
    "Created": "2022-01-26T08:58:35.041664322Z",
    "Container": "f7debc76f8a9f5c1eb8bad0366aaa442b92d4dd0569989d99b0900b2192879ea",
    "ContainerConfig": {
      "Hostname": "f7debc76f8a9",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "ExposedPorts": {
        "80/tcp": {}
      },
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "NGINX_VERSION=1.21.6",
        "NJS_VERSION=0.7.2",
        "PKG_RELEASE=1~bullseye"
      ]
    }
  ]
]
```

4.3. Etiquetas de imágenes

Las imágenes no se identifican con un nombre, lo podemos comprobar al ejecutar el siguiente comando (no existe ninguna columna "Name" o similar):

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	5285cb69ea55	13 days ago	698MB
ubuntu	latest	54c9d81cbb44	13 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	2 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	2 weeks ago	141MB
nginx	stable	d6c9558ba445	2 weeks ago	141MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
alpine	latest	c059bfaa849c	2 months ago	5.59MB
nginx	1.20.2-alpine	373f8d4d4c60	3 months ago	23.2MB
dataiku/dss	latest	f24fbb6d95fe	15 months ago	2.75GB

La forma de identificarlas es mediante un identificador de imagen (IMAGE ID) o la combinación de repositorio y tag. El repositorio, a su vez, puede contener el nombre de la organización. Comparemos, por ejemplo, las siguientes imágenes marcadas en rojo:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	5285cb69ea55	13 days ago	698MB
ubuntu	latest	54c9d81cbb44	13 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	2 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	2 weeks ago	141MB
nginx	stable	d6c9558ba445	2 weeks ago	141MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
alpine	latest	c059bfaa849c	2 months ago	5.59MB
nginx	1.20.2-alpine	373f8d4d4c60	3 months ago	23.2MB
dataiku/dss	latest	f24fbb6d95fe	15 months ago	2.75GB

Empecemos por la última: dataiku/dss. En este caso, al nombre del repositorio (dss) se antepone el nombre de la organización (dataiku).

La imagen de mongo, sin embargo, no contiene el nombre de la organización. Esto se debe a que es una imagen oficial. Las imágenes oficiales no llevan el nombre de la organización que las ha desarrollado.

The screenshot shows the Docker Hub search results for 'nginx'. The search bar at the top contains 'nginx'. Below the search bar, there are filters for 'Verified Publisher' and 'Official Images'. The search results are sorted by 'Best Match'. The first result is 'nginx' by 'Official Image', which is highlighted with a red box. It shows 'Updated 20 days ago' and 'Official build of Nginx.' The second result is 'bitnami/nginx' by 'Verified Publisher', which is also highlighted with a red box. It shows 'By Bitnami • Updated 8 minutes ago' and 'Bitnami nginx Docker Image'.

Al buscar nginx, la primera imagen corresponde a la oficial, no tiene el nombre de la organización delante. El segundo resultado corresponde a una imagen de nginx publicada

por la organización bitnami.

NOTA: normalmente, suele existir una imagen igual que la oficial, con el nombre del repositorio perteneciente a la organización que la creó, siendo ambas la misma imagen.

Revisado el concepto de repositorio de una imagen, ¿en qué consiste entonces un tag o etiqueta? **No** se trata de la versión o la rama (como en GitHub), sino un **commit específico** dentro del repositorio. Por ejemplo, mysql tiene los siguientes tags:



8.0.28, 8.0, 8, latest: hacen referencia al mismo commit.

Los tags no hacen referencia solo a versiones de una imagen, representan también nombres que identifican las versiones. Si descargamos dos veces la misma imagen con dos tags diferentes, la segunda vez se detecta que ya existe esa imagen en la caché local, por el identificador de la imagen que es común a los dos tags.

4.4. Etiquetado y pull a Docker Hub

En este apartado vamos a etiquetar una de las imágenes de que disponemos en la caché local y subirla a un repositorio propio de Docker Hub.

El comando utilizado para etiquetar imágenes es el siguiente:

`docker image tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]`

Con ello, vamos a renombrar la imagen oficial de nginx de la que disponemos en local, para poder subirla a Docker Hub, anteponiendo el nombre de nuestra organización. Ejecutamos el siguiente comando:

`docker image tag nginx manuprofeinfo/nginx`

En este caso, "nginx" es el nombre del repositorio de Docker Hub, de nuestra cuenta. Puede o no existir en este momento, más abajo se explica esto con más detalle.

Al no especificar el tag tras el nombre de las imágenes, Docker entiende que el tag es "latest". Si listamos las imágenes vemos que la columna IMAGE ID tiene el mismo valor para varias de ellas:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	5285cb69ea55	13 days ago	698MB
ubuntu	latest	54c9d81cbb44	13 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	2 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	2 weeks ago	141MB
nginx	stable	d6c9558ba445	2 weeks ago	141MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
manuprofeinfo/nginx	latest	c316d5a335a5	2 weeks ago	142MB
alpine	latest	c059bf8a849c	2 months ago	5.59MB
nginx	1.20.2-alpine	373f8d4d4c60	3 months ago	23.2MB
dataiku/dss	latest	f24fbb6d95fe	15 months ago	2.75GB

Antes de hacer el push hemos de hacer login en nuestra cuenta de Docker Hub, mediante el comando `docker login`. Podemos ver las diferentes formas en que podemos utilizar este comando poniendo `--help` al final. En este caso vamos a especificar el usuario, para que nos pida el password:

`docker login -u manuprofeinfo`


```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker login -u manuprofeinfo
Password:
WARNING! Your password will be stored unencrypted in /home/manu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

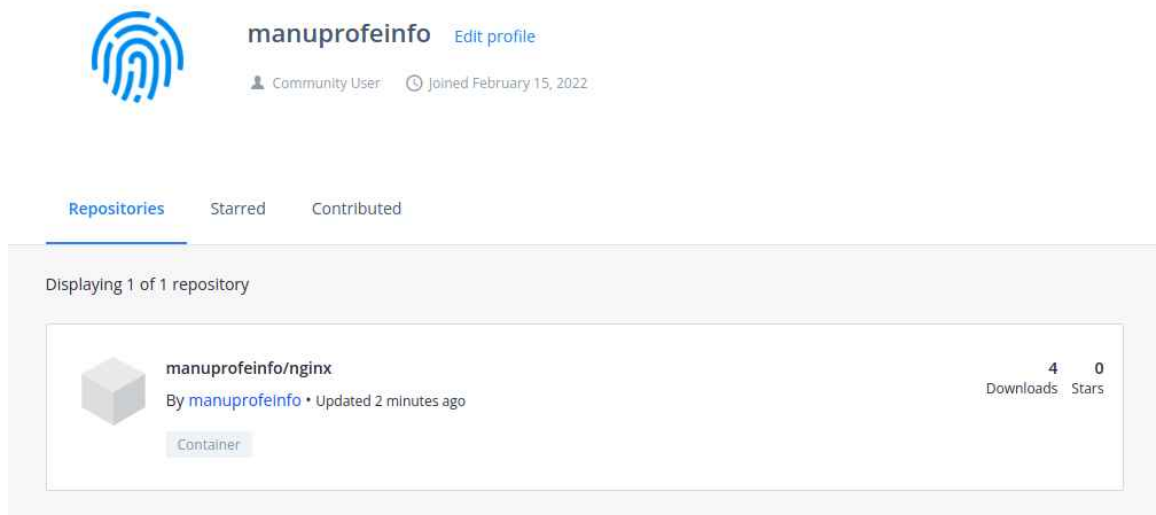
Login Succeeded
```

Ahora ya podemos volver a ejecutar el push:

`docker image push manuprofeinfo/nginx`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image push manuprofeinfo/nginx
The push refers to repository [docker.io/manuprofeinfo/nginx]
762b147902c0: Pushed
235e04e3592a: Pushed
6173b6fa63db: Pushed
9a94c4a55fe4: Pushed
9a3a6af98e18: Pushed
7d0ebbe3f5d2: Pushed
latest: digest: sha256:bb129a712c2431ecce4af8dde831e980373b26368233ef0f3b2bae9e9ec515ee size: 1570
```

Si vamos a nuestro repositorio de Docker Hub podremos ver que se ha creado esta nueva imagen:



Pinchando en el repositorio, vemos los tags:

manuprofeinfo/nginx

By [manuprofeinfo](#) • Updated 3 minutes ago

Container

[Manage Repository](#)

Pulls 5

Overview **Tags**

Advanced Image Management
View all your images and tags in this repository, clean up unused content, recover untagged images. Available with Pro, Team and Business subscriptions. [View preview](#)

Sort by Newest

TAG	DIGEST	OS/ARCH	LAST PULL	COMPRESSED SIZE
latest Last pushed 3 minutes ago by manuprofeinfo	bb129a712c24	linux/amd64	3 minutes ago	54.09 MB

[docker pull manuprofeinfo/nginx:lat...](#)



Ahora vamos a crear un tag que no sea latest:



`docker image tag manuprofeinfo/nginx manuprofeinfo/nginx:testing`

`docker image push manuprofeinfo/nginx:testing`

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image tag manuprofeinfo/nginx manuprofeinfo/nginx:testing
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image ls
REPOSITORY          TAG                IMAGE ID           CREATED           SIZE
mongo               latest            5285cb69ea55      13 days ago      698MB
ubuntu              latest            54c9d81cbb44      13 days ago      72.8MB
mysql               latest            d1dc36cf8d9e      2 weeks ago      519MB
nginx               1.20              d6c9558ba445      2 weeks ago      141MB
nginx               1.20.2            d6c9558ba445      2 weeks ago      141MB
nginx               stable            d6c9558ba445      2 weeks ago      141MB
nginx               latest            c316d5a335a5      2 weeks ago      142MB
manuprofeinfo/nginx latest            c316d5a335a5      2 weeks ago      142MB
manuprofeinfo/nginx testing            c316d5a335a5      2 weeks ago      142MB
alpine              latest            c059bf8a849c      2 months ago     5.59MB
nginx               1.20.2-alpine     373f8d4d4c60      3 months ago     23.2MB
dataiku/dss         latest            f24fbb6d95fe      15 months ago    2.75GB
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker image push manuprofeinfo/nginx:testing
The push refers to repository [docker.io/manuprofeinfo/nginx]
762b147902c0: Layer already exists
235e04e3592a: Layer already exists
6173b6fa63db: Layer already exists
9a94c4a55fe4: Layer already exists
9a3a6af98e18: Layer already exists
7d0ebbe3f5d2: Layer already exists
testing: digest: sha256:bb129a712c2431ecce4af8dde831e980373b26368233ef0f3b2bae9e9ec515ee size: 1570
manu@manu-HP-Laptop-15s-fq1xxx:~$
```

En este caso, como la imagen ya existe en nuestro repositorio, al hacer el push nos informa de que todas las capas ya existen. Si ahora consultamos los tags de nuestro repositorio se incluye el de testing:

TAG			docker pull manuprofeinfo/nginx:lat... 		
latest					
Last pushed 8 minutes ago by manuprofeinfo					
DIGEST	OS/ARCH	LAST PULL	COMPRESSED SIZE 		
bb129a712c24	linux/amd64	8 minutes ago	54.09 MB		

TAG			docker pull manuprofeinfo/nginx:tes... 		
testing					
Last pushed 2 minutes ago by manuprofeinfo					
DIGEST	OS/ARCH	LAST PULL	COMPRESSED SIZE 		
bb129a712c24	linux/amd64	8 minutes ago	54.09 MB		

Si quisiésemos subir una imagen a un repositorio privado de Docker Hub, no podríamos crearlo directamente desde nuestra máquina, sino que deberíamos crearlo primero en Docker Hub y después hacer el push de nuestra imagen local al repositorio de Docker Hub.

4.5. Construcción de imágenes

En este apartado vamos a explorar cómo se definen y construyen imágenes, así como también extender imágenes existentes.

4.5.1. Dockerfile

El Dockerfile es un archivo de texto plano que contiene una serie de instrucciones necesarias para crear una imagen que, posteriormente, se traducirá en una pieza software utilizada para un determinado propósito. Se podría también ver como una especie de receta para construir una imagen.

Vamos a tomar como ejemplo el Dockerfile de este [enlace](#). Podemos ver las siguientes directivas:

- **FROM:** es requerido, y se trata de una distribución mínima a partir de la cual crear la imagen. La más común es alpine. Si se necesita un packet manager es común utilizar distribuciones como Ubuntu, Debian, Fedora...
- **ENV:** indica una variable de entorno. De esta forma, la construcción de la imagen

se puede realizar en cualquier sistema operativo.

- **RUN:** se utiliza normalmente para instalar paquetes adicionales en el contenedor, edición de ficheros dentro del contenedor, etc. Establece un punto inicial del contenedor que lo dota de todo lo necesario para cumplir los requerimientos para nuestra aplicación. Se pueden ejecutar scripts que hayan sido previamente copiados dentro del contenedor.
- **EXPOSE:** puertos a abrir, pero no se van a abrir hasta que se levante el contenedor con -p.
- **CMD:** comando final que se ejecutará cuando se cree un contenedor (o arranque uno existente) a partir de una imagen.

NOTA : Docker se ocupa de los logs dentro del contenedor (stdout, stderr), simplemente hay que extraer estos logs del contenedor.

4.5.2. Docker build

Vamos a crear una imagen a partir del fichero analizado en el apartado anterior. Lo descargamos primero a un directorio de nuestra máquina. El nombre del archivo será "Dockerfile", aunque se podría utilizar otro nombre para este fichero. Para construir la imagen, ejecutamos el siguiente comando:

```
docker image build -t my_app .
```

Este comando lo ejecutamos en el mismo directorio en el que se encuentre el Dockerfile, añadiéndole el tag "my_app". Con el punto final indicamos que el Dockerfile se encuentra en la ruta actual.

Al construir la imagen se puede ver que se crea una capa para determinadas instrucciones del Dockerfile. Por cada capa crea un hash, de forma que si volvemos a hacer un build de la imagen y esa capa no ha cambiado (el hash es el mismo), no se va a

realizar el build de esa capa. Esto hace que aumente la eficiencia. Por esta razón, el build puede llevar más tiempo la primera vez. Esto se puede ver en la traza del build:

```
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM debian:stretch-slim
stretch-slim: Pulling from library/debian
1cb79db8a9e7: Pull complete
Digest: sha256:d87734c97bfd3681c64137af7754d03865c2d013eb05d42f81aa52a6516fc12b
Status: Downloaded newer image for debian:stretch-slim
---> ed2f93606030
Step 2/7 : ENV NGINX_VERSION 1.13.6-1~stretch
---> Running in 49fe8118b1b7
Removing intermediate container 49fe8118b1b7
---> 6b0a01694359
Step 3/7 : ENV NJS_VERSION 1.13.6.0.1.14-1~stretch
---> Running in cc5afc21ce69
Removing intermediate container cc5afc21ce69
---> 420c020ba260
Step 4/7 : RUN apt-get update && apt-get install --no-install-recommends --no-inst
```

Las opciones de docker build se pueden encontrar en el siguiente [enlace](#).

Tras la construcción de la imagen, consultamos las imágenes en nuestra máquina, y vemos que se ha creado una nueva:

```
manu@manu-HP-Laptop-15s-fq1xxx:~/1_Backup/1_Profesor/1_Severo_Ochoa/2_DDAW/Unidades/UD6/DockerBuild$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my_app	latest	7d1931d53be1	2 minutes ago	109MB
mongo	latest	5285cb69ea55	13 days ago	698MB
ubuntu	latest	54c9d81cbb44	13 days ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	2 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	2 weeks ago	141MB
nginx	stable	d6c9558ba445	2 weeks ago	141MB
nginx	latest	c316d5a335a5	2 weeks ago	142MB
manuprofeinfo/nginx	latest	c316d5a335a5	2 weeks ago	142MB
manuprofeinfo/nginx	testing	c316d5a335a5	2 weeks ago	142MB
debian	stretch-slim	ed2f93606030	2 weeks ago	55.3MB
alpine	latest	c059bfaa849c	2 months ago	5.59MB
nginx	1.20.2-alpine	373f8d4d4c60	3 months ago	23.2MB
dataiku/dss	latest	f24fbb6d95fe	15 months ago	2.75GB

Vamos a reconstruir la imagen tras modificar el Dockerfile. Para ello, vamos a exponer un nuevo puerto, dejando la línea 48 del siguiente modo:

EXPOSE 80 443 8080

Volvemos a hacer build:


```
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM debian:stretch-slim
--> ed2f93606030
Step 2/7 : ENV NGINX_VERSION 1.13.6-1-stretch
--> Using cache
--> 6b0a01694359
Step 3/7 : ENV NJS_VERSION 1.13.6.0.1.14-1-stretch
--> Using cache
--> 420c020ba260
Step 4/7 : RUN apt-get update && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 && NGINX_GPGKEY=573BFD6B3D8FBC641
079A6ABAF5BD827BD9BF62; found=''; for server in ha.pool.sks-keyserver.net hkp://keyserver.ubuntu.com:80 h
kp://p80.pool.sks-keyserver.net:80 pgp.mit.edu ; do echo "Fetching GPG key $NGINX_GPGKEY from $server"; apt-ke
y adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$NGINX_GPGKEY" && found=yes && break; done; test -z "$found" && echo >82 "
error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; apt-get remove --purge -y gnupg1 && apt-get -y --purge autoremove && rm -rf /var/lib/a
pt/lists/* && echo "deb http://nginx.org/packages/mainline/debian/ stretch nginx" >> /etc/apt/sources.list && apt-get update && apt
-get install --no-install-recommends --no-install-suggests -y nginx=${NGINX_VERSION} n
ginx-module-xslt=${NGINX_VERSION} nginx-module-geoip=${NGINX_VERSION}
ginx-module-image-filter=${NGINX_VERSION} nginx-module-njs=${NJS_VERSION}
n
g
ettext-base && rm -rf /var/lib/apt/lists/*
--> Using cache
--> d01c8428a152
Step 5/7 : RUN ln -sf /dev/stdout /var/log/nginx/access.log && ln -sf /dev/stderr /var/log/nginx/error.log
--> Using cache
--> 759968fef6d5
Step 6/7 : EXPOSE 80 443 8080
--> Running in 5f39d15f5828
Removing intermediate container 5f39d15f5828
--> c6c4daf40858
Step 7/7 : CMD ["nginx", "-g", "daemon off;"]
--> Running in c0bbf942eae0
Removing intermediate container c0bbf942eae0
--> ec01d85fe99d
Successfully built ec01d85fe99d
Successfully tagged my_app:latest
```

En cada capa que ya estaba construida, se especifica "Using cache". Al llegar a la parte del puerto reconstruye esa línea y las posteriores. Las líneas susceptibles de cambiar mucho, como copiar el código de una aplicación dentro de un contenedor, se ponen al final para que impacte lo menos posible en las capas anteriores dentro del proceso de construcción de la imagen.

4.5.3. Extensión de imágenes

Hemos dicho anteriormente que, para construir una imagen, se necesita al menos una distribución mínima a partir de la cual crear una imagen. En este apartado vamos a ver que también se pueden construir imágenes a partir de otras imágenes (de hecho, es lo más común).

Por lo general, preferiblemente partimos de una imagen oficial. Si no podemos partir de una imagen oficial porque los requerimientos de nuestro proyecto hacen que se haya de adaptar más, se puede investigar en Docker Hub una imagen fiable que se acerque más a nuestros requerimientos, y, de ser necesario, modificarla.

Partimos del Dockerfile y el fichero index.html del siguiente [enlace](#) de Github. El Dockerfile

consta de tres líneas:

```
FROM nginx:latest
```

```
WORKDIR /usr/share/nginx/html
```

```
COPY index.html index.html
```

Cabe notar lo siguiente:

- La directiva WORKDIR especifica un cambio de directorio en el sistema de archivos del contenedor, de forma que en la siguiente línea copiamos el fichero index.html desde el sistema host (index.html está en la misma ruta que el Dockerfile en este caso) al directorio de trabajo del contenedor.
- En este caso no se necesita la directiva EXPOSE o CMD, ya que se encuentran especificadas dentro del Dockerfile de la imagen de nginx.
- Con la copia de index.html en de la ruta indicada del contenedor pretendemos sobrecribir la página de bienvenida de nginx.

Vamos a probarlo. Ejecutamos el siguiente comando para construir nuestra imagen (previamente nos situamos, desde la línea de comandos, donde se encuentran los dos ficheros):

```
docker image build -t my_nginx .
```

```
Sending build context to Docker daemon 3.072kB
Step 1/3 : FROM nginx:latest
--> c316d5a335a5
Step 2/3 : WORKDIR /usr/share/nginx/html
--> Running in 1b55988cbc78
Removing intermediate container 1b55988cbc78
--> 5f77c18f11ce
Step 3/3 : COPY index.html index.html
--> 6c78b78eec1c
Successfully built 6c78b78eec1c
```

Y vemos que se ha creado nuestra imagen con docker image ls:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my_nginx	latest	6c78b78eec1c	3 minutes ago	142MB
my_app	latest	ec01d85fe99d	15 hours ago	109MB
<none>	<none>	7d1931d53be1	15 hours ago	109MB
mongo	latest	5285cb69ea55	2 weeks ago	698MB
ubuntu	latest	54c9d81cbb44	2 weeks ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB

Tras esto vamos a levantar un contenedor a partir de nuestra nueva imagen:

```
docker container run -p 80:80 --rm my_nginx
```

NOTA: Con el parámetro `--rm` el contenedor se elimina cuando se pare.

Finalmente, consultamos localhost con el navegador, y vemos el resultado:

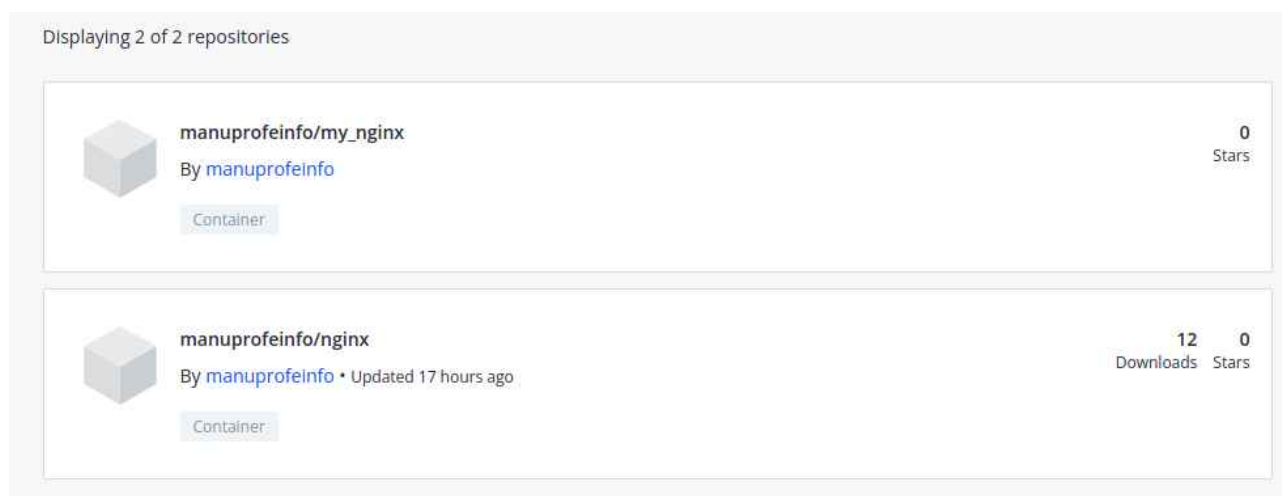


Si queremos subir nuestra nueva imagen a Docker Hub:

```
docker image tag my_nginx:latest manuprofeinfo/my_nginx
```

```
docker image push manuprofeinfo/my_nginx
```

Tras lo cual consultamos nuestra cuenta de Docker Hub y podemos ver un nuevo repositorio con la imagen de nginx extendida:



4.5.4. Borrar imágenes

Para borrar una imagen, utilizamos el comando:

```
docker image rm [REPOSITORY:TAG|IMAGE ID]
```

Si especificamos el repositorio y el tag pero hay más etiquetas, se eliminará esa etiqueta pero no la imagen. Por ejemplo, de la lista que tenemos, eliminamos el primer tag:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my_nginx	latest	6c78b78eec1c	19 minutes ago	142MB
manuprofeinfo/my_nginx	latest	6c78b78eec1c	19 minutes ago	142MB
my_app	latest	ec01d85fe99d	15 hours ago	109MB
<none>	<none>	7d1931d53be1	15 hours ago	109MB
mongo	latest	5285cb69ea55	2 weeks ago	698MB
ubuntu	latest	54c9d81cbb44	2 weeks ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	3 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	3 weeks ago	141MB
nginx	stable	d6c9558ba445	3 weeks ago	141MB
nginx	latest	c316d5a335a5	3 weeks ago	142MB

```
docker image rm my_nginx:latest
```

Tras el borrado, consultamos de nuevo, y vemos que aún nos queda la segunda imagen, que es la misma que la borrada pero con otro tag:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
manuprofeinfo/my_nginx	latest	6c78b78eec1c	20 minutes ago	142MB
my_app	latest	ec01d85fe99d	15 hours ago	109MB
<none>	<none>	7d1931d53be1	15 hours ago	109MB
mongo	latest	5285cb69ea55	2 weeks ago	698MB
ubuntu	latest	54c9d81cbb44	2 weeks ago	72.8MB
mysql	latest	d1dc36cf8d9e	2 weeks ago	519MB
nginx	1.20	d6c9558ba445	3 weeks ago	141MB
nginx	1.20.2	d6c9558ba445	3 weeks ago	141MB
nginx	stable	d6c9558ba445	3 weeks ago	141MB

Si intentamos eliminar una imagen con su IMAGE ID pero tiene varias etiquetas, nos va a devolver un error. Vamos a probar con la que tiene el identificador d6c9558ba445:


```
manu@manu-HP-Laptop-15s-fq1xxx: $ docker image ls
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
manuprofeinfo/my_nginx latest       6c78b78eec1c 20 minutes ago 142MB
my_app               <none>       ec01d85fe99d 15 hours ago 109MB
<none>              <none>       7d1931d53be1 15 hours ago 109MB
mongo                latest       5285cb69ea55 2 weeks ago 698MB
ubuntu              latest       54c9d81cbb44 2 weeks ago 72.8MB
mysql                latest       d1dc36cf8d9e 2 weeks ago 519MB
nginx                1.20        d6c9558ba445 3 weeks ago 141MB
nginx                1.20.2      d6c9558ba445 3 weeks ago 141MB
nginx                stable      d6c9558ba445 3 weeks ago 141MB
nginx                latest      c316d5a335a5 3 weeks ago 142MB
manuprofeinfo/nginx latest      c316d5a335a5 3 weeks ago 142MB
manuprofeinfo/nginx testing     c316d5a335a5 3 weeks ago 142MB
debian               stretch-slim ed2f93606030 3 weeks ago 55.3MB
alpine               latest      c059bf8aa849c 2 months ago 5.59MB
nginx                1.20.2-alpine 373f8d4d4c60 3 months ago 23.2MB
dataiku/dss          latest      f24fbb6d95fe 15 months ago 2.75GB

manu@manu-HP-Laptop-15s-fq1xxx: $ docker image rm d6c9558ba445
error response from daemon: conflict: unable to delete d6c9558ba445 (must be forced) - image is referenced in multiple repositories
```

En este caso habría que eliminar las etiquetas primero, para poder eliminar la imagen:

```
manu@manu-HP-Laptop-15s-fq1xxx: $ docker image rm nginx:1.20 nginx:1.20.2 n
nginx:stable
Untagged: nginx:1.20
Untagged: nginx:1.20.2
Untagged: nginx:stable
Untagged: nginx@sha256:02923d65cde08a49380ab3f3dd2f8f90aa51fa2bd358bd85f89345848f6e6623
Deleted: sha256:d6c9558ba4456741fc4ee304e1a75a561e1c8d92f5107a715b6224bb7844f507
Deleted: sha256:055624127108c9595ef76afb2fdcb8286c37684e00e175d30c59a63bf3cf90c
Deleted: sha256:51cc5e700f73e1d4ae88e0ca8d094743ac39ab9db881bd2d6f04c87a0f4341e6
Deleted: sha256:82b7f14ee2763df71987ec4d5ce5ad846e91cd59204fdee6467a634619c263ad
Deleted: sha256:d1217a4a7e17bc7fefe318f17a84a3649ab716a7a370bc992bc8fe8d0ef04c9d
Deleted: sha256:5c3da4d189e1edff6a1babc21a57332a6a1ee09e79bbfc30fd1bad0a8096ce89

manu@manu-HP-Laptop-15s-fq1xxx: $ docker image ls
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
manuprofeinfo/my_nginx latest       6c78b78eec1c 21 minutes ago 142MB
my_app               latest      ec01d85fe99d 15 hours ago 109MB
<none>              <none>       7d1931d53be1 15 hours ago 109MB
mongo                latest       5285cb69ea55 2 weeks ago 698MB
ubuntu              latest       54c9d81cbb44 2 weeks ago 72.8MB
mysql                latest       d1dc36cf8d9e 2 weeks ago 519MB
nginx                latest      c316d5a335a5 3 weeks ago 142MB
manuprofeinfo/nginx latest      c316d5a335a5 3 weeks ago 142MB
manuprofeinfo/nginx testing     c316d5a335a5 3 weeks ago 142MB
debian               stretch-slim ed2f93606030 3 weeks ago 55.3MB
alpine               latest      c059bf8aa849c 2 months ago 5.59MB
nginx                1.20.2-alpine 373f8d4d4c60 3 months ago 23.2MB
dataiku/dss          latest      f24fbb6d95fe 15 months ago 2.75GB
```




GENERALITAT
VALENCIANA
Conselleria de Educació,
Cultura i Esport

TOTS
A UNA
VEU



Carrer d'Illueca, 28
03206 Elx, Espanya
03013224.secret@gva.es
(+34) 966.912.260



Fondo Social Europeo
El FSE invierte en tu futuro
UNIÓN EUROPEA

4.6. EJERCICIOS

TEST

1. ¿Cuál de los siguientes es un ejemplo de una imagen que podemos ejecutar?
linux
docker hub
container.jpg
nginx
2. ¿Qué flag se utiliza para referenciar un docker file diferente del por defecto "Dockerfile"?
-f
--dockerfile
-t
--specify-file
3. ¿Qué comando de Dockerfile es la mejor opción para cambiar de directorio:
DIR
CD
RUN cd
WORKDIR

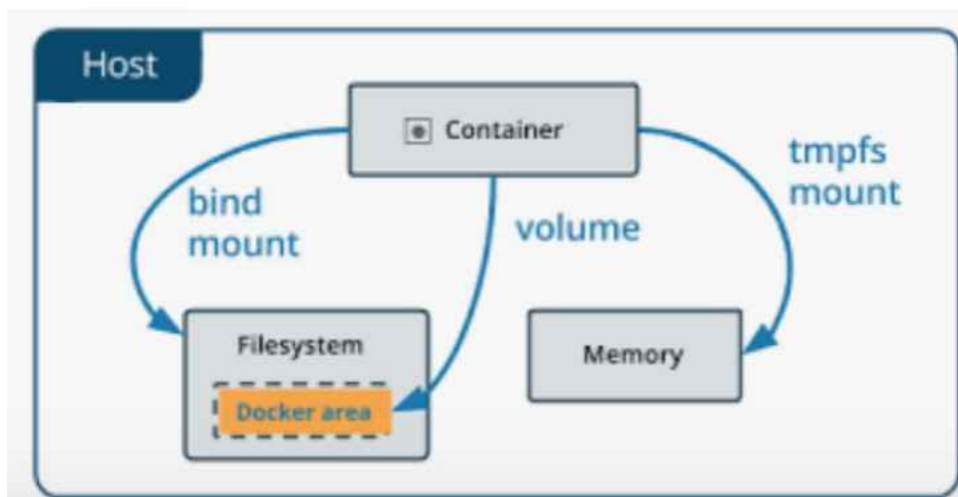
5. Volúmenes

Docker simplifica enormemente la creación de contenedores, y eso lleva a tratar los contenedores como un elemento efímero, que se crea cuando se necesita y que no importa que se destruya puesto que puede ser reconstruido una y otra vez a partir de su imagen.

Si la aplicación o aplicaciones incluidas en el contenedor generan datos y esos datos se guardan en el propio contenedor, en el momento en que se destruyera el contenedor perderíamos esos datos.

El objetivo principal de los volúmenes es no perder datos si borro el contenedor y mejorar rendimiento del Docker. Para conseguir la persistencia de los datos, se pueden emplear dos técnicas:

- Los directorios enlazados (bind mount), en la que la información se guarda fuera de Docker, en la máquina host (por ejemplo si lo ejecutamos en la máquina virtual de Ubuntu o la máquina física de Lliurex en clase). En este tipo, una ruta del contenedor enlaza con una ruta en el sistema de archivos del host
- Los volúmenes, en la que la información se guarda mediante Docker, pero en unos elementos llamados volúmenes, independientes de las imágenes y los contenedores. Además los volumens se pueden catalogar en dos tipos.
 - Volúmenes de Datos: es como si montará un disco en el contenedor y por defecto se realizan en un path temporal.
 - Volúmenes de Host: Mismo concepto pero indicándole el path.



NOTA:

Aconsejable utilizar la técnica de volúmenes, ya que, La ventaja frente a los directorios enlazados es que pueden ser gestionados por Docker. Otro detalle importante es que el acceso al contenido de los volúmenes sólo se puede hacer a través de algún contenedor que utilice el volumen.

Los volúmenes son independientes de los contenedores, por lo que también podemos conservar los datos aunque se destruya el contenedor, reutilizarlos con otro contenedor, etc.

Ventajas Volúmenes

Los volúmenes tienen varias ventajas sobre los directorios enlazados:

- Los volúmenes son más fáciles de respaldar o migrar que enlazar montajes.
- Es posible administrar volúmenes mediante los comandos de la CLI de Docker o la API de Docker.
- Los volúmenes funcionan tanto en contenedores de Linux como de Windows.
- Los volúmenes se pueden compartir de forma más segura entre varios contenedores.
- Los controladores de volumen permiten almacenar volúmenes en hosts remotos o proveedores en la nube, para cifrar el contenido de los volúmenes o para agregar otras funciones.

- Los nuevos volúmenes pueden tener su contenido precargado por un contenedor.
- Los volúmenes en Docker Desktop tienen un rendimiento mucho mayor que los directorios enlazados de hosts de Mac y Windows.

NOTA:

Además, los volúmenes suelen ser una mejor opción que los datos persistentes en la capa de escritura de un contenedor, porque un volumen no aumenta el tamaño de los contenedores que lo usan y el contenido del volumen existe fuera del ciclo de vida de un contenedor determinado.

Los volúmenes tienen varias ventajas sobre los directorios enlazados:

5.1. Operaciones con volúmenes

En este apartado vamos a utilizar una imagen de mysql para realizar operaciones con los volúmenes de Docker.

Primero buscamos en Docker Hub la imagen oficial de mysql, consultamos el Dockerfile de la última versión, y extraemos la ruta de la directiva VOLUME, que este caso es:

`VOLUME /var/lib/mysql`

En esta ruta se guardarán los datos que persistirán aunque se elimine el contenedor. La ruta hace referencia al sistema de archivos del contenedor.

NOTA: para más seguridad, los volúmenes necesitan ser eliminados manualmente.

Primero descargamos la imagen:

`docker pull mysql`

Si ejecutamos el siguiente comando, podemos ver el volumen especificado (entre otra información):

`docker image inspect mysql`

```
"Image": "sha256:78bfc0514d6f8bb3ab6e480b90a2f2aeb34b0cb2d3ab9088fecc71c9e2983c46",
"Volumes": {
  "/var/lib/mysql": {}
},
"Mounts": [
  {
    "Type": "volume",
    "Name": "8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d",
    "Source": "/var/lib/docker/volumes/8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d/_data",
    "Destination": "/var/lib/mysql",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Lanzamos un contenedor basado en esta imagen, como hemos hecho anteriormente:

```
docker container run -d --name mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=True mysql
```

Ahora inspeccionamos el volumen, para ver el volumen asociado:

```
docker container inspect mysql
```

```
"Mounts": [
  {
    "Type": "volume",
    "Name": "8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d",
    "Source": "/var/lib/docker/volumes/8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d/_data",
    "Destination": "/var/lib/mysql",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Vamos ahora a ver los volúmenes que tenemos en el sistema:

```
docker volume ls
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume ls
DRIVER          VOLUME NAME
local           8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
```

Vamos a inspeccionar los detalles de este volumen:

```
docker volume inspect [VOLUME ID]
```

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume inspect 8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
[
  {
    "CreatedAt": "2022-02-16T17:27:30+01:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d/_data",
    "Name": "8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d",
    "Options": null,
    "Scope": "local"
  }
]
```

Ahora creamos otro contenedor de mysql:

```
docker container run -d --name mysql2 -e MYSQL_ALLOW_EMPTY_PASSWORD=True mysql
```

Y volvemos a listar los volúmenes, encontrando uno nuevo:


```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume ls
DRIVER          VOLUME NAME
local           8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
local           5796c2758d3165889229da28bdb84037dcd955b734dd84505a84d14921e9eb20
```

Ahora vamos a parar los dos contenedores:

```
docker container stop mysql mysql2
```

Si consultamos de nuevo los volúmenes, vemos que no desaparecen:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container stop mysql mysql2
mysql
mysql2
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume ls
DRIVER          VOLUME NAME
local           8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
local           5796c2758d3165889229da28bdb84037dcd955b734dd84505a84d14921e9eb20
```

Con el fin de gestionar mejor los volúmenes, podemos nombrarlos. Mediante el siguiente comando lo conseguimos:

```
docker container run -d --name mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=True -v mysql-  
db:/var/lib/mysql mysql
```

Con el parámetro -v damos un nombre al volumen "mysql-db", asignado a la ruta /var/lib/mysql del contenedor.

Vamos a consultar los contenedores y vemos el que acabamos de crear, con su nombre:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume ls
DRIVER          VOLUME NAME
local           8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
local           5796c2758d3165889229da28bdb84037dcd955b734dd84505a84d14921e9eb20
local           mysql-db
```

Vamos a hacer una última prueba: vamos a eliminar el contenedor mysql que hemos creado más recientemente, y vamos a crear uno nuevo mysql3, y asignarle el volumen del contenedor anterior "mysql-db", con el fin de comprobar si este nuevo contenedor reutiliza

el volumen del contenedor anterior, o sin embargo se crea otro volumen.

Primero paramos y borramos el contenedor mysql. A continuación creamos el nuevo:

```
docker container run -d --name mysql3 -e MYSQL_ALLOW_EMPTY_PASSWORD=True -v mysql-  
db:/var/lib/mysql mysql
```

Y visualizamos los volúmenes:

```
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
70cc97eab88c   mysql    "docker-entrypoint.s..." 6 minutes ago  Up 6 minutes  3306/tcp, 33060/tcp      mysql
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container stop mysql
mysql
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container rm mysql
mysql
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker container run -d --name mysql3 -e MYSQL_ALLOW_EMPTY_PASSWORD=True -v mysql-db:/var/lib/mysql mysql
1f1c78cecf80a43a01530a3b1c2fa14fa485fc4edd3e31775bafb7f0d98fcfe0
manu@manu-HP-Laptop-15s-fq1xxx:~$ docker volume ls
DRIVER          VOLUME NAME
local          8f90b92c5f67b310424aef1904259817dc915fc2ed92a39fd149dcb0b494978d
local          5796c2758d3165889229da28bdb84037dcd955b734dd84505a84d14921e9eb20
local          mysql-db
```

Por tanto, no tenemos dos volúmenes con nombre "mysql-db", hemos asignado ese volumen al contenedor mysql3.

Inspeccionando el contenedor vemos más clara esta información:

```
"Mounts": [
  {
    "Type": "volume",
    "Name": "mysql-db",
    "Source": "/var/lib/docker/volumes/mysql-db/_data",
    "Destination": "/var/lib/mysql",
    "Driver": "local",
    "Mode": "z",
    "RW": true,
    "Propagation": ""
  }
]
```

Finalmente, cabe mencionar que es posible crear volúmenes por separado, mediante el comando:

```
docker volume create
```

Posteriormente, se puede enlazar con un contenedor, como hemos visto anteriormente.

Esto se puede llevar a cabo para poder para cambiar su configuración por defecto. Si hacemos `docker volume create --help` vemos que hay distintas opciones (drivers), y tags

para casos como puesta en producción. En este [enlace](#) se puede ver la referencia completa de este comando.

5.2. Operaciones con bind mounts

Se trata básicamente de dos ubicaciones apuntando a los mismos ficheros. Hay que tener en consideración, además, que los ficheros introducidos desde el host sobrescriben a los del contenedor.

Los bind mounts son específicos del host (no del contenedor) y se han de establecer cuando se lanza el contenedor, no en el Dockerfile, de la forma:

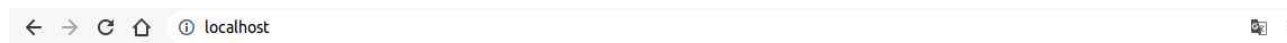
- ... run /Users/manu/docker:/path/contenedor (**mac/linux**)
- ... run //c/Users/manu/docker:/path/contenedor (**windows**)

Vamos a ver un ejemplo con la imagen my_nginx creada en el apartado de extensión de imágenes oficiales.

Primero nos situamos en la ruta donde tenemos el fichero index.html, en nuestro sistema de archivos host, y a continuación ejecutamos el siguiente comando:

```
docker container run -d --name my_nginx -p 80:80 -v $(pwd):usr/share/nginx/html  
manuprofeinfo/my_nginx
```

Vamos a localhost y vemos el contenido del index.html:



Ahora vamos a abrir un nuevo terminal para abrir una sesión bash en este contenedor:

```
docker container exec -it my_nginx bash
```

Y ejecutamos (en el contenedor):

```
cd /usr/share/nginx/html
```

```
ls -al -- > vemos tanto el Dockerfile como el index.html
```

Ahora, desde el host (en una nueva terminal), en el directorio en el que tenemos los dos ficheros (el directorio que hemos compartido con el contenedor) creamos un nuevo fichero:

```
touch test.txt
```

Ahora, desde el contenedor, con la sesión bash que habíamos abierto en la terminal inicial, volvemos a ejecutar ls -al y veremos que nos aparece el fichero test.txt.

Por último, borramos el fichero desde el host, y comprobamos desde el bash del contenedor que se ha eliminado también.

TEST

1. ¿Qué tipo de datos persistentes permite enlazar un directorio en el host con un directorio en el contenedor?

- Volumen con nombre
- Bind mount

2. Cuando se añade un bind mount a un comando de lanzamiento de contenedor, se puede utilizar \$ (pwd), (o \${pwd} dependiendo del shell). ¿Qué hace exactamente?

- Cambia al directorio raíz del usuario en el host
- Realiza pruebas con la impresora
- Imprime el directorio actual del host en el comando
- Publica el contenido del directorio actual para que el código esté disponible en un puerto específico

3. Cuando se crea un nuevo volumen para un contenedor mysql, ¿dónde se puede investigar en qué ruta se localizan los datos en el contenedor?

- docker container inspect
- Docker Hub
- docker ps
- docker pull mysql

EJERCICIO

En este ejercicio vamos a realizar un upgrade de un contenedor de bases de datos. Normalmente se haría un apt-get upgrade para actualizar el software de la BBDD, pero aquí el software está en el contenedor. Lo que tendremos que hacer es lo siguiente:

- Mirar en Docker Hub para analizar las instrucciones de cómo utilizar postgres
- Crear un contenedor de postgres (versión 13.6) con un volumen con nombre (psql-data)
- Investigar, para esta versión, en qué volumen se dejan los datos del contenedor postgres
- Analizar los logs del contenedor para saber cuándo ha creado la base de datos y la configuración por defecto
- Para contenedor
- Crear un nuevo contenedor postgres (versión 14.2) con el mismo volumen que el anterior
- Analizar los logs, no ha de haber rehecho todo como antes

6. System prune

- docker image prune: elimina imágenes que al construirse hayan dado un error y hayan quedado en un estado corrupto o intermedio.
- docker system prune: elimina todos los objetos de docker (contenedores, imágenes, volúmenes...) que no estén siendo utilizados.
- docker image prune -a: elimina imágenes que no estén siendo utilizadas.
- docker system df: para ver la utilización de espacio de los objetos Docker.