Introducción a Docker

Introducción

Instalación

Instalación de Docker en Ubuntu 20.04

Hello World:

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

Vemos las interfaces de red. Veremos que contamos con una nueva interfaz, la de **Docker**.

```
ifconfig
```

Para ser **root** hay teclear el comando:

```
sudo su -i
```

Así no tenemos que poner **sudo** delante de todas las instrucciones.

¿Qué es Docker?

- Es una herramienta que permite desplegar aplicaciones en contenedores de forma rápida y portable.
- Podemos crear contenedores que podemos llevar a otra máquina y correrán igual.

Para mostrar su potencia, con este ejemplo podemos crear un servidor Apache con un solo comando:

```
docker run -d -p 8080:80 --name apache httpd
```

Como no tenemos la imagen de apache, Docker se la baja automáticamente de **Docher HUB**

Si ponemos el comando:

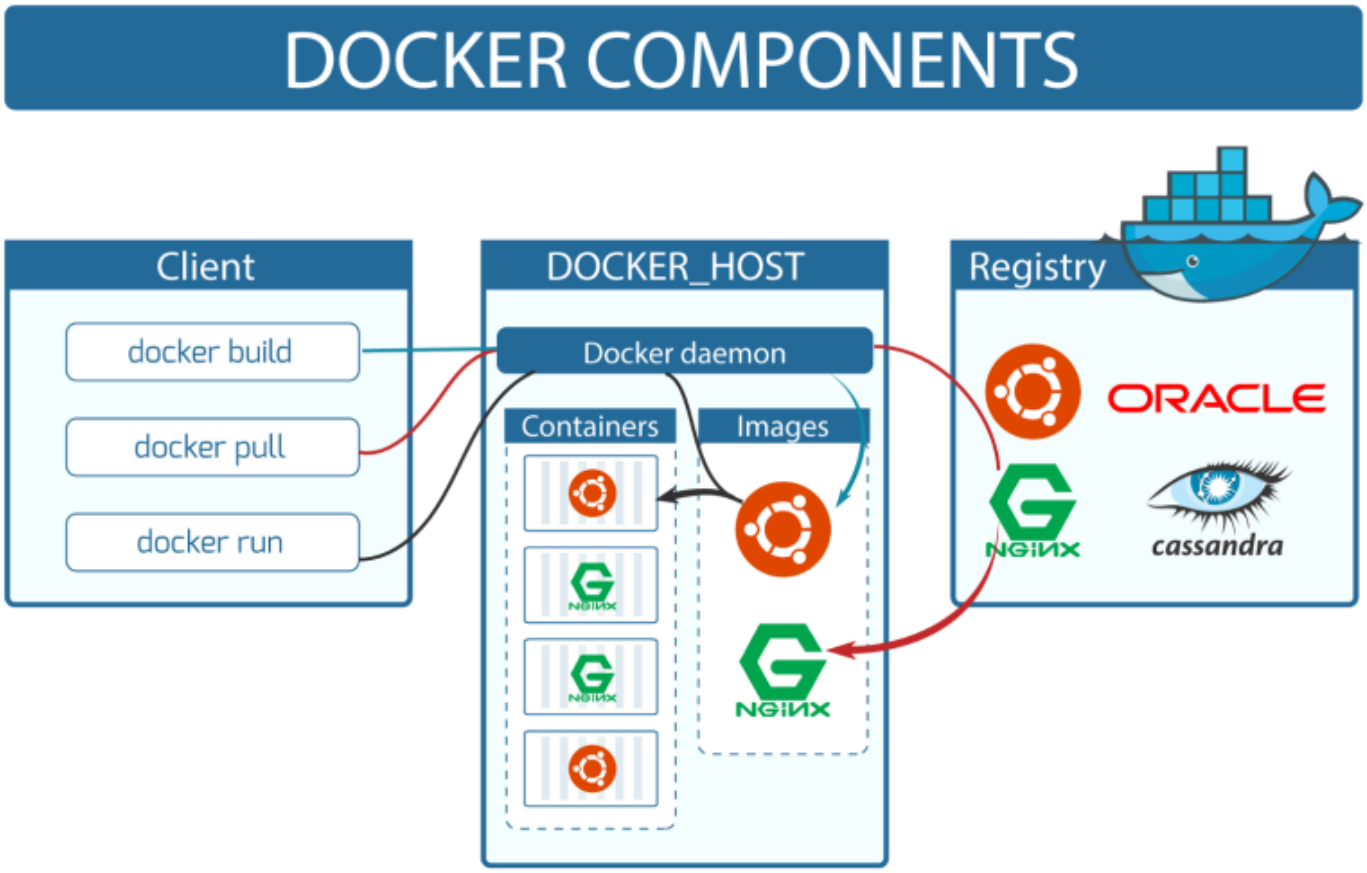
```
docker ps
```

Veremos que ya tenemos un servidor web corriendo:

```
CONTAINER ID   IMAGE     COMMAND              CREATED         STATUS         PORTS
e8a47ba48c73   httpd     "httpd-foreground"   2 minutes ago   Up 2 minutes   0.0.0.0:8080->80/tcp, :::8
```

Si ahora abro un navegador web y pongo la dirección *localhost:8080* veré que tengo un servidor web instalado.

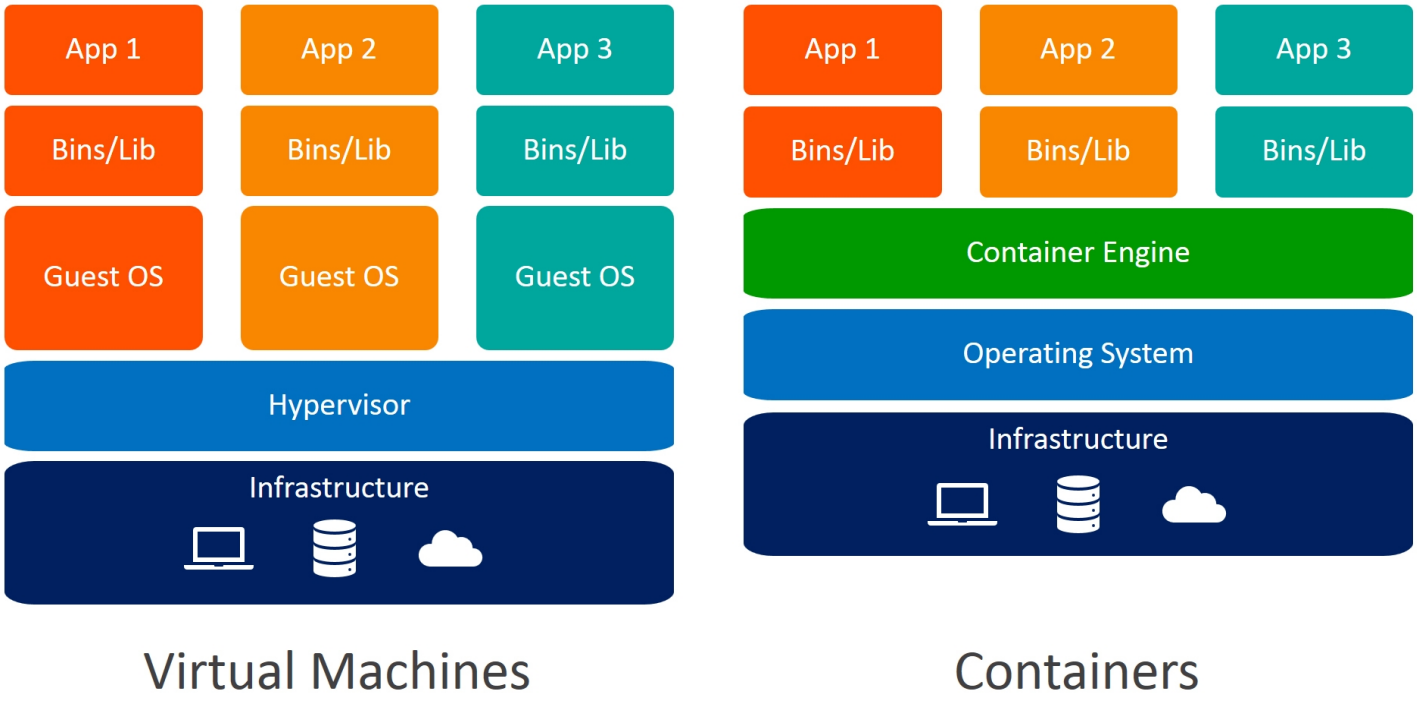
Arquitectura y componentes



- **Docker Host:** Es la máquina **donde va a correr docker**.
- **Docker CLI:** Es el cliente de comandos que nos sirve para manejar Docker, es decir, el comando *docker*. Con este comando podremos manejar contenedors, imágenes, volúmenes y redes.
- **Rest API:** Es una API para poder administrar docker mediante peticiones GET y POST. Ideal para conectarlo a otras aplicaciones como **Jenkins**, que permiten automatizar procesos.
- **Docker Daemon**: Es el demonio que presta servicio de docker (el servidor).

Contenedores vs Máquinas virtuales

Los contenedores son procesos que corren sobre la máquina, por lo que **utilizan menos recursos** que los que utiliza una máquina virtual completa: *Si sólo quiero Apache, ¿por qué tengo que tener todas las herramientas del Sistema Operativo?*



- Información (EN): Contenedores frente a Máquinas Virtuales
- Concepto: Hipervisor

Los contenedores son más rápidos de levantar, pesan menos y son visibles desde el kernel del host, por lo que no están tan aislados unos de otros como en el caso de las máquinas virtuales (y por eso son un poco menos seguros).

Introducción a las imágenes

Una imagen es un paquete que contiene **toda la configuración necesaria** para que **una aplicación se pueda ejecutar**.

Las imágenes son de **solo lectura** por lo que una vez creadas, **no son modificables**.

Para construir imágenes se utiliza un fichero de texto llamado **Dockerfile**.

Ejemplo de Dockerfile:

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

Están compuestas por capas: FROM, RUN, CMD...

- **FROM** Crea una capa a partir de la imagen de Docker de Ubuntu en la versión 18.04.

- **COPY** añade los archivos del directorio actual al contenedor.
- **RUN** Construye la aplicación utilizando **make**.
- **CMD** Especifica un comando a correr en el contenedor. En este caso, ejecuta el script app.py en Python.
- Mejores prácticas para escribir Dockerfiles

Podemos ver que Docker no solo vale para *montar servidores*. Se trata de una tecnología también interesante para programadores, tanto para correr apps como para contar con un entorno común de desarrollo y que se acabe la vieja escusa de "*Pues en mi PC funciona*"...

Otro ejemplo de Dockerfile sería el siguiente:

```
FROM Centos:7
RUN yum -y install httpd
CMD ["apachectl", "-DFOREGROUND"]
```

El **CMD** siempre se debe de ejecutar en primer plano, para que mantenga vivo a nuestro contenedor.

Contenedores

Un contenedor podemos verlo como otra capa adicional en la que se van a ejecutar las imágenes.

Un contenedor **está formado por**:

1. **Imágenes**.
2. **Volúmenes**.
3. **Redes**.

Es el entorno en donde va a correr nuestra aplicación.

Es importante saber que los contenedores son de **lectura y escritura** pero es **TEMPORAL**. Una vez que volvamos a levantar el contenedor a partir de la imagen, si hemos hecho algún cambio, tendremos que volver a hacerlo.

La forma de hacer **persistente la información** es por medio de los **volúmenes**.

Introducción a los volúmenes

Sirven para que los datos de un contenedor puedan ser **permanentes**.

Hay tres tipos de volúmenes:

1. **Host**: Se indica un directorio de nuestro Docker Host.
2. **Anónimos**: Docker genera un volúmen aleatorio.
3. **Volúmenes nombrados**: Son volúmenes que creamos nosotros pero que maneja Docker.

Los veremos en profundidad un poco más adelante.

Introducción a las redes

Las redes son el mecanismo que nos permitirá **acceder** a contenedores y **comunicar** contenedores entre sí.

Cuando instalamos Docker ya vienen 3 tipos de redes instalados que son:

1. **Bridge**: Red estandar que utilizarán los contenedores.
2. **Hosts**: Todas las tarjetas de red y el nombre de nuestro Docker Host se asignarán al contenedor.
3. **None**: No se asigna ninguna red al contenedor.

Existen más tipos de redes, que veremos en profundidad más adelante.

Imágenes

Docker HUB

Es un **repositorio de imágenes** público/privado, donde los desarrolladores suben sus **imágenes ya creadas**. Estas imágenes suelen ser lo más **optimizadas** posible.

Del software popular hay **imágenes oficiales**. No hace falta perder el tiempo construyendo imágenes cuando ya hay imágenes oficiales que probablemente estén mejor que las que uno mismo pueda construir.

- https://hub.docker.com

Docker Hub testea automáticamente las imágenes en busca de software peligroso.

Muchas imágenes están creadas sobre alpine, una distribución de Linux que tiene la filosofía de ser pequeña, simple y segura. Ocupa **entre 5 y 10MB**.

Para descagar imágenes utilizamos el comando:

```
docker pull <nombre-imagen[:etiqueta]>
```

Si no especifico la etiqueta, la etiqueta por defecto es **latest**.

50MB:

```
docker pull nginx
```

9MB:

```
docker pull nginx:alpine
```

La diferencia en tamaño es muy grande. Ahorramos recursos, por lo que podemos correr más contenedores.

En ocasiones, tendremos que instalar los paquetes que necesitemos, porque alpine no los traerá.

Podemos tener repositorios privados con las imágenes que creemos.

- Imágenes de Docker Hub: Ubuntu, alpine, postgres, mysql, jenkins, python...

Para ver las imágenes que tenemos en nuestro Docker Host:

```
docker images
```

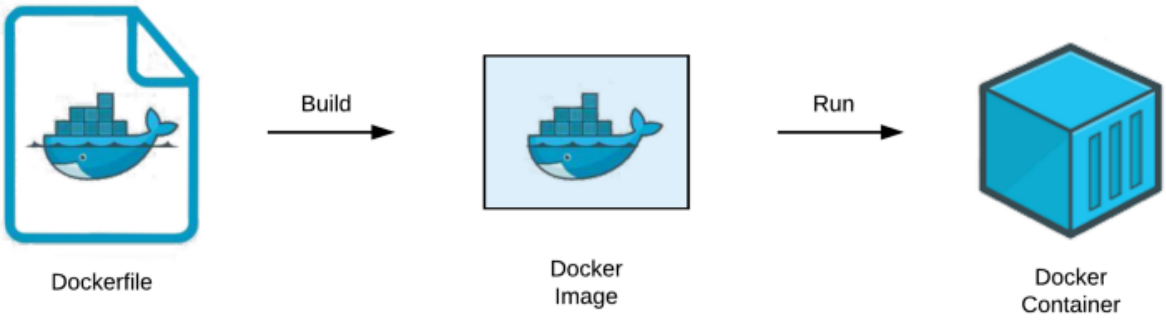
Si hacemos un **docker run** de una imagen que no tenemos, Docker tratará de encontrarla y descargarla para nosotros de forma automática.

```
root@ubuntu2004:~# docker pull jenkins/jenkins
Using default tag: latest
latest: Pulling from jenkins/jenkins
1671565cc8df: Pull complete
c6f3c84ca4c1: Pull complete
f8a3c266a08a: Pull complete
2c3baf4a649e: Pull complete
159829d9bab8: Pull complete
62a0807c5365: Pull complete
c683c0b7aea0: Pull complete
f0a5ea08aa7e: Pull complete
69de96b86dd2: Pull complete
a5edabafdc15: Pull complete
20135332d2dd: Pull complete
f63c5a3f7d66: Pull complete
b1dbc531816e: Pull complete
a68e822bc251: Pull complete
Digest: sha256:394dcabdef3e47fc358c125713028dd74c72289c68e93662350966d68d3262d5
Status: Downloaded newer image for jenkins/jenkins:latest
docker.io/jenkins/jenkins:latest
root@ubuntu2004:~# docker pull jenkins/jenkins:alpine
alpine: Pulling from jenkins/jenkins
213ec9aee27d: Pull complete
f1ce65c285ea: Pull complete
c02868da2250: Pull complete
fe342135932d: Pull complete
44d839b970a9: Pull complete
0e55e5530bec: Pull complete
8e190a0e2fc4: Pull complete
db736a60fe84: Pull complete
8edfda27e5a6: Pull complete
7256213fd07d: Pull complete
9e2847080e84: Pull complete
5a78307cee1a: Pull complete
Digest: sha256:4d31b60c86fbf0459378056d660bad963f953f624de7899d8cfb8d17157a963f
Status: Downloaded newer image for jenkins/jenkins:alpine
docker.io/jenkins/jenkins:alpine
root@ubuntu2004:~# docker images
REPOSITORY        TAG       IMAGE ID       CREATED         SIZE
jenkins/jenkins   alpine    06ee5e5070ba   9 hours ago     262MB
jenkins/jenkins   latest    f93ab9469cd1   9 hours ago     463MB
httpd             latest    a981c8992512   2 weeks ago     145MB
hello-world       latest    feb5d9fea6a5   11 months ago   13.3kB
```

Dockerfile

El **Dockerfile** es un fichero de texto que utilizaremos para crear una imagen. No tiene porqué llamarse *Dockerfile* pero si no lo llamamos así tendremos que especificar su nombre como

argumento del comando.



Está basado en capas. Es como una **receta de cocina** donde se le van especificando los ingredientes. Algunas capas son: FROM, RUN, COPY/ADD, ENTRYPOINT, CMD...

Capa FROM

La capa **FROM** inicia una nueva **etapa de construcción** y establece una **imagen base** para las instrucciones posteriores.

```
FROM <image>[<:tag>] [AS <name>]
```

- Puede aparecer **varias veces** dentro de un Dockerfile para crear múltiples imágenes o crear una etapa de compilación como dependencia de otra.
- Se le puede asignar un nombre con la instrucción **as**.
- Se utiliza en el multi-state build que es un concepto avanzado, para reducir el tamaño de la imagen. Por ejemplo, podríamos tener un FROM que compilase un programa (necesita el entorno de compilación) y otro FROM que tomase ese programa compilado y lo ejecutase (y solo necesitase el entorno de ejecución, no de compilación).
- Los **tags** (etiquetas) son opcionales y si no se indican tomará *latest*, que es la última versión de la imagen.

Capa RUN

La capa **RUN** ejecuta un comando en una nueva capa encima de la imagen actual.

```
RUN <command>
```

- Utiliza la **shell por defecto**:

    - Linux: **/bin/sh -c**
    - Windows: **cmd /S /C**
- Los comandos que utilicemos deben de ser **desatendidos**, por ejemplo:

```
yum –y install
```

No podemos instalar un paquete con **yum install** simplemente, porque nos preguntaría si estamos seguros de que queremos instalar un paquete. Y eso, interrumpiría la construcción.

Capa COPY/ADD

Las capas **COPY** y **ADD** Se utilizan para copiar archivos o direcctorios al contenedor.

```
COPY/ADD [––chown=<user>:<group>] <src>... <dst>
```

- En Linux se puede pasar la órden **chown**.
- La diferencia entre **COPY** y **ADD** es que con ADD se puede copiar un fichero desde una **URL** e incluso si el fichero es un **archivo comprimido**, lo descomprime.

Capa ENTRYPOINT

La capa **ENTRYPOINT** nos permite configurar un contenedor para que sea ejecutable.

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

Capa CMD

La capa **CMD** se utiliza para indicar el comando que **levanta el servicio**.

- Debe levantar el servicio en **primer plano** (foreground) para mantener vivo al contenedor.
- Sólo puede haber una capa **CMD** en el Dockerfile, si hay más sólo la última tendrá efecto.

Ejemplo de Dockerfile

```
FROM richarvey/nginx-php-fpm
EXPOSE 80
EXPOSE 443
ENV WEBROOT=/var/www/html/public
ENV MARIADB_DATABASE=webapp
ENV MARIADB_USERNAME=codeigniter
ENV MARIADB_PASSWORD=password
ENV MARIADB_HOST=mariadb
ENV BASE_URL=http://local.dev/
COPY ./ /var/www/html
COPY ./nginx.conf /etc/nginx/sites-available/default.conf
```

La capa **Expose** es informativa. Indica los puertos que se van a exponer, pero no los expone por sí misma. Ya veremos cómo se exponen.

La capa **ENV** permite pasar variables de entorno, y con ella podemos configurar aplicaciones o servicios.

.dockerignore

Por defecto, cuando se construye una imágen con **docker build** todo el contenido del directorio se envía al contexto de docker.

El fichero **.dockerignore** (es oculto, comienza con un "."), se utiliza para excluír aquellos archivos y directorios que no se utilizarán en la construcción de la imagen. De esta forma, la construcción de la imagen es más rápida. (Es parecido a *.gitignore*, para quien tenga experiencia usando GIT).

Solo se deben de llevar al contexto de docker aquellos ficheros que se vayan a utilizar para la construcción.

Ejemplo:

Vamos a construír un contenedor que corra una aplicación Python.

Localizamos nuestro binario de Python:

```
ubuntu@ubuntu2004:~$ which python3
/usr/bin/python3
```

Creamos un hello world en Python en el archivo **app.py** dentro de una carpeta **appdemo** que creemos. Importamos la librería **time** para poder hacer un **sleep** de 5 minutos. Esto es debido a que si no ponemos a la aplicación a esperar, terminará su ejecución.

La capa **CMD** va a ejecutar el comando que va a levantar nuestra aplicación, y si no pongo el sleep, el contenedor moriría. El sleep va a seguir manteniendo vivo al contenedor durante 5 minutos.

```
#!/usr/bin/python3
import time

print("Hello World!")
time.sleep(600)
```

Vamos a crear nuestro **Dockerfile**.

```
FROM ubuntu:latest
```

Tomamos como base la imagen de Ubuntu. No haría falta poner *latest* ya que se tomaría por defecto.

```
RUN apt update && install -y python3
```

Vamos a actualizar los repositorios y a instalar python3 de forma desatendida **-y**.

```
COPY app.py /app.py
```

Copiamos el fichero de nuestra máquina al contenedor.

Ahora puedo poner:

```
CMD ["python3","/app.py"]
```

Otra opción sería cambiar el ENTRYPOINT, que por defecto es */bin/sh* y decirle que use *python*.

```
ENTRYPOINT["python3"]
CMD ["/app.py"]
```

El fichero **Dockerfile** nos quedaría de la siguiente manera:

```
FROM ubuntu:latest

RUN apt update && apt install -y python3

COPY app.py /app.py

CMD ["python3","/app.py"]
```

Ahora creamos nuestra imagen ejecutando el siguiente comando:

```
docker build -t app-python:latest .
```

La opción *-t* la utilizamos para ponerle un tag.

El "." final es el contexto. Hace referencia al directorio actual.

Si lanzamos el comando:

```
docker images
```

Veremos que tenemos una nueva imagen.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker images
REPOSITORY        TAG       IMAGE ID       CREATED              SIZE
app-python        latest    9b38010ed31b   About a minute ago   144MB
jenkins/jenkins   alpine    06ee5e5070ba   24 hours ago         262MB
jenkins/jenkins   latest    f93ab9469cd1   24 hours ago         463MB
ubuntu            latest    2dc39ba059dc   5 days ago           77.8MB
httpd             latest    a981c8992512   2 weeks ago          145MB
hello-world       latest    feb5d9fea6a5   11 months ago        13.3kB
```

Si nuestro **Dockerfile** tuviera otro nombre, tendríamos que indicarlo con la opción *-f* en el comando.

```
docker build -t app-python:latest -f <Dockerfile_Con_Otro_Nombre> .
```

La utilidad del **.dockerignore** la veremos a continuación. Para ello crearemos un archivo "basura" de 6GB:

```
fallocate -l 6G basura
```

Si hacemos un listado largo veremos que se ha creado un fichero de 6 GB:

```
root@ubuntu2004:/home/ubuntu/appdemo# ls -la
total 6291476
drwxrwxr-x  2 ubuntu ubuntu       4096 Sep  7 11:19 .
drwxr-xr-x 17 ubuntu ubuntu       4096 Sep  7 10:56 ..
-rw-rw-r--  1 ubuntu ubuntu         70 Sep  7 10:55 app.py
-rw-r--r--  1 root   root   6442450944 Sep  7 11:19 basura
-rw-rw-r--  1 ubuntu ubuntu        109 Sep  7 11:08 Dockerfile
```

Si ahora hago un docker build veremos que se envían al contexto todos los ficheros del directorio y que el proceso lleva mucho más tiempo.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker build -t app-pythonv2:latest .
Sending build context to Docker daemon  6.442GB
Step 1/4 : FROM ubuntu:latest
 ---> 2dc39ba059dc
Step 2/4 : RUN apt update && apt install -y python3
 ---> Using cache
 ---> bf247ff5a078
Step 3/4 : COPY app.py /app.py
 ---> Using cache
 ---> 75f23aa4784e
Step 4/4 : CMD ["python3","/app.py"]
 ---> Using cache
 ---> 9b38010ed31b
Successfully built 9b38010ed31b
Successfully tagged app-pythonv2:latest
```

Aunque podemos ver que en algunas capas utiliza **caché** para acelerar el proceso, porque si ya las ha construido, no vuelve a hacerlo.

Para evitarlo podemos crear un fichero llamado **.dockerignore** con el contenido:

```
basura
```

Y ahora vuelvo a ejecutar el docker build veré que ya no se envía al contexto.

Es de buenas prácticas utilizar el **.dockerignore** y enviar al contexto sólo aquellos ficheros necesarios para construir la imagen.

Subir imágenes al repositorio

Para subir nuestra imagen al repositorio necesitamos disponer de una **cuenta de usuario** en Docker Hub.

Debemos de etiquetar nuestra imagen:

```
docker tag app-python:latest jgestal/app-python:latest
```

Si ejecutamos:

```
docker images | grep app
```

Veremos que ha creado una nueva imagen. Recordamos que las imágenes son de **sólo lectura**. Una vez una imagen está construida no la puedo modificar. Lo que puedo hacer es modificar el Dockerfile y crear otra imagen nueva.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker images | grep app
jgestal/app-python   latest    e015621f343b   49 seconds ago   144MB
app-python           latest    e015621f343b   49 seconds ago   144MB
```

Ahora tenemos que **iniciar sesión** en Docker Hub e introducir nuestro usuario y contraseña.

```
docker login
```

Y una vez iniciada sesión hacemos un **push** de nuestra imagen:

```
docker push jgestal/app-python:latest
```

La imagen se subirá y podemos ver nuestras imágenes en la dirección https://hub.docker.com/repositories.

Ahora voy a borrar las dos imagenes de la app de Python que tengo en mi ordenador:

```
docker rmi app-python:latest jgestal/app-python:latest
```

Si están siendo utilizadas, es posible que el comando de borrado dé error, por lo que podemos forzar el borrado con la opción *-f*.

```
docker rmi -f app-python:latest jgestal/app-python:latest
```

Ahora si listamos las imágenes con la app de python, veremos que ya no están:

```
docker images | grep app
```

Pero podemos descargarnos la imagen de nuestro repositorio mediante **pull**. Es posible que debamos autenticarnos antes otra vez.

```
docker pull jgestal/app-python
```

Y si hacemos:

```
docker images | grep app
```

Veremos que volvemos a disponer de nuestra imagen.

Contenedores en Docker

Un **contenedor** es el lugar en el que se van a ejecutar las imágenes. Se puede modificar su contenido, pero los **cambios** son **temporales**.

Vamos a ver los **comandos** más importantes para manejar contenedores.

Listar contenedores

```
docker ps
```

El comando **docker ps** nos permite ver los **contenedores en ejecución**.

Si añadimos la opción *-a* podemos ver también aquellos que estén parados.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker ps –a
CONTAINER ID   IMAGE         COMMAND              CREATED        STATUS                     PORTS     NA
e8a47ba48c73   httpd         "httpd-foreground"   18 hours ago   Exited (0) 15 hours ago            ap
8e4cf5a779d4   hello-world   "/hello"             18 hours ago   Exited (0) 18 hours ago            dr
```

Crear un contenedor

Se utiliza el comando **docker run**. Se le pueden pasar varios argumentos, y los más utilizados son:

- *--name*: Indica un nombre del contenedor.
- *-d*: Corre el contenedor en segundo plano (demonio).
- *-p*: Expone un puerto.
- *-e*: Crea variables de entorno en el contenedor.
- *--network*: Indica una red al contenedor, por defecto **bridge**.
- *--ip*: Asigna una IP al contenedor.
- *--hostname*: Asigna un nombre de dominio para el contenedor.
- *--memory*: Establece límites de memoria.
- *--cpuset-cpus*: Limita el número de CPUs que puede utilizar el contenedor.

Ejemplo:

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

- Corre un contenedor en segundo plano *-d*.
- Expone el puerto 80 del contenedor al 8080 del host *-p 8080:80*.
    - ¿Cómo sabemos que *nginx* utiliza el puerto **80**? Pues podemos verlo en Docker Hub en la documentación de la propia imagen.
- Se le asigna el nombre de nginx *--name nginx*.
- Se utiliza la imagen de nginx.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker run -d -p 8080:80 --name nginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
7a6db449b51b: Already exists
ca1981974b58: Pull complete
d4019c921e20: Pull complete
7cb804d746d4: Pull complete
e7a561826262: Pull complete
7247f6e5c182: Pull complete
Digest: sha256:b95a99feebf7797479e0c5eb5ec0bdfa5d9f504bc94da550c2f58e839ea6914f
Status: Downloaded newer image for nginx:latest
b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9d7914ae9af9fe34
```

Si ahora escribimos **docker ps** podremos ver que tenemos activo el contenedor de nginx:

```
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS
b31b82b705f1   nginx     "/docker-entrypoint.…"   4 minutes ago   Up 4 minutes   0.0.0.0:8080->80/tcp,
```

Podemos ir al navegador y escribir:

```
localhost:8080
```

Y veremos que nginx está funcionando. Si pone "*It works*", el navegador tiene cacheado a Apache. Hay que refrescar para ver el mensaje de "*Welcome to nginx!*".

Pasando variables de entorno

Ahora vamos a levantar otro nginx pero esta vez le vamos a pasar variables de entorno. Como el puerto 8080 ya lo estoy utilizando, voy a poner el 8081; y como el nombre nginx ya lo estoy usando, voy a poner nginx-vars.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker run -d -p 8081:80 -e NOMBRE=JUAN --name nginx-vars nginxb9
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
b9fd84a15287   nginx     "/docker-entrypoint.…"   4 seconds ago    Up 4 seconds    0.0.0.0:8081->80/tcp
b31b82b705f1   nginx     "/docker-entrypoint.…"   10 minutes ago   Up 10 minutes   0.0.0.0:8080->80/tcp
```

Ahora podríamos probar que funciona escribiendo en el navegador:

```
localhost:8081
```

Conectándonos a la consola de un contenedor

Ahora vamos a contectarnos a un contenedor para enviar comandos con **docker exec*:

```
docker exec -it nginx-vars bash
```

La opción *-i* es para que nos conectemos de forma interactiva y la opción *-t* es para instanciar una terminal. Como último parámetro le pasamos la shell que queremos utilizar, en este caso *bash*.

Una vez dentro, con el comando **env** podemos listar las variables de entorno del contenedor.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker exec -it nginx-vars bash
root@b9fd84a15287:/# env
HOSTNAME=b9fd84a15287
PWD=/
PKG_RELEASE=1~bullseye
HOME=/root
NJS_VERSION=0.7.6
NOMBRE=JUAN
TERM=xterm
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_VERSION=1.23.1
_=/usr/bin/env
```

Con el comando **exit** podemos salir de la consola del contenedor y volver a la consola de nuestro sistema.

Siempre podemos utilizar la ayuda para ir familiarizándonos con los comandos y las opciones de cada uno de ellos.

Obtención de ayuda sobre comandos

```
root@ubuntu2004:/home/ubuntu/appdemo# docker exec --help

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

Run a command in a running container

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

Podemos añadir más variables de entorno pasando más parámetros *-e*.

```
docker run -d -p 8082:80 -e NOMBRE=JUAN -e APELLIDO=GESTAL --name nginx-vars
```

Limitando los recursos

Por defecto los contenedores pueden acceder a **toda la memoria** del sistema. Podemos obtener información sobre los recursos que utilizan los contenedores con el comando **docker stats**.

```
docker stats


CONTAINER ID   NAME         CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
b9fd84a15287   nginx-vars   0.00%     4.215MiB / 3.834GiB   0.11%     5.91kB / 1.86kB   4.13MB / 16.4kB
b31b82b705f1   nginx        0.00%     7.121MiB / 3.834GiB   0.18%     6.21kB / 1.65kB   18.6MB / 16.4kB
```

Podemos ver los recursos de nuestro sistema con la orden **free -g**. La opción *-g* nos muestra los valores en GB:

```
root@ubuntu2004:/home/ubuntu/appdemo# free -g
              total        used        free      shared  buff/cache   available
Mem:              3           1           0           0           2           2
Swap:             0           0           0
```

Podemos limitar la memoria de un contenedor utilizando la opción **--memory***:

```
docker run -d -p 8083:80 --name nginx-memory --memory 200m nginx
```

De esta forma limitaríamos la memoria del contenedor a 200 MB.

Podríamos limitar el procesador con la opción **--cpuset-cpus** . Funciona por núcleos, empezando a contar en 0.

```
docker run -d -p 8084:80 --name nginx-cpu --cpuset-cpus 0 nginx
```

Si ponemos el comando **top** podemos ver los programas que están corriendo y su uso de recursos. También nos mostrará las CPUs de las que dispone nuestro sistema.

```
top
```

Asignando un hostname

Podemos asignar un **hostname** en la creación de un contenedor mediante la opción *--hostname*.

```
docker run -d --hostname contenedor-apache httpd
```

Aparentemente no pasa nada, pero si entramos de forma iterativa al contenedor veremos que tiene ese hostname.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker run -d --hostname contenedor-apache httpd
b24881c1518d05f67140dd85ddd3c8028ada4009f6e38fc0e59907f23c175a79
root@ubuntu2004:/home/ubuntu/appdemo# ^C
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
b24881c1518d   httpd     "httpd-foreground"       12 minutes ago   Up 12 minutes   80/tcp
3cd3036914dc   nginx     "/docker-entrypoint.…"   20 minutes ago   Up 20 minutes   0.0.0.0:8084->80/tcp
e5bd5b1f14e8   nginx     "/docker-entrypoint.…"   52 minutes ago   Up 52 minutes   0.0.0.0:8083->80/tcp
b9fd84a15287   nginx     "/docker-entrypoint.…"   3 hours ago      Up 3 hours      0.0.0.0:8081->80/tcp
b31b82b705f1   nginx     "/docker-entrypoint.…"   3 hours ago      Up 3 hours      0.0.0.0:8080->80/tcp
root@ubuntu2004:/home/ubuntu/appdemo# docker exec -it optimistic_dewdney bash
root@contenedor-apache:/usr/local/apache2#
```

El comando **docker ps -l** muestra el último contenedor creado y **docker ps -n 2** muestra los dos últimos.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker ps -l
CONTAINER ID   IMAGE     COMMAND              CREATED          STATUS          PORTS     NAMES
b24881c1518d   httpd     "httpd-foreground"   15 minutes ago   Up 15 minutes   80/tcp    optimistic_dew
root@ubuntu2004:/home/ubuntu/appdemo# docker ps -n 2
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
b24881c1518d   httpd     "httpd-foreground"       15 minutes ago   Up 15 minutes   80/tcp
3cd3036914dc   nginx     "/docker-entrypoint.…"   23 minutes ago   Up 23 minutes   0.0.0.0:8084->80/tcp
```

También se le puede asignar una **IP** a un contenedor, pero lo veremos más adelante cuando hablemos de redes.

Renombrar un contenedor

Para renombrar un contenedor, la sintaxis es:

```
docker rename <oldname> <newname>
```

Vamos a renombrar el contenedor *optimistic_dewdney* que es un nombre que ha elegido Docker ya que no utilicé la opción **--name**.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker rename optimistic_dewdney httpd-con-nombre
```

Y si ahora hacemos un **docker ps -l** veremos que ha cambiado de nombre.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker ps -l
CONTAINER ID   IMAGE     COMMAND              CREATED          STATUS          PORTS     NAMES
b24881c1518d   httpd     "httpd-foreground"   18 minutes ago   Up 18 minutes   80/tcp    httpd-con-nomb
```

Arrancar, parar y reiniciar un contenedor

La diferencia entre **start** y **run** es que **run** crea y arranca y **start** solo arranca un contenedor ya creado y que está parado.

Para **detener** un contenedor se utiliza el comando **stop**.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker stop httpd-con-nombre
httpd-con-nombre
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS             PORTS
3cd3036914dc   nginx     "/docker-entrypoint.…"   50 minutes ago      Up 50 minutes      0.0.0.0:8084->
e5bd5b1f14e8   nginx     "/docker-entrypoint.…"   About an hour ago   Up About an hour   0.0.0.0:8083->
b9fd84a15287   nginx     "/docker-entrypoint.…"   3 hours ago         Up 3 hours         0.0.0.0:8081->
b31b82b705f1   nginx     "/docker-entrypoint.…"   3 hours ago         Up 3 hours         0.0.0.0:8080->
root@ubuntu2004:/home/ubuntu/appdemo# docker start httpd-con-nombre
httpd-con-nombre
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS             PORTS
b24881c1518d   httpd     "httpd-foreground"       42 minutes ago      Up 2 seconds       80/tcp
3cd3036914dc   nginx     "/docker-entrypoint.…"   50 minutes ago      Up 50 minutes      0.0.0.0:8084->
e5bd5b1f14e8   nginx     "/docker-entrypoint.…"   About an hour ago   Up About an hour   0.0.0.0:8083->
b9fd84a15287   nginx     "/docker-entrypoint.…"   3 hours ago         Up 3 hours         0.0.0.0:8081->
b31b82b705f1   nginx     "/docker-entrypoint.…"   3 hours ago         Up 3 hours         0.0.0.0:8080->
```

Cabe recordar que con la orden **docker ps** se muestran los contenedores activos y para ver también los que están parados.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED       STATUS       PORTS                                  
3cd3036914dc   nginx     "/docker-entrypoint.…"   2 hours ago   Up 2 hours   0.0.0.0:8084->80/tcp, :::8
e5bd5b1f14e8   nginx     "/docker-entrypoint.…"   3 hours ago   Up 3 hours   0.0.0.0:8083->80/tcp, :::8
b9fd84a15287   nginx     "/docker-entrypoint.…"   4 hours ago   Up 4 hours   0.0.0.0:8081->80/tcp, :::8
b31b82b705f1   nginx     "/docker-entrypoint.…"   5 hours ago   Up 5 hours   0.0.0.0:8080->80/tcp, :::8
root@ubuntu2004:/home/ubuntu/appdemo# docker ps -a
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS                       PORTS
b24881c1518d   httpd         "httpd-foreground"       2 hours ago     Exited (0) 14 seconds ago
3cd3036914dc   nginx         "/docker-entrypoint.…"   2 hours ago     Up 2 hours                   0.0.0.
e5bd5b1f14e8   nginx         "/docker-entrypoint.…"   3 hours ago     Up 3 hours                   0.0.0.
b9fd84a15287   nginx         "/docker-entrypoint.…"   4 hours ago     Up 4 hours                   0.0.0.
b31b82b705f1   nginx         "/docker-entrypoint.…"   5 hours ago     Up 5 hours                   0.0.0.
e8a47ba48c73   httpd         "httpd-foreground"       23 hours ago    Exited (0) 20 hours ago
8e4cf5a779d4   hello-world   "/hello"                 23 hours ago    Exited (0) 23 hours ago
```

El comando **docker restart**, como su nombre indica, reinicia un contenedor. Lo para y lo levanta.

Borrar un contenedor

El comando **docker rm** se utiliza para borrar un contenedor.

Podemos pasarle algunos parámetros:

- *-f*: Fuerza a eliminarlo si el contenedor está en ejecución. El comando sin *-f* devolvería error si el contenedor estuviera corriendo.
- *-v*: Borra el volumen anónimo asociado al contenedor (en el caso de que tenga un volumen anónimo asociado, si no, no hace nada).

Ejecutar comandos en un contenedor

El comando **docker exec** se utiliza para ejecutar comandos.

Se puede utilizar para *entrar* en el contenedor, si se le pasa las opciones *-it* y un shell.

Ejecutar comando:

```
docker exec <contenedor> <comando>
```

Por ejemplo:

```
docker exec nginx-cpu ls
```

Entrar en un contenedor y utilizar la shell bash:

```
docker exec -it contenedor bash
```

Podemos salir de los contenedores con **CTRL D** o con **exit**.

Ver los logs de un contenedor

El comando **docker logs** nos permite ver los **logs** de un contenedor.

Podemos pasarle algunos parámetros:

- *-f*: Permite ver los logs en tiempo real, como un *tail -200f*.
- *-t*: Añade un timestamp (marca de tiempo) a los logs. Viene bien porque hay aplicaciones que no añaden marca de tiempo a sus logs, y así podemos forzar que aparezca.

```
docker logs nginx-memory
```

Crear una imagen a partir de un contenedor

Para crear una imagen a partir de un contenedor utilizaremos **docker commit**.

```
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Esto puede ser útil en ciertas ocasiones, pero en general la buena práctica es crear la imagen que queramos con un Dockerfile bien definido.

Inspeccionar un contenedor

Para inspeccionar un contenedor utilizamos el comando **docker inspect**.

```
root@ubuntu2004:/home/ubuntu/appdemo# docker inspect nginx
[
    {
        "Id": "b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9d7914ae9af9fe34",
        "Created": "2022-09-07T16:33:01.619869877Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 91050,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2022-09-07T16:33:01.950491886Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:2b7d6430f78d432f89109b29d88d4c36c868cdbf15dc31d2132ceaa02b993763",
        "ResolvConfPath": "/var/lib/docker/containers/b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9
        "HostnamePath": "/var/lib/docker/containers/b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9d7
        "HostsPath": "/var/lib/docker/containers/b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9d7914
        "LogPath": "/var/lib/docker/containers/b31b82b705f1ca6f95c581ba4e18671c31cb12408c59905e9d7914ae
        "Name": "/nginx",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "docker-default",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
            "PortBindings": {
                "80/tcp": [
                    {
                        "HostIp": "",
                        "HostPort": "8080"
```

```
        }
    ]
},
"RestartPolicy": {
    "Name": "no",
    "MaximumRetryCount": 0
},
"AutoRemove": false,
"VolumeDriver": "",
"VolumesFrom": null,
"CapAdd": null,
"CapDrop": null,
"CgroupnsMode": "host",
"Dns": [],
"DnsOptions": [],
"DnsSearch": [],
"ExtraHosts": null,
"GroupAdd": null,
"IpcMode": "private",
"Cgroup": "",
"Links": null,
"OomScoreAdj": 0,
"PidMode": "",
"Privileged": false,
"PublishAllPorts": false,
"ReadonlyRootfs": false,
"SecurityOpt": null,
"UTSMode": "",
"UsernsMode": "",
"ShmSize": 67108864,
"Runtime": "runc",
"ConsoleSize": [
    0,
    0
],
"Isolation": "",
"CpuShares": 0,
"Memory": 0,
"NanoCpus": 0,
"CgroupParent": "",
"BlkioWeight": 0,
"BlkioWeightDevice": [],
"BlkioDeviceReadBps": null,
"BlkioDeviceWriteBps": null,
"BlkioDeviceReadIOps": null,
"BlkioDeviceWriteIOps": null,
"CpuPeriod": 0,
"CpuQuota": 0,
"CpuRealtimePeriod": 0,
"CpuRealtimeRuntime": 0,
"CpusetCpus": "",
```

```
        "CpusetMems": "",
        "Devices": [],
        "DeviceCgroupRules": null,
        "DeviceRequests": null,
        "KernelMemory": 0,
        "KernelMemoryTCP": 0,
        "MemoryReservation": 0,
        "MemorySwap": 0,
        "MemorySwappiness": null,
        "OomKillDisable": false,
        "PidsLimit": null,
        "Ulimits": null,
        "CpuCount": 0,
        "CpuPercent": 0,
        "IOMaximumIOps": 0,
        "IOMaximumBandwidth": 0,
        "MaskedPaths": [
            "/proc/asound",
            "/proc/acpi",
            "/proc/kcore",
            "/proc/keys",
            "/proc/latency_stats",
            "/proc/timer_list",
            "/proc/timer_stats",
            "/proc/sched_debug",
            "/proc/scsi",
            "/sys/firmware"
        ],
        "ReadonlyPaths": [
            "/proc/bus",
            "/proc/fs",
            "/proc/irq",
            "/proc/sys",
            "/proc/sysrq-trigger"
        ]
    },
    "GraphDriver": {
        "Data": {
            "LowerDir": "/var/lib/docker/overlay2/0c5f2a1e1c74286adbbc9ddc36ba060b842ab6d609a9ac01f
            "MergedDir": "/var/lib/docker/overlay2/0c5f2a1e1c74286adbbc9ddc36ba060b842ab6d609a9ac01
            "UpperDir": "/var/lib/docker/overlay2/0c5f2a1e1c74286adbbc9ddc36ba060b842ab6d609a9ac01f
            "WorkDir": "/var/lib/docker/overlay2/0c5f2a1e1c74286adbbc9ddc36ba060b842ab6d609a9ac01f8
        },
        "Name": "overlay2"
    },
    "Mounts": [],
    "Config": {
        "Hostname": "b31b82b705f1",
        "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=1.23.1",
            "NJS_VERSION=0.7.6",
            "PKG_RELEASE=1~bullseye"
        ],
        "Cmd": [
            "nginx",
            "-g",
            "daemon off;"
        ],
        "Image": "nginx",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": [
            "/docker-entrypoint.sh"
        ],
        "OnBuild": null,
        "Labels": {
            "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>"
        },
        "StopSignal": "SIGQUIT"
    },
    "NetworkSettings": {
        "Bridge": "",
        "SandboxID": "5d096e7377c723dcc7c9ea4a04a1e7e5e661973719fd71b613cab840f1a0dd6d",
        "HairpinMode": false,
        "LinkLocalIPv6Address": "",
        "LinkLocalIPv6PrefixLen": 0,
        "Ports": {
            "80/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8080"
                },
                {
                    "HostIp": "::",
                    "HostPort": "8080"
                }
            ]
        },
        "SandboxKey": "/var/run/docker/netns/5d096e7377c7",
        "SecondaryIPAddresses": null,
```

```
            "SecondaryIPv6Addresses": null,
            "EndpointID": "ef394fa807ac99446414401ef106870956fed56e883353a9d2306bdba4ab55df",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "21a4dbd33e81f60712725e3f93997a634cd4adef679efca908204a5838352d5b",
                    "EndpointID": "ef394fa807ac99446414401ef106870956fed56e883353a9d2306bdba4ab55df",
                    "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
                }
            }
        }
    }
]
```


Podemos ver su **IP**, su **gateway**, si tiene algún **volúmen** asociado, su **estado**, etc.

Volúmenes

Los **volúmenes** se utilizan para hacer persistentes los datos de un contenedor. Existen varios tipos.

Volúmenes de Host

Los **volúmenes de host** mapean un directorio del **docker host** al directorio donde la aplicación guarda los datos en el contenedor.

Este volúmen **no se borra** con el comando:

```
docker rm -fv <contenedor>
```

La opción *-v* borraría el volumen anónimo.

Para probarlo creamos un directorio llamado nginx-conf.

```
mkdir nginx-conf
```

Ahora vamos a copiar un archivo de configuración de un nginx que se instale por defecto. Para ello creo un contenedor de nginx temporal:

```
docker run -d --name nginx-temp nginx
```

Ahora utilizamos el comando **docker cp** que vale para copiar ficheros del host al contenedor o viceversa.

```
docker cp nginx-temp:/etc/nginx/nginx.conf nginx-conf/
```

Ahora si entro a mi directorio nginx-conf veré que tengo un fichero de configuración.

```
root@ubuntu2004:/home/ubuntu/# cd nginx-conf/
root@ubuntu2004:/home/ubuntu/nginx-conf# ls
nginx.conf
```

El fichero **nginx.conf** es donde nginx guarda la configuración. Vamos a entrar en el fichero y descomentar la linea de # *gzip on;* tal que quede:

```
gzip on
```

Así podemos ver que la configuración que sustituye.

Ahora borramos el contenedor temporal que acabamos de crear:

```
docker rm -fv nginx-temp
```

Y ahora voy a mapear el fichero de configuración que tengo de nginx a la configuración de un nuevo contenedor:

```
docker run -d --name nginx-volume -v $PWD/nginx-conf/nginx.conf:/etc/nginx/nginx.conf nginx
```

Y ahora haciendo un **docker exec** donde veamos el contenido del archivo de configuración del contenedor, veremos que la línea de *gzip on* está descomentada.

```
root@ubuntu2004:/home/ubuntu# docker exec nginx-volume cat /etc/nginx/nginx.conf

user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    gzip  on;

    include /etc/nginx/conf.d/*.conf;
}
```

Los cambios se producen en los dos sentidos. Si editamos el fichero del contenedor, cambiará el fichero del host.

Volúmenes anónimos

Los **volúmenes anónimos** son creados y manejados por docker. La información se guarda en un directorio que crea docker en el **Docker Root Dir** con un nombre aleatorio.

Para ver cuál es el Docker Root dir podemos escribir el comando:

```
docker info | grep -i root
```

Y se imprimirá en consola el directorio:

```
Docker Root Dir: /var/lib/docker
```

Los volúmenes se guardarían en una carpeta llamada **volumes**:

```
root@ubuntu2004:/home/ubuntu# ls /var/lib/docker/volumes/
backingFsBlockDev  metadata.db
```

El Dockerfile tiene la capa **VOLUME** que no la hemos visto, pero que se puede especificar un volumen.

Si no ponemos nada en el docker run, se crea un volumen anónimo automáticamente.

Ahora que ya sé donde guarda la configuración nginx, puedo crear un contenedor con un volumen anónimo de la siguiente forma:

```
docker run -d --name nginx-vol-anon -v /etc/nginx
```

Solo estoy pasando la ruta del directorio del contenedor y ninguna ruta del host.

```
root@ubuntu2004:/home/ubuntu# docker run -d --name nginx-vol-anon -v /etc/nginx nginx
75b6e12b471ad3197da96c65926452192c98f9dfafea969245a619b94b91aa40
root@ubuntu2004:/home/ubuntu# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS
75b6e12b471a   nginx     "/docker-entrypoint.…"   3 seconds ago   Up 3 seconds   80/tcp
dd6b6426ecd4   nginx     "/docker-entrypoint.…"   2 hours ago     Up 2 hours     80/tcp
3cd3036914dc   nginx     "/docker-entrypoint.…"   24 hours ago    Up 24 hours    0.0.0.0:8084->80/tcp,
e5bd5b1f14e8   nginx     "/docker-entrypoint.…"   24 hours ago    Up 24 hours    0.0.0.0:8083->80/tcp,
b9fd84a15287   nginx     "/docker-entrypoint.…"   26 hours ago    Up 26 hours    0.0.0.0:8081->80/tcp,
b31b82b705f1   nginx     "/docker-entrypoint.…"   26 hours ago    Up 26 hours    0.0.0.0:8080->80/tcp,
```

Si ahora hago un *ls* de /var/lib/docker/volumes veremos que aparece un "chorizo" (una carpeta con nombre muy largo de caracteres aleatorios).

```
root@ubuntu2004:/home/ubuntu# ls -l /var/lib/docker/volumes
total 28
drwx-----x 3 root root   4096 Sep  8 14:41 a34029e7b4153b0f41c9dbd81eb381f06fb6cb9663e3dfdee62f570fe385
```

Si entramos en esa ruta, veremos que tendremos dentro la carpeta **_data** el contenido de nuestro contenedor que contiene en la carpeta /etc/nginx.

```
root@ubuntu2004:/home/ubuntu# ls /var/lib/docker/volumes/a34029e7b4153b0f41c9dbd81eb381f06fb6cb9663e3df
total 28
drwxr-xr-x 2 root root 4096 Sep  8 14:41 conf.d
-rw-r--r-- 1 root root 1007 Jul 19 10:05 fastcgi_params
-rw-r--r-- 1 root root 5349 Jul 19 10:05 mime.types
lrwxrwxrwx 1 root root   22 Jul 19 11:06 modules -> /usr/lib/nginx/modules
-rw-r--r-- 1 root root  648 Jul 19 11:06 nginx.conf
-rw-r--r-- 1 root root  636 Jul 19 10:05 scgi_params
-rw-r--r-- 1 root root  664 Jul 19 10:05 uwsgi_params
```

Con los volumenes anónimos hay que tener mucho cuidado porque si ahora pongo la instrucción:

```
docker rm -fv nginx-vol-anon
```

Lo que va a hacer Docker va a ser **borrar el contenedor y el volumen anónimo asociado** a ese contenedor.

Por este motivo, **los volúmenes anónimos no se recomiendan**. Porque podemos borrar sin querer los datos y porque el nombre del directorio no es descriptivo.

Volúmenes nombrados

Los **volúmenes nombrados** son una mezcla entre un volumen de host y un volúmen anónimo. Nosotros lo creamos con el comando **docker volume create** y se guarda en el **Docker Root Dir**.

Este tipo de volúmenes **no se borran** con el comando:

```
docker rm -fv <contenedor>
```

Son el tipo de volúmenes más cómodos de usar.

Puedo **crear un volumen** con:

```
docker volume create <nombre>
```

Y para **listar** los que tengo:

```
docker volume ls
```

```
root@ubuntu2004:/home/ubuntu# docker volume create jenkins_home
jenkins_home
root@ubuntu2004:/home/ubuntu# docker volume ls
DRIVER    VOLUME NAME
local     jenkins_home
```

Si hacemos un ls al directorio de volumes, veremos que aparece:

```
root@ubuntu2004:/home/ubuntu# ls -l /var/lib/docker/volumes/
total 28
brw------- 1 root root 253, 0 Sep  7 10:47 backingFsBlockDev
drwx-----x 3 root root   4096 Sep  8 15:01 jenkins_home
-rw------- 1 root root  32768 Sep  8 15:01 metadata.db
```

Ahora vamos a crear un contenedor que utilice el volumen que ya tenenemos:

```
docker run -d --name jenkins-vol-nombrado -p 8089:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home
```

Si me voy a un navegador y pongo localhost:8089 podremos ver el jenkins que nos pide una clave que podemos obtener con el comando:

```
docker exec jenkins-vol-nombrado cat /var/jenkins_home/secrets/initialAdminPassword
```

Los puertos de jenkins utilizados están en la documentación del la imagen.

Y si ahora hago un:

```
docker rm -fv jenkins-vol-nombrado
```

Borro el contenedor, pero la información persiste.

```
docker volume ls
```

Redes en Docker

Las **redes** en Docker es la forma en la que vamos a poder acceder al contenedor y la forma que van a tener los contenedores para comunicarse entre sí.

Existen diferentes **tipos de redes**.

Redes Bridge

Es el **driver por defecto**. Se utiliza para comunicar contenedores en el mismo docker host.

No puedo comunicar contenedores que están en **diferentes Docker Host**. Para eso se utiliza el tipo de driver **overlay**.

La puerta de enlace de esta red es la interfaz **docker0** que se crea en la instalación de Docker.

Si ejecutamos:

```
docker network ls
```

Obtendremos:

```
NETWORK ID     NAME      DRIVER    SCOPE
21a4dbd33e81   bridge    bridge    local
5e564c5f9ec5   host      host      local
1cd50adbd28b   none      null      local
```

Y si en el host ejecutamos el comando **ifconfig** veremos que hay una interfaz **docker0**.

```
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:b3ff:fee3:8d5c  prefixlen 64  scopeid 0x20<link>
        ether 02:42:b3:e3:8d:5c  txqueuelen 0  (Ethernet)
        RX packets 15434  bytes 1507632 (1.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18597  bytes 64837893 (64.8 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- En este tipo de red no se puede hacer **ping** de los contenedores **por su nombre**. Hay que poner la dirección IP.
- Tampoco se puede asignar una **IP fija** a un contenedor con este tipo de red.

Si hacemos un:

```
docker inspect jenkins-vol-nombrado | grep IPAddress
```

Obtenemos la dirección IP de un contenedor. Así que podríamos hacerle ping desde otro (suponiendo que ese contenedor dispone del comando **ping**).

```
root@ubuntu2004:/home/ubuntu# docker exec e4aaea0c78c7 ping 172.17.0.7
PING 172.17.0.7 (172.17.0.7): 56 data bytes
64 bytes from 172.17.0.7: seq=0 ttl=42 time=0.029 ms
64 bytes from 172.17.0.7: seq=1 ttl=42 time=0.048 ms
64 bytes from 172.17.0.7: seq=2 ttl=42 time=0.070 ms
64 bytes from 172.17.0.7: seq=3 ttl=42 time=0.046 ms
```

Crear una red

Para crear una red:

```
docker network create red-casa --subnet 192.168.0.0/24 --gateway 192.168.0.1
```

Si hacemos un **docker network ls**:

```
oot@ubuntu2004:/home/ubuntu# docker network ls
NETWORK ID     NAME       DRIVER    SCOPE
21a4dbd33e81   bridge     bridge    local
5e564c5f9ec5   host       host      local
1cd50adbd28b   none       null      local
c315c37eaad0   red-casa   bridge    local
```

Todos los contenedores que creemos y que asignemos a la red se van a crear dentro de la **subred 192.168.0.0/24**.

```
docker run -d --network red-casa --name nginx-con-red nginx
```

Ahora hacemos un inspect:

```
root@ubuntu2004:/home/ubuntu# docker inspect nginx-con-red | grep IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "192.168.0.2",
```

¿Y qué es la **192.168.0.1**? Pues al crear la red, ha creado una **interfaz** de red en nuestro **host**.

Si escribimos el comando:

```
ip address
```

Podemos encontrar la interfaz de red:

```
32: br-c315c37eaad0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:99:78:ee:a9 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 brd 192.168.0.255 scope global br-c315c37eaad0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:99ff:fe78:eea9/64 scope link
       valid_lft forever preferred_lft forever
```

Eso permite que los contenedores se puedan comunicar con el **mundo exterior** y con los contenedores que **pertenezcan a esa red**.

Puedo crear otro contenedor asociado a la misma subred con la IP que quiera (y que no esté siendo usada).

```
docker run -d --network red-casa --name nginx-con-ip --ip 192.168.0.10 nginx
```

Si hacemos un docker inspect del contenedor **nginx-con-ip** veremos que pertenece a la misma red.

```
root@ubuntu2004:/home/ubuntu# docker inspect nginx-con-ip | grep IPv4Address
                        "IPv4Address": "192.168.0.10"
```

En las redes que nosotros creamos sí que podemos hacer **ping por el nombre**. Vamos a verlo creando dos contenedores de **centos**:

Creamos el primer contenedor:

```
docker run -dti --name centos1 --network red-casa --ip 192.168.0.20 centos
```

Creamos el segundo:

```
docker run -dti --name centos2 --network red-casa --ip 192.168.0.30 centos
```

Hacemos ping desde el primero al segundo por el nombre:

```
root@ubuntu2004:/home/ubuntu# docker exec centos1 ping centos2
PING centos2 (192.168.0.30) 56(84) bytes of data.
64 bytes from centos2.red-casa (192.168.0.30): icmp_seq=1 ttl=64 time=0.077 ms
64 bytes from centos2.red-casa (192.168.0.30): icmp_seq=2 ttl=64 time=0.073 ms
64 bytes from centos2.red-casa (192.168.0.30): icmp_seq=3 ttl=64 time=0.112 ms
```

Y vemos que el segundo responde.

También funcionaría si hiciéramos **ping a la IP**.

Resumen de redes

Tenemos las **red bridge**, que ya vimos. También las **red host** que lo que hacen es meter todas las interfaces de red y el host a un contenedor. La **red none** que lo que hace es no asignar red ni ip a un contenedor. Y si queremos comunicar contenedores que están en **distintos docker host** el tipo de red debería de ser **overlay**.

Vamos a crear ahora una **red de clase B**:

```
docker network create --subnet 172.16.0.0/16 --gateway 172.16.0.1 claseb
```

Si ahora ponemos:

```
ip address
```

Veremos que se crea una interfaz que va a ser la interfaz puente para que se comuniquen todos los contenedores que creemos en esa interfaz.

```
47: br-a7bb9f5bd27f: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group defaul
    link/ether 02:42:29:21:1d:41 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.1/16 brd 172.16.255.255 scope global br-a7bb9f5bd27f
       valid_lft forever preferred_lft forever
```

Ahora si creamos un contenedor llamado **centos3** que asignamos a la red **claseb** no debería poder verse con los contenedores de **red-casa**.

```
root@ubuntu2004:/home/ubuntu# docker run -dti --name centos3 --network claseb  centos
4a55b9fcdfcc754c50a868396b8c21ff1c28aba99c0977946aaf2a25dd9b81f6
root@ubuntu2004:/home/ubuntu# docker exec centos1 ping centos3
ping: centos3: Name or service not known
```

Conectar contenedores a diferentes redes

Docker nos permite **conectar contenedores a diferentes redes**.

Podemos conectar **centos1** a la red **claseb** y así ya podría comunicarse con **centos3**.

```
docker network connect claseb centos1
```

Y ahora ya podría hacer **ping** desde **centos1** a **centos3**.

```
root@ubuntu2004:/home/ubuntu# docker exec centos1 ping centos3
PING centos3 (172.16.0.2) 56(84) bytes of data.
64 bytes from centos3.claseb (172.16.0.2): icmp_seq=1 ttl=64 time=0.071 ms
64 bytes from centos3.claseb (172.16.0.2): icmp_seq=2 ttl=64 time=0.071 ms
64 bytes from centos3.claseb (172.16.0.2): icmp_seq=3 ttl=64 time=0.068 ms
```

Si entro al contenedor de **centos1**:

```
docker exec -it centos1 bash
```

Y hago un **ip a**:

```
root@ubuntu2004:/home/ubuntu# docker exec -it centos1 bash
[root@526cc463fc5d /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
45: eth0@if46: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:00:14 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.0.20/24 brd 192.168.0.255 scope global eth0
       valid_lft forever preferred_lft forever
50: eth1@if51: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:10:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.16.0.3/16 brd 172.16.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

Veremos que hay una interfaz de red conectada a la subred **192.168.0.0/24** y otra conectada a la subred **172.16.0.0/16**.

El contenedor **centos2** no está conectado a la red **claseb** por lo que no puede comunicarse con **centos3**.

Docker Compose

Docker Compose es una herramienta que nos permite crear **aplicaciones multicontenedor** de una forma muy sencilla y práctica.

Utiliza un **archivo** de configuración **yaml** que contiene información sobre los contenedores.

El fichero por defecto se va a llamar **DockerCompose.yaml**, pero si no lo nombramos de esta manera, tendremos que pasar como parámetro del comando el nombre del fichero con la opción *-f*.

Instalación

Docker Compose es una herramienta independiente y debemos de instalarla siguiendo los pasos indicados en la documentación.

Yo lo he instalado directamente:

```
apt install docker-compose
```

Fichero docker-compose.yml

Para la creación del fichero **docker-compose.yml** es recomendable tomar como referencia la documentación de docker. El motivo es que el formato del fichero puede variar entre versiones y porque además hay multitud de opciones.

El fichero **docker-compose.yml** suele configurar una serie de etiquetas principales:

1. Version
2. Services
3. Volumes
4. Networks

Ficheros YAML

YAML significa **Yet Another Markup Language** y es un **lenguaje de declaración de datos** que facilita la legibilidad.

Los **espacios** o **tabulaciones** son importantes y a la hora de escribir el fichero hay que tener esto en cuenta y ser ordenado.

Wordpress: App multicontenedor

Wordpress necesita un **servidor web** y un **servidor de bases de datos**.

Si vamos a Docker Hub y buscamos Wordpress en su documentación encontraremos un ejemplo de archivo **yaml**.

```
version: '3.1'

services:

  wordpress:
    image: wordpress
    restart: always
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass
      WORDPRESS_DB_NAME: exampledb
    volumes:
      - wordpress:/var/www/html

  db:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_DATABASE: exampledb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
    volumes:
      - db:/var/lib/mysql

volumes:
  wordpress:
  db:
```

Vamos a ir haciendo este ejemplo paso a paso.

Lo primero, creamos un directorio wordpress:

```
mkdir wordpress
cd wordpress
```

Creamos dentro del directorio un fichero **docker-compose.yml** y lo editamos:

```
nano docker-compose.yaml
```

Tenemos que configurar las 4 etiquetas principales:

```
version:
services:
volumes:
networks:
```

En version vamos a poner la "3.8".

```
version: "3.8"
services:
volumes:
networks:
```

Ahora vamos con los servicios. Vamos a crear una **base de datos** que va a usar **mysql**. La llamamos "dbwordpress" y va a ser el nombre que va a tener nuestro servicio.

Podríamos abrir algún puerto, pero como la base de datos es accedida solo por el otro contenedor, no tenemos que hacerlo.

```
version: "3.8"
services:
  dbwordpress:
    container_name: wordpressdb
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: "123456"
      MYSQL_DATABASE: wpdatabase
      MYSQL_USER: wpuser
      MYSQL_PASSWORD: "654321"

    volumes:
      - wordpress-data:/var/lib/mysql
    networks:
      - net-wordpress
volumes:
networks:
```

Le damos el nombre al contenedor. Especificamos la imagen que queremos usar, y le pasamos varias variables de entorno para configurar el servicio Mysql.

Si quisiéramos hacerlo con un comando, nos quedaría bastante largo...

```
docker run -d --name wordpressdb -e MYSQL_ROOT_PASSWORD=123456 -e MYSQL_DATABASE=wpdatabase -e (...)
```

También especificamos el volumen nombrado donde Mysql va a guardar sus ficheros: */var/lib/mysql* y está referenciado en la documentación en el apartado **Where to Store Data**.

Creamos por último una red "net-wordpress".

Para esto sirve el **Docker Compose**, para evitar usar comandos y automatizar este tipo de tareas.

Finalizamos nuestro archivo:

```
root@ubuntu2004:/home/ubuntu# cat docker-compose.yaml
version: "3.2"
services:
  dbwordpress:
    container_name: wordpressdb
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: "123456"
      MYSQL_DATABASE: wpdatabase
      MYSQL_USER: wpuser
      MYSQL_PASSWORD: "654321"
    volumes:
      - "wordpress-data:/var/lib/mysql"
    networks:
      - net_wordpress

  webwordpress:
    container_name: wordpressweb
    image: wordpress:latest
    ports:
      - "8090:80"
    environment:
      WORDPRESS_DB_HOST: dbwordpress
      WORDPRESS_DB_USER: wpuser
      WORDPRESS_DB_PASSWORD: "654321"
      WORDPRESS_DB_NAME: wpdatabase
    volumes:
      - "wordpress-web:/var/www/html"
    networks:
      - net_wordpress
    depends_on:
      - dbwordpress
volumes:
  wordpress-data:
  wordpress-web:
networks:
  net_wordpress:
```

La opción **depends_on:** nos permite hacer depender la web de la base de datos. Si el contenedor de la base de datos no se levanta, no se levanta tampoco el contenedor de la web.

En el contenedor web mapeamos su puerto 80 al puerto 8089 del host. En el contenedor de la base de datos no nos interesa exponer puertos.

En las etiquetas finales de volumes y networks añadimos los volúmenes y las redes que hemos dicho que íbamos a usar.

Una vez el archivo listo, ejecutamos:

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

La opción *-d* es para que los levante en segundo plano.

Si vamos al navegador y escribimos: **localhost:8090** deberíamos ver nuestra instalación de Wordpress.

Y con el comando:

```
docker-compose ps
```

Podemos ver los contenedores levantados.

Si añado una entrada a Wordpress, tras configurarlo, y luego hago un:

```
docker-compose down
```

Lo que voy a hacer es parar los contenedores y eliminar la red, pero no elimina los volúmenes.

```
docker volume ls
DRIVER    VOLUME NAME
local     jenkins_home
local     wordpress_wordpress-data
local     wordpress_wordpress-web
```

Los volúmenes siguen, y tienen como prefijo el nombre del directorio en el que se encuentra el docker compose.

Por último, si mi fichero en lugar de llamarse docker-compose.yaml se llamase **docker-compose-wordpress.yaml**, tendría que utilizar la opción *-f*.

```
docker-compose up -d -f docker-compose-wordpress.yaml
```

Fuente: Docker desde CERO a Experto