

# DOCKER

## Contenido

|   |    |
|---|----|
| DOCKER .....  | 1  |
| Introducción .....  | 4  |
| Arquitectura de Docker.....                                 | 4  |
| ¿Qué es una imagen? .....                                   | 4  |
| <b>¿Cómo creo las capas?</b> .....                          | 5  |
| ¿Qué es un contenedor? .....                                | 6  |
| Contenedores vs Máquinas Virtuales.....                     | 7  |
| Docker Images .....   | 8  |
| <b>Imágenes oficiales</b> .....                             | 8  |
| <b>Creando nuestra primer imagen</b> .....                  | 10 |
| <b>Creando un contenedor con nuestra imagen</b> .....       | 13 |
| <b>Dockerfile</b> .....                                     | 15 |
| <b>Construyendo una imagen Apache + PHP + TLS/SSL</b> ..... | 22 |
| <b>Eliminar imágenes</b> .....                              | 26 |
| <b>Cambiar el nombre del Dockerfile</b> .....               | 27 |
| <b>Dangling images</b> .....                                | 27 |
| <b>Multi-Stage-Build</b> .....                              | 29 |
| Docker Containers .....                                     | 31 |
| <b>Listar / Mapear puertos</b> .....                        | 31 |
| <b>Iniciar / Reiniciar / Detener</b> .....                  | 34 |
| <b>Crear un contenedor MySQL</b> .....                      | 37 |
| <b>Variables de entorno</b> .....                           | 41 |
| <b>Crear un contenedor Mongo</b> .....                      | 44 |
| <b>Crear un contenedor Apache / Nginx / Tomcat</b> .....    | 46 |
| <b>Crear un contenedor PostgreSQL</b> .....                 | 48 |
| <b>Crear un contenedor Jenkins</b> .....                    | 49 |
| <b>Administristrar usuarios</b> .....                       | 51 |
| <b>Limitar recursos de un contenedor</b> .....              | 52 |

|   |    |
|---|----|
| <b>Copiar archivos a un contenedor .....</b>                          | 54 |
| <b>Convierte un contenedor en una imagen .....</b>                    | 56 |
| <b>Sobrescribe el CMD de una imagen sin un Dockerfile.....</b>        | 58 |
| <b>Aprende a destruir contenedores automáticamente.....</b>           | 59 |
| <b>Cambiar el Document Root de Docker.....</b>                        | 59 |
| Docker Volumes .....  | 60 |
| <b>¿Por qué son importantes los volúmenes? .....</b>                  | 60 |
| <b>Volúmenes de host – Caso práctico MySQL .....</b>                  | 62 |
| <b>Volúmenes anónimos – Caso práctico MySQL.....</b>                  | 64 |
| <b>Instrucción VOLUME dentro de un Dockerfile .....</b>               | 65 |
| <b>Volúmenes nombrados – Caso práctico MySQL.....</b>                 | 66 |
| <b>Dangling volumes .....</b>   | 67 |
| <b>Persistiendo data en MongoDB.....</b>                              | 68 |
| <b>Persistiendo data en Jenkins.....</b>                              | 70 |
| <b>Persistiendo logs de Nginx .....</b>                               | 72 |
| <b>Comparte volúmenes entre uno o más contenedores.....</b>           | 72 |
| Docker Network.....   | 74 |
| <b>¿Cuál es la red por defecto? .....</b>                             | 74 |
| <b>Crear una red definida por el usuario.....</b>                     | 78 |
| <b>Inspeccionar Redes .....</b>                                       | 80 |
| <b>Agregar contenedores a una red distinta a la por defecto .....</b> | 80 |
| <b>Conectar contenedores en la misma red.....</b>                     | 81 |
| <b>Conectar contenedores en distintas redes .....</b>                 | 83 |
| <b>Eliminar redes .....</b>   | 85 |
| <b>Asignar IP a un contenedor .....</b>                               | 86 |
| <b>La red host .....</b>  | 87 |
| <b>La red none .....</b>  | 87 |
| Docker Compose.....   | 88 |
| <b>Instalación.....</b>   | 88 |
| <b>Primeros pasos.....</b>  | 89 |
| <b>Variables de entorno en Compose .....</b>                          | 91 |
| <b>Volúmenes en Compose .....</b>                                     | 93 |
| <b>Redes en Compose .....</b>   | 95 |

|   |     |
|---|-----|
| <b>Construye imágenes en Compose .....</b>                    | 98  |
| <b>Sobreescribe el CMD de un contenedor con Compose.....</b>  | 99  |
| <b>Limitar recursos en contenedores (Compose v2).....</b>     | 100 |
| <b>Política de reinicio de contenedores .....</b>             | 101 |
| <b>Personaliza el nombre de tu proyecto en Compose.....</b>   | 105 |
| <b>Usar un nombre distinto en el docker-compose.yml .....</b> | 105 |
| <b>Más opciones en Compose .....</b>                          | 106 |
| <b>Instalando WordPress + MySQL .....</b>                     | 107 |
| <b>Instalando Drupal + PostgreSQL .....</b>                   | 110 |
| <b>Instalando PrestaShop + MySQL .....</b>                    | 113 |
| <b>Instalando Joomla + MySQL .....</b>                        | 115 |
| <b>Instalando Reaction Ecommerce – NodeJS + MongoDB .....</b> | 116 |
| <b>Instalando Guacamole .....</b>                             | 118 |
| <b>Instalando Zabbix en Compose.....</b>                      | 119 |
| <b>Docker Registry .....</b>                                  | 121 |
| <b>Crea tu propio Docker Registry.....</b>                    | 121 |
| <b>Sube tus imágenes .....</b>                                | 121 |
| <b>Comparte las imágenes en tu red.....</b>                   | 123 |

## Introducción

Es una herramienta que permite desplegar aplicaciones en contenedores, de forma rápida y portable.

Permite generar aplicaciones de bolsillo, debido a que su arquitectura utiliza containers e imágenes, en estas últimas se definen toda la configuración, el software, las librerías y demás cosas que necesita la aplicación para funcionar y en un container lo vuelve realidad.

Las imágenes son muy portables y te permiten desplegar y escalar aplicaciones, como así también destruir y recrear imágenes de una manera muy fácil y rápido.

Las imágenes son como un estilo de snapshot.

## Arquitectura de Docker

Toda base se compone usando un Docker host (es el servidor físico/real donde se encuentra instalado Docker), el cual hace referencia a la casa/servidor donde se aloja el servicio de Docker.

Dentro del servidor vive el servidor de Docker, que se llama Docker Daemon, también tenemos una Rest API y un Docker CLI.

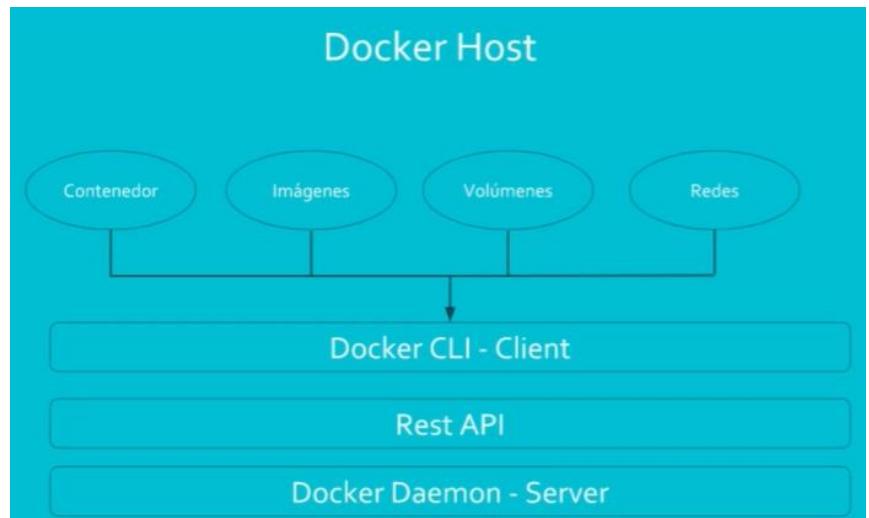
Su interacción es cuando se utiliza el CLI nos conectamos por medio de la API hacia el server, y por el medio del server se utiliza la API para contestarle al cliente, por lo que la misma funciona como canal de comunicación entre los otros dos componentes.

Docker client y Docker server viven en el mismo Docker host. Con el Docker client podemos manejar contenedores, imágenes, volúmenes y podemos manejar redes.

## ¿Qué es una imagen?

Las imágenes viven dentro del Docker Host, una imagen **es un paquete que contiene toda la configuración necesaria para que funcione el servicio**.

Las imágenes se componen por **N capas**.



Ejemplo:

- Capa 1 normalmente tiene un FROM, define que SO voy a utilizar.
- Capa 2 tenemos un RUN, define que va a haber luego del SO, por ejemplo, Apache.
- Capa 3 tenemos el CMD, es lo que va a ejecutar el servicio. Por ejemplo, un servicio que inicie lo que está en la capa 2.

Estas capas de la imágenes **son de solo lectura**.

## ¿Cómo creo las capas?

Estas capas se crean utilizando un archivo llamado **Dockerfile**, archivo de texto plano en el cual definimos las capas que queramos utilizar. Este es el archivo por default que el Docker va a buscar.



### Ejemplo de DockerFile:

FROM centos:7 //sistema operativo

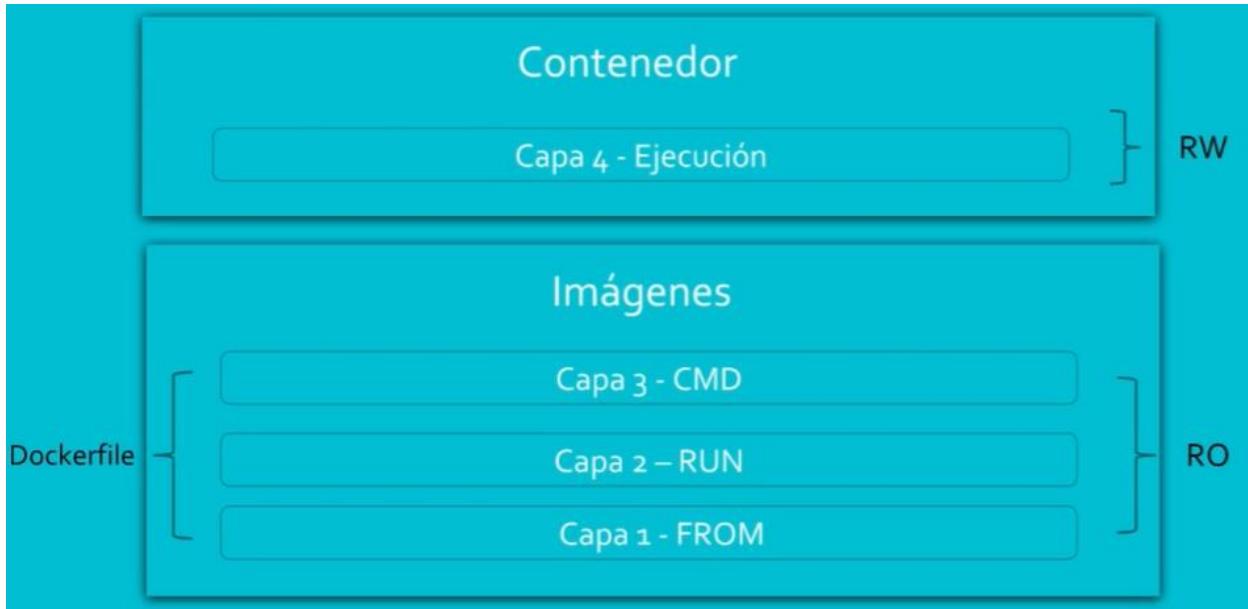
RUN yum -y install httpd //así se instala tipicamente apache en centos

CMD ["apachectl","-DFOREGROUND"] //va a iniciar el servicio de apache que instalamos en la capa anterior en el primer plano.

Es importante que el CMD este en primer plano para mantener vivo el contenedor de Docker.

## ¿Qué es un contenedor?

Un contenedor es una capa adicional que lo que hace es traer una ejecución de tiempo real de las capas de la imagen.



Siguiendo con el ejemplo anterior, la capa 4 de nuestra ejecución va a tener centos corriendo como un sistema base dentro del contenedor.

Va a tener un apache instalado y un CMD que el contenedor va a ejecutar para vivir la primera vez, mientras el CMD esté vivo, es decir mientras el output este en pantalla, el container va a vivir.

La capa 4 es de escritura, debido a que es una capa de ejecución. Si estamos en la capa 4 vamos a poder acceder a las capas anteriores con lectura y escritura, pero todos los cambios que hagamos en la capa 4 van a ser **temporales**, debido a que en realidad las capas de la imagen son de sólo lectura, y si modificamos algo no lo estamos modificando en la imagen, lo estamos modificando en la capa 4.

Si por ejemplo eliminamos apache en la capa 4, lo que va a suceder es que ese container no va a tener eso instalado, pero la imagen en sí va a seguir viva, va a seguir sin modificación porque es de solo lectura.

Un contenedor es una capa adicional a las 3 capas de las imágenes que es R/W y que es temporal, lo que quiere decir que podemos borrar la capa del contenedor y volver a crearla debido a que es **temporal**.

**No es bueno ni recomendable meter mano en la capa 4, debido a que toda su configuración va a estar viva temporalmente.**

Un contenedor tiene las 3 capas anteriores de las imágenes, volúmenes que sirven para tener persistencia en esta capa temporal. También tenemos dentro del contenedor Redes que permiten comunicar contenedores entre sí.

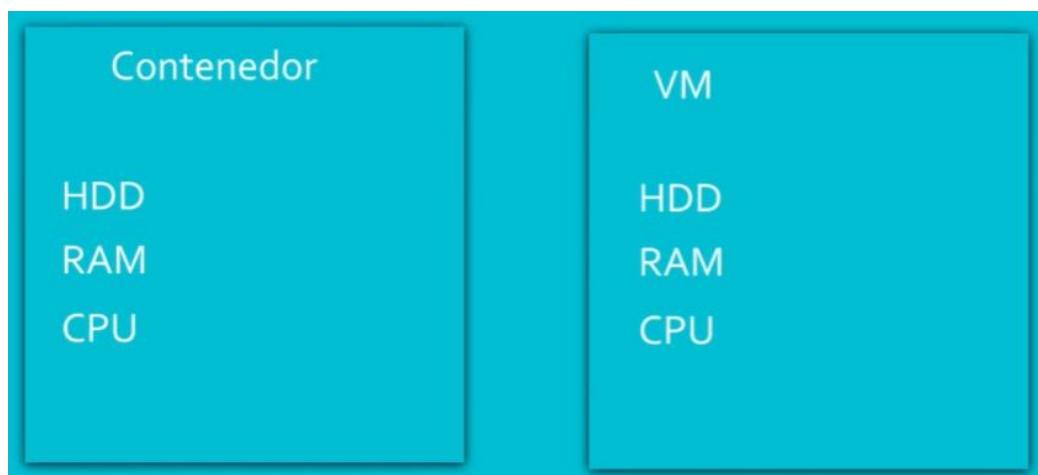
En resumen, una imagen es un paquete con las configuración y un contenedor es una capa adicional de ejecución que inicia todo lo que está definido en la imagen.



## Contenedores vs Máquinas Virtuales

Un contenedor es una instancia en ejecución de lo que es una imagen. El mismo es como un proceso más del sistema, por lo que va a utilizar la misma RAM, el mismo disco duro y la misma CPU del sistema.

Al ser un proceso va a consumir una mínima cantidad de RAM.



Para instalar una VM debemos instalar un Software virtualizador, la ISO del SO, crear un disco virtual, instalar el SO, agregarle RAM y CPU a la máquina. La desventaja de esto contra los containers es su increíble peso y todo los componentes que hay que instalar para manejar la compatibilidad si instalo por ejemplo Apache, debo instalar Ubuntu y todos los programas necesarios para que corran en dicha VM.

En cambio, un contenedor no es más que un proceso aislado, que no va a consumir más que un proceso común consumiría. La ventaja de flexibilidad y peso son destacables. A su vez los contenedores se pueden generar y eliminar con mucha facilidad dándole mucha flexibilidad y rapidez en su uso.

Otra de las ventajas de los contenedores es que yo podría tener un contenedor con Apache, otro PHP y otro con MySQL, por ejemplo. Y consumir mucho menos RAM que una VM.

## Docker Images

### Imágenes oficiales

**docker images:** lista todas las imágenes de Docker que tenemos en nuestro sistema.

En las **imágenes oficiales** (Ubuntu, apache, mongo) vienen todos los recursos que necesita un contenedor para funcionar.

Las imágenes se almacenan en Docker hub que es un repositorio público donde podemos subir nuestras imágenes.

Si realizamos un Docker pull mongo por ejemplo estaremos descargando la imagen de mongo de Docker hub a nuestra máquina local, por defecto si no le especificamos ningún tag se va a descargar la última imagen que mongo haya subido a docker hub.

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker>docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
f22ccc0b8772: Pull complete
3cf8fb62ba5f: Pull complete
e80c964ece6a: Pull complete
329e632c35b3: Pull complete
3e1bd1325a3d: Pull complete
4aa6e3d64a4a: Pull complete
035bca87b778: Pull complete
874e4e43cb00: Pull complete
08cb97662b8b: Pull complete
f623ce2ba1e1: Pull complete
f100ac278196: Pull complete
461b064aece5: Pull complete
Digest: sha256:00878f3d8e0a61997f2ea67351934b815a77c5ff8985df3ec041bca1c88258f4
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest
```

Si quisieramos descargar un tag posterior nos copiamos el nombre del mismo y lo definimos con el mismo comando docker pull "nombre de imagen": "tag versión"

Por ejemplo: docker pull mongo:4.2.11-bionic

(Como ya habíamos descargado mongo previamente Docker detecta que ya tiene varias capas que ya existen localmente, por lo que solo modifica las partes que son diferentes, esto se llama **Copy on Write**)

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker>docker pull mongo:4.2.11-bionic
4.2.11-bionic: Pulling from library/mongo
f22ccc0b8772: Already exists
3cf8fb62ba5f: Already exists
e80c964ecea6a: Already exists
329e632c35b3: Already exists
3e1bd1325a3d: Already exists
4aa6e3d64a4a: Already exists
035bca87b778: Already exists
441b974c007b: Pull complete
e8a6135ee562: Pull complete
20a5cbea1a51: Pull complete
9176a2e7af6: Pull complete
41ce2fbc2aea: Pull complete
Digest: sha256:d18fc373825e51d789a866841d4f4114270e9ee15cf96251f0b6fe9d72edaf4b
Status: Downloaded newer image for mongo:4.2.11-bionic
docker.io/library/mongo:4.2.11-bionic
```

Estos tags los obtenemos de la página oficial de docker hub. En caso de que la versión que queremos descargar no está disponible podríamos nosotros crear una imagen personalizada.

Utilizamos imágenes oficiales cuando ya existe una imagen con la que necesitamos, en caso contrario creamos una imagen personalizada.

Si hacemos un docker images | grep mongo, podemos observar que tenemos 2 imágenes de mongo que comparten capas que son iguales.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker images | grep mongo
mongo      latest          609301053242        42 hours ago     493MB
mongo      4.2.11-bionic    747c6655809e        42 hours ago     388MB
```

Si intentamos descargar nuevamente la imagen de mongo, lo que va a pasar es que nos va a aparecer un mensaje por consola avisándonos de que ya tenemos dicha imagen descargada y que la misma no presenta cambios para descargar.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
Digest: sha256:00878f3d8e0a61997f2ea67351934b815a77c5ff8985df3ec041bca1c88258f4
Status: Image is up to date for mongo:latest
docker.io/library/mongo:latest
```

Veamos otro ejemplo con mysql.

Descargamos la última imagen de mysql con el comando docker pull mysql:

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql
852e50cd189d: Pull complete
29969ddb0ffb: Pull complete
a43f41a44c48: Pull complete
5cdd802543a3: Pull complete
b79b040de953: Pull complete
938c64119969: Pull complete
7689ec51a0d9: Pull complete
a880ba7c411f: Pull complete
984f656ec6ca: Pull complete
9f497bce458a: Pull complete
b9940f97694b: Pull complete
2f069358dc96: Pull complete
Digest: sha256:4bb2e81a40e9d0d59bd8e3dc2ba5e1f2197696f6de39a91e90798dd27299b093
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest
```

¿Qué pasa si tengo dos imágenes que se llaman igual (ver caso de mongo) ?, se aplica un concepto de imágenes colgadas (dangling images), dejando a las imágenes que ya no debería usarse como huérfanas quitándole la referencia al tag:

|       |        |              |              |       |
|-------|--------|--------------|--------------|-------|
| mysql | latest | a8a59477268d | 3 weeks ago  | 445MB |
| mysql | 5.6    | e09f6de95634 | 4 weeks ago  | 256MB |
| mysql | <none> | 8d65ec712c69 | 5 weeks ago  | 445MB |
| mysql | 5.7    | f008d8ff927d | 4 months ago | 409MB |
| mysql | 5.7.20 | 7d83a47ab2d2 | 5 months ago | 408MB |

## Creando nuestra primer imagen

En primer instancia necesitamos crear un Dockerfile que es un archivo de texto normal, que se puede abrir con cualquier editor de texto.

La primera instrucción de un Dockerfile es un FROM que nos permite indicar que SO vamos a querer que contenga nuestras aplicaciones. Por lo que vamos a buscar una imagen oficial para un SO que queremos instalar en la imagen.

Por ejemplo, instalamos un centos con la siguiente línea

**FROM centos** -> esto va a descargar una imagen de la última versión de centos subida en docker hub

Luego debemos indicar en la sección de RUN que archivos, aplicaciones necesitamos en nuestra imagen, por ejemplo, apache, que se instala de la siguiente manera en centos:

**RUN yum install httpd**

Para construir nuestra imagen en base al Docker file que generamos, se debe ejecutar el comando docker build con un nombre de tag que indica el nombre de la imagen resultante.

**docker build - -tag <imagen>:<tag> <pathDockerfile>**

Ejemplo:

**docker build - -tag apache-centos .**

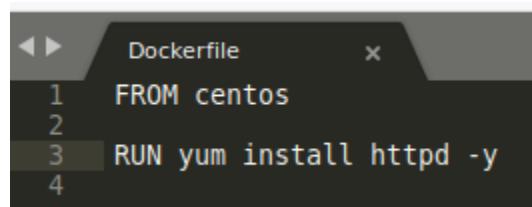
El proceso como tenemos el docker file actualmente vamos a empezar a correr los pasos que definimos en el docker file, por ejemplo:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build --tag apache-centos .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM centos
latest: Pulling from library/centos
3c72a8ed6814: Pull complete
Digest: sha256:76d24f3ba3317fa945743bb3746fbaf3a0b752f10b10376960de01da70685fb
Status: Downloaded newer image for centos:latest
--> 0d120b6ccaa8
```

La primer capa Step 1/2 instala la imagen de centos por medio de un docker pull centos.

Lo que sucede en el paso 2 es tratar de descargar apache, pero desde el SO de centos y aquí va a fallar porque todo lo que creamos en docker debe ser lo más automatizado posible debido a que docker file no debería tener interacción con nosotros sino simplemente ejecutar comandos, por lo que debemos incluir que aceptamos las preguntas yes or no dentro de nuestro docker file como que aceptamos la instalación.

Por ejemplo, nuestro Docker File debe quedar algo así:



```
1 FROM centos
2
3 RUN yum install httpd -y
4
```

Una vez ejecutado nuevamente el comando docker build, se descargarán los paquetes necesarios para instalar apache dando como resultado un complete successfully, listando todo lo instalado:

```
Installed:
  apr-1.6.3-11.el8.x86_64
  apr-util-1.6.1-6.el8.x86_64
  apr-util-bdb-1.6.1-6.el8.x86_64
  apr-util-openssl-1.6.1-6.el8.x86_64
  brotli-1.0.6-2.el8.x86_64
  centos-logos-httpd-80.5-2.el8.noarch
  httpd-2.4.37-30.module_el8.3.0+561+97fdbbcc.x86_64
  httpd-filesystem-2.4.37-30.module_el8.3.0+561+97fdbbcc.noarch
  httpd-tools-2.4.37-30.module_el8.3.0+561+97fdbbcc.x86_64
  mailcap-2.1.48-3.el8.noarch
  mod_http2-1.15.7-2.module_el8.3.0+477+498bb568.x86_64

Complete!
Removing intermediate container 4fc6d6ad312
--> 2701c7d6c483
Successfully built 2701c7d6c483
Successfully tagged apache-centos:latest
```

En Windows:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\docker-images>docker build --tag apache-centos .
[+] Building 34.0s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 78B
=> [internal] load .dockerrignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/centos:latest
=> [1/2] FROM docker.io/library/centos@sha256:76d24f3ba3317fa945743bb3746fbaf3a0b752f10b10376960de01da70685fb
=> => resolve docker.io/library/centos@sha256:76d24f3ba3317fa945743bb3746fbaf3a0b752f10b10376960de01da70685fb
=> => sha256:fc4a234b91cc4b542bac8a6ad23b2ddce60aae68fc4dbd4a52efbf10baad71 529B / 529B
=> => sha256:dd120b6ccaa8c5e1491767983501d4dd1885f961922497cd0abef155c869566 2.18KB / 2.18KB
=> => sha256:c372a8ed68140139e483fe37368ae4d9651422749e91483557cbd5ecf99a96110 74.87MB / 74.87MB
=> => sha256:76d24f3ba3317fa945743bb3746fbaf3a0b752f10b10376960de01da70685fb 762B / 762B
=> => extracting sha256:c372a8ed68140139e483fe37368ae4d9651422749e91483557cbd5ecf99a96110
=> [2/2] RUN yum install httpd -y
=> exporting to image
=> => exporting layers
=> => writing image sha256:878bb48fe662b5bd38571aa23272f05f4895e540a98b5ff75a12121a6eb6c39
=> => naming to docker.io/library/apache-centos
0.0s
0.05s
0.1s
0.05s
3.5s
6.9s
0.05s
0.05s
0.05s
0.05s
0.05s
0.05s
0.05s
4.3s
0.0s
2.1s
23.2s
0.3s
0.3s
0.05s
0.05s
```

Si revisamos si se creó correctamente la imagen mediante el comando docker images veremos que el tag que se le puso fue latest:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED       SIZE
apache-centos  latest        2701c7d6c483  11 minutes ago  255MB
```

Si queremos especificarle algún tag podemos construirla con un nombre del mismo poniendo luego del nombre de la imagen :<nombre del tag>. Docker maneja un tipo de cache, por lo que se da cuenta que ya está construida la imagen, pero va a realizar un tag de esta:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build --tag apache-centos:primera .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM centos
--> 0d120b6ccaa8
Step 2/2 : RUN yum install httpd -y
--> Using cache
--> 2701c7d6c483
Successfully built 2701c7d6c483
Successfully tagged apache-centos:primera
```

Si realizamos un docker images nuevamente veremos que tenemos 2 imágenes que en realidad son la misma imagen pero que tienen una etiqueta diferente pero que fueron creadas desde el mismo dockerfile:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker images | grep apache-centos
apache-centos      latest          2701c7d6c483    26 minutes ago   255MB
apache-centos      primera         2701c7d6c483    26 minutes ago   255MB
```

También podemos revisar el docker history para ver las capas que fueron creadas con el comando:

**docker history -H <image>:<tag>**

En nuestro ejemplo:

**docker history -H apache-centos:latest**

| IMAGE        | CREATED        | CREATED BY                                      | SIZE   | COMMENT |
|--------------|----------------|---|--------|---------|
| 2701c7d6c483 | 28 minutes ago | /bin/sh -c yum install httpd -y                 | 40.2MB |         |
| 0d120b6ccaa8 | 3 months ago   | /bin/sh -c #(nop) CMD ["/bin/bash"]             | 0B     |         |
| <missing>    | 3 months ago   | /bin/sh -c #(nop) LABEL org.label-schema.sc...  | 0B     |         |
| <missing>    | 3 months ago   | /bin/sh -c #(nop) ADD file:538afc0c5c964ce0d... | 215MB  |         |

Lo que podemos observar es los pasos de la creación de capas.

## Creando un contenedor con nuestra imagen

Vamos a crear un contenedor de la imagen de apache-centos que creamos en el paso anterior con el comando:

**docker run -d <nombre\_imagen>:<tag>**

Ejemplo:

**docker run -d apache-centos**

|  |  |
|--|--|
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker run -d apache-centos | 32dfd293ec5420c143835efa0515a0511e77842ecf06952df295328ac3835cb0 |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps                   |  |
| CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS  |  |
| NAMES  |  |

|   |              |               |             |               |                          |       |
|---|--------------|---------------|-------------|---------------|--------------------------|-------|
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps -a | 32dfd293ec54 | apache-centos | "/bin/bash" | 7 seconds ago | Exited (0) 6 seconds ago |       |
| Archivos  | ID           | IMAGE         | COMMAND     | CREATED       | STATUS                   | PORTS |
| NAMES   |              |               |             |               |                          |       |

Lo que va a suceder es que este contenedor se va a crear y morir en segundos, esto ocurre porque para crear un contenedor necesitamos la capa CMD para que se mantenga vivo, por lo que debemos definirlo debido a que si no tomaría el CMD de la imagen del SO.

Por lo que modificaríamos nuestro docker file agregando la capa de CMD de la siguiente manera:

### CMD apachectl -DFOREGROUND

Este comando lo que hace es ejecutar el servicio de apache en primer plano.

Lo que debemos hacer primero es construir nuestra imagen con ese cambio y luego crear un contenedor ya que si no se crearía un contenedor con las imágenes sin este comando y se destruiría al cabo de unos segundos.

Por lo que creamos nuestra nueva imagen con el comando

**docker build -t apache-centos:apache-cmd .**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t apache-centos:apache-cmd .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM centos
--> 0d120b6ccaa8
Step 2/3 : RUN yum install httpd -y
--> Using cache
--> 2701c7d6c483
Step 3/3 : CMD apachectl -DFOREGROUND
--> Running in 0a7206c5edad
Removing intermediate container 0a7206c5edad
--> 01d5d92c3edf
Successfully built 01d5d92c3edf
Successfully tagged apache-centos:apache-cmd
```

Ahora si creamos nuestro contenedor con esta imagen, veremos que al hacer un docker ps, el contenedor seguiría vivo:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker run -d --name apache apache-centos:apache-cmd
192a30d33ec366d4b973568d16b7670c02b1a196f8e971a412022cccefa4d36
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
192a30d33ec3        apache-centos:apache-cmd   "/bin/sh -c 'apachec..."   2 seconds ago      Up 2 seconds
          apache
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ █
```

Lo que vamos a hacer ahora es borrar el contenedor con el comando:

**docker rm -fv <nombre-contenedor>**

Ejemplo:

**docker rm -fv apache**

Para poder crear el mismo seteandole puertos de la siguiente manera:

**docker run -d --name apache -p 80:80 apache-centos:apache-cmd**

Lo que va a hacer esto es mapear nuestro puerto 80 de nuestra máquina con el puerto 80 del contenedor y podremos acceder mediante un browser a nuestra imagen para corroborar que esté funcionando correctamente.



## Dockerfile

### Introducción

Es un archivo donde definimos la configuración de una imagen, que recordando es un paquete que contiene aplicaciones necesarias para que funcione un servicio.

La imagen se crea a partir del Dockerfile el cual se divide en varias secciones con distintos argumentos:

- **FROM:** especificamos que SO queremos en nuestra imagen o incluso podemos especificar una imagen misma desde la que queramos comenzar.
- **RUN:** instrucciones que se pueden ejecutar desde la terminal, se puede ejecutar cualquier comando de Linux.
- **COPY/ADD:** utilizado para copiar archivos desde nuestra máquina hacia la imagen.
- **ENV:** variables de entornos.
- **WORKDIR:** directorios de trabajo.
- **EXPOSE:** sirve para exponer puertos.
- **LABEL**
- **USER**
- **VOLUME**
- **CMD** .dockerignore

### FROM / RUN / COPY / ADD

El argumento **COPY** se utiliza para copiar archivos de nuestra máquina local a la imagen, el mismo se utiliza de la siguiente manera:

COPY <nombreArchivoACopiar> destino

En el ejemplo de apache: COPY beryllium /var/www/html (document root de apache)

El argumento **ADD** se utiliza para agregar urls hacia una imagen, cualquier cosa que estuviese en internet se puede colocar la url como fuente. Lo que hace ADD es descargar el archivo de la URL y lo copia donde le indiquemos. En el caso de que sea un archivo local el ADD funciona como un COPY.

Comando:

ADD <url o archivo> destino

En el ejemplo de apache: ADD startbootstrap-freelancer-master /var/www/html

## **ENV / WORKDIR / EXPOSE**

**ENV** la utilizamos para agregar variables de entorno, que deberá utilizar nuestra imagen.

Ejemplo:

```
ENV contenido prueba  
RUN echo "$contenido" > /var/www/html/prueba.html
```

Lo que hace este argumento es declarar una variable de entorno llamada contenido y grabar su contenido propiamente dicho en este caso “prueba” en un archivo prueba.html

En el caso del argumento **WORKDIR** va a ser nuestro espacio de trabajo y podemos mediante el mismo posicionarnos en el directorio destino que le envíemos como parámetro, funciona parecido a un cd de la línea de comandos.

Ejemplo:

```
WORKDIR /var/www/html  
COPY <nameProject> .
```

El argumento **EXPOSE** lo que nos permite hacer es exponer un puerto distinto al que por defecto usaría el servidor, en este caso apache.

Ejemplo:

```
EXPOSE 8080
```

## **LABEL / USER / VOLUME**

**LABEL** es una etiqueta que puede ir en cualquier parte de la imagen, y sirve para dar metadata a la imagen. Cuando poseen espacios deben ir entre comillas.

Por ejemplo:

```
LABEL version =1.0
```

```
LABEL description = "This is an apache image"
```

Cuando hagamos un docker build si agregamos estos labels arriba de los atributos de RUN y CMD se va a recrear la imagen.

La directiva **USER** nos va a indicar que usuario está ejecutando la tarea en ese momento. Setea el usuario que va a estar en ejecución en ese momento.

Se puede setear por defecto un usuario por medio de:

```
RUN echo "$(whoami)" > destino
```

```
RUN useradd facundo
```

```
USER facundo
```

```
RUN echo "$(whoami)" > destino
```

```
USER root
```

La directiva **VOLUME** es una manera de colocar la data persistente dentro del contenedor para que cuando ese contenedor se elimine esa data siga viva dentro de nuestra máquina.

Ejemplo:

```
VOLUME /var/www/html
```

## CMD / dockerignore

El **CMD** es la directiva que mantiene vivo el contenedor, pero también puede utilizarse como un script.

Por ejemplo, se podría crear un script que mantenga vivo el contenedor y correrlo con la directiva CMD dentro del Dockerfile:

Run.sh

```
#!/bin/bash
echo "Iniciando container"
apachectl -DFOREGROUND
```

y en nuestro Dockerfile llamarlo de la siguiente manera:

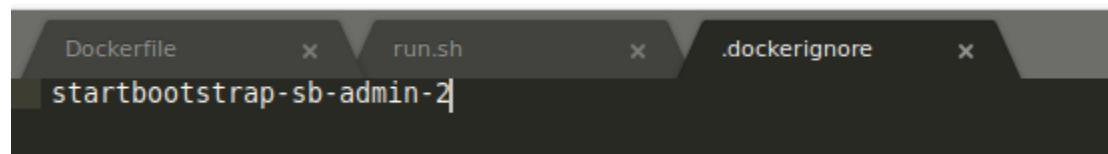
```
COPY run.sh /run.sh
CMD sh /run.sh
```

El **dockerignore** es un archivo normalmente oculto en el cual se indica que archivos se quieren ignorar para la construcción de la imagen, por ejemplo tenemos estos archivos en la ruta Docker/docker-images y si hacemos un docker build -t <nombreImagen> se va a construir la imagen con el peso total de todo lo contenido en ese directorio pero comprimido en una imagen.

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ du -shc *
2,6M    beryllium
4,0K    Dockerfile
4,0K    Dockerfile-OLD
4,0K    run.sh
5,8M    startbootstrap-freelancer-master
5,2M    startbootstrap-sb-admin-2
14M    total
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$
```

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t apache .
Sending build context to Docker daemon 13.2MB
Step 1/14 : FROM centos
--> 0d120b6ccaa8
Step 2/14 : LABEL version=1.0
--> Using cache
--> 248445caee72
Step 3/14 : LABEL description="This is an apache image"
-->
```

Si queremos por ejemplo ignorar uno de los archivos de dicho directorio porque no lo queremos incluir en nuestra imagen lo que se hace es poner el nombre del mismo en un archivo **.dockerignore**:



Y cuando construyamos nuevamente la imagen se va a excluir el mismo:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t apache .
Sending build context to Docker daemon 8.241MB
Step 1/14 : FROM centos
--> 0d120b6ccaa8
```

## Creando una imagen con todos los argumentos

Se puede crear una imagen con todas las instrucciones del Dockerfile vistas hasta ahora. No es necesario usar todas las instrucciones para generar una imagen sino solo las que consideremos necesarias, pero a fines prácticos vamos a crear un dockerfile con todas las directivas.

Ejemplo Dockerfile:

```
FROM nginx

RUN useradd facundo

COPY fruit /usr/share/nginx/html

ENV archivo docker

WORKDIR /usr/share/nginx/html

RUN echo "$archivo" > /usr/share/nginx/html/env.html

EXPOSE 90

LABEL version=1

USER facundo

RUN echo "Yo soy $(whoami)" > /tmp/yo.html

USER root

RUN cp /tmp/yo.html /usr/share/nginx/html/docker.html

VOLUME /var/log/nginx

CMD nginx -g 'daemon off;'
```

## Buenas prácticas

- La imagen o el servicio que está instalado debe ser efímero, es decir que se debe poder destruir con gran facilidad.
- Debería haber un solo servicio por contenedor o un solo servicio instalado por imagen.
- Si queremos excluir archivos que no queremos que estén en el contexto de Docker, cuando vayamos a construir la imagen es importante que agreguemos los mismos al dockerignore.
- Reducir el número de capas que tiene la imagen.
- Separar argumentos en multilínea para que sea más legible. ("\\")
- Varios argumentos en una sola capa.
- No instalar paquetes innecesarios.
- Uso de labels para aplicarle metadata a la imagen.

Ejemplo práctico:

Para reducir la cantidad de capas que tiene la imagen se puede hacer la misma instrucción en una sola línea concatenando los comandos en el caso de que sea posible, por ejemplo:

Tenemos una imagen con nginx que tiene 3 comandos RUN por lo que al construir tendríamos 4 capas:

### Dockerfile

```
FROM nginx
RUN echo "1" >> /usr/share/nginx/html/test.txt
RUN echo "2" >> /usr/share/nginx/html/test.txt
RUN echo "3" >> /usr/share/nginx/html/test.txt
```

### Docker build

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t nginx:bad .
Sending build context to Docker daemon 9.946MB
Step 1/4 : FROM nginx
--> bc9a0695f571
Step 2/4 : RUN echo "1" >> /usr/share/nginx/html/test.txt
--> Running in ae473b3cda92
Removing intermediate container ae473b3cda92
--> 5cac4df2d092
Step 3/4 : RUN echo "2" >> /usr/share/nginx/html/test.txt
--> Running in d064127fe087
Removing intermediate container d064127fe087
--> 6cfdd7f5a5d5
Step 4/4 : RUN echo "3" >> /usr/share/nginx/html/test.txt
--> Running in f6629291c7d7
Removing intermediate container f6629291c7d7
--> 6d341aab9be1
Successfully built 6d341aab9be1
Successfully tagged nginx:bad
```

Podemos reducir esa cantidad de capas si concatenamos en el Dockerfile las directivas RUN de la siguiente manera:

```
FROM nginx
RUN echo "1" >> /usr/share/nginx/html/test.txt && echo "2" >> /usr/share/nginx/html/test.txt && echo "3" >> /usr/share/nginx/html/test.txt
```

Dando como resultado en un docker build que se construyan 2 capas y no 4 como en el caso anterior.

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t nginx:bad .
Sending build context to Docker daemon 9.946MB
Step 1/2 : FROM nginx
--> bc9a0695f571
Step 2/2 : RUN echo "1" >> /usr/share/nginx/html/test.txt && echo "2" >> /usr/share/nginx/html/test.txt && echo "3" >> /usr/share/nginx/html/test.txt
--> Using cache
--> 9b7b73caeba8
Successfully built 9b7b73caeba8
Successfully tagged nginx:bad
```

Esto se puede mejorar aún más utilizando buenas prácticas separando las líneas por un escape ("\"), dejando el Dockerfile de la siguiente manera:

```
FROM nginx
RUN \
  echo "1" >> /usr/share/nginx/html/test.txt && \
  echo "2" >> /usr/share/nginx/html/test.txt && \
  echo "3" >> /usr/share/nginx/html/test.txt
```

De esta forma lo que se hace es mejorar el ámbito visual, todas las tareas quedan dentro de la misma capa, el backslash significa que continua todo en la misma línea.

Para mejorar el Dockerfile y organizarlo lo mejor posible podemos enviar todo el directorio a una variable de entorno y utilizar dicha variable:

### Dockerfile:

```
FROM nginx
ENV dir /usr/share/nginx/html/test.txt
RUN \
  echo "4" >> $dir && \
  echo "5" >> $dir && \
  echo "6" >> $dir
```

## Docker build:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t nginx:good .
Sending build context to Docker daemon 9.946MB
Step 1/3 : FROM nginx
--> bc9a0695f571
Step 2/3 : ENV dir /usr/share/nginx/html/test.txt
--> Running in 8a4ac039efbf
Removing intermediate container 8a4ac039efbf
--> 5925758d2caf
Step 3/3 : RUN echo "4" >> $dir &&     echo "5" >> $dir &&     echo "6" >> $dir
--> Running in 649d51e2a9cb
Removing intermediate container 649d51e2a9cb
--> 4941249f274f
Successfully built 4941249f274f
Successfully tagged nginx:good
```

## Construyendo una imagen Apache + PHP + TLS/SSL

```
Dockerfile      x
FROM centos:7

RUN \
    yum -y install httpd php php-cli php-common

RUN echo "<?php phpinfo(); ?>" > /var/www/html/hola.php

CMD apachectl -DFOREGROUND
```

El comando yum se encarga de instalar httpd y el php ya que soporta varios argumentos de entrada. Una vez corrido el docker build con su tag correspondiente yum también se encarga de instalar las dependencias necesarias que necesite la imagen.

## Agregando seguridad a nuestra imagen con SSL:

Openssl req -x509 -nodes -day 365 -newkey rsa:2048 -keyout **mysitename.key** -out **mysitename.crt**

En windows primero hay que configurar openssl en las variables de entorno del sistema.

Donde mysitename es el nombre de nuestra aplicación, cuando corremos el comando se nos harán un par de solicitudes de ingreso de datos para incorporar en el certificado de seguridad, el más importante es el parámetro Common Name donde se coloca el nombre del sitio de como se va a llamar, pero como estamos creando un sitio de prueba le pondremos localhost.

```
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'docker.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:
```

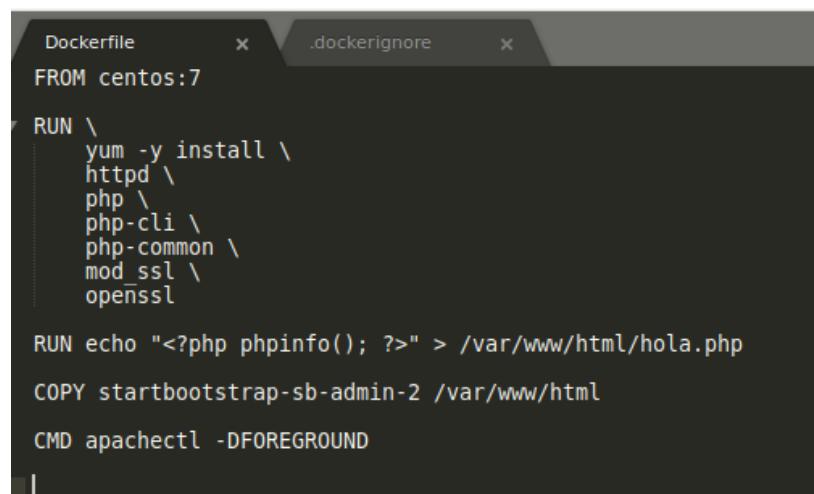
Este comando nos va a generar un par de archivos que son:

- Docker.crt
- Docker.key

Estos archivos son los certificados SSL que necesitamos instalar en nuestro web server.

En apache se instalan de la siguiente manera: [LINK](#)

Agregamos los dos paquetes que debemos instalar en nuestro Dockerfile y haciendo uso de buenas prácticas vamos a utilizar los slash multilínea:



The screenshot shows a terminal window with two tabs: 'Dockerfile' and '.dockerignore'. The Dockerfile tab contains the following content:

```
FROM centos:7
RUN \
    yum -y install \
        httpd \
        php \
        php-cli \
        php-common \
        mod_ssl \
        openssl
RUN echo "<?php phpinfo(); ?>" > /var/www/html/hola.php
COPY startbootstrap-sb-admin-2 /var/www/html
CMD apachectl -DFOREGROUND
```

The .dockerignore tab is empty.

Al correr docker build se instalarán los paquetes correspondientes y se minimizarán las capas gracias al uso del slash multilínea:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t apache:ssl .
Sending build context to Docker daemon 4.982MB
Step 1/5 : FROM centos:7
--> 8652b9f0cb4c
Step 2/5 : RUN yum -y install httpd php php-cli php-common mod_ssl openssl
--> Using cache
--> 05e7d9f51447
Step 3/5 : RUN echo "<?php phpinfo(); ?>" > /var/www/html/hola.php
--> Using cache
--> 1f9626622674
Step 4/5 : COPY startbootstrap-sb-admin-2 /var/www/html
--> Using cache
--> 41f120356cf8
Step 5/5 : CMD apachectl -DFOREGROUND
--> Using cache
--> a08846c510ff
Successfully built a08846c510ff
Successfully tagged apache:ssl
```

Para configurar una conexión segura por https y ssl debemos configurar un vhost para apache, su formato lo podemos obtener desde el siguiente [link](#), y sería algo así:

### ssl.conf

```
<VirtualHost *:443>
    ServerName localhost
    DocumentRoot /var/www/html
    SSLEngine on
    SSLCertificateFile /docker.crt
    SSLCertificateKeyFile /docker.key
</VirtualHost>
```

Archivo que luego incluimos en nuestro Dockerfile para poder construir la imagen, como así también la copia del certificado y key previamente generados hacia la imagen, debido a que si leemos la configuración del virtual host hace referencia a un archivo de certificado y un archivo de key que deben existir en la imagen.

Como el puerto SSL usado es el 443, debemos exponer nuestra imagen por dicho puerto por medio de la directiva EXPOSE.

Por lo que nuestro Dockerfile quedará de la siguiente forma armado:

```
FROM centos:7

RUN \
    yum -y install \
        httpd \
        php \
        php-cli \
        php-common \
        mod_ssl \
        openssl

RUN echo "<?php phpinfo(); ?>" > /var/www/html/hola.php

COPY startbootstrap-sb-admin-2 /var/www/html

COPY ssl.conf /etc/httpd/conf.d/default.conf

COPY docker.crt /docker.crt

COPY docker.key /docker.key

EXPOSE 443

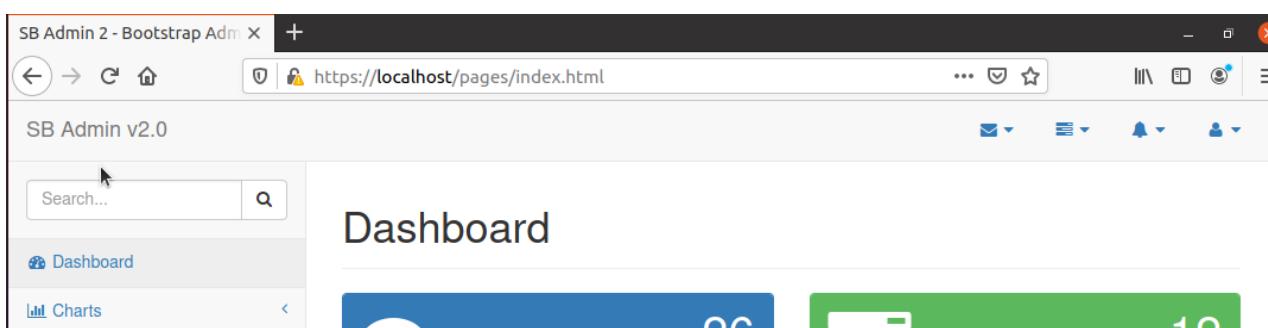
CMD apachectl -DFOREGROUND
```

Luego solo resta correr los comandos docker build y docker run para poder acceder a nuestro sitio de forma segura.

En este caso nuestro comando de docker run que veníamos utilizando debemos exponer el puerto seguro 443 y ya no el puerto 80, por lo que nuestro comando será de la siguiente manera:

**docker run -d -p 443:443 apache:ssl-ok**

Una vez levantado el contenedor, accedemos a nuestro localhost con conexión https y veríamos nuestro sitio levantado con conexión segura:



## Eliminar imágenes

Para poder listar las imágenes que tenemos en nuestro sistema ya sean creadas o descargadas desde docker hub, podemos utilizar el comando docker images:

| REPOSITORY | TAG     | IMAGE ID     | CREATED        | SIZE  |
|------------|---------|--------------|----------------|-------|
| apache     | ssl-ok  | 8b8e7f9ded06 | 26 minutes ago | 347MB |
| apache     | boot    | 100d583db1a5 | 7 hours ago    | 346MB |
| apache     | phpinfo | b415cbeaae47 | 7 hours ago    | 341MB |
| nginx      | good    | 4941249f274f | 7 hours ago    | 133MB |
| apache     | latest  | 9540c6ddfc5c | 5 days ago     | 261MB |
| nginx      | latest  | bc9a0695f571 | 2 weeks ago    | 133MB |
| centos     | 7       | 8652b9f0cb4c | 4 weeks ago    | 204MB |
| centos     | latest  | 0d120b6ccaa8 | 4 months ago   | 215MB |

Para eliminar alguna imagen que ya no queremos utilizar debemos utilizar el comando

**Docker rmi argumento**, donde argumento puede ser el id de la imagen o el nombre + el tag, por ejemplo queremos eliminar la imagen apache:boot, por lo que corremos el comando **docker rmi apache:boot** y si listamos nuevamente las imágenes ya no va a estar disponible la misma para su uso.

| REPOSITORY | TAG     | IMAGE ID     | CREATED        | SIZE  |
|------------|---------|--------------|----------------|-------|
| apache     | ssl-ok  | 8b8e7f9ded06 | 26 minutes ago | 347MB |
| apache     | boot    | 100d583db1a5 | 7 hours ago    | 346MB |
| apache     | phpinfo | b415cbeaae47 | 7 hours ago    | 341MB |
| nginx      | good    | 4941249f274f | 7 hours ago    | 133MB |
| apache     | latest  | 9540c6ddfc5c | 5 days ago     | 261MB |
| nginx      | latest  | bc9a0695f571 | 2 weeks ago    | 133MB |
| centos     | 7       | 8652b9f0cb4c | 4 weeks ago    | 204MB |
| centos     | latest  | 0d120b6ccaa8 | 4 months ago   | 215MB |

| REPOSITORY | TAG     | IMAGE ID     | CREATED        | SIZE  |
|------------|---------|--------------|----------------|-------|
| apache     | ssl-ok  | 8b8e7f9ded06 | 28 minutes ago | 347MB |
| apache     | phpinfo | b415cbeaae47 | 7 hours ago    | 341MB |
| nginx      | good    | 4941249f274f | 7 hours ago    | 133MB |
| apache     | latest  | 9540c6ddfc5c | 5 days ago     | 261MB |
| nginx      | latest  | bc9a0695f571 | 2 weeks ago    | 133MB |
| centos     | 7       | 8652b9f0cb4c | 4 weeks ago    | 204MB |
| centos     | latest  | 0d120b6ccaa8 | 4 months ago   | 215MB |

## Cambiar el nombre del Dockerfile

Para poder utilizar un Dockerfile con un nombre diferente, lo que se debe hacer es al construir la imagen con el comando docker build es agregarle un parámetro -f (flag) y a continuación el nombre del archivo dockerfile que queremos que se tome para la construcción de nuestra imagen, por ejemplo:

```
docker build -t test -f my-dockerfile .
```

```
my-dockerfile      x
FROM centos
RUN echo hola
```

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t test -f my-dockerfile .
Sending build context to Docker daemon 4.984MB
Step 1/2 : FROM centos
--> 0d120b6ccaa8
Step 2/2 : RUN echo hola
--> Running in 5641a9b91527
hola
Removing intermediate container 5641a9b91527
--> b3e93dd04a60
Successfully built b3e93dd04a60
Successfully tagged test:latest
```

## Dangling images

Una dangling image es una imagen huérfana o sin referenciar, la misma se genera cuando tenemos una imagen creada por medio del comando docker build tomando como referencia un Dockerfile, pero luego si modifíco alguna de las capas del dockerfile y se vuelve a construir la imagen con el mismo nombre y tag, lo que ocurrirá es que se dejara de referenciar a la antigua imagen y se comenzara a referenciar a la nueva dejando a la antigua como huérfana o sin referenciar:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker build -t test -f Dockerfile-DI .
Sending build context to Docker daemon 4.985MB
Step 1/4 : FROM centos
--> 0d120b6ccaa8
Step 2/4 : RUN echo "hola" > /tmp/hola && echo hola >> /tmp/hola1
--> Using cache
--> 70df875d5e67
Step 3/4 : RUN echo "bye" > /tmp/bye1
--> Running in 8b45c859414f
Removing intermediate container 8b45c859414f
--> e237bd59a228
Step 4/4 : RUN echo "test" > /tmp/test
--> Running in 2c589259d308
Removing intermediate container 2c589259d308
--> eaa944c76e4c
Successfully built eaa944c76e4c
Successfully tagged test:latest
```

| REPOSITORY  | TAG     | IMAGE ID     | CREATED            | SIZE  |
|-------------|---------|--------------|--------------------|-------|
| test        | latest  | eaa944c76e4c | About a minute ago | 215MB |
| <none>      | <none>  | 3b66cdebeb6e | 2 minutes ago      | 215MB |
| <none>      | <none>  | b3e93dd04a60 | 22 hours ago       | 215MB |
| apache      | ssl-ok  | 8b8e7f9ded06 | 22 hours ago       | 347MB |
| apache      | phpinfo | b415cbeeae47 | 29 hours ago       | 341MB |
| nginx       | good    | 4941249f274f | 29 hours ago       | 133MB |
| apache      | latest  | 9540c6ddfc5c | 6 days ago         | 261MB |
| nainx       | latest  | bc9a0695f571 | 2 weeks ago        | 133MB |
| c Rhythmbox | 7       | 8652b9f0cb4c | 4 weeks ago        | 204MB |
| centos      | latest  | 0d120b6ccaa8 | 4 months ago       | 215MB |

Esto sucede porque las capas de la imagen son de **SOLO LECTURA**, por lo tanto, las capas no pueden modificarse por lo que al modificar un Dockerfile, docker crea otra imagen totalmente nueva y le quita la referencia a la imagen anterior.

## ¿Cómo podríamos evitar este problema?

Definiendo tags en las imágenes:

| REPOSITORY | TAG     | IMAGE ID     | CREATED        | SIZE  |
|------------|---------|--------------|----------------|-------|
| test       | v1      | 70996c72fab2 | 2 minutes ago  | 215MB |
| <none>     | <none>  | bd2d254851fb | 3 minutes ago  | 215MB |
| test       | latest  | 98b7b2bde5b2 | 5 minutes ago  | 215MB |
| <none>     | <none>  | d6d9cccdeb44 | 10 minutes ago | 215MB |
| <none>     | <none>  | eaa944c76e4c | 23 minutes ago | 215MB |
| <none>     | <none>  | 3b66cdebeb6e | 24 minutes ago | 215MB |
| <none>     | <none>  | b3e93dd04a60 | 22 hours ago   | 215MB |
| apache     | ssl-ok  | 8b8e7f9ded06 | 23 hours ago   | 347MB |
| apache     | phpinfo | b415cbeeae47 | 29 hours ago   | 341MB |
| nginx      | good    | 4941249f274f | 30 hours ago   | 133MB |
| apache     | latest  | 9540c6ddfc5c | 6 days ago     | 261MB |
| nginx      | latest  | bc9a0695f571 | 2 weeks ago    | 133MB |
| centos     | 7       | 8652b9f0cb4c | 4 weeks ago    | 204MB |
| centos     | latest  | 0d120b6ccaa8 | 4 months ago   | 215MB |

| REPOSITORY | TAG    | IMAGE ID     | CREATED        | SIZE  |
|------------|--------|--------------|----------------|-------|
| test       | v1     | 70996c72fab2 | 12 seconds ago | 215MB |
| <none>     | <none> | bd2d254851fb | 46 seconds ago | 215MB |
| test       | latest | 98b7b2bde5b2 | 3 minutes ago  | 215MB |

Podríamos eliminar estas imágenes huérfanas mediante los siguiente comandos:

**docker images -f dangling = true**

**docker rmi <imagenIds a eliminar>**

También podemos listar solo los images ids y utilizar el comando xargs docker rmi:

**docker images -f dangling=true -q | xargs docker rmi**

Luego de eliminar todas las dangling images veremos que no existe ninguna imagen huérfana:

| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker images                  |         |              |                |       |
|---|---------|--------------|----------------|-------|
| REPOSITORY  | TAG     | IMAGE ID     | CREATED        | SIZE  |
| test  | v1      | 70996c72fab2 | 7 minutes ago  | 215MB |
| test  | latest  | 98b7b2bde5b2 | 10 minutes ago | 215MB |
| apache  | ssl-ok  | 8b8e7f9ded06 | 23 hours ago   | 347MB |
| apache  | phpinfo | b415cbeaae47 | 29 hours ago   | 341MB |
| nginx   | good    | 4941249f274f | 30 hours ago   | 133MB |
| apache  | latest  | 9540c6ddfc5c | 6 days ago     | 261MB |
| nginx   | latest  | bc9a0695f571 | 2 weeks ago    | 133MB |
| centos  | 7       | 8652b9f0cb4c | 4 weeks ago    | 204MB |
| centos  | latest  | 0d120b6ccaa8 | 4 months ago   | 215MB |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker images -f dangling=true |         |              |                |       |
| REPOSITORY  | TAG     | IMAGE ID     | CREATED        | SIZE  |

## Multi-Stage-Build

En las nuevas versiones de docker se nos permite utilizar varias veces la directiva FROM dentro del mismo Dockerfile para construir imágenes diferentes con temas de dependencias, por ejemplo, si quiero construir un jar desde una imagen Maven y luego quiero copiar ese jar hacia una imagen Java ahora lo puedo hacer en el mismo Dockerfile.

Lo que nos permite hacer el multi-stage-build es obviar las dependencias basuras que no necesitamos en la imagen para construir el jar.

Nuestro Dockerfile de ejemplo para construir un jar inicial y poder generar una imagen sería el siguiente:

```
FROM maven:3.5-alpine as builder

COPY app /app

RUN cd /app && mvn package

FROM openjdk:8-alpine

COPY --from=builder /app/target/my-app-1.0-SNAPSHOT.jar /opt/app.jar

CMD java -jar /opt/app.jar
```

Otro punto en donde podemos ver las ventajas del uso de multi-stage-build es el peso de la imagen, donde podemos reducir el tamaño del mismo utilizando esta estrategia.

Para explicar lo anterior con un ejercicio práctico, pondremos el siguiente Dockerfile de ejemplo:

```
FROM centos as test

RUN fallocate -l 10M /opt/file1

RUN fallocate -l 20M /opt/test2

RUN fallocate -l 30M /opt/test3

FROM alpine

COPY --from=test /opt/test2 /opt/myfile
```

En el mismo se puede ver el uso de la directiva FROM más de una vez lo que nos indica que estamos ante un multi-stage-build.

Si interpretamos un poco las líneas del Dockerfile veremos que en la líneas de RUN se están ejecutando fallocate que crea un archivo de texto con el peso que le pasemos de parámetro, por lo que si sumamos tenemos 60M creados en la imagen más el peso de centos que es algo así como 215MB por lo que en total deberíamos tener una imagen

de 275MB más el peso de la imagen de alpine, sin embargo cuando corremos un docker images en la consola veremos que la imagen que estamos creando pesa solamente 26.5MB, esto es debido a que el multi-stage-build no utiliza las dependencias que no necesitamos para crear nuestra imagen, entonces el peso resultante de la imagen sería el peso de alpine + el peso de la instrucción COPY que estamos realizando en el Dockerfile, en el ejemplo 20MB (/opt/test2)

| REPOSITORY | TAG    | IMAGE ID     | CREATED            | SIZE   |
|------------|--------|--------------|--------------------|--------|
| test       | latest | d6c54a38d591 | About a minute ago | 26.5MB |
| java       | maven  | fda1db0e774f | 2 hours ago        | 105MB  |
| alpine     | latest | 389fef711851 | 27 hours ago       | 5.58MB |

## Docker Containers

Son una instancia de ejecución de una imagen, que como ya vimos empaqueta todo lo que el contenedor necesita para funcionar, así que lo que hace el contenedor es traer a ejecución todo lo que definimos en la imagen.

Los contenedores son temporales por lo que si queremos que un cambio sea persistente debemos definirlo en el Dockerfile en la imagen. Nunca debemos hacer cambios en el contenedor ya que, si se elimina el contenedor, los cambios también se van a eliminar.

Las imágenes poseen capas de solo lectura, por lo que no podemos modificarlas, por lo que si queremos modificarlas lo que debemos hacer es generar una nueva imagen con los cambios nuevos. Los contenedores, al contrario, son instancias con una capa de lectura y escritura, por lo que podemos modificar, crear y eliminar archivos.

Otra ventaja que presentan los contenedores es que podemos crear varios de ellos partiendo desde una misma imagen.

### Listar / Mapear puertos

Para listar los contenedores activos utilizamos el comando:

**docker ps**

Y para listar todos los contenedores incluso los detenidos:

**docker ps -a**

Uno de los requisitos para crear un contenedor es tener una imagen.

Para crear un contenedor utilizaremos el comando **docker run** que recibe como argumentos varios parámetros que pasamos a explicar los más usuales.

**-d** -> Corre el contenedor en segundo plano.

Ejemplo:

**docker run -d jenkins**

Si listamos los contenedores activos veremos un contenedor corriendo con la imagen de Jenkins:

| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps |                 |                          |                |               |               |
|--|-----------------|--------------------------|----------------|---------------|---------------|
| CONTAINER ID   | IMAGE           | COMMAND                  | CREATED        | STATUS        | PORTS         |
| 8b98c9c31631   | jenkins/jenkins | "/sbin/tini -- /usr/..." | 18 seconds ago | Up 16 seconds | 8080/tcp, 500 |
| 00/tcp   | gallant_poitras |                          |                |               |               |

Vamos a analizar detenidamente la salida del comando docker ps:

- **Container ID:** secuencia de caracteres que actúan como identificador del contenedor.
- **Image:** imagen que se encuentra desplegada en el contenedor.
- **Command:** comando que ejecuta la imagen
- **Created:** tiempo que paso desde que fue creado.
- **Status:** estado del contenedor
- **Ports:** puertos que está exponiendo el contenedor
- **Names:** nombre del contenedor

Todos estos atributos son modificables desde el comando docker run.

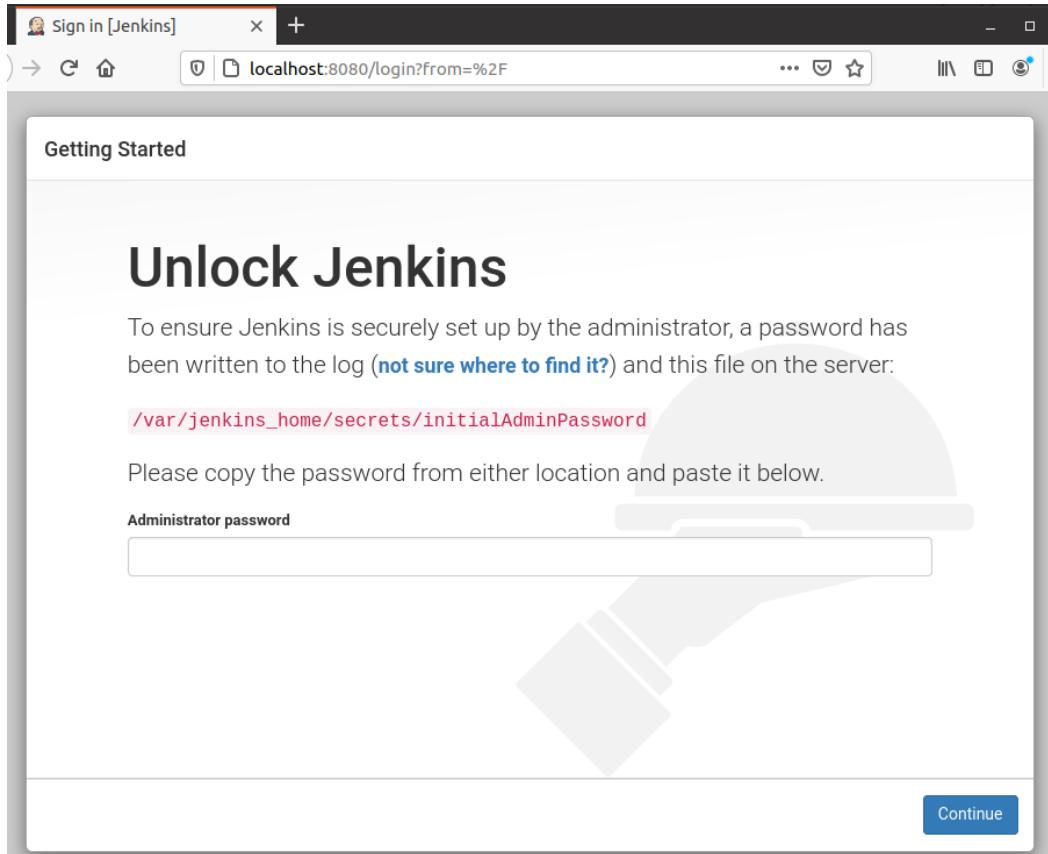
Para poder ver nuestra instancia en el navegador tenemos que hacer un **mapeo de puertos o ports mapping**.

Para poder hacer este mapeo se debe utilizar el comando docker run con el flag **-p** el cuál nos permite hacer ese mapeo.

Ejemplo:

**docker run -d -p 8080:8080 jenkins/jenkins**

Esto significa que estoy mapeando el puerto 8080 de mi maquina con el puerto 8080 del contenedor, por lo que si ahora accedo a mi browser como localhost:8080 voy a poder ver mi Jenkins funcionando.



Si observamos los contenedores activos veremos que el que acabamos de mapear tiene lo siguiente seteado en ports: **0.0.0.0:8080 -> 8080/tcp, 50000/tcp**

| CONTAINER ID | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS        | PORTS                             |
|--------------|-----------------|--------------------------|----------------|---------------|-----------------------------------|
| 75606ff6f12e | jenkins/jenkins | "/sbin/tini -- /usr/..." | 3 seconds ago  | Up 2 seconds  | 0.0.0.0:8080->8080/tcp, 50000/tcp |
| 8b98c9c31631 | jenkins/jenkins | "/sbin/tini -- /usr/..." | 23 minutes ago | Up 23 minutes | 8080/tcp, 50000/tcp               |

Lo que significa esto es que todas las interfaces de nuestra máquina en el puerto 8080 están siendo mapeadas del contenedor.

También podemos utilizar un puerto que no sea el estándar en nuestra máquina y mapearle el puerto 8080 del contenedor.

Por ejemplo:

```
docker run -d -p 9090:8080 jenkins/Jenkins
```

De esta manera podemos tener varios contenedores corriendo al mismo tiempo por distintos puertos, en este caso con el mismo servicio utilizando la misma imagen de docker sin ningún problema.

| CONTAINER ID          | IMAGE NAMES          | COMMAND                  | CREATED            | STATUS            | PORTS                             |
|-----------------------|----------------------|--------------------------|--------------------|-------------------|-----------------------------------|
| 6c1b0cb32d23          | jenkins/jenkins      | "/sbin/tini -- /usr/..." | About a minute ago | Up About a minute | 50000/tcp, 0                      |
| .0.0.0:9091->8080/tcp | beautiful_ardinghell |                          |                    |                   |                                   |
| 7c957ba8e0b1          | jenkins/jenkins      | "/sbin/tini -- /usr/..." | 10 minutes ago     | Up 10 minutes     | 50000/tcp, 0                      |
| .0.0.0:9090->8080/tcp | nervous_rhodes       |                          |                    |                   |                                   |
| 75606ff6f12e          | jenkins/jenkins      | "/sbin/tini -- /usr/..." | 16 minutes ago     | Up 16 minutes     | 0.0.0.0:8080->8080/tcp, 50000/tcp |
| 8b98c9c31631          | jenkins/jenkins      | "/sbin/tini -- /usr/..." | 40 minutes ago     | Up 40 minutes     | 8080/tcp, 5000/tcp                |
| 000/tcp               | gallant_poitras      |                          |                    |                   |                                   |

Si queremos borrar estos contenedores podemos utilizar el comando:

**docker rm -f <listNamesContainer>**

Y poder levantarlos nuevamente con un docker run en pocos segundos que es una de las facilidades que posee docker, y **esto es debido a que la configuración, dependencias y paquetes que necesitamos para las aplicaciones se encuentran empaquetadas en la imagen y el contenedor toma esa plantilla y va a ejecutar todo lo que hay dentro.**

## Iniciar / Reiniciar / Detener

Si por algún motivo nosotros queremos renombrar un contenedor que tenemos creado podemos hacer uso del comando:

**docker rename <oldname> <newname>**

| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps                            |                 |                          |                |               |                                   |
|---|-----------------|--------------------------|----------------|---------------|-----------------------------------|
| CONTAINER ID  | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS        | PORTS                             |
| 88cedd7a2a5b  | jenkins/jenkins | "/sbin/tini -- /usr/..." | 6 seconds ago  | Up 4 seconds  | 50000/tcp, 0.0.0.0:9091->8080/tcp |
| wizardly_brown  |                 |                          |                |               |                                   |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker rename wizardly_brown jenkins |                 |                          |                |               |                                   |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps                            |                 |                          |                |               |                                   |
| CONTAINER ID  | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS        | PORTS                             |
| 88cedd7a2a5b  | jenkins/jenkins | "/sbin/tini -- /usr/..." | 20 seconds ago | Up 19 seconds | 50000/tcp, 0.0.0.0:9091->8080/tcp |
| Rhythmbox   | jenkins         |                          |                |               |                                   |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$                                      |                 |                          |                |               |                                   |

Otra de las operaciones con contenedores que podemos realizar es detener el contenedor, pero no eliminarlo, para poder lograr esto debemos utilizar el comando:

**docker stop <containerID or containerName>**

| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps                |                 |                          |                |                             |                                   |
|---|-----------------|--------------------------|----------------|-----------------------------|-----------------------------------|
| CONTAINER ID  | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS                      | PORTS                             |
| 88cedd7a2a5b  | jenkins/jenkins | "/sbin/tini -- /usr/..." | 20 seconds ago | Up 19 seconds               | 50000/tcp, 0.0.0.0:9091->8080/tcp |
| jenkins   |                 |                          |                |                             |                                   |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker stop 88cedd7a2a5b |                 |                          |                |                             |                                   |
| 88cedd7a2a5b  |                 |                          |                |                             |                                   |
| CONTAINER ID  | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS                      | PORTS                             |
| fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images\$ docker ps                |                 |                          |                |                             |                                   |
| CONTAINER ID  | IMAGE NAMES     | COMMAND                  | CREATED        | STATUS                      | PORTS                             |
| TS  |                 |                          |                |                             |                                   |
| 88cedd7a2a5b  | jenkins/jenkins | "/sbin/tini -- /usr/..." | 2 minutes ago  | Exited (143) 17 seconds ago |                                   |
| jenkins   |                 |                          |                |                             |                                   |

Si queremos iniciar de nuevo debemos utilizar el comando:

**docker start <containerID or containerName>**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
88cedd7a2a5b      jenkins/jenkins     "/sbin/tini -- /usr..."   4 minutes ago       Exited (143) 2 minutes ago
jenkins
jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker start jenkins
jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
88cedd7a2a5b      jenkins/jenkins     "/sbin/tini -- /usr..."   4 minutes ago       Up 6 seconds       50000/tcp, 0.
0.0.0:9091->8080/tcp  jenkins
```

Si por alguna razón queremos reiniciar nuestro contenedor, porque se quedó colgado o está consumiendo mucha RAM, o simplemente está muy lento, se hace uso del comando:

**docker restart <containerID or containerName>**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
88cedd7a2a5b      jenkins/jenkins     "/sbin/tini -- /usr..."   6 minutes ago       Up 2 minutes       50000/tcp, 0.
0.0.0:9091->8080/tcp  jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker restart jenkins
jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
88cedd7a2a5b      jenkins/jenkins     "/sbin/tini -- /usr..."   6 minutes ago       Up 2 seconds       50000/tcp, 0.
0.0.0:9091->8080/tcp  jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$
```

Por el momento solo hemos visto comandos desde afuera del contenedor, ¿qué pasa si queremos ingresar dentro del contenedor?

La respuesta es mediante los sistemas operativos que instalamos desde la capa FROM de la imagen, por lo que tendríamos una Shell o una terminal.

Para poder acceder dentro del contenedor utilizaremos el comando:

**docker exec -ti <nombreContenedor> bash**

**-ti** -> terminal interactive

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
88cedd7a2a5b      jenkins/jenkins     "/sbin/tini -- /usr..."   16 minutes ago      Up 10 minutes      50000/tcp, 0.
0.0.0:9091->8080/tcp  jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -ti jenkins bash
jenkins@88cedd7a2a5b:/$
jenkins
jenkins@88cedd7a2a5b:/$
```

Podemos observar que el usuario es Jenkins y esta seguido por el id del contenedor que está actuando como hostname del contenedor:

```
jenkins@88cedd7a2a5b:/$ hostname  
88cedd7a2a5b  
jenkins@88cedd7a2a5b:/$
```

Para salir de la terminal del contenedor tipeamos **exit**.

También podemos acceder dentro del contenedor como un usuario root agregando al comando anterior el flag **-u username**:

Comando: **docker exec -u root -ti jenkins bash**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -u root -ti jenkins bash  
root@88cedd7a2a5b:/#
```

El entrar y salir del contenedor no lo afecta de ninguna manera.

Ahora ¿de qué me sirve entrar y salir del contenedor?

Nos es útil esto para poder ver cosas que estén corriendo dentro del contenedor, como, por ejemplo, un archivo, librería o lo que necesitemos observar.

Por ejemplo, si queremos desbloquear Jenkins nuestra aplicación localhost nos está diciendo que busquemos la password dentro del contenedor en un path específico.

## Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

Administrator password

Por lo que debemos acceder a nuestro contenedor para obtenerla con el comando ya visto previamente.

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -ti jenkins bash
jenkins@88cedd7a2a5b:/$ cat /var/jenkins_home/secrets/initialAdminPassword
4fe8fe97289148ee9e02cef7f917e157
jenkins@88cedd7a2a5b:/$
```

En el caso de que no podamos acceder al archivo con el usuario por defecto (definido en el Dockerfile), podemos acceder como un usuario root. Su uso es para determinadas acciones como por ejemplo dar permisos, ver archivos que no poseemos permisos con otro usuario o realizar acciones que están restringidas solo para este usuario.

Por ejemplo, puedo hacer el siguiente ejemplo práctico:

Se crea un archivo de texto con la palabra test en el path /tmp/test del contenedor

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -u root -ti jenkins bash
root@88cedd7a2a5b:/# echo "test" > /tmp/test
root@88cedd7a2a5b:/# chmod 400 /tmp/test
root@88cedd7a2a5b:/# cat /tmp/test
test
root@88cedd7a2a5b:/# exit
exit
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ █
```

Ahora si queremos ver el mismo con otro usuario que no sea el root, no vamos a poder accederlo por tema de permisos, pero si accedemos nuevamente como root si lo veríamos:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -ti jenkins bash
jenkins@88cedd7a2a5b:/$ cat /tmp/test
cat: /tmp/test: Permission denied
jenkins@88cedd7a2a5b:/$ whoami
jenkins
jenkins@88cedd7a2a5b:/$ exit
exit
fmediotte@fmediotte-Virtual-Machine:~/Docker/docker-images$ docker exec -u root -ti jenkins bash
root@88cedd7a2a5b:/# cat /tmp/test
test
root@88cedd7a2a5b:/# █
```

## Crear un contenedor MySQL

### Requisitos previos para testear MySQL

Dentro de nuestros sistemas operativos que se encuentran corriendo dentro de nuestro contenedor no tenemos instalado el MySQL client para poder hacer uso de él, por lo que debemos acceder a nuestro contenedor como root e instalarlo mediante un comando de instalación.

Centos:

**yum install mysql -y**

Ubuntu:

**apt-get update**

**apt-get install mysql-client -y**

### Creando el contenedor

El primer paso es acceder al [Docker hub](#) y descargar una imagen de mysql con el comando **docker pull mysql**.

Para levantar el contenedor debemos seguir las [instrucciones](#) de docker hub donde explica como iniciar un servicio de mysql:

Donde se hace uso del comando:

**docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret -d mysql:tag**

Donde some-mysql es el nombre que queramos darle al contenedor y Mysql\_root\_password es una variable de entorno.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d --name my-db1 -e "MYSQL_ROOT_PASSWORD=1234" mysql:5.7
0dc6533245a9fa9e2d7a7b103707d49bc3024951e7837854bac4e3f94c353635
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES              NAMES
0dc6533245a9        mysql:5.7          "docker-entrypoint.s..."   4 seconds ago      Up 2 seconds       3306/tcp, 330
60/tcp   my-db1
```

Para leer los logs del contenedor y ver si nuestro motor de mysql ya está listo para aceptar conexiones se hace uso del comando:

**docker logs -f <nombreContenedor>**

Para conectarnos sin definirle un mapeo de puertos lo que debemos hacer es utilizar el comando **docker inspect < nombreContenedor >** y obtener la IP del contenedor.

```
"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "3d7c85b03fc1e22edc30be57c0e854b03f8b289d4c738d1c79cf807c549e470",
        "EndpointID": "eff23034ba6fe5a9c7c89885bd5288eb41a3fd0248984a07839b1d75ae25264b",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:03",
        "DriverOpts": null
    }
}
```

Una vez obtenida debemos conectarnos a MySQL con el comando:

```
mysql -u root -h <ipContenedor> -p<password>
```

```
fmediotte@fmediotte-Virtual-Machine:~$ mysql -u root -h 172.17.0.3 -p1234
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.32 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> ■
```

### ¿Cómo hacemos para poder correr este mysql pero en localhost?

Al crear el contenedor realizamos el mapeo de puerto agregando también variables de entorno para poder crear una base de datos cuando se esté iniciando el contenedor.

```
docker run -d -p 3333:3306 //mapeo de puertos
--name some-mysql //nombre del contenedor
-e "MYSQL_ROOT_PASSWORD=my-secret"
-e "MYSQL_DATABASE=docker-db"
-e "MYSQL_USER=docker-user"
-e "MYSQL_PASSWORD=1234"
mysql:tag
```

Las variables de entorno las obtenemos desde la página oficial de MySQL en [docker hub](#).

Una vez arriba el contenedor debemos conectarnos desde nuestra máquina local al MySQL levantado en el contenedor por medio del comando:

```
mysql -u root -h <localhost> -p<password> --port 3333
```

Donde localhost sería en mi caso 127.0.0.1 y port el puerto seteado en la creación del contenedor.

Una vez conectados podemos ver si se creó la base de datos que pusimos en la creación del contenedor con el comando show databases; de MySQL:

```
fmediotte@fmediotte-Virtual-Machine:~$ mysql -u root -p1234 -h 127.0.0.1 --port 3333
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.32 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| docker-db      |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.00 sec)

mysql> █
```

Adicionalmente en las variables de entorno definimos un usuario y contraseña para conectarnos a MySQL por lo que procedemos a probar los mismos en el comando de conexión a la BD.

**mysql -u <user> -h <localhost> -p<password> --port 3333**

```
fmediotte@fmediotte-Virtual-Machine:~$ mysql -u docker-user -p1234 -h 127.0.0.1 --port 3333
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.32 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| docker-db      |
+-----+
2 rows in set (0.00 sec)

mysql> █
```

Observamos que vemos menos instancias de bases de datos esto es debido porque no estamos conectados con el usuario root que ve bases de datos del sistema.

**Tip:** otro comando para eliminar todos los contenedores es: **docker rm -fv \$(docker ps -aq)**

## Variables de entorno

Una variables de entorno es aquella la cual podemos acceder desde cualquier parte del contenedor.

Las podemos definir en 2 lugares:

- En el Dockerfile
- Al crear el contenedor

**TIP:** ¿Qué pasa si tenemos muchos contenedores corriendo al mismo tiempo y quiero dejar de utilizarlos y borrarlos todos?

Para estos casos podemos hacer uso del comando:

**docker rm -f <containerIDs>** pero concatenandolo de una manera que se borren sin copiar todos los ids, por ejemplo, con el comando

**docker ps -q** listamos todos los containersIDs y lo podemos pasar como parámetro al un segundo comando que elimine los contenedores que liste, como por ejemplo en Linux:

**docker ps -q | xargs docker rm -f**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
12630a33039c        jenkins/jenkins    "/sbin/tini -- /usr..."   32 seconds ago    Up 30 seconds      0.0.0.0:8080-
>8080/tcp, 50000/tcp   affectionate_brown
88cedd7a2a5b        jenkins/jenkins    "/sbin/tini -- /usr..."   56 minutes ago   Up 50 minutes     50000/tcp, 0.
0.0.0:9091->8080/tcp   jenkins
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps -q
12630a33039c
88cedd7a2a5b
f Rhythmbox fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps -q | xargs docker rm -f
12630a33039c
88cedd7a2a5b
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
```

Volviendo al tema de las definiciones de las variables de entorno habíamos dicho que se podía definir en el Dockerfile, por ejemplo:

```
FROM centos:7
```

```
ENV prueba 1234
```

```
RUN useradd facundo
```

Si construimos una imagen en base a este Dockerfile, un contenedor con dicha imagen y accedemos al sistema operativo del contenedor por medio de una terminal podemos observar que la variable de entorno posee el valor 1234:

(Creamos el docker run con el flag -dti ya que es una imagen de un SO)

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker run -dti --name env env  
361c3a8aa98b311d2d288cdced0571a392aae14eedab6bfec21d3299591084d8  
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS  
NAMES  
361c3a8aa98b env "/bin/bash" 2 seconds ago Up 2 seconds
```

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker exec -ti env bash  
[root@8bc39c061f37 /]# echo $prueba  
1234  
[root@8bc39c061f37 /]#
```

También se pueden crear variables de entorno al crear el contenedor agregando el flag **-e** que sirve para crear variables de entorno, ejemplo:

```
docker run -dti -e "prueba1=4321" --name env2 env
```

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker run -dti -e "prueba1=4321" --name env2 env  
f74eec1cea34cdef16b9db46b48edac85177a344f4ee82ece9cf92c2277d294  
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS  
NAMES  
f74eec1cea3 env2 "/bin/bash" 2 seconds ago Up 1 second  
361c3a8aa98b env "/bin/bash" 4 minutes ago Up 4 minutes
```

Si ahora nos metemos dentro del contenedor y listamos las variables de entorno del sistema veremos nuestra variable prueba1 definida con el valor pasado por parámetro y nueva variable de entorno creada desde el Dockerfile.

```
[root@f748eec1cea3 /]# env
HOSTNAME=f748eec1cea3
TERM=xterm
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:01:or=40;31:01:mi=01;05;37;41:s
u=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.arc=01;31:*.arj=01;31:*.taz=01
;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=0
1;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*
*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;3
1:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01
;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg
=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:
*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35
:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;3
5:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=01;36:*.au=01;36:*.flac=01;36:*.mid=01
;36:*.midi=01;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;36:*.oga=01;36:*.spx=
01;36:*.xspf=01;36:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
prueba=1234
prueba1=4321
_=~/usr/bin/env
[root@f748eec1cea3 /]#
```

## Crear un contenedor Mongo

Para construir un contenedor con Mongo, vamos a descargar la imagen oficial de Mongo del repositorio de docker hub, accediendo al siguiente [link](#) y realizando un docker pull.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
f22ccc0b8772: Pull complete
3cf8fb62ba5f: Pull complete
e80c964ece6a: Pull complete
329e632c35b3: Pull complete
3e1bd1325a3d: Pull complete
4aa6e3d64a4a: Pull complete
035bca87b778: Pull complete
874e4e43cb00: Pull complete
08cb97662b8b: Pull complete
f623ce2ba1e1: Pull complete
f100ac278196: Pull complete
6f5539f9b3ee: Pull complete
Digest: sha256:02e9941ddcb949424fa4eb01f9d235da91a5b7b64feb5887eab77e1ef84a3bad
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest
```

Para crear el contenedor con la imagen oficial de mongo utilizamos el comando **docker run** de la siguiente manera:

```
docker run -d --name my-mongo -p 27017:27017 mongo
```

(por defecto el Puerto que utiliza mongo es 27017, esto lo podemos consultar en docker hub)

```
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d --name my-mongo -p 27017:27017 mongo
8f6c92ad736e80942bc75ce82815a75b7519aa00968b195d2b32284ed276edce
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
8f6c92ad736e        mongo              "docker-entrypoint.s..."   10 seconds ago    Up 9 seconds       0.0.0.0:27017>27017/tcp
my-mongo
fmediotte@fmediotte-Virtual-Machine:~$
```

**Tip:** Con el comando docker stats <nombreContenedor> podemos ver cuando memoria y cpu está consumiendo nuestro contenedor:

| CONTAINER ID | NAME      | CPU % | MEM USAGE / LIMIT   | MEM % | NET I/O     |
|--------------|-----------|-------|---------------------|-------|-------------|
| BLOCK I/O    | PIDS      |       |                     |       |             |
| b77fb592f6e5 | my-mongo2 | 0.49% | 57.04MiB / 3.843GiB | 1.45% | 3.22kB / 0B |
| 0B / 438kB   | 32        |       |                     |       |             |

La cantidad de memoria ram y cpu que está usando se puede limitar lo cual veremos en otra sección.

Si levantamos otro contenedor, pero ahora mapeándole el puerto 27018 de la siguiente forma:

```
docker run -d --name my-mongo2 -p 27018:27017 mongo
```

| CONTAINER ID | IMAGE | COMMAND                  | CREATED       | STATUS       | PORTS                    | NAMES     |
|--------------|-------|--------------------------|---------------|--------------|--------------------------|-----------|
| 906282c797fe | mongo | "docker-entrypoint.s..." | 5 minutes ago | Up 5 minutes | 0.0.0.0:27018->27017/tcp | my-mongo2 |
| 6ad6e4b1a85f | mongo | "docker-entrypoint.s..." | 8 minutes ago | Up 8 minutes | 0.0.0.0:27017->27017/tcp | my-mongo  |

Y si nos levantamos un mongo gui client como por ejemplo Mongo Compass o Robo 3T, y realizamos la conexión hacia esos puertos (27017 o 27018) veremos que tenemos instancias de mongo para utilizar

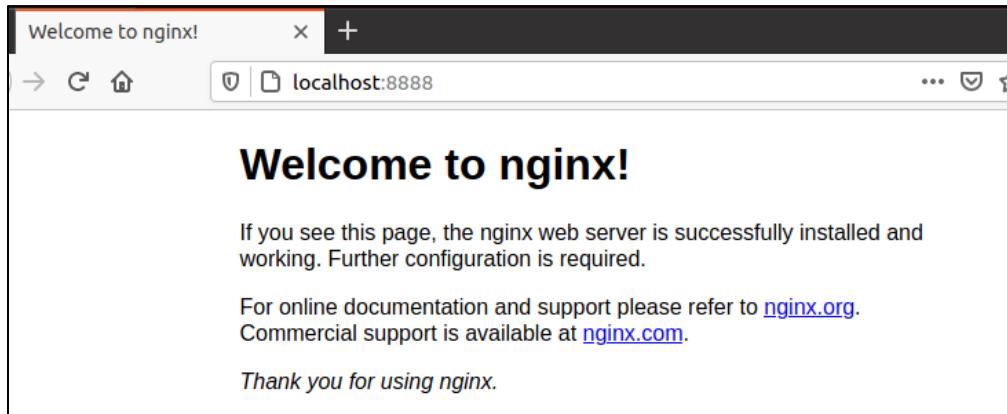
The screenshot shows the MongoDB Compass interface. On the left, there is a configuration panel with fields for Hostname (localhost), Port (27018), SRV Record (disabled), and Authentication (None). A green 'CONNECT' button is at the bottom right of this panel. To the right, the main interface displays the 'Local' database. It shows cluster information: HOST localhost:27018, CLUSTER Standalone, and EDITION MongoDB 4.4.2 Community. Below this, a search bar says 'Filter your data' and lists three databases: admin, config, and local.

## Crear un contenedor Apache / Nginx / Tomcat

Buscamos la imagen oficial de nginx en [docker hub](#) y hacemos un docker pull en nuestra máquina.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
6ec7b7d162b2: Already exists
cb420a90068e: Pull complete
2766c0bf2b07: Pull complete
e05167b6a99d: Pull complete
70ac9d795e79: Pull complete
Digest: sha256:4cf620a5c81390ee209398ecc18e5fb9dd0f5155cd82adcbae532fec94006fb9
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d -p 8888:80 nginx
9df3df59d89e3447a3e935ac28d7f16d21b3f27bd6f95084ae7a89d18e02bdbd
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
9df3df59d89e        nginx              "/docker-entrypoint...."   2 seconds ago      Up 1 second       0.0.0.0:8888->80/tcp
fmediotte@fmediotte-Virtual-Machine:~$
```

Si accedemos a localhost:8888 en nuestro browser veremos nginx levantado.



Ahora ¿cómo creamos un Apache sin eliminar nuestro nginx?

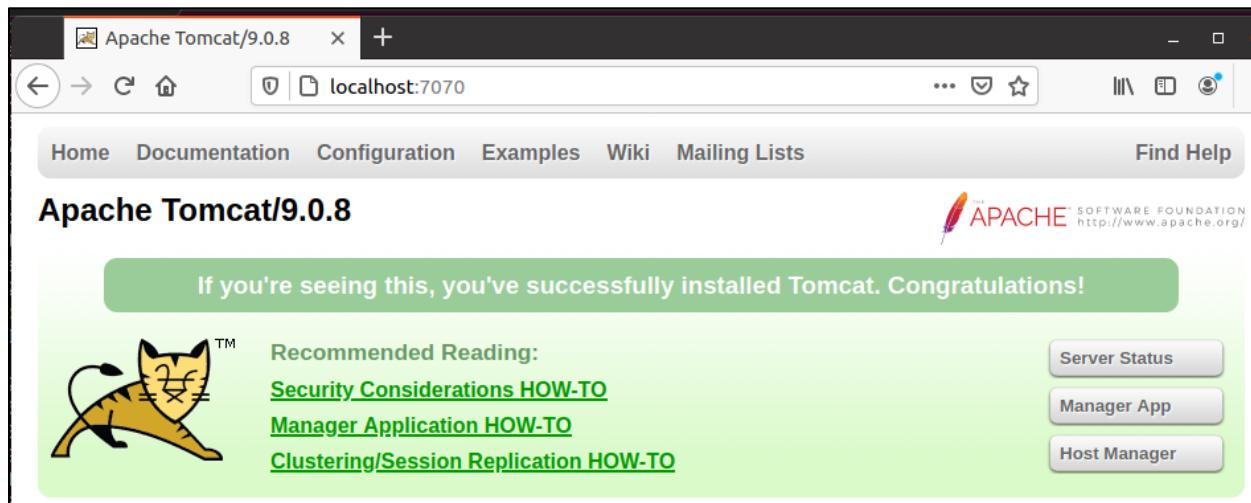
Realizamos un `docker run -d -p <otropuerto>:80 --name apache httpd`

```
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d -p 9999:80 --name apache httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
6ec7b7d162b2: Already exists
17e233bac21e: Pull complete
130aad5bf43a: Pull complete
81d0a34533d4: Pull complete
da240d12a8a4: Pull complete
Digest: sha256:a3a2886ec2500194804974932eaf4a4ba2b77c4e7d551ddb63b01068bf70f4120
Status: Downloaded newer image for httpd:latest
953373fdd024f0bc52c43ddda7666f38c7719c4987d6a6f8ead9d5b195badcf
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
953373fdd024        httpd              "httpd-foreground"   9 seconds ago      Up 7 seconds       0.0.0.0:9999->80/tcp
>80/tcp apache      nginx              "/dock...r-entrypoint...."  7 minutes ago     Up 7 minutes       0.0.0.0:8888->80/tcp
fmediotte@fmediotte-Virtual-Machine:~$
```

Si accedemos a localhost, pero al puerto 9999 veríamos apache levantado y corriendo.

Ahora creamos un contenedor que contenga la [imagen oficial de tomcat](#) realizando un **docker run -d -p <otropuesto>:8080 tomcat**

```
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d -p 7070:8080 --name my-tomcat tomcat:9.0.8-jre8-alpine
Unable to find image 'tomcat:9.0.8-jre8-alpine' locally
9.0.8-jre8-alpine: Pulling from library/tomcat
ff3a5c916c92: Pull complete
a8906544047d: Pull complete
590b87a38029: Pull complete
844a026fab6d: Pull complete
11818b451ba0: Pull complete
6a1b6c87412e: Pull complete
Digest: sha256:c9d8c2e06404f97b9dff7c4b80f9a41392fb73398d44051bca54878fdc2293eb
Status: Downloaded newer image for tomcat:9.0.8-jre8-alpine
9b73c25d6cb8caa628e49f0d76e804843b78299a70de07b1dfcf239cc8cb722e
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
9b73c25d6cb8        tomcat:9.0.8-jre8-alpine   "catalina.sh run"   11 seconds ago    Up 9 seconds       0.0.0.
0:7070->8080/tcp   my-tomcat
953373fdd024        httpd
0:9999->80/tcp     apache
9df3df59d89e        nginx
0:8888->80/tcp     nginx
fmediotte@fmediotte-Virtual-Machine:~$
```



De esta manera creamos contenedores de Apache, Nginx y Tomcat.

## Crear un contenedor PostgreSQL

Buscamos la [imagen oficial de Postgres](#) y realizamos un docker pull como se indica en la página oficial del docker hub.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
Digest: sha256:c846f37f65fd0d3d8b43040df5ebdc5856a038415b019ba596864848fb717a8b
Status: Image is up to date for postgres:latest
docker.io/library/postgres:latest
```

Y levantamos un contenedor con distintas variables de entorno necesarias para la configuración de la base de datos:

```
Docker run -d --name postgres
-e "POSTGRES_PASSWORD=1234"
-e "POSTGRES_USER=docker"
-e "POSTGRES_DB=docker-db"
-p 5432:5432
postgres
```

Estas variables de entorno se encuentran en la documentación oficial.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker run -d --name postgres -e "POSTGRES_PASSWORD=1234" -e "POSTGRES_USER=docker" -e "POSTGRES_DB=docker-db" -p 5432:5432 postgres
f388614e6b2a40c499ba0189e5b16fbcd1cef3d257768c743d8714902eba090
fmediotte@fmediotte-Virtual-Machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
f388614e6b2a        postgres            "docker-entrypoint.s..."   4 seconds ago      Up 3 seconds       0.0.0.0:5432->5432/tcp
postgres
```

Para comprobar que quedo todo bien creado el user y la base de datos podemos acceder al contenedor con el comando ya visto:

```
docker exec -ti <nombrecontenedor> bash
```

Y loguearnos a postgres con los parámetros de la base de datos y el usuario definido en la creación del container.

Una vez logueados con el comando:

```
psql -d docker-db -U docker
```

Ya nos encontraremos dentro del motor postgres y podríamos listar las base de datos como ejemplo.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker exec -ti postgres bash
root@f388614e6b2a:/# psql -d docker-db -U docker
psql (13.1 (Debian 13.1-1.pgdg100+1))
Type "help" for help.

docker-db=# \l
          List of databases
   Name    | Owner | Encoding | Collate | Ctype | Access privileges
-----+-----+-----+-----+-----+-----+
 docker-db | docker | UTF8     | en_US.utf8 | en_US.utf8 |
 Rhythmbox | docker | UTF8     | en_US.utf8 | en_US.utf8 |
 template0 | docker | UTF8     | en_US.utf8 | en_US.utf8 | =c/docker      +
 template1 | docker | UTF8     | en_US.utf8 | en_US.utf8 | =c/docker      +
               |       |           |           |       | docker=CTc/docker
(4 rows)

docker-db=#
```

## Crear un contenedor Jenkins

Buscamos la [imagen oficial de Jenkins](#) y realizamos un docker pull como se indica en la página oficial del docker hub.

```
fmediotte@fmediotte-Virtual-Machine:~$ docker pull jenkins/jenkins
Using default tag: latest
latest: Pulling from jenkins/jenkins
3192219af04: Pull complete
17c160265e75: Pull complete
cc4fe40d0e61: Pull complete
9d647f502a07: Pull complete
d108b8c498aa: Pull complete
1bfe918b8aa5: Pull complete
dafa1a7c0751: Pull complete
a10933ea2f2f: Pull complete
dacec5718df4: Pull complete
0cd1192f374e: Pull complete
bac0875b818f: Pull complete
fc7126ecc5e0: Pull complete
65580027eff4: Pull complete
ea01a82194c6: Pull complete
be9da8492eef: Pull complete
8351e2eec838: Pull complete
909eab257f21: Pull complete
9107ef57f0b5: Pull complete
f925e67af94f: Pull complete
54b9422507ad: Pull complete
Digest: sha256:7648cdca09867d87462a82e6a2aac39942bd2c6deb687da2025cae3f928966cb
Status: Downloaded newer image for jenkins/jenkins:latest
docker.io/jenkins/jenkins:latest
```

Una vez descargada la imagen levantamos un contenedor con dicha imagen y mapearle un puerto que tengamos disponible, en el caso del ejemplo 7070:

```
docker run -d -p 7070:8080 --name jenkins jenkins
```

Una vez arriba el contenedor nos solicitará que ingresemos la password para desbloquear el servicio y nos informa la ruta de donde obtener desde el contenedor.

# Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

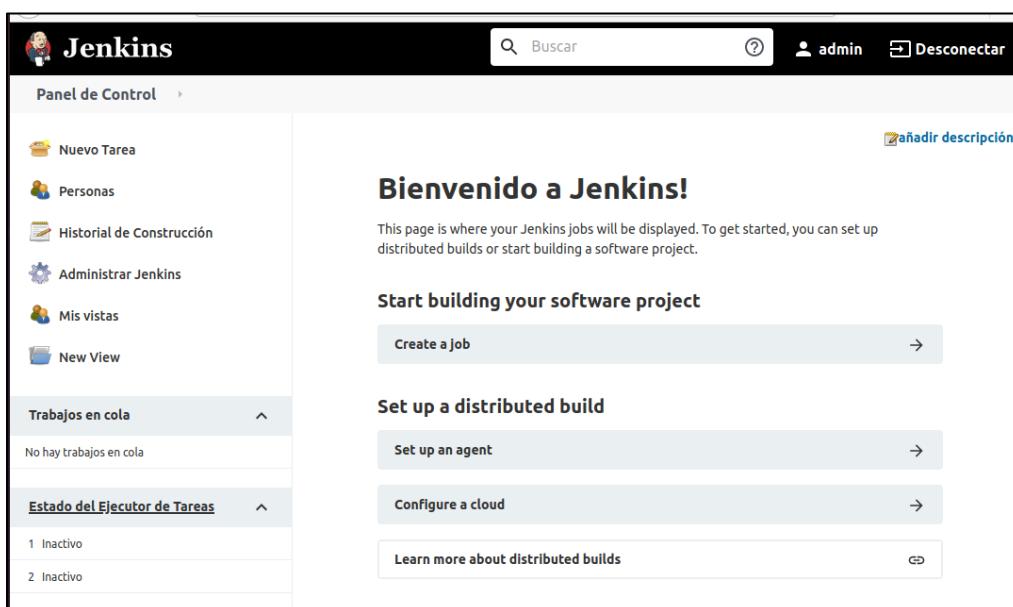
`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

Administrator password

```
fmediotte@fmediotte-Virtual-Machine:~$ docker exec -ti jenkins bash
jenkins@4d66e13df665:/$ cat /var/jenkins_home/secrets/initialAdminPassword
3446acffed234585b578b9f4a44526b3
jenkins@4d66e13df665:/$
```

Una vez ingresada la contraseña de administrador Jenkins nos solicitará instalar varios plugins y crear un usuario, una vez terminada la configuración inicial podríamos ver jenkins funcionando en nuestro localhost:



## Administrar usuarios

Podemos agregar usuarios a los contenedores desde un Dockerfile agregando una instrucción RUN useradd username

Por ejemplo, nuestro Dockerfile sería algo así:

```
FROM centos:7

ENV prueba 1234

RUN useradd fmediotte

USER fmediotte
```

En esta dockerfile le estamos diciendo a nuestro SO centos que va a estar utilizando el user creado fmediotte, por lo que si accedemos al contenedor con el comando:

**docker exec -ti <nombrecontenedor> bash**

Veremos que el usuario ya no es más root sino el que seteamos en el Dockerfile:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker run -d -ti --name prueba centos:prueba
5416d23f8a5a46e3fe4e7d2c248dfba7c1fbe986ebf98cdce63db34e3d726b10
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
5416d23f8a5a        centos:prueba     "/bin/bash"         2 seconds ago      Up 1 second
prueba
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker exec -ti prueba bash
[fmediotte@5416d23f8a5a /]$ █
```

Si por ejemplo comentásemos el Dockerfile la instrucción USER tendríamos agregado el usuario fmediotte pero no seteado por defecto por lo que si creamos nuevamente la imagen con el nombre centos:prueba2 y otro contenedor pero con esa imagen veremos que el usuario por defecto que toma el contenedor al ingresar al mismo es **root**:

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
ad8351723727        centos:prueba2    "/bin/bash"         3 minutes ago      Up 3 minutes
prueba2
5416d23f8a5a        centos:prueba     "/bin/bash"         17 minutes ago    Up 17 minutes
prueba
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker exec -ti prueba2 bash
[root@ad8351723727 /]#
```

Este usuario se puede modificar al ingresar al contenedor de prueba2 poniendo el argumento **-u username** en el flag de entrada al mismo:

**docker exec -u fmediotte -ti prueba2 bash**

```
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker exec -ti prueba2 bash
[root@ad8351723727 /]# exit
exit
fmediotte@fmediotte-Virtual-Machine:~/Docker/containers$ docker exec -u fmediotte -ti prueba2 bash
[fmediotte@ad8351723727 /]$
```

Si nuestro usuario por defecto es distinto de root porque en el Dockerfile hemos seteado uno, podemos acceder como root al contenedor poniendo el argumento **-u root** en el docker exec -ti.

## Limitar recursos de un contenedor

Para poder adentrarnos en este tema vamos a crear un contenedor de ejemplo con una imagen de mongo:

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

Una vez creado el mismo utilizaremos el comando **docker stats <nameContainer>** que nos permite ver cuántos recursos está consumiendo el contenedor enviado como parámetro.

| CONTAINER ID | NAME  | CPU % | MEM USAGE / LIMIT   | MEM % | NET I/O    | BLOCK I/O      | PIDS |
|--------------|-------|-------|---------------------|-------|------------|----------------|------|
| 5a444c0a26ea | mongo | 0.41% | 60.19MiB / 1.943GiB | 3.03% | 1.1kB / 0B | 62.4MB / 152kB | 33   |

Podemos observar que está utilizando 59.48MB de memoria de un límite de 2GB, esto es debido a que en Windows se asigna una porción de la memoria RAM de la computadora y por defecto son 2GB, mientras que en otras distribuciones como Ubuntu utiliza todos los recursos de la computadora.

En los casos donde se quiere limitar el uso de RAM y uso de CPU que ocupan los contenedores se puede lograr de la siguiente manera:

## Limitar Memoria RAM

Para la memoria RAM existe una opción en docker run que se llama **memory** y que posee las siguientes variantes:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker run --help | grep memo
--kernel-memory bytes          Kernel memory limit
-m, --memory bytes             Memory limit
--memory-reservation bytes     Memory soft limit
--memory-swap bytes            Swap limit equal to memory
--memory-swappiness int        Tune container memory
```

Si nosotros creamos un contenedor utilizando el argumento **-m <valor>** estaremos limitando la memoria que puede consumir ese contenedor:

Ejemplo:

```
docker run -d -m "500mb" --name mongo2 mongo
```

Podemos ejecutar el comando **docker stats mongo2** para ver si se aplicó ese límite de memoria a 500mb.

| CONTAINER ID | NAME   | CPU % | MEM USAGE / LIMIT | MEM %  | NET I/O   | BLOCK I/O     | PIDS |
|--------------|--------|-------|-------------------|--------|-----------|---------------|------|
| c322db0b58f0 | mongo2 | 0.37% | 55.39MiB / 500MiB | 11.08% | 656B / 0B | 131kB / 160kB | 33   |

También podemos especificar el valor en GB:

```
docker run -d -m "1gb" --name mongo3 mongo
```

| CONTAINER ID | NAME   | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O   | BLOCK I/O      | PIDS |
|--------------|--------|-------|-------------------|-------|-----------|----------------|------|
| 5f64aeaf5598 | mongo3 | 0.56% | 55.1MiB / 1GiB    | 5.38% | 766B / 0B | 2.69MB / 152kB | 33   |

De esta manera nosotros podemos limitar cuanta memoria RAM posee como límite cada contenedor.

## Limitar Uso de CPU

Para ver cuantas CPU tenemos en una máquina Linux podemos hacer uso del comando:

```
grep "model name" /proc/cpuinfo | wc -l
```

Lo que nos devolverá la cantidad de procesadores que tenemos disponibles en nuestra computadora.

Por defecto Docker le asigna todas las CPUs al contenedor, lo cual podemos limitar con argumentos que se le concatenan al docker run:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker run --help | grep cpu
  --cpu-period int          Limit CPU CFS (Completely Fair Scheduler) period
  --cpu-quota int           Limit CPU CFS (Completely Fair Scheduler) quota
  --cpu-rt-period int       Limit CPU real-time period in microseconds
  --cpu-rt-runtime int      Limit CPU real-time runtime in microseconds
  -c, --cpu-shares int      CPU shares (relative weight)
  --cpus decimal             Number of CPUs
  --cpuset-cpus string      CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems string      MEMs in which to allow execution (0-3, 0,1)
```

Por ejemplo, para limitar la cantidad de CPUs que utiliza un contenedor utilizamos el argumento **--cpuset-cpus [rangoCPUs]** donde rangoCPUs son la CPUs que se quieren compartir para el contenedor:

```
docker run -d -m "1gb" --cpuset-cpus 0-1 --name mongo4 mongo
```

De esta manera restringimos cuantas CPUs queremos utilizar para cada contenedor.

## Copiar archivos a un contenedor

Para poder copiar archivos de nuestra máquina al contenedor debemos hacer uso del comando:

**docker cp <archivoAcopiar> <pathDestino>**

*El cuál nos permite copiar desde fuera del contenedor hacia dentro del mismo y viceversa.*

Para hacer un ejemplo vamos a crear un contenedor con una imagen de apache levantando en el puerto 80 de nuestra máquina:

| CONTAINER ID | IMAGE | COMMAND            | CREATED            | STATUS            | PORTS              | NAMES  |
|--------------|-------|--------------------|--------------------|-------------------|--------------------|--------|
| 3df48a86aa4f | httpd | "httpd-foreground" | About a minute ago | Up About a minute | 0.0.0.0:80->80/tcp | apache |

Y vamos a copiar un archivo index.html hacia el contenedor, por lo que hacemos uso del comando:

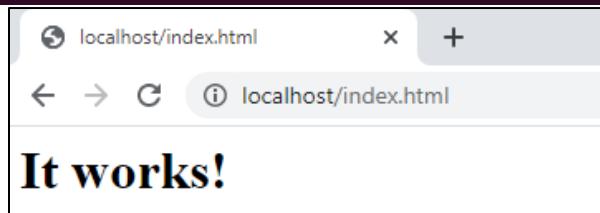
**docker cp index.html apache:/tmp**

Al acceder al contenedor al path especificado veremos que se encuentra el archivo index.html copiado exitosamente.

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ docker cp index.html apache:/tmp
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ docker exec -ti apache bash
root@3df48a86aa4f:/usr/local/apache2# cd /tmp
root@3df48a86aa4f:/tmp# ls -l
total 4
-rwxrwxrwx 1 1000 1000 3 Dec 20 21:58 index.html
root@3df48a86aa4f:/tmp# ls
index.html
root@3df48a86aa4f:/tmp# |
```

Para que tenga más sentido en el ejemplo que estamos haciendo vamos a obtener el path donde se encuentra el index.html que se levanta en el browser, el mismo en apache es **/usr/local/apache2/htdocs** donde veremos si hacemos un cat de index.html lo que observamos en el browser.

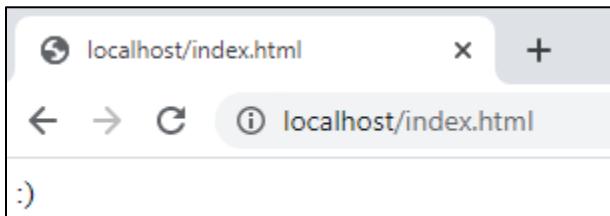
```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ docker exec -ti apache bash
root@3df48a86aa4f:/usr/local/apache2# cd /usr/local/apache2/htdocs/
root@3df48a86aa4f:/usr/local/apache2/htdocs# ls
index.html
root@3df48a86aa4f:/usr/local/apache2/htdocs# cat index.html
<html><body><h1>It works!</h1></body></html>
root@3df48a86aa4f:/usr/local/apache2/htdocs# |
```



Por lo que vamos a copiar nuestro index.html hacia dicha ruta para poder cambiar el html por defecto de apache:

```
docker cp index.html apache:/usr/local/apache2/htdocs
```

Si recargamos nuestra localhost veremos el contenido de nuestro index.html que teníamos fuera del contenedor.



Ahora probemos hacerlo a la inversa, es decir **desde el contenedor hacia un path en nuestra máquina**, por ejemplo, vamos a traernos un log de apache para verlo en nuestra máquina, lo que podría ser muy útil para revisar algún error de la aplicación:

Accedemos a la carpeta /var/log de apache y nos copiamos por ejemplo el archivo dpkg.log y utilizamos el comando:

```
docker cp <archivoAcopiar> <pathDestino>
```

En el ejemplo:

```
docker cp apache:/var/log/dpkg.log .
```

El “.” del final nos indica que queremos copiarlo en el directorio donde estamos parados:

```
root@3df48a86aa4f:/var/log# ls
alternatives.log  apt  btmp  [dpkg.log]  faillog  lastlog  wtmp
root@3df48a86aa4f:/var/log# exit
exit
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ docker cp apache:/var/log/dpkg.log .
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ ls
Dockerfile          Dockerfile-OLD          Nginx-php-fpm  docker.key  fruit      my-dockerfile  startbootstrap-freelancer-master
Dockerfile-Arguments  DockerfileAPACHE      beryllium    [dpkg.log]  index.html  run.sh       startbootstrap-sb-admin-2
Dockerfile-DI        DockerfileBuenasPracticas  docker.crt   exercise    multi      ssl.conf
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-images$ |
```

## Convierte un contenedor en una imagen

Para lograr realizar esto existe un comando de docker llamado **docker commit** que lo que hace es tomar el estado de un contenedor que está corriendo y transformarlo en una imagen.

Si realizamos un cambio en un contenedor y queremos que esos datos persistan por lo que usamos el comando **docker commit**, lo que no es muy buena práctica ya que lo mejor sería utilizar volúmenes.

Como ya sabemos los cambios que haga dentro de un contenedor van a ser temporales, ya que al eliminarlo se van a eliminar esos cambios también.

Vamos a probar un ejemplo

Generamos una imagen con un SO centos con un Dockerfile y levantamos la misma en un contenedor:

### Dockerfile

```
FROM centos:7  
  
VOLUME /opt/volumen
```

### Container:

| CONTAINER ID | IMAGE       | COMMAND     | CREATED            | STATUS            | PORTS | NAMES  |
|--------------|-------------|-------------|--------------------|-------------------|-------|--------|
| fe7886cd7f42 | centos:test | "/bin/bash" | About a minute ago | Up About a minute |       | centos |

Vamos a crear dos archivos dentro del contenedor accediendo al mismo con el comando **docker exec -ti centos bash**:

```
[root@fe7886cd7f42 volumen]# touch file1.txt  
[root@fe7886cd7f42 volumen]# ls  
file1.txt  
[root@fe7886cd7f42 volumen]# cd ..  
[root@fe7886cd7f42 opt]# touch file1.txt  
[root@fe7886cd7f42 opt]# ls  
file1.txt  volumen  
[root@fe7886cd7f42 opt]# |
```

Si eliminamos el contenedor y lo creamos nuevamente veremos que los cambios que hicimos ya no están, es decir los archivos creados ya no existen.

```
[root@b6b84a111607 /]# cd opt  
[root@b6b84a111607 opt]# ls  
volumen  
[root@b6b84a111607 opt]# cd volumen  
[root@b6b84a111607 volumen]# ls  
[root@b6b84a111607 volumen]# |
```

Lo que haremos es crearlos nuevamente:

```
[root@b6b84a111607 volumen]# cd ..
[root@b6b84a111607 opt]# ls
file1.txt  volumen
[root@b6b84a111607 opt]# cd volumen
[root@b6b84a111607 volumen]# ls
file1.txt
[root@b6b84a111607 volumen]# |
```

Lo que debemos hacer es capturar el estado actual del contenedor para crear una imagen nueva, por lo que procedemos a usar el comando:

**docker commit <nameContainer> <nombreNuevalImagen>**

Al ejecutar esto se crea una nueva imagen con el estado actual del contenedor:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ docker commit centos centos-resultante
sha256:d8041b13309610ec99b30c28112260b693d14bef14e9e6e3691d5974e8f1579d
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
b6b84a111607        centos:test      "/bin/bash"        5 minutes ago     Up 5 minutes          centos
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED            SIZE
centos-resultante   latest              d8041b133096   4 seconds ago    204MB
httpd               latest              dd85cd8b9987   9 days ago       138MB
mongo               latest              3068f6bb852e   10 days ago      493MB
centos              test                b5e9da74f3dc   5 weeks ago      204MB
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ |
```

Por lo que si ahora borramos nuestro contenedor y creamos uno nuevo en base a la imagen que generamos, deberíamos ver nuestros cambios dentro del contenedor:

**docker run -dti –name centos centos-resultante /bin/bash**

Se agrega el comando /bin/bash a la instrucción porque a veces con el commit se pierde el comando, podemos revisar si es correcto el mismo mediante el comando docker ps luego de crear el nuevo contenedor.

Si accedemos al contenedor veremos que el archivo file1.txt se encuentra en la carpeta opt pero no en la carpeta /opt/volumen:

```
[root@1a0d5b637746 /]# cd opt
[root@1a0d5b637746 opt]# ls
file1.txt  volumen
[root@1a0d5b637746 opt]# cd volumen
[root@1a0d5b637746 volumen]# ls
[root@1a0d5b637746 volumen]# exit
exit
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ cat Dockerfile
FROM centos:7

VOLUME /opt/volumen
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/containers/commit$ |
```

Esto se debe a que en nuestro Dockerfile pusimos al path /opt/volumen como un VOLUME y al hacer docker commit dichos cambios del volumen no se van a guardar, todo lo que esta fuera de volumen si se va a guardar, debido a que lo está dentro del path de un volumen no queremos que se guarde al hacer un docker commit.

## Sobrescribe el CMD de una imagen sin un Dockerfile

Todos los contenedores de una imagen de un SO tienen un CMD definido por defecto, por ejemplo, en el caso de centos es /bin/bash:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker run -dti centos
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
7a0437f04f83: Pull complete
Digest: sha256:5528e8b1b1719d34604c87e11dc1c0a20bedf46e83b5632cdeac91b8c04efc1
Status: Downloaded newer image for centos:latest
d1a749b54440aa5ad6323d8944ee3230eee83b96a29a5f51134f55d9799fdc4f
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d1a749b54440 centos "/bin/bash" 4 seconds ago Up 3 seconds fervent_allen
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ |
```

Si queremos sobrescribir este comando desde la terminal debemos agregar un argumento más al **docker run** que ya venimos utilizando, el mismo se agrega posterior al nombre de la imagen que queremos que quede corriendo en segundo plano en el contenedor, por ejemplo, si escribiésemos luego de la imagen algo como:

**docker run -dti centos echo hola mundo**

Nuestro contenedor tomaría el echo hola mundo como comando:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker run -dti centos echo hola mundo
2ce55f57f5354bc2fe9048e270ef860bca56e911fb3b79483c0295b45ccf334c
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d1a749b54440 centos "/bin/bash" 3 minutes ago Up 3 minutes fervent_allen
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
2ce55f57f535 centos "echo hola mundo" 11 seconds ago Exited (0) 10 seconds ago practical_shannon
d1a749b54440 centos "/bin/bash" 3 minutes ago Up 3 minutes fervent_allen
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ |
```

Si vemos los docker logs de dicho contenedor veremos la ejecución del hola mundo:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker logs practical_shannon
hola mundo
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ |
```

Complejizándolo un poco y para que nuestro contenedor quede vivo en segundo plano vamos a agregar un comando de Python y realizar el mapeo de puertos correspondiente, observamos que le pone en el comando del contenedor lo que le pusimos como parámetro luego de la imagen:

**docker run -d -p 8080:8080 centos python -m SimpleHTTPServer 8080**

```
COMMAND
"python -m SimpleHTTPServer 8080"
```

## Aprende a destruir contenedores automáticamente

El objetivo de esta sección es aprender cómo hacer que un contenedor se autodestruya.

Para lograr esto debemos agregar al comando docker run el argumento **--rm** el cuál instruye a docker que el contenedor que estas creando es temporal y que una vez el contenedor se salga o que te salgas de la sesión el contenedor debería finalizar.

Ejemplo:

```
docker run --rm -ti --name centos centos bash
```

Podemos observar que ya no estamos agregando el parámetro **-d** porque ya no quiero que corra en background, sino que quiero que sea un contenedor temporal.

Por lo que una vez que salgamos de la sesión del contenedor el mismo debería autodestruirse:

```
exit
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED   STATUS    PORTS     NAMES
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu$ |
```

## Cambiar el Document Root de Docker

El document root de Docker es el directorio donde se guardan todas las imágenes, contenedores, volúmenes, redes, etc.

Para ubicar cual es nuestro Document Root debemos utilizar el comando:

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

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker info | grep -i root
Docker Root Dir: /var/lib/docker
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ sudo su
[sudo] password for fmediotte:
root@NTBK-TECH678:/mnt/c/Users/fmediotte# cd /var/lib/docker
root@NTBK-TECH678:/var/lib/docker# ls
containerd  containers  image  network  overlay2  plugins  runtimes  tmp  trust  volumes
root@NTBK-TECH678:/var/lib/docker# |
```

Cada vez que nosotros tipeamos **docker images** estamos haciendo una consulta al document root.

Por defecto nuestro Document root se ubica en **/var/lib/docker**. Para modificar esta ruta debemos editar un archivo de configuración de Docker que se encuentra en **/lib/systemd/system/docker.service**, y editamos la línea que dice ExecStart agregando al final de la misma **--data-root <nuevoDocumentRoot>**, por ejemplo:

## --data-root /opt

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --data-root /opt
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

## Docker Volumes

Los volúmenes son herramientas que nos permiten almacenar datos del contenedor de una manera persistente en nuestra máquina local, lo cual es muy útil para mantener los datos persistentes del contenedor en caso de que el mismo se elimine.

Un ejemplo podría ser una imagen de una base de datos debido a que al recrear la imagen se estaría instanciando una base de datos limpia, por lo que, si le cargamos datos y luego eliminamos el contenedor, la información y los registros también se perderían.

Existen tres tipos de volúmenes:

- **Host:** se almacenan en nuestro docker host, y viven dentro de una carpeta de file system que nosotros definimos.
- **Anonymus:** no definimos una carpeta, sino que docker genera una carpeta random y dentro de la misma persiste la información.
- **Named Volumes:** son volúmenes que nosotros creamos, que no son carpetas nuestras, sino que son carpetas administradas por Docker, pero que a diferencia de los Anonymus si tienen un nombre y son manejados totalmente por Docker.

## ¿Por qué son importantes los volúmenes?

Por ejemplo, creamos un contenedor con una imagen de MySQL como lo hemos visto anteriormente:

```
docker run -d -p 3306:3306 --name my-db -e
"MYSQL_ROOT_PASSWORD=1234" -e "M
YSQL_DATABASE=docker-db" -e "MYSQL_USER=docker-user" -e
"MYSQL_PASSWORD=1234" mysql:5.7
```

Una vez el servicio este aceptando conexiones accedemos al motor con el comando **mysql -u root -h 127.0.0.1 -p** e ingresamos la password definida para el usuario root previamente, observamos que se haya creado bien la base de datos y salimos de la consola de mysql.

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| docker-db      |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.00 sec)

mysql> use docker-db
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> exit
```

Hacemos un dump de la base sys con el comando:

```
mysqldump -u root -h 127.0.0.1 -p1234 sys > dump.sql --column-statistics=0
```

Lo cual nos generara un archivo dump.sql el cual vamos a importarlo en nuestra base de datos creada **docker-db**, con el siguiente comando:

```
mysql -u root -h 127.0.0.1 -p docker-db < dump.sql
```

Una vez realizado accedemos nuevamente a nuestra base de datos docker-db y ejecutamos el comando **show tables**, el cual nos mostrará las tablas nuevas importadas del dump.sql.

```
mysql> show tables;
+-----+
| Tables_in_docker-db |
+-----+
| host_summary
| host_summary_by_file_io
| host_summary_by_file_io_type
| host_summary_by_stages
| host_summary_by_statement_latency
| host_summary_by_statement_type
| innodb_buffer_pool_stats
+-----+
```

Este paso es para validar que tengamos información en docker-db.

Si nosotros tenemos información importante en docker-db y procedemos a eliminar el contenedor y crearlo nuevamente:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0ca0b8e7b8f6 mysql:5.7 "docker-entrypoint.s..." 35 minutes ago Up 35 minutes 0.0.0.0:3306->3306/tcp, 33060/tcp my-db
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker rm -f my-db
my-db
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker run -d -p 3306:3306 --name my-db -e "MYSQL_ROOT_PASSWORD=1234" -e "MYSQL_DATABASE=docker-db" -e "MYSQL_USER=docker-user" -e "MYSQL_PASSWORD=1234" mysql:5.7
f6dbd96547a0e41336a07823e6154365cddf05f83fd95e38075bb9c0294f0b66
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f6dbd96547a0 mysql:5.7 "docker-entrypoint.s..." 3 seconds ago Up 3 seconds 0.0.0.0:3306->3306/tcp, 33060/tcp my-db
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$
```

Al conectarnos nuevamente a nuestra base de datos docker-db y ejecutar el comando show tables, veremos que hemos perdido toda la información que teníamos previamente ya que la misma se perdió al eliminar el contenedor:

```
mysql> show tables;
Empty set (0.00 sec)

mysql> |
```

Para evitar que nos suceda esto, lo que debemos hacer es indicarle a docker que cuando elimine el contenedor borre absolutamente todo menos las carpetas que queramos resguardar la información, esto lo hacemos con **volúmenes**, los cuales nos permiten guardar información persistente y clave para nuestra operatoria.

## Volúmenes de host – Caso práctico MySQL

En el caso práctico de MySQL la ruta donde se persiste la información persistente de la base de datos la podemos obtener de la documentación oficial de la imagen de mysql de Docker hub, donde nos indica que MySQL por defecto persistirá sus datos en la ruta **/var/lib/mysql** dentro del contenedor, por lo que si nosotros guardamos este directorio en un volumen persistente vamos a guardar toda la configuración y bases de datos que creemos en MySQL.

Para lograr persistir esos datos por el medio de la configuración de un volumen de host, lo que debemos indicar es en el argumento **-v** del docker run la carpeta del file system de la máquina local donde queremos guardar la información persistente y mapearla con la carpeta persistente del contenedor, por ejemplo:

```
docker run -d --name db -p 3306:3306 -e "MYSQL_ROOT_PASSWORD=1234" -v /opt/mysql/:/var/lib/mysql mysql:5.7
```

Donde **/opt/mysql** es nuestro directorio que estamos mapeando, por lo que al correr dicho comando ya se empezarán a crear las carpetas correspondientes a **/var/lib/mysql** en el directorio.

Podríamos crear bases de datos de ejemplo como test, test1 y test2 y posteriormente borrar el contenedor:

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
| test           |
| test1          |
| test2          |
+-----+
7 rows in set (0.01 sec)

mysql> exit
Bye
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker rm -fv db
```

Ahora si lo creamos nuevamente con el comando:

```
docker run -d --name db -p 3306:3306 -e "MYSQL_ROOT_PASSWORD=1234" -v
/opt/mysql/:/var/lib/mysql mysql:5.7
```

Docker lo que hará es obtener la información del volumen persistente **/opt/mysql** que guardo información del contenedor anterior, por lo que al acceder nuevamente al nuevo contenedor veremos que las bases de datos siguen creadas:

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
| test           |
| test1          |
| test2          |
+-----+
7 rows in set (0.00 sec)

mysql> |
```

## Volúmenes anónimos – Caso práctico MySQL

En el caso de volúmenes anónimos es similar a los volúmenes de host, pero en este caso al utilizar el argumento **-v** no vamos a especificar la ruta de nuestra máquina que actuará de volumen, solo vamos a referenciar el volumen del contenedor de la siguiente forma:

```
docker run -d --name db -p 3306:3306 -e "MYSQL_ROOT_PASSWORD=1234" -v  
/var/lib/mysql mysql:5.7
```

Lo que hará Docker es mapear una carpeta como volumen local en nuestra máquina, pero va a ser al azar, dicho volumen lo va a alojar en nuestro Document root de Docker en la carpeta volumen, con un nombre aleatorio que podemos saber si inspeccionamos el contenedor con el comando **docker inspect <nombreContenedor>** veremos en la sección Mounts los volúmenes que tenemos persistiendo para este contenedor, su nombre y donde están alojados:

```
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "33cc665a97a3efe78a377f8c506542083cacaea7ad8105bddc1514ed14a70d67",  
    "Source": "/var/lib/docker/volumes/33cc665a97a3efe78a377f8c506542083cacaea7ad8105bddc1514ed14a70d67/_data",  
    "Destination": "/var/lib/mysql",  
    "Driver": "local",  
    "Mode": "",  
    "RW": true,  
    "Propagation": ""  
  }  
]
```

No es aconsejable de utilizar estos volúmenes anónimos debido a dos puntos:

- Nombre aleatorio y difícil de acceder
- Si eliminamos el contenedor con el comando **docker rm -fv <ContainerName>** el volumen también se va a eliminar porque le estamos indicando **-v** que significa que borre el volumen que Docker, podemos solucionar esto solo borrando con el argumento **-f**.

## Instrucción VOLUME dentro de un Dockerfile

La instrucción VOLUME como hemos visto en la sección de Docker images nos sirve para indicar donde guardar la data persistente dentro de nuestra máquina.

Por ejemplo, tenemos el siguiente dockerfile:

```
FROM centos:7  
  
VOLUME /opt/
```

La instrucción VOLUME /opt nos permite crear un volumen anónimo cuando creamos el contenedor.

Para ver los volumes persistentes que tenemos en nuestra máquina podemos hacer uso del comando:

**docker volumen ls**

Para realizar un ejemplo creamos una imagen con el Dockerfile de antes y creamos un contenedor con el mismo, con el comando:

**docker run -dti --name test test-vol**

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker run -dti --name test test-vol  
67c7089ad24e1e8e0659629fbe2a610681b02dd95c44f7d7e1ab2ae5cde94acf  
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
67c7089ad24e test-vol "/bin/bash" 4 seconds ago Up 3 seconds test  
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ |
```

Si ahora consultamos los volúmenes existentes veremos que tenemos un nuevo volumen anónimo creado:

| DRIVER | VOLUME NAME  |
|--------|--|
| local  | 6a70dd7d2961d9732de936baab86d52a67b46f2bb57edbfff432dcf32e66769fb5 |

Creamos dos archivos en la carpeta que definimos como Volumen en el Dockerfile:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker exec -ti test bash  
[root@67c7089ad24e /]# ls  
anaconda-post.log bin dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var  
[root@67c7089ad24e /]# cd opt/  
[root@67c7089ad24e opt]# ls  
[root@67c7089ad24e opt]# touch file1.txt  
[root@67c7089ad24e opt]# touch file2.txt  
[root@67c7089ad24e opt]# ls  
file1.txt file2.txt  
[root@67c7089ad24e opt]# |
```

Si accedemos al volumen veremos los dos archivos creados.

- Para borrar un volumen podemos borrarlo junto al contenedor con el comando:  
**docker rm -fv test**
- Para borrar un volumen podemos borrarlo con el comando: **docker volumen rm <volumeName>**

## Volúmenes nombrados – Caso práctico MySQL

Es una unión entre el volumen de host y un volumen anónimo.

Para crear un nuevo volumen nombrado utilizamos el siguiente comando:

**docker volume create <volumeName>**

Ejemplo:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume create mysql-data
mysql-data
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume ls
DRIVER      VOLUME NAME
local      mysql-data
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ |
```

Para eliminar el mismo se utiliza el comando:

**docker volume rm <volumeName>**

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume rm mysql-data
mysql-data
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume ls
DRIVER      VOLUME NAME
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ |
```

### ¿Cómo asignamos un volumen nombrado a un contenedor?

Creamos un volumen nuevo con el comando visto previamente llamado my-vol

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume create my-vol
my-vol
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume ls
DRIVER      VOLUME NAME
local      my-vol
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ |
```

Y en la instrucción docker run hacemos uso del argumento -v para realizar el mapeo de los volúmenes local y del contenedor, de la siguiente forma:

```
docker run -d --name mysql -v my-vol:/var/lib/mysql -p 3306:3306 -e
"MYSQL_ROOT_PASSWORD=1234" -e "MYSQL_DATABASE=docker-db" mysql:5.7
```

El cambio con respecto a los volúmenes de host es que en esta forma utilizamos el nombre del volumen y no la ruta.

Otra ventaja es que si nosotros eliminamos el contenedor con la opción -v en este caso el volumen no se va a eliminar, sino que el mismo va a ser persistente:

```
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
435f4a779f2b mysql:5.7 "docker-entrypoint.s..." 5 minutes ago Up 5 minutes 0.0.0.0:3306->3306/tcp, 33060/tcp mysql
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker rm -fv mysql
mysql
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ docker volume ls
DRIVER VOLUME NAME
local my-vol
fmediotte@NTBK-TECH678:/mnt/c/Users/fmediotte$ |
```

### Tip para inspeccionar volúmenes en Windows:

```
docker run --rm -it -v /:/vm-root alpine:edge ls -l /vm-
root/var/lib/docker/volumes/my-vol/_data
```

### Access the MobyLinux VM's file system:

```
# Run this from your regular terminal on Windows / MacOS:
docker container run --rm -it -v /:/host alpine

# Once you're in the container that we just ran, run this:
chroot /host
```

## Dangling volumes

Un dangling volumen es un volumen huérfano o sin referenciar, el mismo se genera cuando creamos varios contenedores con volúmenes definidos para cada uno de ellos y posteriormente los eliminamos sin la opción de -v, los volúmenes siguen existiendo en nuestro document root, pero sin referenciar a ningún contenedor.

Para eliminar los dangling volumes podemos listarlos con el argumento -f dangling=true de la siguiente forma:

**docker volume ls -f dangling=true**

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker volume ls -f dangling=true
DRIVER VOLUME NAME
local b6c7b5f1f53bb5144afbd02160b5f6e5cf718696fd533d1d33818dd7d53416d7
local c4ec5906ceb08586730674c1b7150705219c1e571e0496ac1852de70b422bf45
local dc7ec9715e3a3d1d3e85c8d81badfedc6b0158b2731338be3147ad12f711ad4e
local my-vol
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Para eliminar todos los dangling volumes podemos utilizar el siguiente comando:

```
docker volume ls -f dangling=true -q | xargs docker volume rm
```

## Persistiendo data en MongoDB

Creamos una carpeta llamada mongo en la ruta opt que vamos a utilizar de volumen, por lo que nos quedará algo así:

```
/opt/mongo # |
```

Creamos un contenedor con una imagen de mongo mapeada al puerto 27017, y mapeamos el volumen de nuestra path local con el volumen persistente de mongo, el cual podemos consultar en la documentación oficial de mongo en docker hub. En el caso de mongo la misma se guarda en la carpeta **/data/db**, por lo que nuestro docker run será el siguiente:

```
docker run -d -p 27017:27017 -v /opt/mongo:/data/db mongo
```

Una vez creado nuestro contenedor, accedemos al mismo mediante el comando **docker exec -ti <containerName> bash** y tipeamos la palabra mongo para poder acceder a nuestro motor de base de datos.

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec -ti charming_yonath bash
root@2cb31b1e065f:/# mongo
MongoDB shell version v4.4.2
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("eb46b64d-5815-478a-8af8-00c2b7ddae4") }
MongoDB server version: 4.4.2
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    https://docs.mongodb.com/
Questions? Try the MongoDB Developer Community Forums
    https://community.mongodb.com
---
The server generated these startup warnings when booting:
  2020-12-27T20:43:37.745+00:00: Access control is not enabled for the database. Read and write ac
ed
---
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
```

Insertamos un registro en la base de datos siguiendo el siguiente tutorial:

[https://www.tutorialspoint.com/mongodb/mongodb\\_create\\_database.htm](https://www.tutorialspoint.com/mongodb/mongodb_create_database.htm)

Veremos que ahora nuestra base de datos contiene datos:

```

> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
mydb     0.000GB
> use mydb
switched to db mydb
> show collections
movie
> db.movie.find()
{ "_id" : ObjectId("5fe8f28c5432a9c13be48335"), "name" : "tutorials point" }
>

```

Ahora lo que haremos es borrar el contenedor y observaremos que en la carpeta /opt/mongo aún tenemos información persistente:

```

fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker ps
CONTAINER ID        IMAGE       COMMAND             CREATED          STATUS           PORTS          NAMES
2cb31b1e065f        mongo      "docker-entrypoint.s..."   9 minutes ago   Up 9 minutes   0.0.0.0:27017->27017/tcp   charming_yonath
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker rm -fv 2cb31b1e065f
2cb31b1e065f
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ 
/opt/mongo # ls
WiredTiger          WiredTigerHS.wt
WiredTiger.lock     _mdb_catalog.wt
WiredTiger.turtle   collection-0-5653090885863898229.wt
WiredTiger.wt       collection-2-5653090885863898229.wt
collection-4-5653090885863898229.wt
collection-7-5653090885863898229.wt
diagnostic.data
index-3-5653090885863898229.wt
index-5-5653090885863898229.wt
index-6-5653090885863898229.wt
index-8-5653090885863898229.wt
journal
mongod.lock
sizeStorer.wt
storage.bson
/opt/mongo #

```

Por lo que si volvemos a recrear el contenedor lo que hará docker es mapearle el volumen persistente que le indiquemos al contenedor como input del volumen propio de mongo, por lo que nuestras bases de datos con sus datos seguirían vivas a pesar de haber eliminado el contenedor:

```

> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
mydb     0.000GB
> use mydb
switched to db mydb
> db.movie.find()
{ "_id" : ObjectId("5fe8f28c5432a9c13be48335"), "name" : "tutorials point" }
>

```

## Persistiendo data en Jenkins

Creamos una carpeta llamada jenkins en la ruta opt que vamos a utilizar de volumen, por lo que nos quedará algo así:

```
/opt/jenkins # |
```

Creamos un contenedor con una imagen de jenkins mapeada al puerto 8080, y mapeamos el volumen de nuestra path local con el volumen persistente de jenkins, el cual podemos consultar en la documentación oficial de jenkins en docker hub. En el caso de jenkins la misma se guarda en la carpeta **/var/jenkins\_home**, por lo que nuestro docker run será el siguiente:

```
docker run -d --name jenkins -p 8080:8080 -v /opt/jenkins/:/var/jenkins_home  
jenkins
```

(En el video esta explicado así, pero la imagen de jenkins fue modificada y no funciona el volumen /opt/jenkins por lo que vamos a utilizar jenkins-data que va a estar ubicado en /var/lib/docker/volumes)

Cómo siempre Jenkins nos solicitará la password que se encuentra dentro del contenedor en el archivo: **/var/jenkins\_home/secrets/initialAdminPassword**

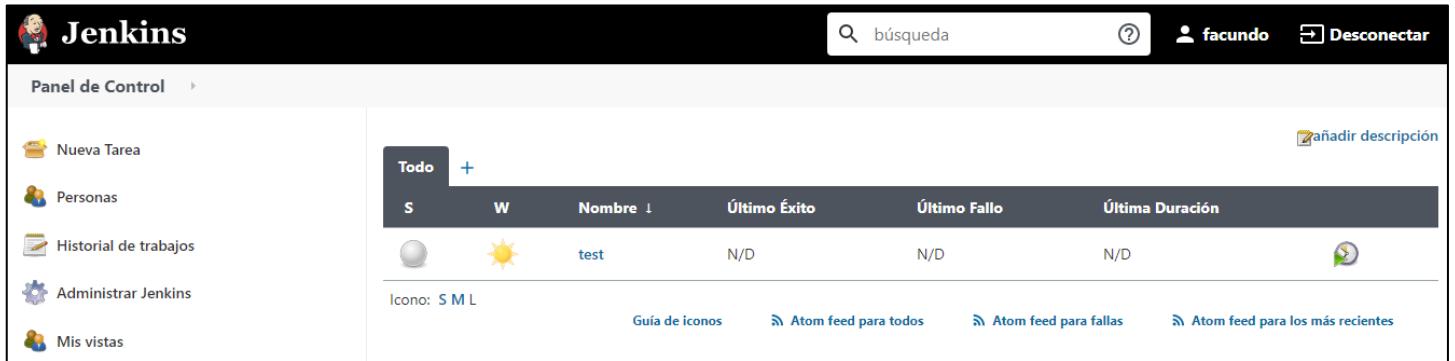


Podríamos acceder con docker exec -ti pero vamos a aprender una forma de que no haga falta acceder al contenedor sino que podemos obtener el contenido de un archivo desde fuera del mismo, esto se logra quitando el argumento -ti (terminal interactive) y agregando el argumento -c (command) en el comando docker exec, este argumento nos permite agregar un comando que se va a ejecutar dentro del contenedor, en este caso utilizaremos un cat del archivo de la siguiente manera:

```
docker exec jenkins bash -c "cat  
/var/jenkins_home/secrets/initialAdminPassword"
```

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec jenkins bash -c "cat /var/jenkins_home/secrets/initialAdminPassword"  
d21e52edd7a042ac9047f08b3e3e8b38  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Creamos una tarea automatizada en Jenkins:



The screenshot shows the Jenkins dashboard. On the left sidebar, there are links for 'Nueva Tarea', 'Personas', 'Historial de trabajos', 'Administrar Jenkins', and 'Mis vistas'. The main area displays a table of jobs. The first job listed is 'test', which has a status icon showing a sun (W), a name of 'test', and the last two columns showing 'Último Éxito' and 'Último Fallo' both as 'N/D'. At the bottom of the table, it says 'Icono: S M L'. Below the table, there are links for 'Guía de iconos', 'Atom feed para todos', 'Atom feed para fallas', and 'Atom feed para los más recientes'. A button 'añadir descripción' is located at the top right of the table.

Borramos nuestro contenedor de jenkins para validar si funciona correctamente nuestro volumen, podemos observar en /var/lib/docker/volumes/jenkins-data/\_data que tenemos aún información del contenedor que acabamos de eliminar:

```
/var/lib/docker/volumes/jenkins-data/_data # ls
config.xml          jenkins.install.InstallUtil.lastExecVersion    jobs           plugins          tini_pub.gpg   war
copy_reference_file.log  jenkins.install.UpgradeWizard.state      logs            secret.key     updates
hudson.model.UpdateCenter.xml  jenkins.model.JenkinsLocationConfiguration.xml nodeMonitors.xml secret.key.not-so-secret userContent
identity.key.enc      jenkins.telemetry.Correlator.xml        nodes           secrets         users
```

Recreamos el contenedor con el comando que utilizamos anteriormente:

```
docker run -d --name jenkins -p 8080:8080 -v jenkins-data:/var/jenkins_home
jenkins
```

Y si accedemos a nuestro localhost:8080 jenkins ya no nos solicitará ninguna password, sino que accedemos al login de jenkins, nos logueamos con el usuario que creamos previamente y veremos que en el home de la aplicación tenemos el job aún creado:



This screenshot is identical to the one above, showing the Jenkins dashboard with the 'test' job listed in the central table. The job has a sun icon (W), name 'test', and 'N/D' in the last two columns. The bottom links and description button are also present.

De esta manera comprobamos que nuestro volumen persistente funciona correctamente y no perdimos toda la información útil que ya teníamos en el volumen.

## Persistiendo logs de Nginx

Creamos un contenedor de Nginx con el siguiente comando:

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

y accedemos al mismo para revisar donde guarda los logs de Nginx:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec -ti nginx bash
root@bcfe292ac882:/# cd /var/log/nginx
root@bcfe292ac882:/var/log/nginx# ls
access.log  error.log
root@bcfe292ac882:/var/log/nginx# |
```

Los mismos como se muestra en la imagen se guardan en /var/log/nginx. De esta forma averiguamos donde se están guardando dentro del contenedor los logs, por lo que borramos el contenedor.

Creamos un nuevo contenedor mapeando un nuevo volumen que creamos previamente llamado nginx:

```
docker run -d --name nginx -p 80:80 -v ./nginx/:/var/log/
nginx/ nginx
```

Dentro del cual vamos a tener los logs de nginx:

```
/var/lib/docker/volumes/nginx/_data # ls
access.log  error.log
/var/lib/docker/volumes/nginx/_data # |
```

Y si borramos el contenedor esta data queda guardada en el volumen persistente.

## Comparte volúmenes entre uno o más contenedores

Un volumen puede ser compartido entre uno o más contenedores, para verlo en la práctica crearemos una carpeta que se llame common, la cual será el source volume de más de un contenedor y dentro de la misma un Dockerfile con el siguiente contenido:

```
FROM centos:7

COPY start.sh /start.sh

RUN chmod +x /start.sh

CMD /start.sh
```

Donde nuestro archivo start.sh será de la siguiente manera:

```
#!/bin/bash

while true; do
    echo "<p>$ (date +%H:%M:%S) </p>" >> /opt/index.html && \
    sleep 10
done
```

Su objetivo es imprimir la hora en un archivo index.html y por cada iteración que realice dormirá 10 segundos.

Si creamos un contenedor utilizando como volumen la carpeta common debemos escribir un comando como el siguiente:

```
docker run -v common:/opt -d --name gen generador
```

Luego si inspeccionamos el archivo index.html veremos que se esta escribiendo la hora cada 10 segundos:

```
/var/lib/docker/volumes/common/_data # cat index.html
03:44:42
03:44:52
03:45:02
/var/lib/docker/volumes/common/_data # |
```

Ahora vamos a crear un contenedor de nginx con el comando:

```
docker run -d -p 80:80 --name nginx -v common:/usr/share/nginx/html nginx:alpine
```

Vamos a tener corriendo nuestro dos contenedores si listamos con **docker ps**:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES
1e6881fca1ce        nginx:alpine       "/docker-entrypoint..."   18 seconds ago      Up 17 seconds   0.0.0.0:80->80/tcp   nginx
f1a049aa0bc0        generador         "/bin/sh -c /start.sh"   5 minutes ago     Up 2 minutes           gen
fmediotte@NTBK-TECH678:/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/volumes$ |
```

Si vamos a nuestro localhost en algún browser veremos el resultado de start.sh:

```

localhost
x
localhost
< > C i localhost
03:59:59
04:00:09
04:00:19
04:00:29
04:00:39
04:00:49

```

Pero además estamos compartiendo el volumen common por lo que el funcionamiento es:

- El generador está generando texto html que se persiste en el volumen common.
- Cuando creamos el segundo contenedor de nginx y le pasamos el mismo volumen como parámetro se le está compartiendo la data (index.html) al document root de nginx (:/usr/share/nginx/html) por lo que cuando levantamos localhost estamos viendo ese contenido index.html en el puerto 80 que expusimos del contenedor.

## Docker Network

En esta sección vamos a ver los siguientes tópicos:

- Crear / Eliminar Redes
- Tipos de redes:
  - Bridge
  - Host
  - None
  - Overlay
- Conectar contenedores

### ¿Cuál es la red por defecto?

La red por defecto de Docker es docker0 que te asigna un rango de subnet.

Por ejemplo, si inspeccionamos un contenedor veremos en la parte de Networks -> bridge la dirección IPAddress asignada al mismo, y una IP Gateway de la red por defecto de Docker:

```

"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",
    "EndpointID": "5e4cd269ca242944791e44fdd0c21abb31a9d5ee7ea05348dff491a46dd90bd",
    "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
  }
}

```

## There is no docker0 bridge on Windows

Because of the way networking is implemented in Docker Desktop for Windows, you cannot see a docker0 interface on the host. This interface is actually within the virtual machine.

Para ver la red por defecto de docker debemos utilizar el comando:

**docker network ls**

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
73a5ae549533    bridge    bridge      local
2ff7bf818df9    host      host       local
dbbf65a1a689    none     null       local
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Y si queremos filtrar por la red **bridge (red por defecto de Docker)** que es la que nos interesa en este momento le agregamos un grep bridge de la siguiente forma:

**docker network ls | grep bridge**

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network ls | grep bridge
73a5ae549533    bridge    bridge      local
```

Hacemos un **docker network inspect bridge** para ver la configuración de red de la misma:

```
"Name": "bridge",
"Id": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",
"Created": "2020-12-27T18:54:16.674300Z",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
        {
            "Subnet": "172.17.0.0/16",
            "Gateway": "172.17.0.1"
        }
    ]
},
```

Si hacemos un docker inspect del contenedor ya creado de la siguiente forma:

```
docker inspect <containerName>
```

Veremos en la sección de Network que la red a la que pertenece es "bridge":

```
"Networks": {  
    "bridge": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null,  
        "NetworkID": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",  
        "EndpointID": "5e4cd269ca242944791e44fdd0c21abb31a9d5ee7ea05348dff491a46dd90bd",  
        "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  
    }  
}
```

La red por defecto que trae docker es **bridge**, es decir que si creamos un nuevo contenedor si no definimos que trabaje con otra red va a asignar siempre el mismo a la red bridge:

```
"Networks": {  
    "bridge": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null,  
        "NetworkID": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",  
        "EndpointID": "1622eb86b10533b774893ecebf8e665d11e2cbb93d3170b31bee3e84371f2ac6",  
        "Gateway": "172.17.0.1",  
        "IPAddress": "172.17.0.3",  
        "IPPrefixLen": 16,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:11:00:03",  
        "DriverOpts": null  
    }  
}
```

También se puede utilizar el comando ping desde un contenedor a otro, debido a que pertenecen a la misma red.

Por ejemplo, si tenemos dos contenedores de centos podemos utilizar el comando ping para solicitar respuesta el otro contenedor:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
8293fce373fa centos:7 "/bin/bash" About a minute ago Up About a minute practical_euclid  
bbc456bcd37 centos:7 "/bin/bash" About a minute ago Up About a minute confident_ramanujan  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker inspect practical_euclid |
```

Al inspeccionar las redes de los 2 contenedores obtenemos lo siguiente:

```
"Networks": {  
    "bridge": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null,  
        "NetworkID": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",  
        "EndpointID": "11fb38962af24de1b8e220f0a0c303ec4cb52572404a472170f7fc91469f64ba",  
        "Gateway": "172.17.0.1",  
        "IPAddress": "172.17.0.3",  
        "IPPrefixLen": 16,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:11:00:03",  
        "DriverOpts": null  
    }  
}  
  
"Networks": {  
    "bridge": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null,  
        "NetworkID": "73a5ae549533bd30b2fd17de8cb58dc7723c36943787327b0eff92c4adcf88b7",  
        "EndpointID": "c36770dd8b04c6874995de5452cb2a2f6ac8219d0b8946c7d99bc61878a420c4",  
        "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  
    }  
}
```

Por lo que podemos utilizar el comando:

**docker exec <containerName> bash -c “ping <ipOtherContainer>”**

En este ejemplo:

```
CONTAINER ID  IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES  
8293fce373fa  centos:7  "/bin/bash"  3 minutes ago  Up 3 minutes   practical_euclid  
bbbcb456bcd37  centos:7  "/bin/bash"  3 minutes ago  Up 3 minutes   confident_ramanujan  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec confident_ramanujan bash -c "ping 172.17.0.3"  
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.  
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.061 ms  
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.047 ms  
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.042 ms  
64 bytes from 172.17.0.3: icmp_seq=4 ttl=64 time=0.046 ms
```

## Crear una red definida por el usuario

Para poder crear una red necesitamos hacer uso del driver bridge, que es el driver por defecto que trae la red de docker, que funciona como una mini red virtual donde podemos agregar contenedores.

Comandos disponibles para trabajar con redes:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network

Usage: docker network COMMAND

Manage networks

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

Run 'docker network COMMAND --help' for more information on a command.
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

En este caso vamos a crear una red por lo que utilizaremos el comando:

**docker network create <networkName>**

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network create test-network
1bf12eb44d0e178f8c8cf93dbab2231354a3f645dcff4facbf3a58a85700e6be
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network ls | grep test
1bf12eb44d0e  test-network  bridge  local
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Si queremos operar con una red nueva y agregarle subredes y rango de ips de Gateway, podemos consultar el help del comando docker network create:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network create --help

Usage: docker network create [OPTIONS] NETWORK

Create a network

Options:
  --attachable      Enable manual container attachment
  --aux-address map Auxiliary IPv4 or IPv6 addresses used by Network driver (default map[])
  --config-from string The network from which to copy the configuration
  --config-only     Create a configuration only network
  -d, --driver string Driver to manage the Network (default "bridge")
  --gateway strings  IPv4 or IPv6 Gateway for the master subnet
  --ingress          Create swarm routing-mesh network
  --internal         Restrict external access to the network
  --ip-range strings Allocate container ip from a sub-range
  --ipam-driver string IP Address Management Driver (default "default")
  --ipam-opt map    Set IPAM driver specific options (default map[])
  --ipv6            Enable IPv6 networking
  --label list       Set metadata on a network
  -o, --opt map      Set driver specific options (default map[])
  --scope string     Control the network's scope
  --subnet strings   Subnet in CIDR format that represents a network segment
```

Por ejemplo, podemos definir el driver (por defecto es “bridge”), una subnet, setearle un Gateway a la misma, por ejemplo:

```
docker network create -d bridge --subnet 172.124.10.0/24 --gateway 172.124.10.1 docker-test-network
```

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker network ls | grep docker-test
96d1367ce6af    docker-test-network    bridge    local
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Docker inspect de la red creada:

```
[{"Name": "docker-test-network", "Id": "96d1367ce6afe3c91d0977f3864ee0497422129dc9ff70445b923855bd9255f6", "Created": "2020-12-28T18:41:20.6985046Z", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": {}, "Config": [{"Subnet": "172.124.10.0/24", "Gateway": "172.124.10.1"}]}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}, "ConfigOnly": false, "Containers": {}, "Options": {}, "Labels": {}}]
```

## Inspeccionar Redes

Para inspeccionar redes se hace uso del comando:

**docker network inspect <netName>**

Dentro de la misma encontraremos el siguiente formato:

```
[  
  {  
    "Name": "docker-test-network",  
    "Id": "96d1367ce6afe3c91d0977f3864ee0497422129dc9ff70445b923855bd9255f6",  
    "Created": "2020-12-28T18:41:20.6985046Z",  
    "Scope": "local",  
    "Driver": "bridge",  
    "EnableIPv6": false,  
    "IPAM": {  
      "Driver": "default",  
      "Options": {},  
      "Config": [  
        {  
          "Subnet": "172.124.10.0/24",  
          "Gateway": "172.124.10.1"  
        }  
      ]  
    },  
    "Internal": false,  
    "Attachable": false,  
    "Ingress": false,  
    "ConfigFrom": {  
      "Network": ""  
    },  
    "ConfigOnly": false,  
    "Containers": {},  
    "Options": {},  
    "Labels": {}  
  }  
]
```

Donde se encuentran definidos ciertos parámetros:

- **Name:** nombre de la red
- **Id:** id de la red
- **Created:** fecha de creación
- **Configuraciones como el scope, driver, subnet, Gateway.**

## Agregar contenedores a una red distinta a la por defecto

Si creamos un contenedor por defecto se va a hostear en la red por defecto de docker llamada **bridge**.

Si queremos que nuestro contenedor se conecte a un red distinta a la por defecto, por ejemplo, a la creada previamente llamada docker-test-network, debemos utilizar el argumento --network en el comando docker run de la siguiente manera:

**docker run --network <netName> -d <imageName>**

En el ejemplo:

```
docker run --network docker-test-network -d --name test2 -ti centos
```

Y si inspeccionamos el contenedor creado veremos que se conectó con la red que pasamos como parámetro heredando las configuraciones que habíamos definido para la red:

```
"Networks": {  
    "docker-test-network": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": [  
            "9d56450f12db"  
        ],  
        "NetworkID": "96d1367ce6afe3c91d0977f3864ee0497422129dc9ff70445b923855bd9255f6",  
        "EndpointID": "622ec752ed0de2a14f2210584d832915fb20b252b73cdffaaf44f0eca288e91a5",  
        "Gateway": "172.124.10.1",  
        "IPAddress": "172.124.10.2",  
        "IPPrefixLen": 24,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:7c:0a:02",  
        "DriverOpts": null  
    }  
}
```

De esta manera creamos un contenedor a una red distinta a la red de docker.

## Conegar contenedores en la misma red

Para explicar esto debemos crear dos contenedores en la misma red, por ejemplo:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker run -d --network docker-test-network --name cont1 -ti centos d2eb85b6b1d9e72d184dbbb065ae8e0834efaedb77214ef79166efab9abed09  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker run -d --network docker-test-network --name cont2 -ti centos a0244d479f903ba5363f0902ad8d276472502edc0fa4525a2ea54122598d5aae  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
a0244d479f90 centos "/bin/bash" 4 seconds ago Up 2 seconds cont2  
d2eb85b6b1d9 centos "/bin/bash" 9 seconds ago Up 8 seconds cont1  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker inspect cont1 | grep IPAddress  
    "SecondaryIPAddresses": null,  
    "IPAddress": "",  
        "IPAddress": "172.124.10.2",  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker inspect cont2 | grep IPAddress  
    "SecondaryIPAddresses": null,  
    "IPAddress": "",  
        "IPAddress": "172.124.10.3",  
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Si por ejemplo mi contenedor 1 es mi WebServer y mi contenedor 2 es mi base de datos.

El cont1 tiene la ip: 172.124.10.2

El cont2 tiene la ip: 172.124.10.3

Verifico que se vean entre ellos con el comando ping:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont1 bash -c "ping 172.124.10.3"
PING 172.124.10.3 (172.124.10.3) 56(84) bytes of data.
64 bytes from 172.124.10.3: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 172.124.10.3: icmp_seq=2 ttl=64 time=0.109 ms
64 bytes from 172.124.10.3: icmp_seq=3 ttl=64 time=0.049 ms
64 bytes from 172.124.10.3: icmp_seq=4 ttl=64 time=0.048 ms
64 bytes from 172.124.10.3: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 172.124.10.3: icmp_seq=6 ttl=64 time=0.070 ms
64 bytes from 172.124.10.3: icmp_seq=7 ttl=64 time=0.057 ms
64 bytes from 172.124.10.3: icmp_seq=8 ttl=64 time=0.125 ms
^C
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont2 bash -c "ping 172.124.10.2"
PING 172.124.10.2 (172.124.10.2) 56(84) bytes of data.
64 bytes from 172.124.10.2: icmp_seq=1 ttl=64 time=0.049 ms
64 bytes from 172.124.10.2: icmp_seq=2 ttl=64 time=0.050 ms
64 bytes from 172.124.10.2: icmp_seq=3 ttl=64 time=0.060 ms
64 bytes from 172.124.10.2: icmp_seq=4 ttl=64 time=0.058 ms
64 bytes from 172.124.10.2: icmp_seq=5 ttl=64 time=0.088 ms
^C
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ |
```

Otra ventaja de crear nuestras propias redes es que en la red por defecto de docker no podemos ver a los contenedores por su nombre, en cambio en las **user define networks** podemos hacer ping por nombre de contenedor (dns), lo cual nos facilita el trabajo:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont1 bash -c "ping cont2"
PING cont2 (172.124.10.3) 56(84) bytes of data.
64 bytes from cont2.docker-test-network (172.124.10.3): icmp_seq=1 ttl=64 time=0.059 ms
64 bytes from cont2.docker-test-network (172.124.10.3): icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from cont2.docker-test-network (172.124.10.3): icmp_seq=3 ttl=64 time=0.100 ms
64 bytes from cont2.docker-test-network (172.124.10.3): icmp_seq=4 ttl=64 time=0.063 ms
^C
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont2 bash -c "ping cont1"
PING cont1 (172.124.10.2) 56(84) bytes of data.
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=1 ttl=64 time=0.041 ms
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=3 ttl=64 time=0.103 ms
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=4 ttl=64 time=0.118 ms
^C
```

## Conecitar contenedores en distintas redes

Para lograr exemplificar como haríamos estos debemos primero tener 2 redes creadas:

```
96d1367ce6af    docker-test-network    bridge    local
1fc0c56e9d26    test1                  bridge    local
```

```
"Name": "docker-test-network",
"Id": "96d1367ce6afe3c91d0977f3864ee0497422129dc9ff70445b923855bd9255f6",
"Created": "2020-12-28T18:41:20.6985046Z",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
        {
            "Subnet": "172.124.10.0/24",
            "Gateway": "172.124.10.1"
        }
    ]
},
{
    "Name": "test1",
    "Id": "1fc0c56e9d26c311174704fefafa2e09f6f1c9a9a28df1a5abc96630b1abbbfb90",
    "Created": "2020-12-28T19:09:46.7439282Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.19.0.0/16",
                "Gateway": "172.19.0.1"
            }
        ]
    }
},
```

```
"Name": "test1",
"Id": "1fc0c56e9d26c311174704fefafa2e09f6f1c9a9a28df1a5abc96630b1abbbfb90",
"Created": "2020-12-28T19:09:46.7439282Z",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
        {
            "Subnet": "172.19.0.0/16",
            "Gateway": "172.19.0.1"
        }
    ]
}
},
```

Como vemos ambas tienen configuraciones de red.

Vamos a tener dos contenedores uno en cada red:

## Cont1

```
"docker-test-network": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
        "d2eb85b6b1d9"
    ],
    "NetworkID": "96d1367ce6afe3c91d0977f3864ee0497422129dc9ff70445b923855bd9255f6",
    "EndpointID": "511278fd2cfad8633e74226f1e60be3708343c7eac12bfa324702c2f8deeb05d",
    "Gateway": "172.124.10.1",
    "IPAddress": "172.124.10.2",
    "IPPrefixLen": 24,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:7c:0a:02",
    "DriverOpts": null
}
```

## Cont3

```
"Networks": {
    "test1": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": [
            "d15d1170bbb1"
        ],
        "NetworkID": "1fc0c56e9d26c311174704fefafa2e09f6f1c9a9a28df1a5abc96630b1abbbfb90",
        "EndpointID": "ed4a548b5097efc923353dc4b020743550f52a7702bfa92a10c4164ca83fd4f9",
        "Gateway": "172.19.0.1",
        "IPAddress": "172.19.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:13:00:02",
        "DriverOpts": null
    }
}
```

Para poder conectar ambos contenedores vamos a utilizar el comando:

**docker network connect <netNameForConnect> <nameContainerToConnect>**

Por lo que en nuestro ejemplo sería de la siguiente forma:

**docker network connect docker-test-network cont3**

Si ahora inspeccionamos nuestro contenedor “cont3” veremos que se mantiene la red original del contenedor, pero además se lo adjunta a la red docker-test-network por lo que este contenedor ahora pertenece a 2 redes actualmente.

Si ahora intentamos hacer un ping desde cont1 a cont3, el mismo responderá correctamente:

```
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont1 bash -c "ping cont3"
PING cont3 (172.124.10.4) 56(84) bytes of data.
64 bytes from cont3.docker-test-network (172.124.10.4): icmp_seq=1 ttl=64 time=0.085 ms
64 bytes from cont3.docker-test-network (172.124.10.4): icmp_seq=2 ttl=64 time=0.088 ms
64 bytes from cont3.docker-test-network (172.124.10.4): icmp_seq=3 ttl=64 time=0.080 ms
^C
fmediotte@NTBK-TECH678:/c/Users/fmediotte$ docker exec cont3 bash -c "ping cont1"
PING cont1 (172.124.10.2) 56(84) bytes of data.
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=1 ttl=64 time=0.108 ms
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=2 ttl=64 time=0.066 ms
64 bytes from cont1.docker-test-network (172.124.10.2): icmp_seq=3 ttl=64 time=0.137 ms
^C
```

De esta manera lo que se hizo fue conectar el cont 3 a la red, pero si ahora lo queremos desconectar debemos utilizar el comando:

**docker network disconnect <netName> <containerNameToDisconnect>**

En nuestro ejemplo:

**docker network disconnect docker-test-network cont3**

Si inspeccionamos el contenedor “cont3” veremos que ahora solo pertenece a la red test1.

```
"Networks": {
    "test1": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": [
            "d15d1170bbbb1"
        ],
        "NetworkID": "1fc0c56e9d26c311174704fefafa2e09f6f1c9a9a28df1a5abc96630b1abbbfb90",
        "EndpointID": "ed4a548b5097efc923353dc4b020743550f52a7702bfa92a10c4164ca83fd4f9",
        "Gateway": "172.19.0.1",
        "IPAddress": "172.19.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:13:00:02",
        "DriverOpts": null
    }
}
```

## Eliminar redes

Para eliminar redes que ya no queremos utilizar hacemos uso del comando:

**docker network rm <netName>**

## Asignar IP a un contenedor

Primero creamos una red de la siguiente forma:

```
docker network create --subnet 172.128.10.0/24 --gateway  
172.128.10.1 -d bridge my-net
```

Y luego creamos un contenedor en la red creada previamente:

```
docker run --network my-net -d --name nginx1 -ti centos
```

Al inspeccionarlo veremos que se creo con una ip al azar:

```
"Networks": {  
    "my-net": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": [  
            "e89576527c61"  
        ],  
        "NetworkID": "e2c10b0a487d2975137e38e624f4810f0ec4724d5a37c0764b13e38ad7995616",  
        "EndpointID": "62af0eeaa9eeeb27d67f032736b8887f2e0347539385251b1af58b9fe1daff237",  
        "Gateway": "172.128.10.1",  
        "IPAddress": "172.128.10.2",  
        "IPPrefixLen": 24,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:80:0a:02",  
        "DriverOpts": null  
    }  
}
```

Ahora si yo quiero modificar la asignación de esa ip por una que yo quiera, lo que debo hacer es en el comando **docker run** pasarle un argumento llamado **--ip** y le asignamos un host de la subred que nosotros queramos, por ejemplo:

```
docker run --network my-net --ip 172.128.10.50 -d --name nginx2 -  
ti centos
```

Si inspeccionamos el contenedor nginx2 veremos que se asigno la ip que definimos como input del comando:

```
"Networks": {  
    "my-net": {  
        "IPAMConfig": {  
            "IPv4Address": "172.128.10.50"  
        },  
        "Links": null,  
        "Aliases": [  
            "39a2a4b0c50a"  
        ],  
        "NetworkID": "e2c10b0a487d2975137e38e624f4810f0ec4724d5a37c0764b13e38ad7995616",  
        "EndpointID": "18d632ba9bdd73ea75581f00ed1d3c5cc74e4a4a0650875de5c26090c4f40b9b",  
        "Gateway": "172.128.10.1",  
        "IPAddress": "172.128.10.50",  
        "IPPrefixLen": 24,
```

## La red host

En docker ya existe por una red llamada host que mapea nuestro host de nuestra máquina por lo que si creo un contenedor en dicha red estaría creándolo en mi host local.

Por lo que podemos que si creamos un contenedor tendríamos el mismo host, hostname y red que mi máquina local:

```
docker run --network host -d --name tst2 -ti centos
```

| CONTAINER ID | IMAGE  | COMMAND     | CREATED        | STATUS        | PORTS | NAMES |
|--------------|--------|-------------|----------------|---------------|-------|-------|
| b188259931d1 | centos | "/bin/bash" | 39 seconds ago | Up 38 seconds |       | tst2  |

fmediotte@NTBK-TECH678:/c/Users/fmediotte\$ |

## La red none

Es una red que viene por defecto con docker, y es utilizada para que los contenedores que hostiemos ahí no tengan red.

Hagamos una prueba:

```
docker run --network none --name hola -d -ti centos
```

Si inspeccionamos el contenedor “hola” veremos que no tiene ninguna IP ni Gateway definido:

```
"Networks": {
    "none": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "dbbf65a1a6899064d9bc155a239b120657276f8d318dc706bfac9498ce134170",
        "EndpointID": "0206da6c9f49a81fa371c7b91333fea4473af1d6c8054f775523457cd0830c88",
        "Gateway": "",
        "IPAddress": "",
        "IPPrefixLen": 0,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "",
        "DriverOpts": null
    }
}
```

## Docker Compose

Es una herramienta de Docker que nos permite crear aplicaciones multicontenedor.

Por ejemplo, un sitio en wordpress que necesita un servidor web, apache o nginx con php instalado y adicional necesita una base de datos para guardar información.

Con lo que vimos hasta ahora crearíamos un contenedor para un web server y crearíamos otro contenedor con una base de datos y estos contenedores los conectaríamos por una red que nosotros creamos y definimos. Esto nos llevaría aproximadamente 10 líneas para hacerlo funcionar.

Con Docker Compose lo que hacemos es definir todo este tipo de cosas como contenedores, imágenes, volúmenes, redes y demás cosas lo definimos en un archivo de texto de extensión .yml y Docker Compose va a tomar ese archivo lo va a leer y va a ejecutar su contenido en lo que se incluye crear los contenedores, las imágenes, crear y construir volúmenes y redes. Por lo tanto, simplifica bastante la operatoria ya que solo con escribir **docker compose app** (comando para crear todo desde el archivo de texto) va a quedar todo lo definido funcionando.

## Instalación

### En Linux:

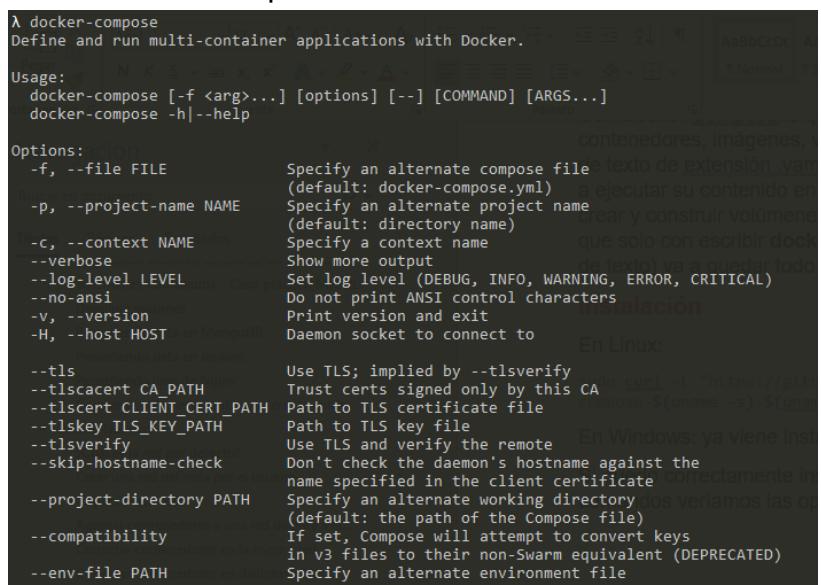
```
sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Convertirse a root con el comando: **sudo su**

Darle permisos de ejecución con: **chmod +x /usr/local/bin/docker-compose**

**En Windows:** ya viene instalado con Docker Desktop.

Si quedo correctamente instalado al escribir docker-compose en una línea de comandos veríamos las opciones que la herramienta nos ofrece:



```
λ docker-compose
Define and run multi-container applications with Docker.

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

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

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

## Primeros pasos

Con Docker-compose escribimos la declaración de un docker run en un archivo de texto que puede llevar cualquier nombre siempre y cuando sea de formato yml, por defecto se llama **docker-compose.yml**.

Este archivo se compone de 4 grandes partes:

- Versión
- Services
- Volumes (opcional)
- Networks (opcional)

La diferencia que existe entre crear un contenedor de la forma normal y crearlo en docker compose es que en la primera se utiliza una línea de comando para crear un contenedor con el ya visto **docker run** y sus argumentos, mientras que en la segunda se debe llenar el archivo docker-compose.yml.

Para saber que parámetros cargar en el archivo docker-compose.yml buscamos en la documentación oficial de docker compose de la versión que estemos utilizando.

### docker-compose.yml

```
version: '3'
services:
  web:
    container_name: nginx1
    ports:
      - "8080:80"
    image: nginx
```

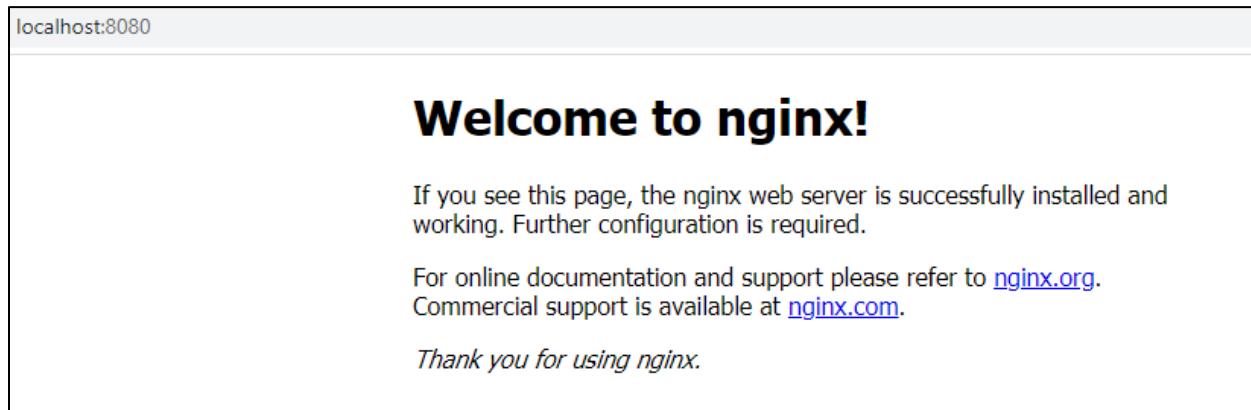
Para poder utilizarlo debemos ir a la terminal y escribir el siguiente comando:

**docker-compose up -d**

Lo que hará este comando es crear y levantar el contenedor, pero previamente docker por defecto crea una red donde aloja este contenedor:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose
λ docker-compose up -d
Creating network "docker-compose_default" with the default driver
Creating nginx1 ... done
```

Si accedemos a nuestro localhost:8080 veremos nginx corriendo con docker-compose:



Si queremos eliminar este docker-compose utilizamos el comando:

**docker-compose down**

Que lo que hace es detener, remover el contenedor y eliminar la red que se creo por defecto que creo el docker-compose:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose
λ docker-compose down
Stopping nginx1 ... done
Removing nginx1 ... done
Removing network docker-compose_default
```

## Variables de entorno en Compose

Para definir variables de entorno en Docker compose debemos agregar en nuestro archivo docker-compose.yml el atributo **environment** de la siguiente manera:

```
version: '3'
services:
  db:
    image: mysql:5.7
    container_name: mysql
    ports:
      - "3306:3306"
    environment:
      - "MYSQL_ROOT_PASSWORD=1234"
```

Ejecutamos el docker-compose up

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ docker-compose up -d
Creating network "varentorno_default" with the default driver
Creating mysql ... done
```

Y queda levantado nuestro contenedor en el puerto 3306:

| CONTAINER ID | IMAGE     | COMMAND                  | CREATED            | STATUS            | PORTS                             | NAMES |
|--------------|-----------|--------------------------|--------------------|-------------------|-----------------------------------|-------|
| 0987c1d5fe43 | mysql:5.7 | "docker-entrypoint.s..." | About a minute ago | Up About a minute | 0.0.0.0:3306->3306/tcp, 33060/tcp | mysql |

Si accedemos al mismo veremos nuestra variable de entorno correctamente seteada:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ docker exec -ti mysql bash
root@0987c1d5fe43:/# env
Y queda levantado el contenedor en el puerto 3306
MYSQL_MAJOR=5.7
HOSTNAME=0987c1d5fe43
PWD=/
MYSQL_ROOT_PASSWORD=1234
HOME=/root
MYSQL_VERSION=5.7.32-1debian10
GOSU_VERSION=1.12
TERM=xterm
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=~/bin/env
root@0987c1d5fe43:/# | Compose
```

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ docker-compose down
Stopping mysql ... done
Removing mysql ... done
Removing network varentorno_default
MYSQL_VERSION=5.7
GOSU_VERSION=1.12
TERM=xterm
```

Hay otra manera de definir una variable de entorno en un docker-compose.yml y es llevar dicho variable de entorno que queramos definir a otro archivo generalmente llamado common.env:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ echo "MYSQL_ROOT_PASSWORD=1234" > common.env

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ cat common.env
"MYSQL_ROOT_PASSWORD=1234"
```

Lo que debemos modificar en nuestro docker-compose.yml es reemplazar el archivo environment por el atributo **env\_file** que recibe como parámetro el archivo common.env:

```
version: '3'

services:
  db:
    image: mysql:5.7
    container_name: mysql
    ports:
      - "3306:3306"
    env_file: common.env
```

Ahora sencillamente recreamos el contenedor y veremos que la variable de entorno sigue definida dentro del mismo ya que la obtiene del archivo common.env:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ docker-compose up -d
Creating network "varentorno_default" with the default driver
Creating mysql ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno
λ docker exec -ti mysql bash
root@2c8e022c7463:/# env
MYSQL_MAJOR=5.7
HOSTNAME=2c8e022c7463
hola=hola12
PWD=/
MYSQL_ROOT_PASSWORD=1234
HOME=/root
MYSQL_VERSION=5.7.32-1debian10
GOSU_VERSION=1.12
TERM=xterm
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/usr/bin/env
root@2c8e022c7463:/#
```

## Volúmenes en Compose

Cómo ya estuvimos viendo existen 3 tipos de volúmenes:

- Host volumes
- Anonymous volumes
- Named volumes

Para esta sección veremos los host y named volumes.

Para crear un **named volume** lo que hacíamos era hacer uso del comando:

**docker volumen create <volumenName>**

**¿Cómo definimos un named volume con docker-compose?**

Sencillamente escribimos el parámetro **volumes**: en nuestro docker-compose.yml y seguido el nombre del volumen.

```
version: '3'
services:
  web:
    container_name: nginx1
    ports:
      - "8080:80"
    image: nginx
volumes:
  vol2:
```

Adicional lo que debemos hacer es colocar nuestro volumen dentro de nuestro contenedor, lo que hacemos es escribir el atributo volumes dentro del nodo web e indicamos que volumen vamos a montar, esto se hace en reemplazo del -v host:container que utilizábamos antes para montar un volumen a un contenedor:

```
version: '3'
services:
  web:
    container_name: nginx1
    ports:
      - "8080:80"
    volumes:
      - "vol2:/usr/share/nginx/html"
    image: nginx
volumes:
  vol2:
```

Lo único que nos falta hacer es levantar el contenedor con un docker-compose up -d:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Volumes (main -> origin)
λ docker-compose up -d
Creating network "volumes_default" with the default driver
Creating volume "volumes_vol2" with default driver
Creating nginx1 ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Volumes (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
337cc1437ba9 nginx "/docker-entrypoint..." 11 seconds ago Up 10 seconds 0.0.0.0:8080->80/tcp nginx1
```

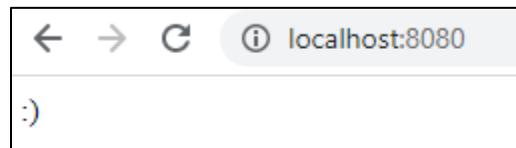
Para validar que funciona nuestro volumen nos iremos al document root obtenido con el comando:

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

```
/var/lib/docker/volumes # ls
backingFsBlockDev metadata.db volumes_vol2
/var/lib/docker/volumes # |
```

Si modificamos nuestro archivo index.html por una carita feliz y bajamos y subimos nuevamente el contenedor con docker-compose el contenido en el volumen no debería modificarse:

```
/var/lib/docker/volumes/volumes_vol2/_data # cat index.html
:)
```



## ¿Cómo definimos un host volume con docker-compose?

Para definir un host volumen se debe agregar en nuestro archivo docker-compose.yml

Buscaremos nuestro código fuente para montar al volumen:

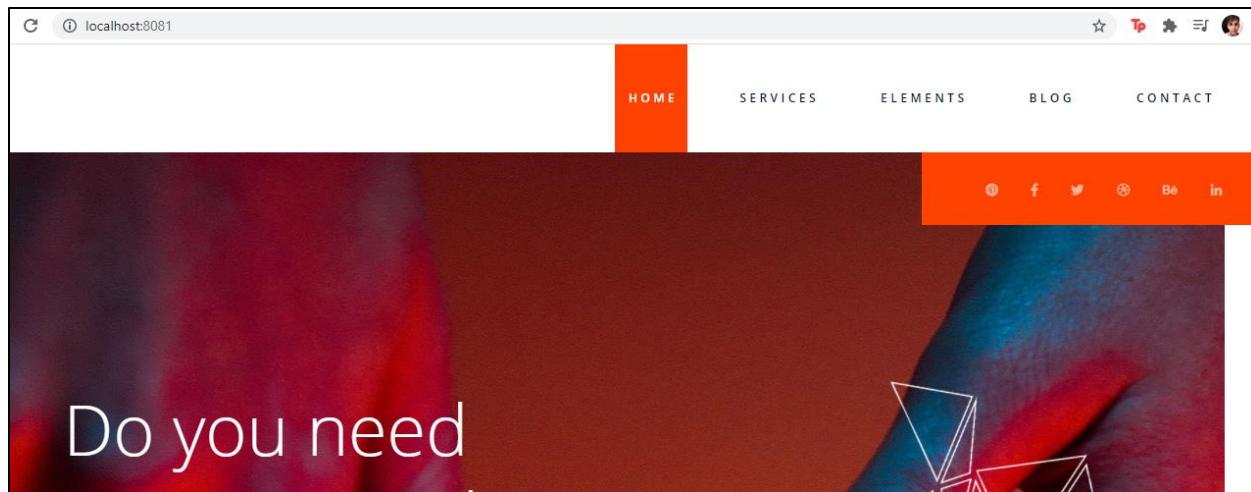
```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Volumes\host volume
λ ls
docker-compose.yml html/
```

Y utilizaremos el mismo para montar el volumen:

```
version: '3'
services:
  web:
    container_name: nginx2
    ports:
      - "8081:80"
    volumes:
      - "/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-compose/Volumes/host volume/html:/usr/share/nginx/html"
    image: nginx
```

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Volumes\host volume
λ docker-compose up -d
Creating network "hostvolume_default" with the default driver
Creating nginx2 ... done
```

Si ahora accedemos a nuestro localhost:8081 veremos levantado nuestra aplicación web:



## Redes en Compose

Repasando conocimientos para crear una red con docker utilizamos el comando **docker network create <netName>**

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose
λ docker network create facu
94a4873a28317671bf54ceab0931963424da340a6803a68bf5b62be5f79d97d2
```

Para hacer lo mismo en docker compose se escribe el parámetro **network** y luego el nombre de la red:

```
version: '3'
services:
  web:
    container_name: nginx2
    ports:
      - "8081:80"
    image: nginx
networks:
  net-test:
```

Para incluir al contenedor dentro de esta red debemos escribir dentro del parámetro web el atributo networks e indicar la lista de redes en la que queremos disponibilizar el contenedor:

```

version: '3'
services:
  web:
    container_name: nginx2
    ports:
      - "8081:80"
    image: nginx
    networks:
      - net-test
networks:
  net-test:

```

Luego ejecutamos el comando docker-compose up -d y tendremos nuestro contenedores corriendo en la red que definimos en el archivo .yml:

```

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Redes (main -> origin)
λ docker-compose up -d
Creating network "redes_net-test" with the default driver
Creating nginx2 ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Redes (main -> origin)
λ docker ps
CONTAINER ID   IMAGE     COMMAND          CREATED        STATUS       PORTS     NAMES
5f5fa64d8d56   nginx     "/docker-entrypoint..."   10 seconds ago   Up 9 seconds   0.0.0.0:8081->80/tcp   nginx2

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\Redes (main -> origin)
λ |

```

```

"Networks": {
  "redes_net-test": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "5f5fa64d8d56",
      "web"
    ],
    "NetworkID": "512e8fb6f28f6a9f8240ede8a0a0ed64f30252ae066020e2efad887c51d88bcf",
    "EndpointID": "cf688161c43e22f700b7f5e476772e8adce2cd8b89ced71a3372db5e1006e023",
    "Gateway": "172.19.0.1",
    "IPAddress": "172.19.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:13:00:02",
    "DriverOpts": null
  }
}

```

Lo que nos permite hacer el docker-compose con redes es crear varias servicios y hostearlos en la misma red de la siguiente manera:

```
version: '3'
services:
  web:
    container_name: nginx2
    ports:
      - "8081:80"
    image: httpd
    networks:
      - net-test
  web2:
    container_name: nginx3
    ports:
      - "8082:80"
    image: httpd
    networks:
      - net-test
networks:
  net-test:
```

Luego desde el servicio de apache podemos hacer un ping desde el contenedor nginx3 hacia el contenedor nginx2 nos va a responder debido a que ambos existen en la misma red:

**docker exec -ti nginx2 bash -c “ping nginx3”**

También podríamos realizar un ping por el nombre del servicio definido en el docker-compose.yml:

**docker exec -ti nginx2 bash -c “ping web2”**

## Construye imágenes en Compose

En esta sección veremos cómo construir una imagen, un dockerfile utilizando un **docker-compose build** que es utilizado en docker compose para construir imágenes.

Por ejemplo, teniendo nuestro docker-compose.yml de la siguiente forma:

```
version: '3'
services:
  web:
    container_name: web
    image: web-test
    build: .
```

El nombre de la imagen es una que queremos construir y el parámetro importante que necesitamos para crear imágenes desde docker compose es **build** donde en este caso al indicar un “.” buscara en el directorio donde estamos parados un Dockerfile para armar una imagen, por lo que creamos un Dockerfile sencillo:

```
FROM centos

RUN mkdir /opt/test
```

Y ejecutamos el comando **docker-compose build** para que se construya nuestra imagen:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-
λ docker-compose build
Building web
Step 1/2 : FROM centos
--> 300e315adb2f
Step 2/2 : RUN mkdir /opt/test
--> Using cache
--> 33cf13304c96

Successfully built 33cf13304c96
Successfully tagged web-test:latest

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-
λ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
web-test        latest   33cf13304c96  30 seconds ago  209MB
exercise        latest   a9fc953983f6  3 days ago    348MB
nginx           latest   ae2feff98a0c  2 weeks ago   133MB
httpd           latest   dd85cd9bb9987  2 weeks ago   138MB
centos          latest   300e315adb2f  3 weeks ago   209MB
```

Si nosotros tenemos un Dockerfile con un nombre distinto, lo que debemos hacer es en build poner una atributo llamado **context** que indica en que carpeta se encuentra el Dockerfile, y otro atributo llamado **dockerfile** el que indica el nombre del Dockerfile con el cuál queremos construir nuestra imagen:

```
version: '3'
services:
  web:
    container_name: web
    image: web-test
    build:
      context: .
      dockerfile: Dockerfile1
```

Por lo que procedemos a crear un Dockerfile1 igual al Dockerfile ya existe y ejecutamos nuevamente el comando **docker-compose build** y se construirá la imagen correctamente:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\build
λ docker-compose build
Building web
  web: Using cache
Step 1/2 : FROM centos
--> 300e315adb2f
Step 2/2 : RUN mkdir /opt/test
--> Using cache
--> 33cf13304c96
Successfully built 33cf13304c96
Successfully tagged web-test:latest
```

Por lo que procedemos nuevamente el comando **docker-compose build** y se construirá la imagen correctamente:

## Sobreescribe el CMD de un contenedor con Compose

En esta sección aprenderemos como modificar un CMD de un contenedor sin la necesidad de crear un Dockerfile para modificarlo.

El CMD de un contenedor se modifica agregando en nuestro docker-compose.yml el atributo command al nivel del contenedor que queramos crear:

```
version: '3'
services:
  web:
    image: centos
    command: python -m SimpleHTTPServer 8080
    ports:
      - "8080:8080"
```

Por ejemplo, si creamos un contenedor con la imagen de centos veremos que el comando por defecto de dicha imagen es **/bin/bash**:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\cmd (main -> origin)
λ docker run -dti centos
89605cbdf3ffe8d2bf8367c402c9b0fe3f774c5942ffa4572f8d0149c7e593d

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\cmd (main -> origin)
λ docker ps
CONTAINER ID   IMAGE      COMMAND           CREATED          STATUS          PORTS     NAMES
89605cbdf3f   centos    "/bin/bash"        7 seconds ago   Up 6 seconds            focused_mirzakhani
```

Pero con nuestro docker compose podemos modificar el mismo y levantaron normalmente con **docker-compose up -d**.

## Limitar recursos en contenedores (Compose v2)

Como ya hemos visto en secciones anteriores podemos limitar la memoria y las cpus utilizadas por el contenedor por medio de los atributos:

**-m <valor>**: limita la memoria del contenedor, por ejemplo:

```
docker run -d -m "500mb" --name mongo2 mongo
```

**--cpuset-cpus [rangoCPUs]**: Cpus que puede utilizar el contenedor, por ejemplo:

```
docker run -d -m "1gb" --cpuset-cpus 0-1 --name mongo4 mongo
```

Para limitar ambos valores en docker compose lo que se utiliza son dos parámetros:

**mem\_limit: <valor>**

**cpuset: <valor>**

Podríamos tener un docker-compose.yml que sea de la siguiente forma:

```
version: '2'
services:
  web:
    container_name: nginx
    mem_limit: 20m
    cpuset: "0"
    image: nginx:alpine
```

Levantamos nuestro servicio con un **docker-compose up -d** y observamos cuanta memoria puede utilizar con el comando docker stats <nameContainer>:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\mem-cpu (main -> origin)
λ docker-compose up -d
Creating network "mem-cpu_default" with the default driver
Creating nginx ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\mem-cpu (main -> origin)
λ docker ps
CONTAINER ID   IMAGE      COMMAND           CREATED          STATUS          PORTS     NAMES
b772187f3c12   nginx:alpine   "/docker-entrypoint..."   4 seconds ago   Up 3 seconds   80/tcp   nginx
```

| CONTAINER ID | NAME  | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O     | BLOCK I/O   | PIDS |
|--------------|-------|-------|-------------------|-------|-------------|-------------|------|
| b772187f3c12 | nginx | 0.00% | 1.875MiB / 20MiB  | 9.38% | 1.24kB / 0B | 0B / 8.19kB | 2    |

## Política de reinicio de contenedores

Son las condiciones que se deben cumplir para que un contenedor deba ser reiniciado, para configurar esta política se agrega el flag **--restart** en el comando docker run que tiene distintas variaciones:

| Flag                        | Description  |
|-----------------------------|--|
| <code>no</code>             | Do not automatically restart the container. (the default)  |
| <code>on-failure</code>     | Restart the container if it exits due to an error, which manifests as a non-zero exit code.  |
| <code>always</code>         | Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in <a href="#">restart policy details</a> ) |
| <code>unless-stopped</code> | Similar to <code>always</code> , except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.   |

Por default la política de reinicio es que el contenedor no se reinicie, esto se puede ejemplificar si por ejemplo creamos un contenedor con centos y luego utilizamos un docker stop el mismo no se va a reiniciar nunca, sino que queda ahí parado excepto que lo reiniciemos manualmente con un docker start:

```
C:\Users\fmediotte
λ docker run -dti centos
9e878e1b7b5337d095e0d5b645ba4eb07bb2980cacc2a186c9fc03d4dbbf91de

C:\Users\fmediotte
λ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
9e878e1b7b53        centos             "/bin/bash"         8 seconds ago      Up 7 seconds          jolly_solomon

C:\Users\fmediotte
λ docker stop jolly_solomon
jolly_solomon

C:\Users\fmediotte
λ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
```

Son las condiciones que se deben cumplir para configurar esta política se agregan distintas variaciones:

Para el caso del **restart always** se reinicia el contenedor cada vez que es detenido, para probar este escenario, tendremos un docker-compose.yml de la siguiente forma:

```
version: '3'
services:
  test:
    container_name: test
    image: restart-image
    build: .
    restart: always
```

El mismo lo que hará es construir una imagen llamada restart-image y tenemos un atributo nuevo llamado restart que por defecto tiene no como valor, pero en este caso tendremos dicho parámetro seteado con el valor always. Adicionalmente tenemos que el build lo construye con un Dockerfile ubicado en la misma ruta que el archivo yml:

## Dockerfile

```
FROM centos

COPY start.sh /start.sh

RUN chmod +x /start.sh

CMD /start.sh
start.sh

#!/bin/bash

echo "Estoy vivo"
sleep 5
echo "Estoy detenido"
```

Procedemos a construir nuestra imagen con un docker-compose build:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart
λ docker-compose build
Building test
Step 1/4 : FROM centos
--> 300e315adb2f
Step 2/4 : COPY start.sh /start.sh
--> 3a17f157e205
Step 3/4 : RUN chmod +x /start.sh
--> Running in bcddd3b1717
Removing intermediate container bcddd3b1717
--> 2187f23ace6a
Step 4/4 : CMD /start.sh
--> Running in 1389e1275e38
Removing intermediate container 1389e1275e38
--> 069a674a6bd4

Successfully built 069a674a6bd4
Successfully tagged restart-image:latest
```

Y creamos nuestro servicio con docker-compose up -d, lo cual nos creara un contenedor que se va a detener a los 5 segundos y se va a volver a iniciar cuando pasen esos 5 segundos:

Podemos observar dicho comportamiento haciendo uso del siguiente comando:

**watch -d docker ps**

```
Every 2.0s: docker ps
```

| CONTAINER ID | IMAGE         | COMMAND                | CREATED       | STATUS                      | PORTS | NAMES |
|--------------|---------------|------------------------|---------------|-----------------------------|-------|-------|
| 7753db913488 | restart-image | "/bin/sh -c /start.sh" | 3 minutes ago | Restarting (0) 1 second ago |       | test  |

Veamos ahora la opción de restart **unless-stopped** la cual nos indica que el contenedor se reiniciara y tendrá un comportamiento similar al always excepto cuando detenemos el contenedor con el comando **docker stop** de dicha manera no se reiniciara el contenedor:

```
version: '3'  
services:  
  test:  
    container_name: test  
    image: restart-image  
    build: .  
    restart: unless-stopped
```

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8ee9be55027b restart-image "/bin/sh -c /start.sh" 17 seconds ago Restarting (0) Less than a second ago

```

Si lo detenemos a mano veremos que el mismo no se reinicia:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8ee9be55027b restart-image "/bin/sh -c /start.sh" 17 seconds ago Restarting (0) Less than a second ago test

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker stop test
test

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Otra política de reinicio que nos provee docker es reiniciar **on-failure** que nos indica que el contenedor se va a reiniciar cuando el mismo tenga un error interno, por lo que en nuestro ejemplo debemos generar un error intencional para que funcione:

```
version: '3'
services:
  test:
    container_name: test
    image: restart-image
    build: .
    restart: on-failure
```

Con el exit 1 en el start.sh indicamos que paso algo mal, que tuvimos un error en el servicio:

```
#!/bin/bash

echo "Estoy vivo"
sleep 5
echo "Estoy detenido"
exit 1
```

Construimos nuevamente la imagen y levantamos el contenedor, observaremos que el mismo se reinicia cada 5 segundos porque el exit 1 simula un error y como nuestra política reinicia el contenedor on-failure el mismo se reinicia correctamente:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker-compose up -d
Recreating test ... done
Construimos nuevamente la imagen y levantamos el contenedor.

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9d10f19b1b5 restart-image "/bin/sh -c /start.sh" 8 seconds ago Up 1 second test

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9d10f19b1b5 restart-image "/bin/sh -c /start.sh" 16 seconds ago Up 3 seconds test

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\restart (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9d10f19b1b5 restart-image "/bin/sh -c /start.sh" 24 seconds ago Restarting (1) Less than a second ago test
```

Estas opciones también están disponibles en el comando **docker run** utilizándolo de la siguiente manera: docker run -dit --restart unless-stopped redis.

## Personaliza el nombre de tu proyecto en Compose

En esta sección veremos como cambiar el prefijo que esta acompañando la creación de los volúmenes y las redes.

Observamos que el prefijo por defecto que le pone docker a la creación de redes y volúmenes es el nombre de la carpeta donde se encuentra el docker-compose.yml, por ejemplo:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno (main -> origin)
λ docker-compose up -d
Creating network "varentorno_default" with the default driver
Pulling db (mysql:5.7)...
```

Para cambiar ese prefijo lo que debemos hacer es utilizar el flag **-p** en el comando **docker-compose up -d**, dicho flag hace referencia al nombre del proyecto:

```
-p, --project-name NAME      Specify an alternate project name
                               (default: directory name)
```

Por lo que si ahora creamos el mismo contenedor de la siguiente manera:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno (main -> origin)
λ docker-compose -p webtest up -d
Creating network "webtest_default" with the default driver
Creating mysql ... done
```

Veremos que el prefijo ahora llega el nombre del proyecto que definimos como input en el comando docker-compose up -d.

## Usar un nombre distinto en el docker-compose.yml

Para poder darle un nombre distinto a nuestro archivo **docker-compose.yml** lo que hacemos es utilizar el flag **-f** que hace referencia a File que nos permite indicarle un compose file alternativo al por defecto, por ejemplo podemos tener un **docker-compose-mysql.yml** y llamarlo con el flag **-f** en el comando docker-compose up -d de la siguiente manera:

```
docker-compose -f <compose file alternative name> up -d
```

### Dockerfile

```
version: '3'
services:
  db:
    image: mysql:5.7
    container_name: mysql
    ports:
      - "3306:3306"
    env_file: common.env
```

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno (main -> origin)
λ ls
common.env docker-compose.yml docker-compose-mysql.yml

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno (main -> origin)
λ docker-compose -f docker-compose-mysql.yml up -d
Creating mysql ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\VarEntorno (main -> origin)
λ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED          STATUS    PORTS          NAMES
5c3c0a0a685d   mysql:5.7  "docker-entrypoint.s..."  8 seconds ago   Up 7 seconds  0.0.0.0:3306->3306/tcp, 33060/tcp   mysql
```

## Más opciones en Compose

Las opciones que nos ofrece docker-compose se pueden consultar haciendo uso del comando **docker-compose** sin argumentos, lo cual nos listará todos los flags y comandos podemos utilizar y que nos ofrece Compose:

```
λ docker-compose
Define and run multi-container applications with Docker.

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

Options:
  -f, --file FILE           Specify an alternate compose file
                            (default: docker-compose.yml)
  -p, --project-name NAME  Specify an alternate project name
                            (default: directory name)
  -c, --context NAME        Specify a context name
  --verbose                 Show more output
  --log-level LEVEL         Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
  --no-ansi                 Do not print ANSI control characters
  -v, --version              Print version and exit
  -H, --host HOST           Daemon socket to connect to
  --assign-ip CONTENEDOR    Asignar IP a un contenedor
  --tls                      Use TLS; implied by --tlsverify
  --tlscacert CA_PATH       Trust certs signed only by this CA
  --tlscert CLIENT_CERT_PATH Path to TLS certificate file
  --tlskey TLS_KEY_PATH     Path to TLS key file
  --tlsverify                Use TLS and verify the remote
  --skip-hostname-check     Don't check the daemon's hostname against the
                            name specified in the client certificate
  --project-directory PATH  Specify an alternate working directory
                            (default: the path of the Compose file)
  --compatibility            If set, Compose will attempt to convert keys
                            in v3 files to their non-Swarm equivalent (DEPRECATED)
  --env-file PATH            Specif an alternate environment file
```

```

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

```

## Instalando WordPress + MySQL

Lo primero es definir el servicio de MySQL para que puedas ser utilizado desde wordpress por lo que en principio nuestro docker-compose será de la siguiente forma:

```

version: '3'
services:
  db:
    container_name: wp-mysql
    image: mysql:5.7
    volumes:
      - $PWD/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    ports:
      - "3306:3306"
    networks:
      - my_net
networks:
  my_net:

```

Como vemos en dicho archivo configuramos un servicio db que tiene seteado el nombre del contenedor, la imagen a utilizar, el volumen que se va a encontrar en la carpeta donde estamos corriendo este docker-compose, variables de entornos de MySQL que podemos consultar en la documentación oficial, puertos donde se va a exponer y la red donde queremos hostear el servicio.

Adicionalmente tendremos la definición de la red en la sección de **networks**.

Luego debemos configurar el servicio de wordpress que va a conectarse con nuestra base de datos, por lo que nuestro Dockerfile quedara de la siguiente manera:

```
version: '3'
services:
  db:
    container_name: wp-mysql
    image: mysql:5.7
    volumes:
      - $PWD/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    ports:
      - "3306:3306"
    networks:
      - my_net

  wp:
    container_name: wp-web
    volumes:
      - "$PWD/html:/var/www/html"
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "80:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    networks:
      - my_net
networks:
  my_net:
```

Podemos observar que tiene el servicio wp de wordpress tiene configurado un nombre de contenedor, un volumen, la imagen a utilizar, los puertos que va a utilizar http, la red a utilizar y dos atributos más bastante importantes que son:

**depends\_on** con este atributo lo que le decimos a Docker Compose es que nuestro servicio depende de los servicios que se pasen como parámetros, es decir se debe crear el servicio actual posterior a la creación de los servicios de los que depende,

**environment** aquí definimos como siempre las variables de entorno, sin embargo el valor de las mismas ahora deben corresponderse con las definidas en MySQL, por ejemplo observamos que en host se puso **nombreServicio:puerto** y esos datos son los configurados para el servicio **db** mientras que lo mismo ocurre para user y password ya que se corresponden con los seteados en el servicio de MySQL.

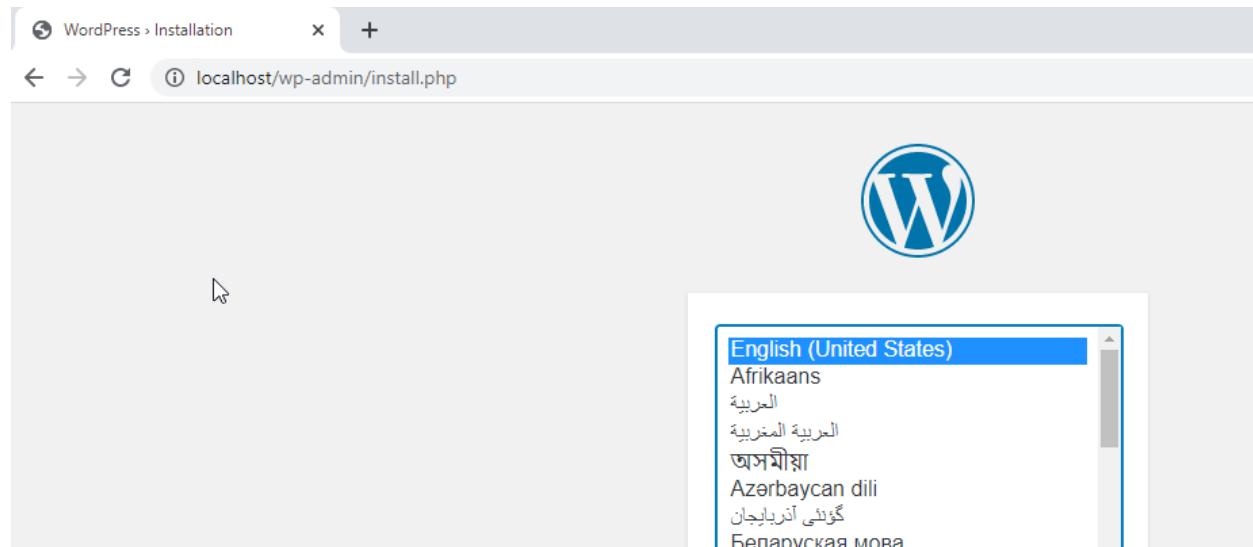
Si ejecutamos el docker-compose up -d

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\wordpress (main -> origin)
λ docker-compose up -d
WARNING: The PWD variable is not set. Defaulting to a blank string.
Creating network "wordpress_my_net" with the default driver
Creating wp-mysql ... done
Creating wp-web ... done
```

Y vemos que los dos contenedores quedaron levantados:

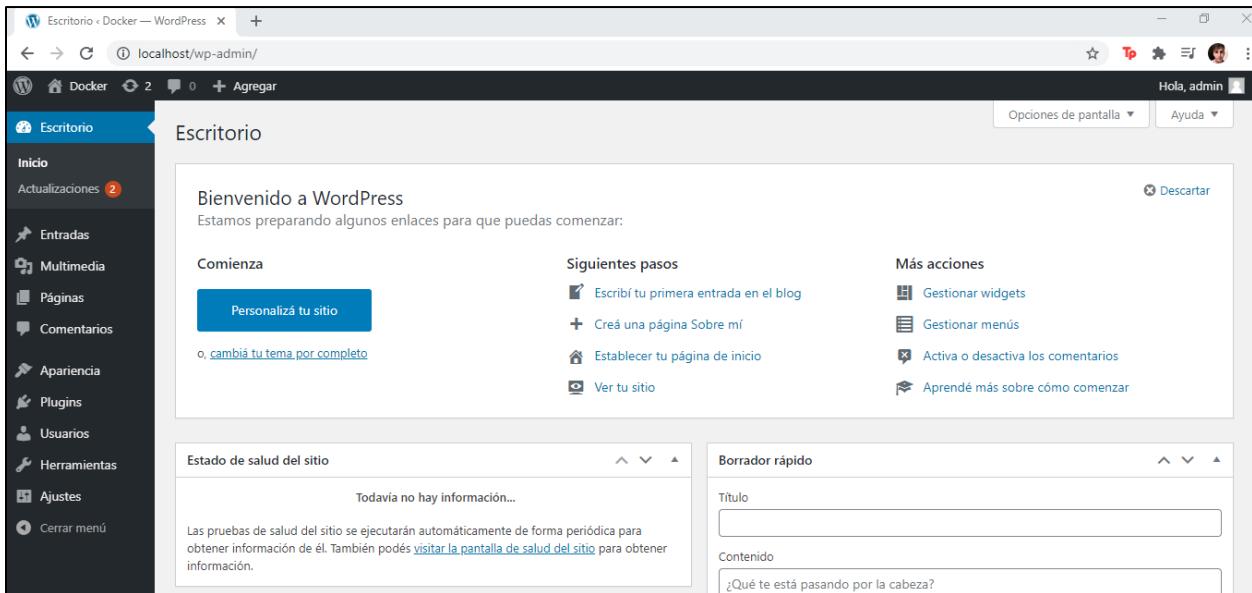
```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\wordpress (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
880ae27d4404 wordpress:latest "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0:80->80/tcp wp-web
b6eac165b7a9 mysql:5.7 "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0:3306->3306/tcp, 33060/tcp wp-mysql
```

Podemos acceder a nuestro localhost y ver nuestro servicio wordpress levantado:



**Tip:** también podemos monitorear el estado de los contenedores con el comando **docker-compose logs -f**

Configuramos wordpress y lo veremos funcionando correctamente:



## Instalando Drupal + PostgreSQL

Para instalar el servicio de Drupal + PostgreSQL tendremos un docker-compose.yml de la siguiente manera:

```
version: '3'
services:
  drupal:
    volumes:
      - drupal:/var/www/html
    image: drupal:8-apache
    ports:
      - 80:80
    networks:
      - net
  postgres:
    image: postgres:10
    environment:
      POSTGRES_PASSWORD: example
    volumes:
      - $PWD/data:/var/lib/postgresql/data
    networks:
      - net
volumes:
  drupal:
networks:
  net:
```

En dicho archivo veremos que tenemos un volumen nombrado llamado drupal que lo mapeamos al contenedor de drupal y tendremos un volumen de host seteado en el servicio de postgres. Adicionalmente tenemos una red llamada net en la cual hosteamos ambos servicios y tenemos definido las variables de entorno necesarias para el servicio de postgres, las cuales podemos consultar en la documentación oficial.

Una vez creado este archivo docker-compose levantamos los servicios con docker-compose up -d y verificamos que quede levantado:

Drupal Translation website. If you do not want this, select English.' A blue button at the bottom right says 'Save and continue'."/>

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\drupal (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
aa1039223f9c drupal:8-apache "docker-php-entrypoi..." 57 seconds ago Up 54 seconds 0.0.0.0:80->80/tcp drupal_drupal_1

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\drupal (main -> origin)
λ docker volume ls
DRIVER VOLUME NAME
local drupal_drupal
local e986d82247df71f88112bca15bc34db9b92e1296a99b33ffe22e113e88875ba5
```

Procedemos a instalar drupal y en la parte de set up database elegimos postgres y la clave es la que tenemos seteada en nuestro archivo yml.

En cuanto a la configuración avanzada debemos utilizar el servidor que creamos en nuestro docker-compose.yml también:

```
postgres:
  image: postgres
  environment:
    POSTGRES_PASSWORD: example
```

Nuestra configuración en drupal debe quedar algo así:

### Configuración de la base de datos

**Tipo de base de datos \***

MySQL, MariaDB, Percona Server o equivalente  
 SQLite  
 PostgreSQL

**Nombre de la base de datos \***

postgres

**Nombre de usuario de la base de datos \***

postgres

**Contraseña de la base de datos**

.....

**OPCIONES AVANZADAS**

**Servidor \***

postgres

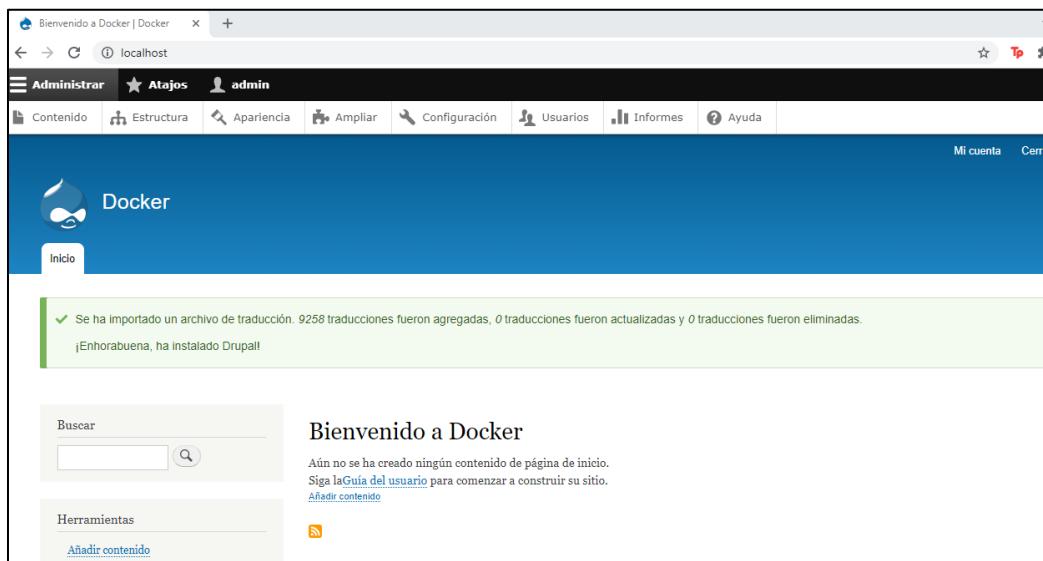
**Número de puerto**

5432

**Prefijo del nombre de la tabla**

Si hay más de una aplicación compartiendo esta base de datos, un prefijo de tablas único – como *drupal\_* – evitará conflictos.

Se va a instalar drupal y una vez instalado nos va a solicitar setear una configuración inicial que involucra nombre del sitio, usuario y contraseña, etc una vez finaliza la instalación correctamente veremos nuestro sitio de drupal correctamente configurado:



## Instalando PrestaShop + MySQL

Para instalar el servicio de PrestaShop + MySQL tendremos un docker-compose.yml de la siguiente manera:

```
version: '3'

services:
  db:
    container_name: ps-mysql
    image: mysql:5.7
    volumes:
      - $PWD/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: ps
      MYSQL_USER: ps
      MYSQL_PASSWORD: ps
    ports:
      - "3306:3306"
    networks:
      - my_net
  ps:
    container_name: ps-web
    volumes:
      - "$PWD/html:/var/www/html"
    depends_on:
      - WORDPRESS_DB_HOST
    image: prestashop/prestashop
    ports:
      - "80:80"
    environment:
      DB_SERVER: db
      DB_USER: ps
      DB_PASSWD: ps
      DB_NAME: ps
    networks:
      - my_net
networks:
  my_net:
```

En el mismo tenemos definidos los dos servicios, en el caso de db (MySQL) tenemos definido el nombre del contenedor, la imagen de mysql, el volumen, variables de entorno, puerto a exponer por el contenedor y la red a la cual pertenece. Mientras que el servicio de prestashop, tiene definido el nombre del contenedor, la imagen de

prestashop, el volumen, las variables de entorno que se deben corresponder con los datos del servicio de MySQL, la red a la que pertenece, puerto a exponer por el contenedor y la dependencia del servicio **db**.

También tenemos definido la declaración de la red **my\_net**.

Una vez armado este docker-compose debemos iniciar los servicios con el comando docker-compose up -d:

```
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
029fc0e4c68b prestashop/prestashop "docker-php-entrypoi..." 2 minutes ago Up 2 minutes 0.0.0.0:80->80/tcp ps-web
dc62ee1e24a3 mysql:5.7 "docker-entrypoint.s..." 2 minutes ago Up 2 minutes 0.0.0.0:3306->3306/tcp, 33060/tcp ps-mysql
```

Una vez arriba el servicio de prestashop accedemos al localhost y configuramos la instalación:

The screenshot shows the initial step of the PrestaShop 1.7.7.0 Installation Assistant. At the top, there's a navigation bar with links for Forum, Support, Documentation, and Blog. Below the header, the title 'Installation Assistant' is displayed, followed by a progress bar consisting of six circles, with the first one filled. On the left, a sidebar lists steps: 'Choose your language', 'License agreements', 'System compatibility', 'Store information', 'System configuration', and 'Store installation'. The main content area starts with a heading 'Welcome to the PrestaShop 1.7.7.0 Installer'. It explains that installing PrestaShop is quick and easy, mentioning over 250,000 merchants. It also provides links for help and documentation. A dropdown menu shows 'Español (Spanish)'. Below it, a note states that the language selection applies only to the Installation Assistant and that users can choose from over 60 translations once the store is installed. In the bottom right corner, a blue 'Next' button is visible.

## Instalando Joomla + MySQL

Para instalar el servicio de Joomla + MySQL tendremos un docker-compose.yml de la siguiente manera:

```
version: '3'
services:
  web:
    volumes:
      - $PWD/html:/var/www/html
    image: joomla
    ports:
      - 8080:80
    environment:
      JOOMLA_DB_HOST: db
      JOOMLA_DB_PASSWORD: example
      JOOMLA_DB_USER: joomla
      JOOMLA_DB_NAME: joomla
  db:
    image: mysql:5.7
    volumes:
      - $PWD/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_USER: joomla
      MYSQL_DATABASE: joomla
      MYSQL_PASSWORD: joomla
```

Donde tendremos definidos dos servicios, el de la base de datos MySQL (db) donde definimos la imagen, volumen y las variables de entorno correspondientes. Mientras que en el servicio web (Joomla) definimos la imagen, puerto, volumen y las variables de entorno correspondiente consultando la imagen oficial de Joomla en docker hub.

Una vez construido este docker-compose.yml lo utilizamos para iniciar los servicios y configurar nuestro servicio web Joomla.

## Instalando Reaction Ecommerce – NodeJS + MongoDB

Para instalar el servicio de Reaction Ecommerce – NodeJS + MongoDB tendremos un docker-compose.yml de la siguiente manera:

```
version: '3'
services:
  reaction:
    image: reactioncommerce/reaction:latest
    networks:
      - net
    depends_on:
      - mongo
    ports:
      - "3000:3000"
    environment:
      ROOT_URL: "http://localhost"
      MONGO_URL: "mongodb://mongo:27017/reaction"

  mongo:
    image: mongo
    volumes:
      - $PWD/data:/data/db
    networks:
      - net
networks:
  net:
```

En el cual tenemos un servicio de base de datos no relacional como lo es Mongo con su imagen correspondiente, su mapeo del volumen y a que red pertenece. Y además, un servicio con una imagen de reactioncommerce, configuración de red y mapeo de puertos del contenedor, como así también la dependencia de mongo y las variables de entornos que se deben setear, las cuales fueron obtenidas de la imagen oficial de docker hub.

Una vez construido el archivo docker-compose.yml procedemos a iniciar los servicios con el comando docker-compose up -d:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\react (main -> origin)
λ docker-compose up -d
WARNING: The PWD variable is not set. Defaulting to a blank string.
Creating network "react_net" with the default driver
Creating react_mongo_1 ... done
Creating react_reaction_1 ... done

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\react (main -> origin)
λ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
93f14f9088db reactioncommerce/reaction:latest "node main.js" 28 seconds ago Up 27 seconds 0.0.0.0:3000->3000/tcp react_reaction_1
74d88723e4f1 mongo "docker-entrypoint.s..." 29 seconds ago Up 27 seconds 27017/tcp react_mongo_1
```



## Sign in

**Email address**

admin@localhost

**Password**

.....

**Sign in**

[Reset password](#)

[Register](#)

## Instalando Guacamole

Guacamole es un servicio en html5 que nos permite conectarnos a escritorios remotos de nuestro navegador. Docker-compose.yml:

```
version: '3'
services:
  db:
    container_name: guacamole-db
    networks:
      - net
    image: mysql:5.7
    volumes:
      - ./conf/initdb.sql:/docker-entrypoint-initdb.d/initdb.sql
      - ./data:/var/lib/mysql
    env_file: .env
  daemon:
    container_name: guacamole-daemon
    networks:
      - net
    image: guacamole/guacd
    depends_on:
      - db
  web:
    container_name: guacamole-web
    networks:
      - net
    image: guacamole/guacamole
    env_file: .env
    depends_on:
      - daemon
  proxy:
    container_name: guacamole-proxy
    hostname: guacamole-proxy
    networks:
      - net
    image: nginx
    ports:
      - "80:80"
    restart: always
    volumes:
      - $PWD/conf/nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - web
networks:
  net:
```

Una vez configurado dicho docker compose file debemos iniciar los servicios con el comando docker-compose up -d:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\apps\guacamole (main -> origin)
λ docker-compose up -d
Creating network "guacamole_net" with the default driver
Creating guacamole-db ... done
Creating guacamole-daemon ... done
Creating guacamole-web ... done
Creating guacamole-proxy ... done
```

| λ docker ps | CONTAINER ID | IMAGE               | COMMAND                  | CREATED            | STATUS                           | PORTS               | NAMES            |
|-------------|--------------|---------------------|--------------------------|--------------------|----------------------------------|---------------------|------------------|
|             | 5b90ce658725 | nginx               | "/docker-entrypoint..."  | 58 seconds ago     | Up 55 seconds                    | 0.0.0.0:80->80/tcp  | guacamole-proxy  |
|             | 9d8fd8df1a00 | guacamole/guacamole | "/opt/guacamole/bin/..." | About a minute ago | Up 57 seconds                    | 8080/tcp            | guacamole-web    |
|             | 20b89f2880a1 | guacamole/guacd     | "/bin/sh -c '/usr/lo..." | About a minute ago | Up 58 seconds (health: starting) | 4822/tcp            | guacamole-daemon |
|             | ae34d819214f | mysql:5.7           | "docker-entrypoint.s..." | About a minute ago | Up 58 seconds                    | 3306/tcp, 33060/tcp | guacamole-db     |

## Instalando Zabbix en Compose

Zabbix es una herramienta para monitorear servidores.

Para instalar el servicio de Zabbix tendremos un docker-compose.yml de la siguiente manera:

```
version: '3'
services:
  zabbix:
    container_name: zabbix-web
    image: zabbix
    build: .
    volumes:
      - "$PWD/html:/usr/share/zabbix"
    ports:
      - "80:80"
    networks:
      - net
  db:
    container_name: zabbix-db
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_USER: zabbix
      MYSQL_PASSWORD: zabbix
      MYSQL_DATABASE: zabbix
    volumes:
      - "$PWD/data:/var/lib/mysql"
      - "$PWD/conf/create.sql:/docker-entrypoint.initdb.d/zabbix.sql"
    ports:
      - "3306:3306"
    networks:
```

```
- net  
networks:  
  net:
```

Como observamos primero tenemos el servicio web de Zabbix el cual va a tener una imagen construida por nosotros de zabbix con un Dockerfile:

```
FROM centos:7  
  
ENV ZABBIX_REPO http://repo.zabbix.com/zabbix/3.4/rhel/7/x86_64/zabbix-release-  
3.4-1.el7.centos.noarch.rpm  
  
RUN  
    yum -y install $ZABBIX_REPO && \  
    yum -y install  
        zabbix-get \  
        zabbix-server-mysql \  
        zabbix-web-mysql \  
        zabbix-agent  
  
EXPOSE 80 443  
  
COPY ./bin/start.sh /start.sh  
  
COPY ./conf/zabbix-http.conf /etc/httpd/conf.d/zabbix.conf  
  
COPY ./conf/zabbix-server.conf /etc/zabbix/zabbix_server.conf  
  
COPY ./conf/zabbix-conf.conf /etc/zabbix/web/zabbix.conf.php  
  
VOLUME /usr/share/zabbix /var/log/httpd  
  
RUN chmod +x /start.sh  
  
CMD /start.sh
```

Va a tener montado un volumen de host que montaba en el contenedor la carpeta html que tenemos en nuestro directorio. Mapeo de puertos y la red net seteada.

También tendremos un servicio db, que va a tener la imagen de mysql, con sus credenciales como variables de entorno. Este servicio va a tener montado dos volúmenes uno de data y otro de conf que va a tener los scripts iniciales que se

importaran en la base de datos cuando se levante la primera vez el servicio. También se mapearon los puertos y unimos nuestra base de datos a nuestra red net.

El siguiente paso es levantar el servicio con docker-compose up -d y revisar que queden los dos servicios levantados con un docker ps y acceder a nuestro localhost/zabbix donde veremos nuestro servicio de monitoreo corriendo correctamente.

## Docker Registry

### Crea tu propio Docker Registry

El Docker Registry es un repositorio donde nosotros podemos subir nuestras imágenes.

Por ejemplo, cuando nosotros hacemos un **docker pull hello-world** lo que hace docker es conectarse al docker registry y descargando esas imágenes.

Si queremos crear un registry local debemos ejecutar el siguiente comando:

```
docker run -d -p 5000:5000 --name registry -v dataImages:/var/lib/registry  
registry:2
```

The screenshot shows a Windows Command Prompt window with the following text:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\ejercicio (main -> origin)
λ docker run -d -p 5000:5000 --name registry -v dataImages:/var/lib/registry registry:2
807eeab1728a056d074a99ea4cf6b994ce93b5f8bba196579e7f9768aca915e0

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\ejercicio (main -> origin)
λ docker ps
CONTAINER ID   IMAGE      COMMAND       CREATED          STATUS          PORTS          NAMES
807eeab1728a   registry:2   "/entrypoint.sh /etc..."   2 minutes ago   Up 2 minutes   0.0.0.0:5000->5000/tcp   registry

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-compose\ejercicio (main -> origin)
```

### Sube tus imágenes

Para subir una imagen al registry primero debemos tagearla lo cual hacemos con el comando **docker tag**, de la siguiente forma:

```
docker tag <nombreActualImagen>:<tag> <tagRegistry>
```

Por ejemplo:

The screenshot shows a Windows Command Prompt window with the following text:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker pull hello-world:latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:1a523af650137b8accdaed439c17d684df61ee4d74feac151b5b337bd29e7ec
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

**C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)**

```
λ docker tag hello-world:latest localhost:5000/hello-world
```

**C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)**

```
λ docker images
REPOSITORY          TAG      IMAGE ID      CREATED          SIZE
exercise           latest   24e0e6d86041  47 minutes ago  348MB
phpmyadmin/phpmyadmin latest   badddfc395a5  5 days ago    469MB
mysql              5.7     f07dfa83b528  2 weeks ago   448MB
registry            2       678dfa38fcfa  2 weeks ago   26.2MB
httpd              latest   dd85cd9bb9987  3 weeks ago   138MB
centos             latest   300e315adb2f  4 weeks ago   209MB
centos             7       8652b9f0cb4c  7 weeks ago   204MB
hello-world         latest   bf756fb1ae65  12 months ago  13.3kB
localhost:5000/hello-world latest   bf756fb1ae65  12 months ago  13.3kB
```

**C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)**

Una vez tageada la imagen, debemos subirla a nuestro registry lo cual hacemos con el comando **docker push** de la siguiente forma:

**docker push <imagenTageada>**

Ejemplo:

**docker push localhost:5000/hello-world**

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker push localhost:5000/hello-world
Using default tag: latest
The push refers to repository [localhost:5000/hello-world]
9c27e219663c: Pushed
latest: digest: sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb826903b02cfa0da11c82cbc042 size: 525

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ ls dataImages\
docker/

C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ ls dataImages\docker\registry\v2repositories\hello-world\
_layers/ _manifests/ _uploads/
```

Procedemos a eliminar la imagen:

```
λ docker rmi hello-world localhost:5000/hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:1a523af650137b8accdaed439c17d684df61ee4d74feac151b5b337bd29e7ec
Untagged: localhost:5000/hello-world:latest
Untagged: localhost:5000/hello-world@sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb826903b02cfa0da11c82cbc042
Deleted: sha256:bf756fb1ae65adf866bd8c456593cd24beb6a0a061dedf42b26a993176745f6b
Deleted: sha256:9c27e219663c25e0f28493790cc0b88bc973ba3b1686355f221c38a36978ac63
```

Y ahora vamos a bajar la imagen pero de nuestro registry, para hacer esto hacemos un **docker pull <nombrelImagenTageada>**

Ejemplo:

**docker pull localhost:5000/hello-world**

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker pull localhost:5000/hello-world
Using default tag: latest
latest: Pulling from hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb826903b02cfa0da11c82cbc042
Status: Downloaded newer image for localhost:5000/hello-world:latest
localhost:5000/hello-world:latest
```

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
exercise            latest   24e0e6d86041  55 minutes ago  348MB
phpmyadmin/phpmyadmin   latest   badddfc395a5  5 days ago    469MB
mysql               5.7     f07dfa83b528  2 weeks ago   448MB
registry             2       678dfa38fcfa  2 weeks ago   26.2MB
httpd                latest   dd85cdbb9987  3 weeks ago   138MB
centos              latest   300e315adb2f  4 weeks ago   209MB
centos              7       8652b9f0cb4c  7 weeks ago   204MB
localhost:5000/hello-world  latest   bf756fb1ae65  12 months ago  13.3kB
```

## Comparte las imágenes en tu red

Para compartir una imagen con nuestra red debemos conocer nuestra ip la cual podemos obtener con los comando ipconfig o ip a | less

```
Wireless LAN adapter Wi-Fi 2: 169.254.255.255 scope global dynamic
    valid_lft forever preferred_lft forever
    Connection-specific DNS Suffix . : fibertel.com.ar
    Link-local IPv6 Address . . . . . : fe80::e93d:cbd0:301b:7d46%7
    IPv4 Address. . . . . : 192.168.0.224
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.0.1
```

```
7: wifi0: <BROADCAST,MULTICAST,UP> mtu 1500 group default qlen 1
    link/ieee802.11 9c:fc:e8:a2:2f:d3
    inet 192.168.0.224/24 brd 192.168.0.255 scope global dynamic
        valid_lft 2995sec preferred_lft 2995sec
    inet6 fe80::e93d:cbd0:301b:7d46%64 scope link dynamic
        valid_lft forever preferred_lft forever
8: wifi1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 group default qlen 1
```

Si yo quiero subir una imagen con un docker push a dicha ip no voy a poder debido a que estoy queriendo subir la misma a un docker registry inseguro, es decir que no tiene autenticación:

```
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
Digest: sha256:1a523af650137b8accdaed439c17d684df61ee4d74feac151b5b337bd29e7eec
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest

Limitar recursos en contenedores (Compose...)
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker tag hello-world 192.168.0.224/hello-world
Personaliza el nombre de tu proyecto en Compose...
C:\Users\fmediotte\Desktop\Facu\Udemy\Docker\DockerRepo\docker-registry (main -> origin)
λ docker push 192.168.0.224/hello-world
Using default tag: latest
The push refers to repository [192.168.0.224/hello-world]
Get https://192.168.0.224/v2/: dial tcp 192.168.0.224:443: connect: connection refused
```

Para que esto funcione en nuestra máquina debemos ir a /lib/systemd/system/docker.service y modificar la línea que dice ExecStart agregándole lo siguiente:

--insecure-registry <ip>:<puertoRegistryLocal>

Ejemplo:

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --insecure-registry 192.168.0.224:5000
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

Reiniciamos los demonios del sistema de la siguiente forma:

**sudo systemctl daemon-reload**

Y reiniciamos el servicio de docker con el comando:

**sudo systemctl restart docker**

Una vez el servicio docker se reinicie, vamos a iniciar nuestro registry nuevamente:

**docker start registry**

Y vamos a intentar realizar nuevamente el docker push:

**docker push 192.168.0.224/hello-world**

Y se subiría la misma correctamente al registry local que definimos.

De esta manera nuestros compañeros pueden descargar la imagen utilizando el comando:

**docker pull <ip>/<imagenTageada>**