# DOCKER

## 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 vuelves realidad.

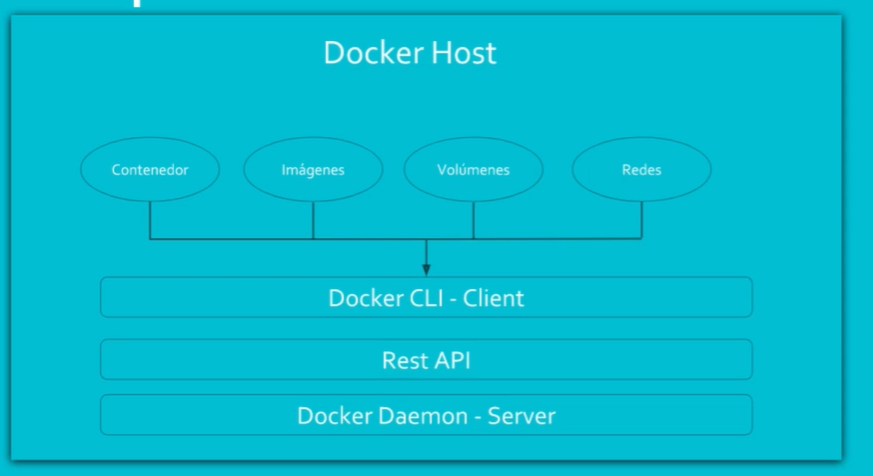
Las imágenes son muy portables y te permiten desplegar y escalar aplicaciones, como así también destruir y recrear imágenes de una manera muy fácil y rápido.

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 tipiciamente 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.

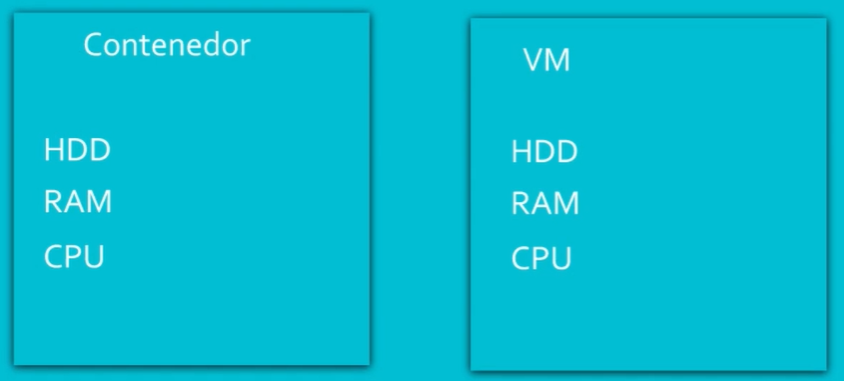
En resumen, una imagen es un paquete con las configuración y un contenedor es una capa adicional de ejecución que inicia todo lo que está definido en la imagen.



## 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 cual 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

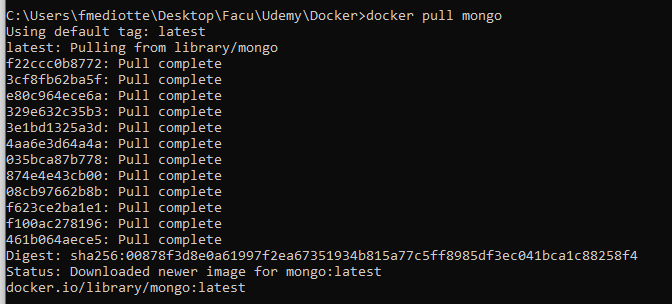
### **Imágenes oficiales**

**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.

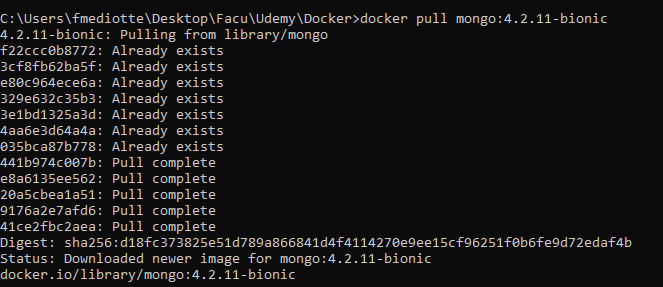
Si realizamos un Docker pull mongo por ejemplo estaremos descargando la imagen de mongo de Docker hub a nuestra máquina local, por defecto si no le especificamos ningún tag se va a descargar la última imagen que mongo haya subido a docker hub.



Si quisiéramos 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

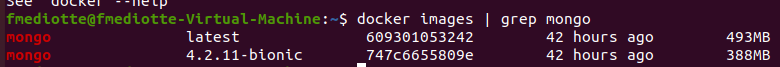
(Como ya habíamos descargado mongo previamente Docker detecta que ya tiene varias capas que ya existen localmente, por lo que solo modifica las partes que son diferentes, esto se llama **Copy on Write**)



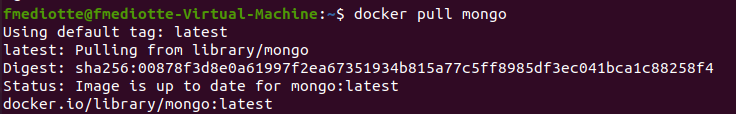
Estos tags los obtenemos de la página oficial de docker hub. En caso de que la versión que queremos descargar no está disponible podríamos nosotros crear una imagen personalizada.

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.

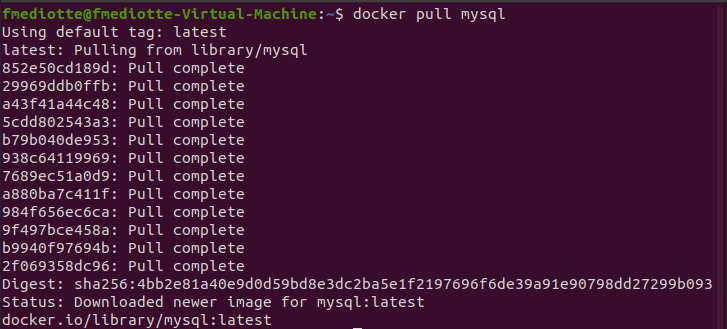


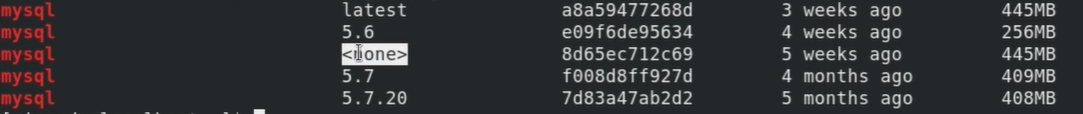
Si intentamos descargar nuevamente la imagen de mongo, lo que va a pasar es que nos va a aparecer un mensaje por consola avisándonos de que ya tenemos dicha imagen descargada y que la misma no presenta cambios para descargar.



Veamos otro ejemplo con mysql.

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



¿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 la imágenes que ya no debería usarse como huérfanas quitándole la referencia al tag:  


### **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 queramos 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**

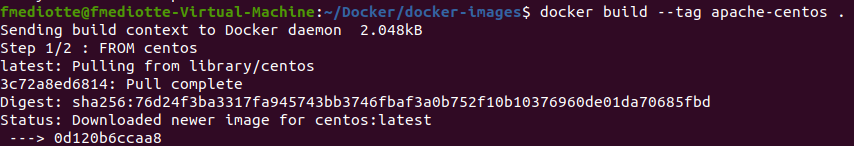
Para construir nuestra imagen en base al Docker file que generamos, se debe ejecutar el comando docker build con un nombre de tag que indica el nombre de la imagen resultante.

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

Ejemplo:

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

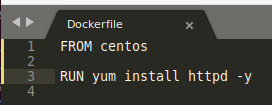
El proceso como tenemos el docker file actualmente vamos a empezar a correr los pasos que definimos en el docker file, por ejemplo:

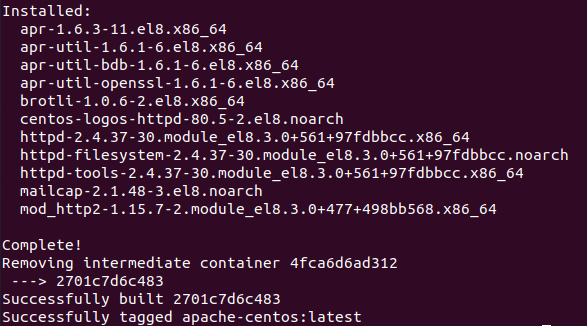


La primer capa Step 1/2 instala la imagen de centos por medio de un docker pull centos.

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:



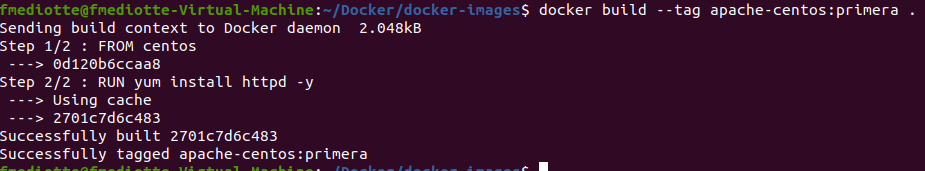
Una vez ejecutado nuevamente el comando docker build, se descargarán los paquetes necesarios para instalar apache dando como resultado un complete successfully, listando todo lo instalado:

En Windows:



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



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:

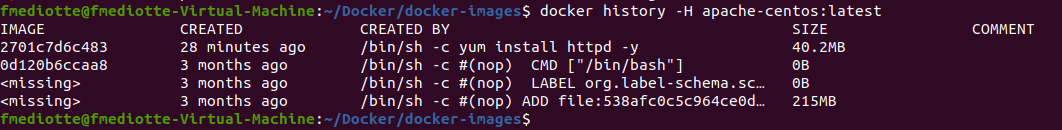
Si realizamos un docker images nuevamente veremos que tenemos 2 imágenes que en realidad son la misma imagen pero que tienen una etiqueta diferente pero que fueron creadas desde el mismo dockerfile:



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**

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

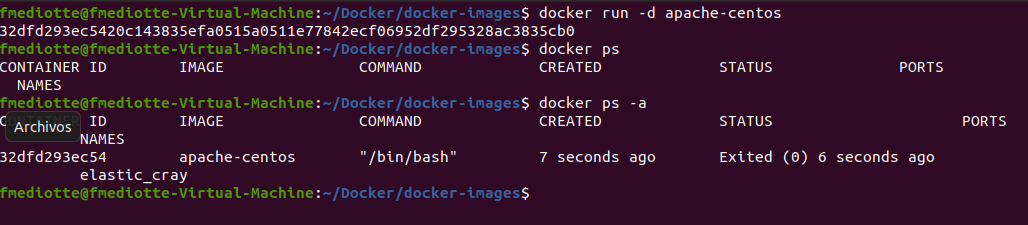
### **Creando un contenedor con nuestra imagen**

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**



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**

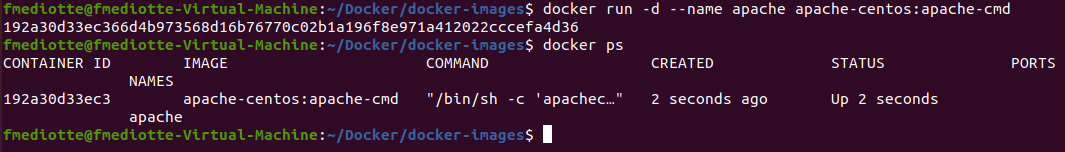
Este comando lo que hace es ejecutar el servicio de apache en primer plano.

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 .**



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

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

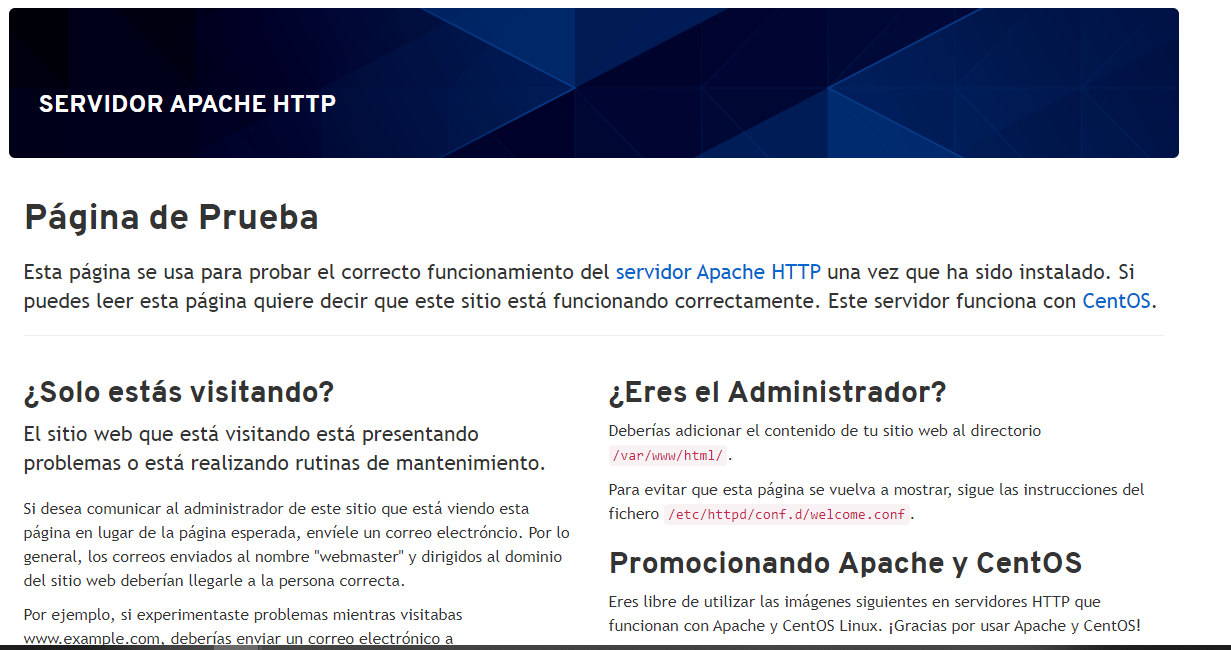
**docker rm -fv <nombre-contenedor>**

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 enviemos 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”

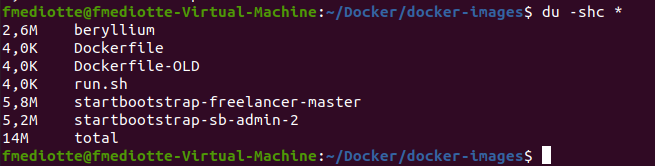
apachectl -DFOREGROUND

y en nuestro Dockerfile llamarlo de la siguiente manera:

COPY run.sh /run.sh

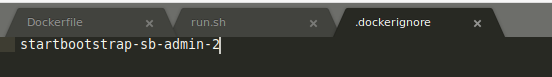
CMD sh /run.sh

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.

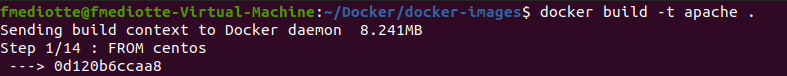




Si queremos por ejemplo ignorar uno de los archivos de dicho directorio porque no lo queremos incluir en nuestra imagen lo que se hace es poner el nombre del mismo en un archivo .dockerignore:



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



#### **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**

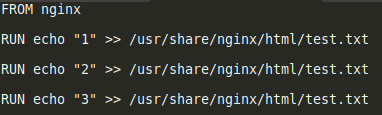
* La imagen o el servicio que está instalado debe ser efímero, es decir que se debe poder destruir con gran facilidad.
* Debería haber un solo servicio por contenedor o un solo servicio instalado por imagen.
* Si queremos excluir archivos que no queremos que estén en el contexto de Docker, cuando vayamos 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:

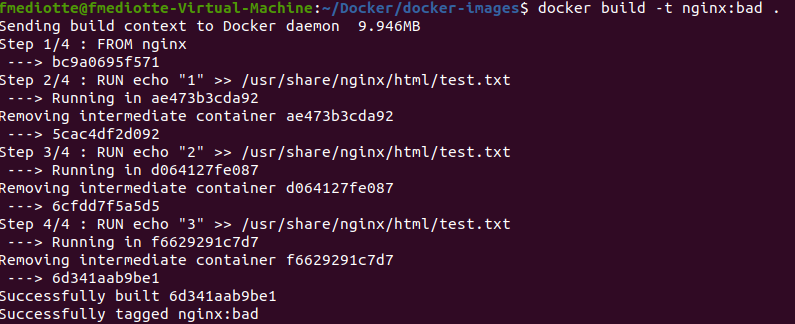
Para reducir la cantidad de capas que tiene la imagen se puede hacer la misma instrucción en una sola línea concatenando los comandos en el caso de que sea posible, por ejemplo:

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

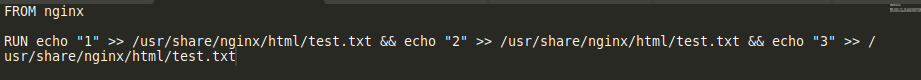
**Dockerfile**



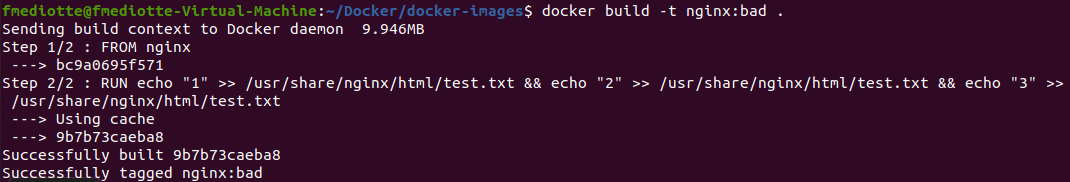
**Docker build**



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



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



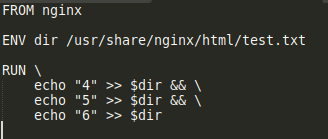
Esto se puede mejorar aún más utilizando buenas prácticas separando las líneas por un *escape* (“\”), dejando el Dockerfile de la siguiente manera:



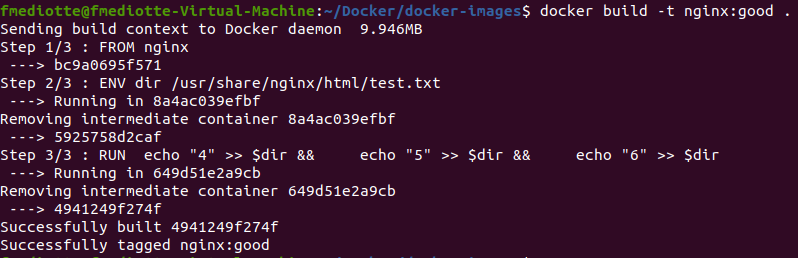
De esta forma lo que se hace es mejorar el ámbito visual, todas las tareas quedan dentro de la misma capa, el backslash significa que continua todo en la misma línea.

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

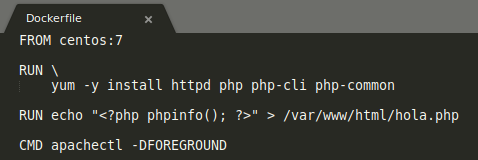
**Dockerfile:**



**Docker build:**



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



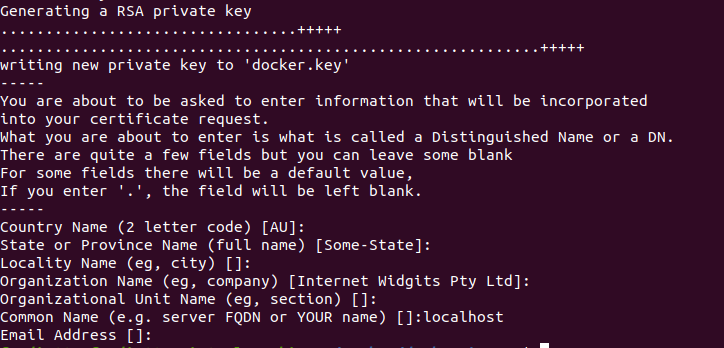
El comando yum se encarga de instalar httpd y el php ya que soporta varios argumentos de entrada. Una vez corrido el docker build con su tag correspondiente yum también se encarga de instalar las dependencias necesarias que necesite la imagen.

**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.



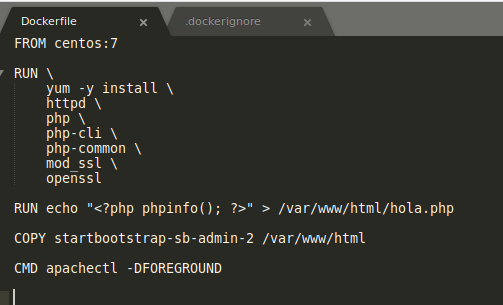
Este comando nos va a generar un par de archivos que son:

* Docker.crt
* Docker.key

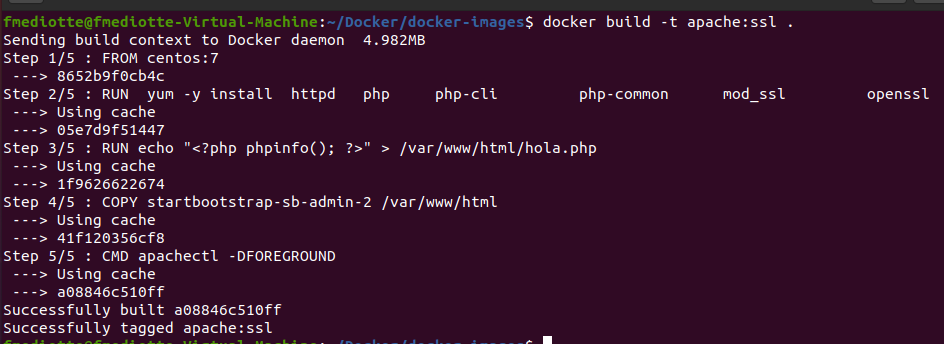
Estos archivos son los certificados SSL que necesitamos instalar en nuestro web server.

En apache se instalan de la siguiente manera: [LINK](https://www.techrepublic.com/article/how-to-enable-https-on-apache-centos/)

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



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



Para configurar una conexión segura por https y ssl debemos configurar un vhost para apache, su formato lo podemos obtener desde el siguiente [link](https://www.digicert.com/kb/ssl-support/apache-multiple-ssl-certificates-using-sni.htm), y sería algo así:

**ssl.conf**

<VirtualHost \*:443>

ServerName localhost

DocumentRoot /var/www/html

SSLEngine on

SSLCertificateFile **/docker.crt**

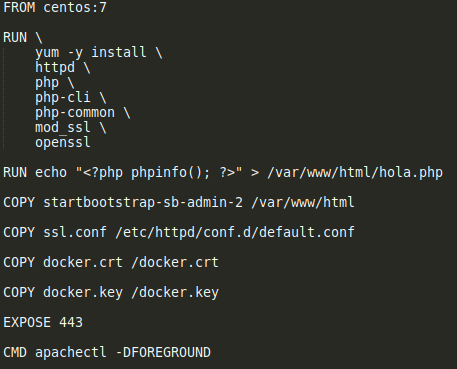
SSLCertificateKeyFile **/docker.key**

</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:

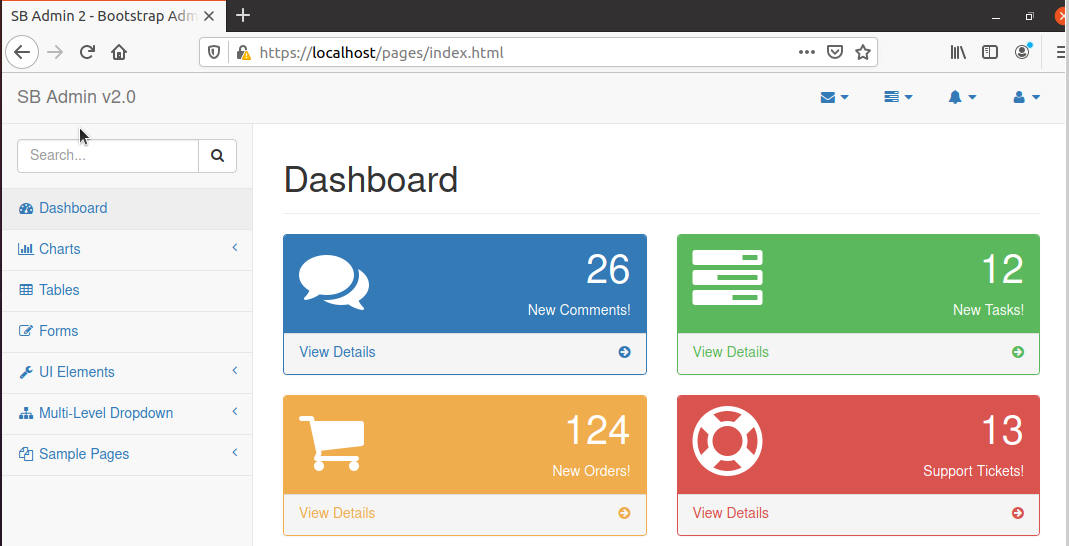


Luego solo resta correr los comandos docker build y docker run para poder acceder a nuestro sitio de forma segura.

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:

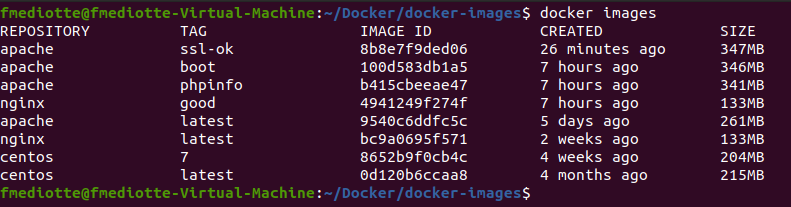
**docker run -d -p 443:443 apache:ssl-ok**

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



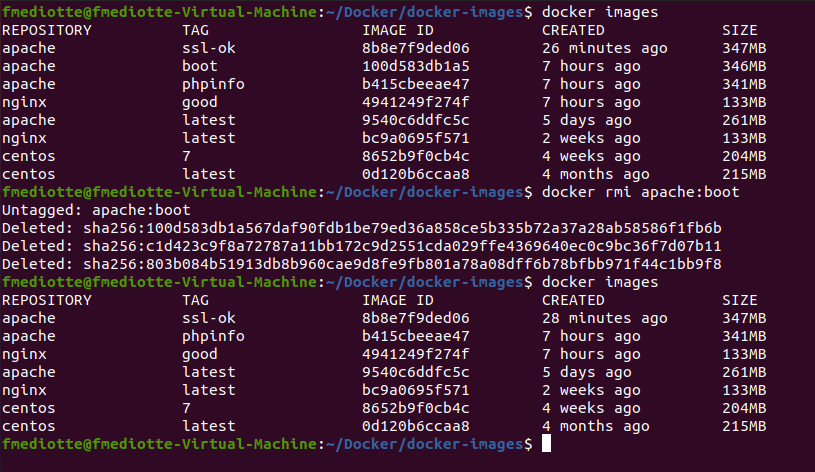
### **Eliminar imágenes**

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



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

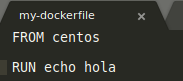
**Docker rmi argumento**, donde argumento puede ser el id de la imagen o el nombre + el tag, por ejemplo queremos eliminar la imagen apache:boot, por lo que corremos el comando **docker rmi apache:boot** y si listamos nuevamente las imágenes ya no va a estar disponible la misma para su uso.

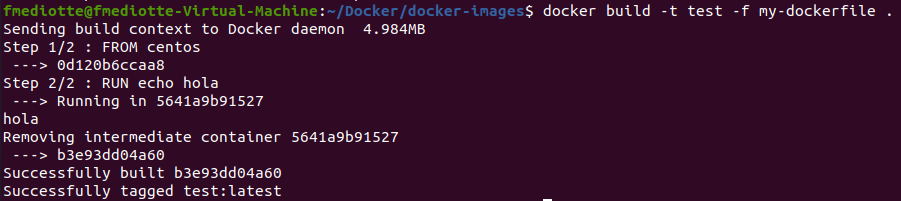


### **Cambiar el nombre del Dockerfile**

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:

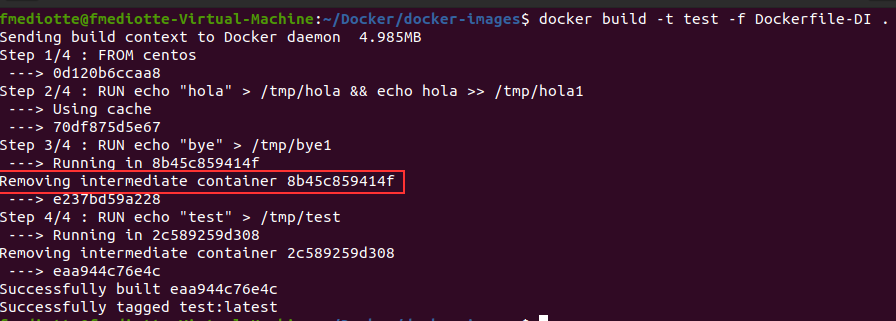
**docker build -t test -f my-dockerfile .**

****



### **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 modifico 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:

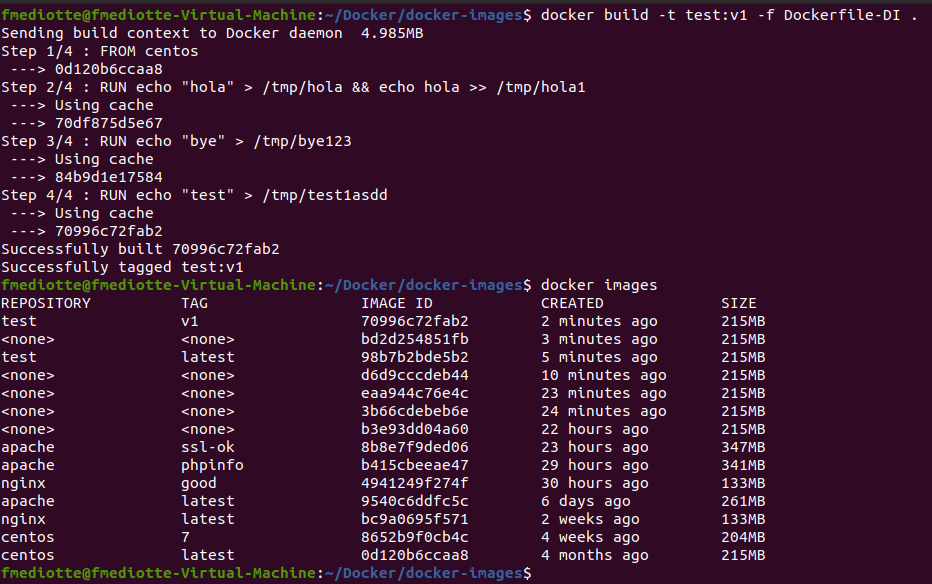


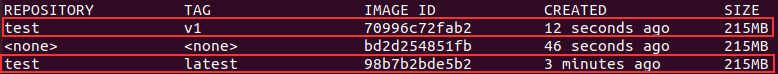


Esto sucede porque las capas de la imagen son de **SOLO LECTURA**, por lo tanto, las capas no pueden modificarse por lo que al modificar un Dockerfile, docker crea otra imagen totalmente nueva y le quita la referencia a la imagen anterior.

**¿Cómo podríamos evitar este problema?**

Definiendo tags en las imágenes:





Podríamos eliminar estas imágenes huérfanas mediante los siguiente comandos:

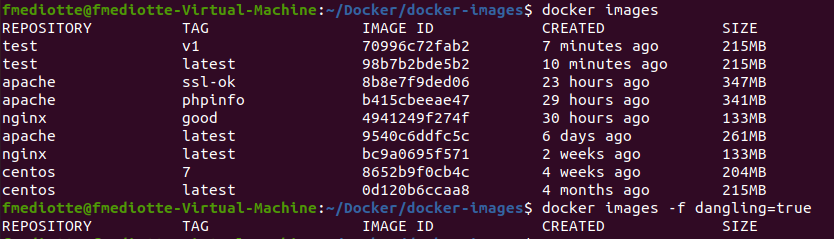
**docker images -f dangling = true**

**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:

****

### **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

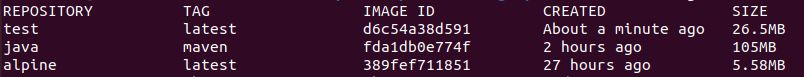
RUN fallocate -l 30M /opt/test3

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 necesitemos 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)



## 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**

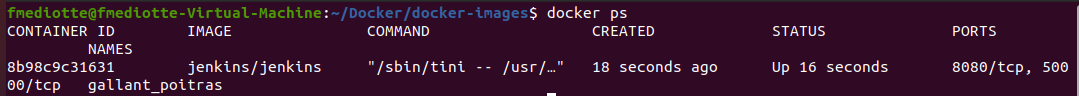
Uno de los requisitos para crear un contenedor es tener una imagen.

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:

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.

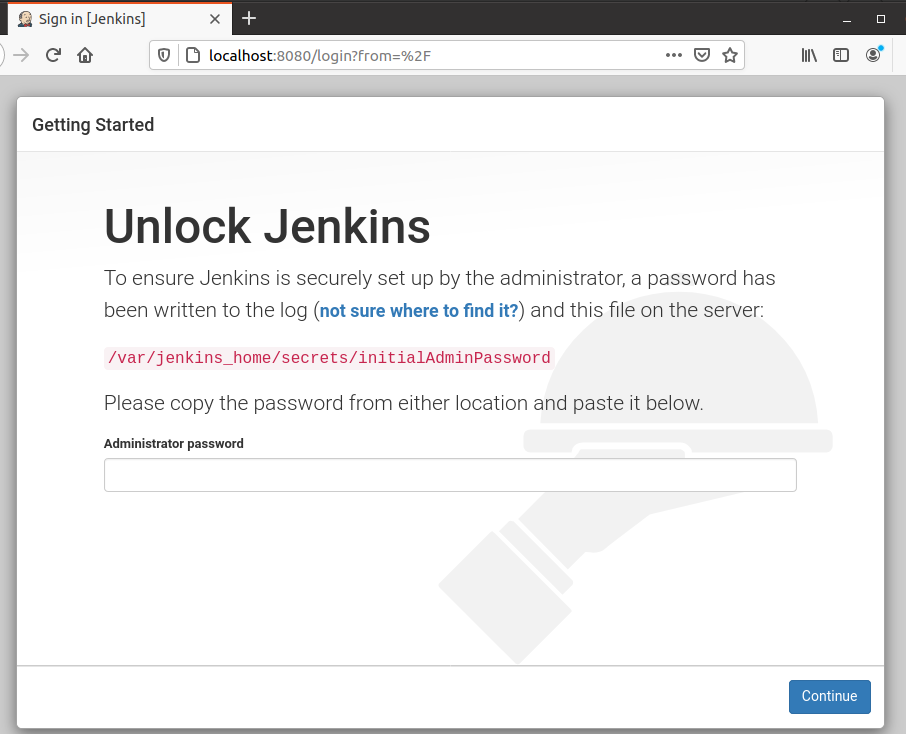
Para poder ver nuestra instancia en el navegador tenemos que hacer un **mapeo de puertos o ports mapping.**

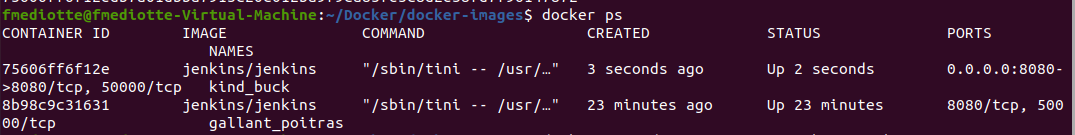
Para poder hacer este mapeo se debe utilizar el comando docker run con el flag **-p** el cuál nos permite hacer ese mapeo.

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**

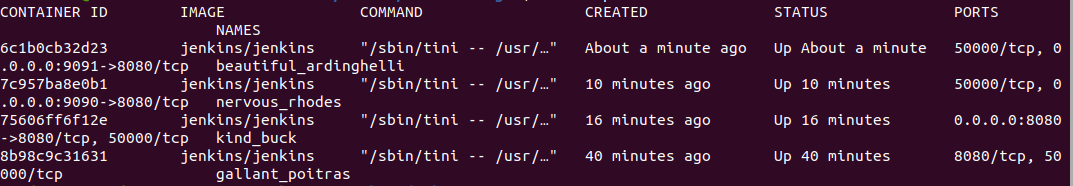
Lo que significa esto es que todas las interfaces de nuestra máquina en el puerto 8080 están siendo mapeadas del contenedor.

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.



Si queremos borrar estos contenedores podemos utilizar el comando:

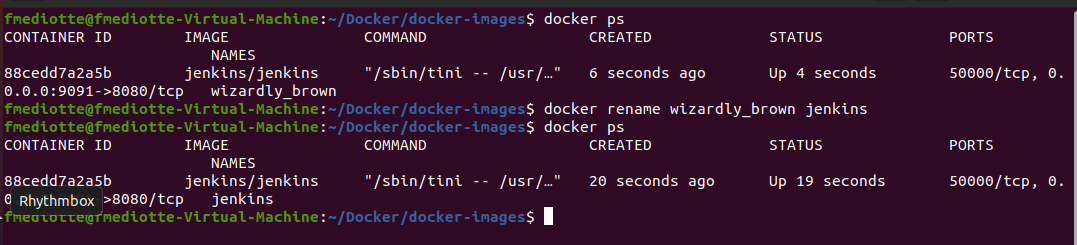
**docker rm -f <listNamesContainer>**

Y poder levantarlo 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**

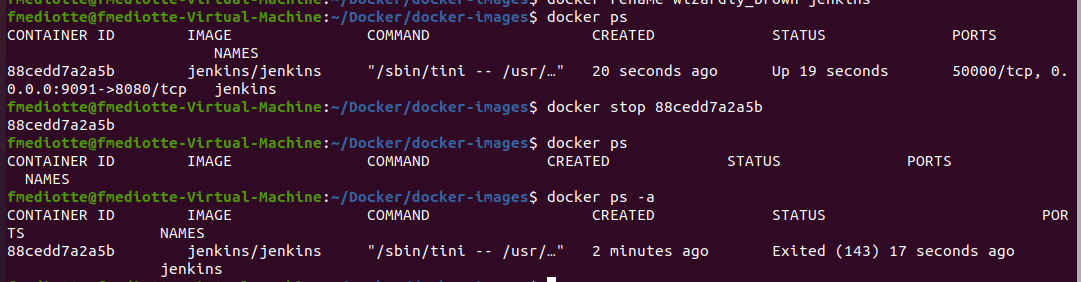
Si por algún motivo nosotros queremos renombrar un contenedor que tenemos creado podemos hacer uso del comando:

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

****

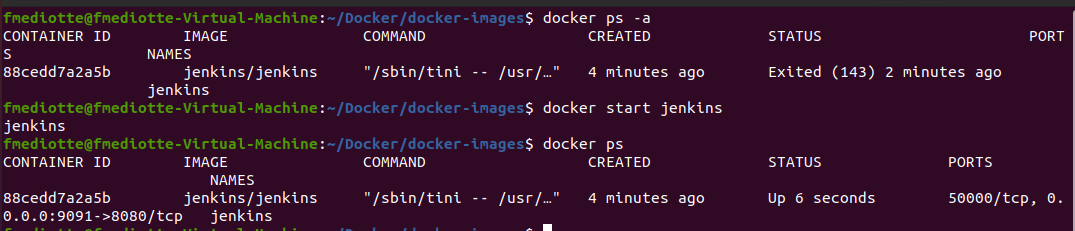
Otra de las operaciones con contenedores que podemos realizar es detener el contenedor, pero no eliminarlo, para poder lograr esto debemos utilizar el comando:

**docker stop <containerID or containerName>**

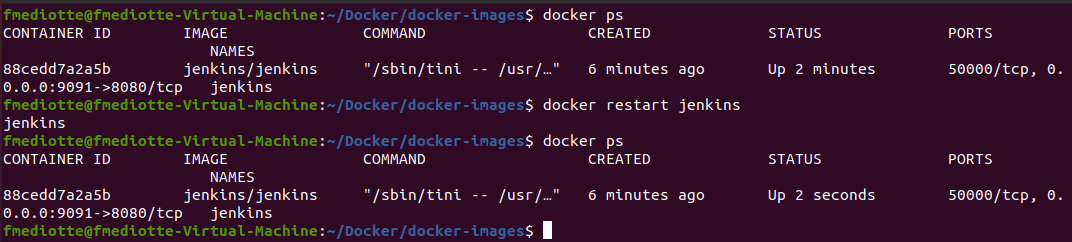
****

Si queremos iniciarlo de nuevo debemos utilizar el comando:

**docker start <containerID or containerName>**

****

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:

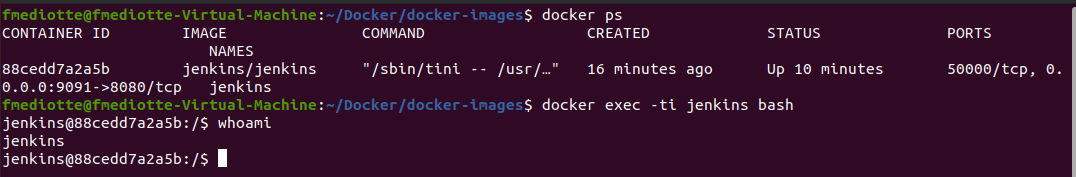
**docker restart <containerID or containerName>**

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

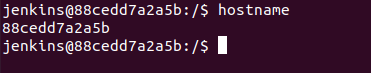
La respuesta es mediante los sistema operativos que instalamos desde la capa FROM de la imagen, por lo que tendríamos una Shell o una terminal.

Para poder acceder dentro del contenedor utilizaremos el comando:

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

**-ti -> t**erminal **i**nteractive

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



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**

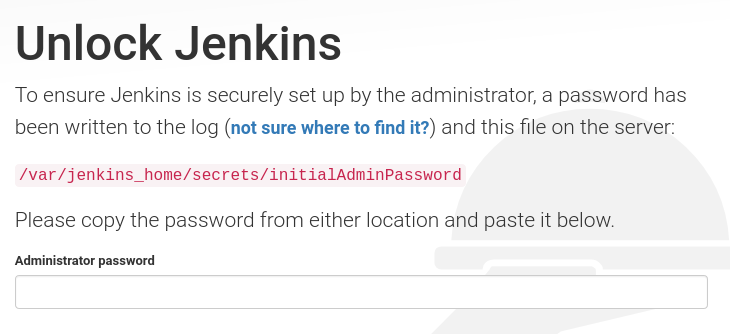


El entrar y salir del contenedor no lo afecta de ninguna manera.

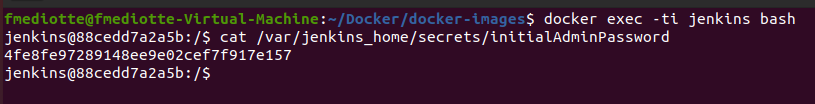
Ahora ¿de qué me sirve entrar y salir del contenedor?

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 especifico.



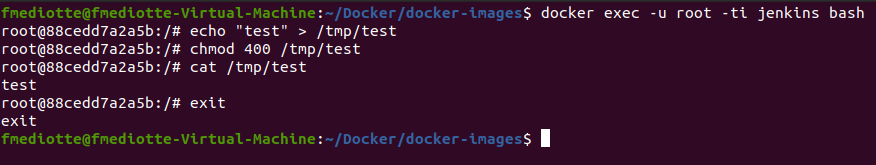
Por lo que debemos acceder a nuestro contenedor para obtenerla con el comando ya visto previamente.



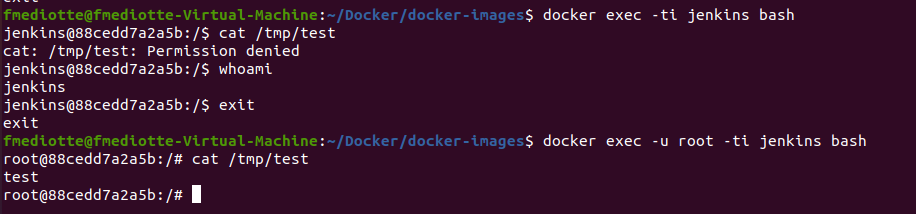
En el caso de que no podamos acceder al archivo con el usuario por defecto (definido en el Dockerfile), podemos acceder como un usuario root. Su uso es para determinadas acciones como por ejemplo dar permisos, ver archivos que no poseemos permisos con otro usuario o realizar acciones que están restringidas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



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:



### **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**

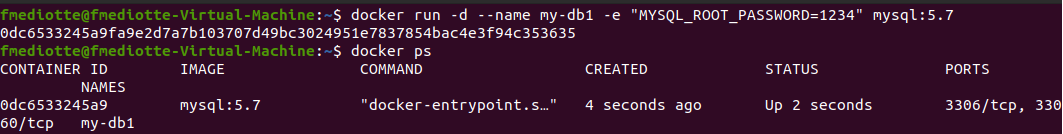
#### **Creando el contenedor**

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

Para levantar el contenedor debemos seguir las [instrucciones](https://hub.docker.com/_/mysql) de docker hub donde explica como iniciar un servicio de mysql:

Donde se hace uso del comando:

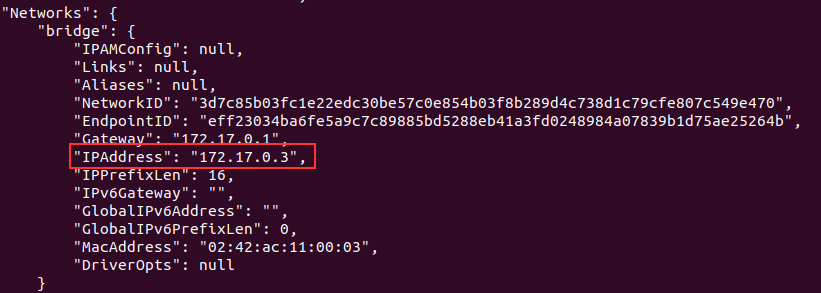
**docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret -d mysql:tag**

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

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

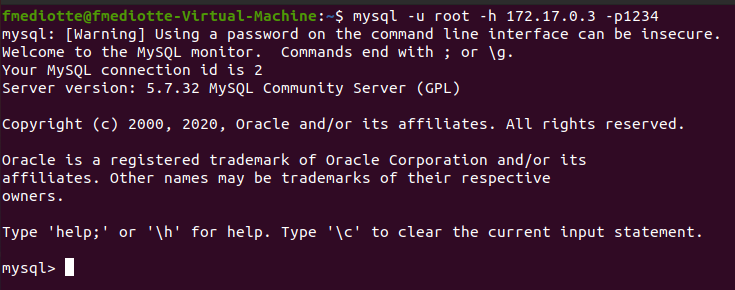
**docker logs -f <nombreContenedor>**

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.



Una vez obtenida debemos conectarnos a MySQL con el comando:

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

****

**¿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**

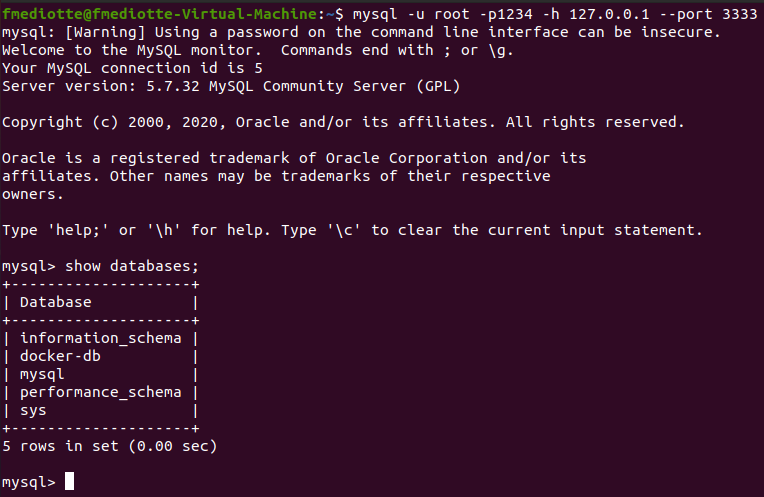
Las variables de entorno las obtenemos desde la página oficial de MySQL en [docker hub](https://hub.docker.com/_/mysql).

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**

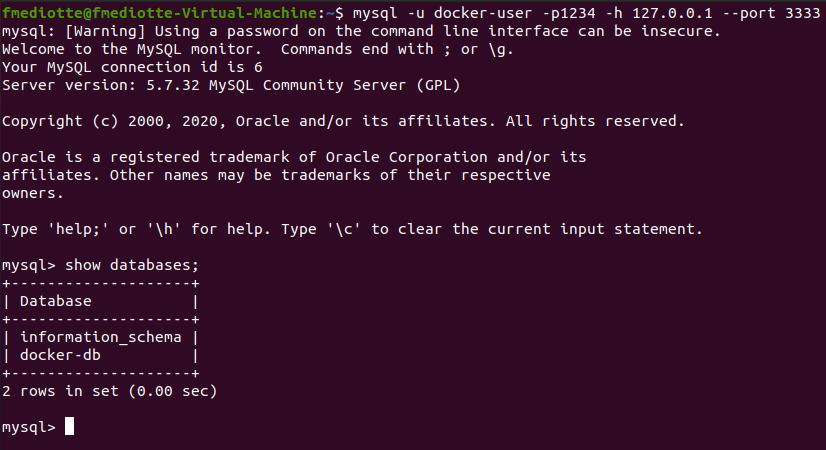
Donde localhost sería en mi caso 127.0.0.1 y port el puerto seteado en la creación del contenedor.

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:



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**



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:

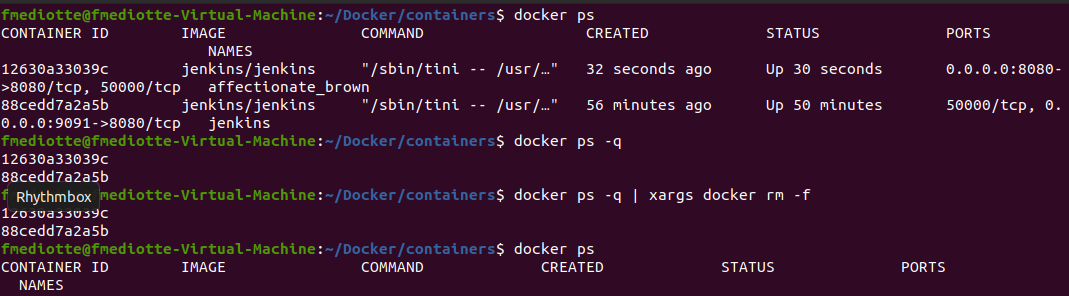
* En el Dockerfile
* Al crear el contenedor

**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**

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

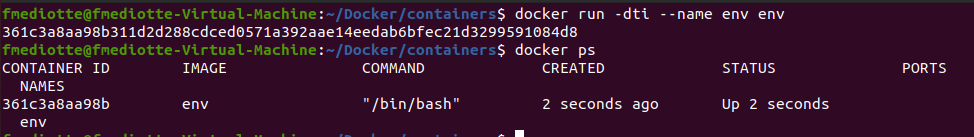
FROM centos:7

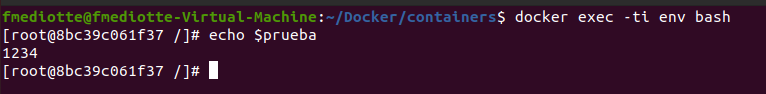
ENV prueba 1234

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:

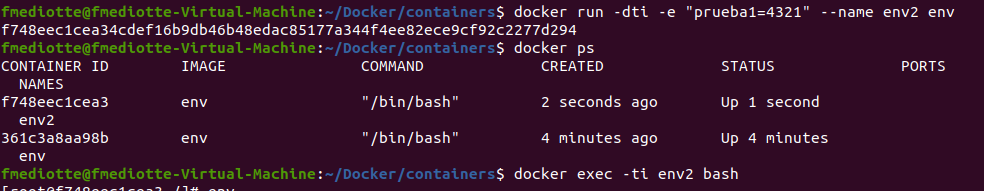
(Creamos el docker run con el flag -dti ya que es una imagen de un SO)





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**

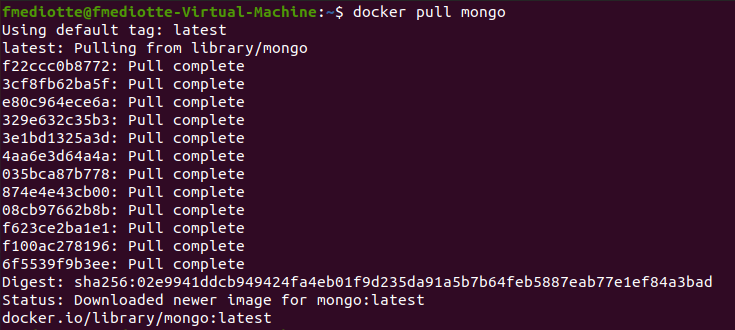


Si ahora nos metemos dentro del contenedor y listamos las variables de entorno del sistema veremos nuestra variable prueba1 definida con el valor pasado por parámetro y nueva variable de entorno creada desde el Dockerfile.



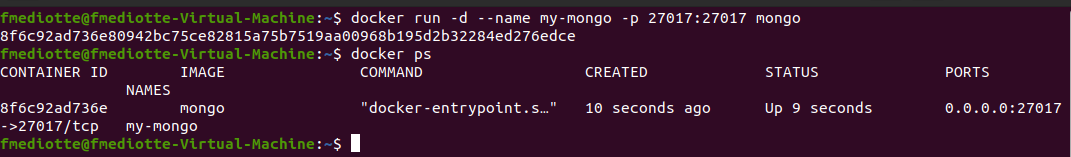
### **Crear un contenedor Mongo**

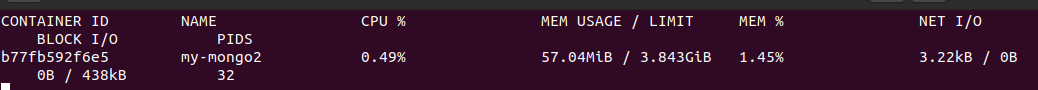
Para construir un contenedor con Mongo, vamos a descargar la imagen oficial de Mongo del repositorio de docker hub, accediendo al siguiente [link](https://hub.docker.com/_/mongo) y realizando un docker pull.



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

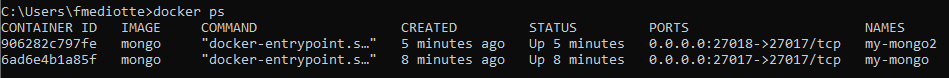
**docker run -d --name my-mongo -p 27017:27017 mongo**

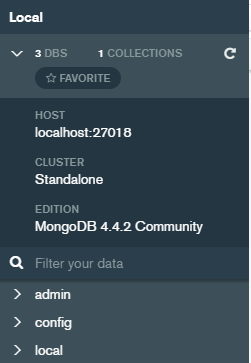
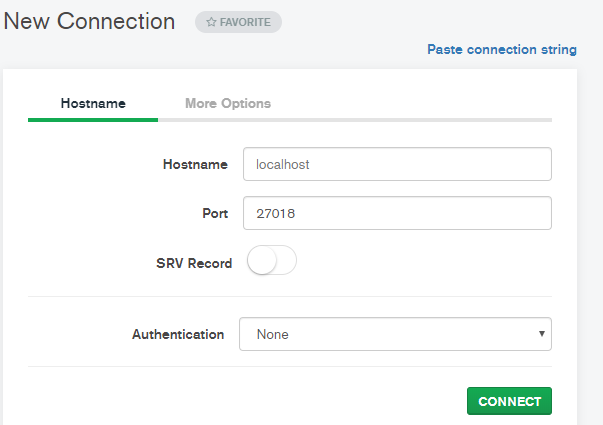
*(por defecto el Puerto que utiliza mongo es 27017, esto lo podemos consultar en docker hub)*

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

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:

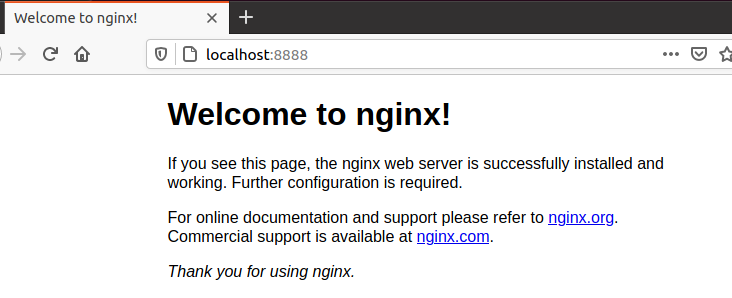
**docker run -d --name my-mongo2 -p 27018:27017 mongo**

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

### **Crear un contenedor Apache / Nginx / Tomcat**

Buscamos la imagen oficial de nginx en [docker hub](https://hub.docker.com/_/nginx) y hacemos un docker pull en nuestra máquina.

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



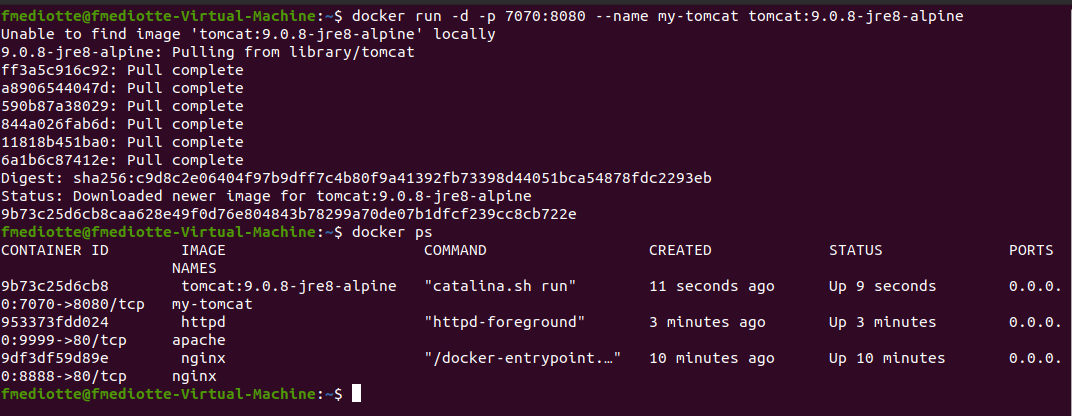
Ahora ¿cómo creamos un Apache sin eliminar nuestro nginx?

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



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

Ahora creamos un contenedor que contenga la [imagen oficial de tomcat](https://hub.docker.com/_/tomcat) realizando un **docker run -d -p <otropuesto>:8080 tomcat**

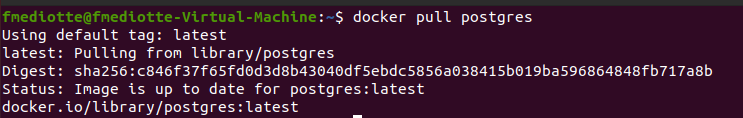




De esta manera creamos contenedores de Apache, Nginx y Tomcat.

### **Crear un contenedor PostgreSQL**

Buscamos la [imagen oficial de Postgres](https://hub.docker.com/_/postgres) y realizamos un docker pull como se indica en la página oficial del docker hub.



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

**Docker run -d --name postgres**

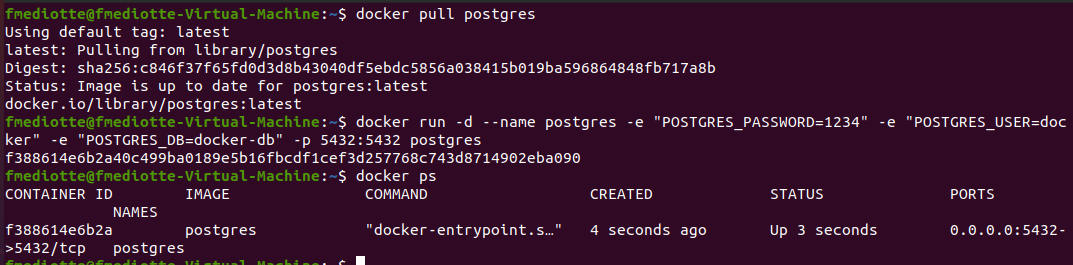
**-e “POSTGRES\_PASSWORD=1234”**

**-e “POSTGRES\_USER=docker”**

**-e “POSTGRES\_DB=docker-db”**

**-p 5432:5432**

**postgres**

Estas variables de entorno se encuentran en la documentación oficial.

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

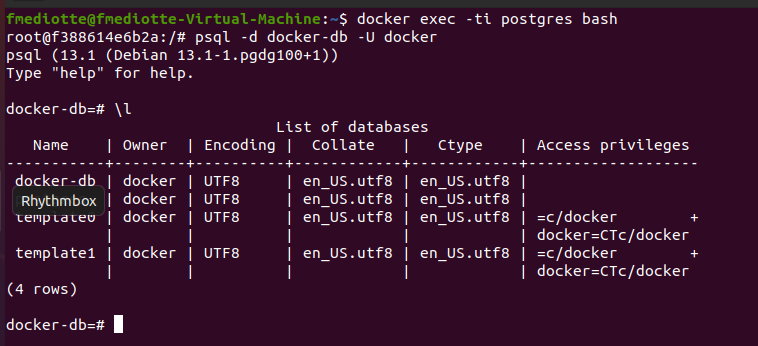
**docker exec -ti <nombrecontenedor> bash**

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:

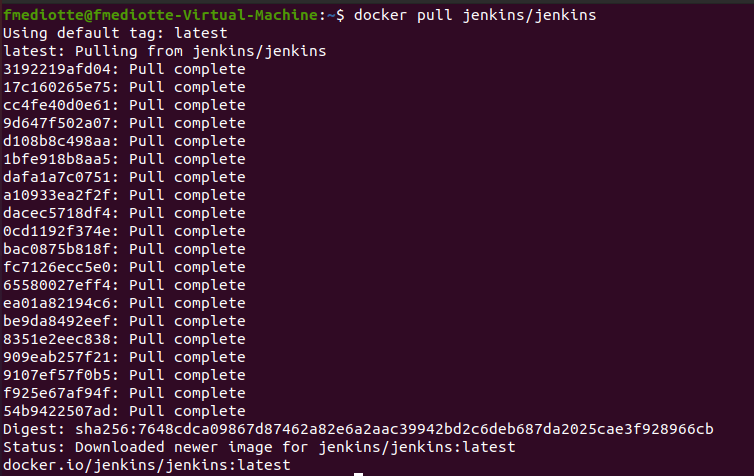
**psql -d docker-db -U docker**

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



### **Crear un contenedor Jenkins**

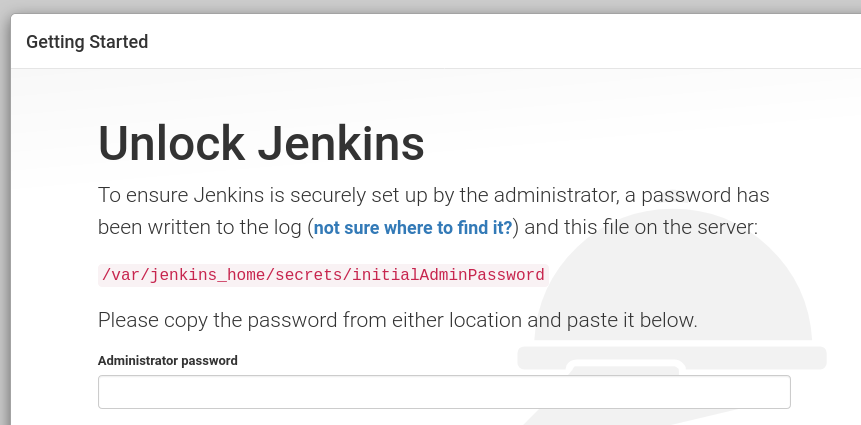
Buscamos la [imagen oficial de Jenkins](https://hub.docker.com/r/jenkins/jenkins) y realizamos un docker pull como se indica en la página oficial del docker hub.

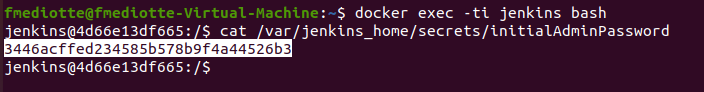


Una vez descargada la imagen levantamos un contenedor con dicha imagen y mapearle un puerto que tengamos disponible, en el caso del ejemplo 7070:

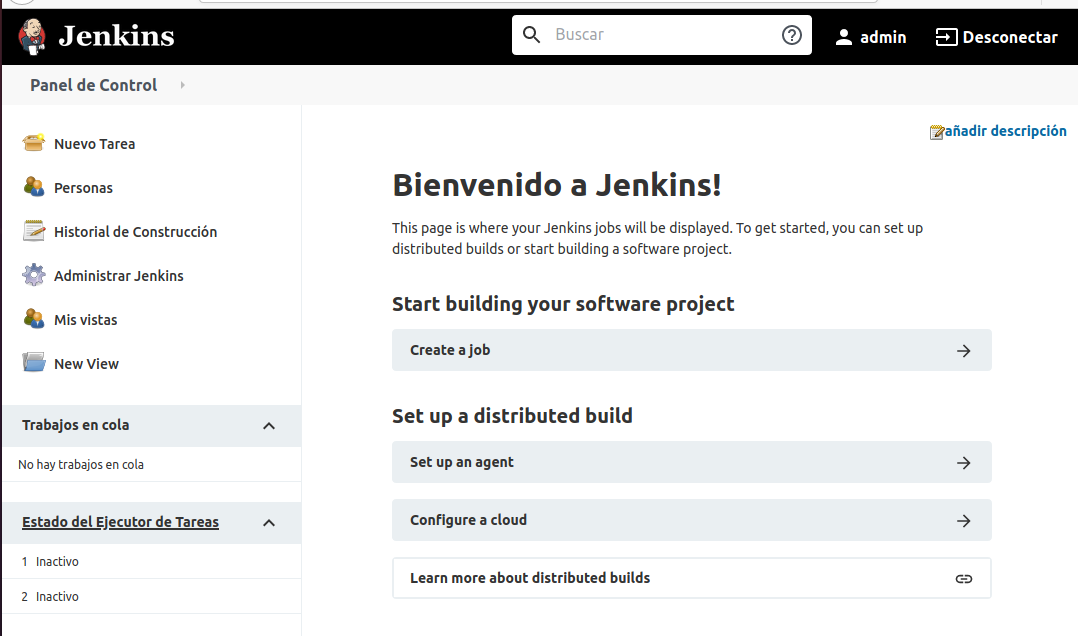
**docker run -d -p 7070:8080 --name jenkins jenkins**

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.





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

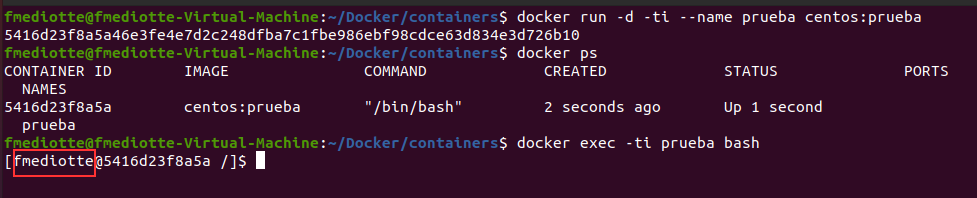
RUN useradd fmediotte

USER fmediotte

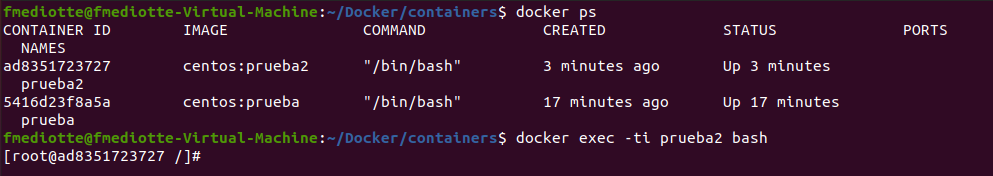
En esta dockerfile le estamos diciendo a nuestro SO centos que va a estar utilizando el user creado fmediotte, por lo que si accedemos al contenedor con el comando:

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

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

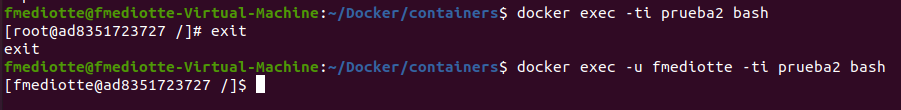


Si por ejemplo comentásemos el Dockerfile la instrucción USER tendríamos agregado el usuario fmediotte pero no seteado por defecto por lo que si creamos nuevamente la imagen con el nombre centos:prueba2 y otro contenedor pero con esa imagen veremos que el usuario por defecto que toma el contenedor al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



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**



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 **-d 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.

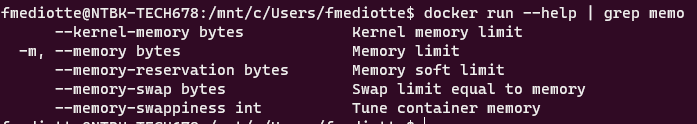


Podemos observar que está utilizando 59.48MB de memoria de un límite de 2GB, esto es debido a que en Windows se asigna una porción de la memoria RAM de la computadora y por defecto son 2GB, mientras que en otras distribuciones como Ubuntu utiliza todos los recursos de la computadora.

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:



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.



También podemos especificar el valor en GB:

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

****

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

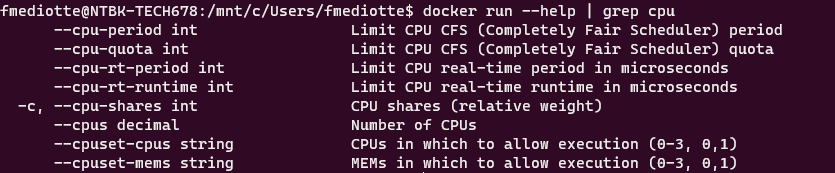
#### **Limitar Uso de CPU**

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:

****

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:

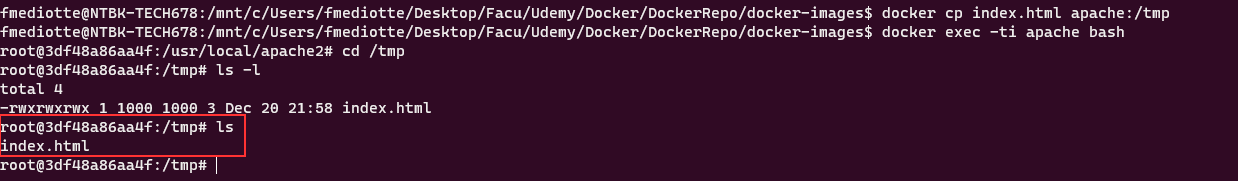
**docker cp <archivoAcopiar> <pathDestino>**

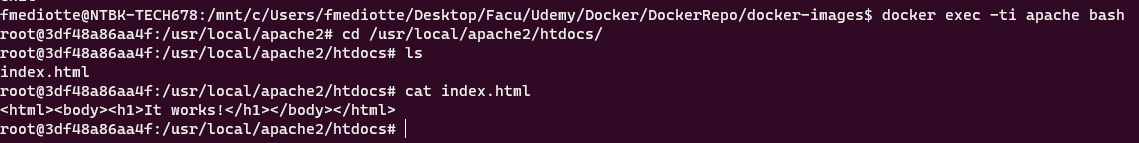
***El cuál nos permite copiar desde fuera del contenedor hacia dentro del mismo y viceversa.***

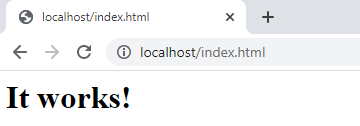
Para hacer un ejemplo vamos a crear un contenedor con una imagen de apache levantando en el puerto 80 de nuestra máquina:

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.

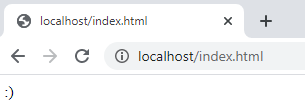
Para que tenga más sentido en el ejemplo que estamos haciendo vamos a obtener el path donde se encuentra el index.html que se levanta en el browser, el mismo en apache es **/usr/local/apache2/htdocs** donde veremos si hacemos un cat de index.html lo que observamos en el browser.



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:

### **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**

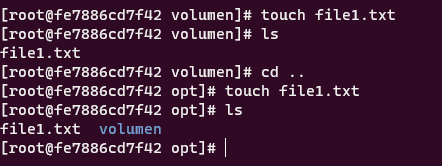
FROM centos:7

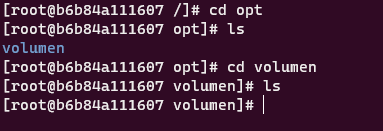
VOLUME /opt/volumen

**Container:**

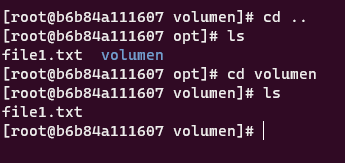


Vamos a crear dos archivos dentro del contenedor accediendo al mismo con el comando **docker exec -ti centos bash**:



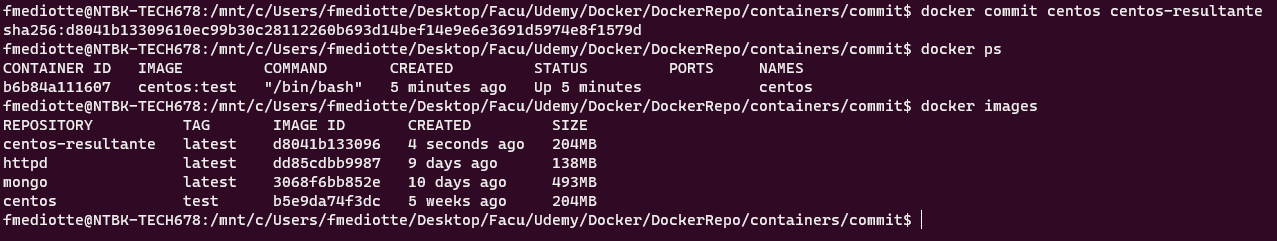
Si eliminamos el contenedor y lo creamos nuevamente veremos que los cambios que hicimos ya no están, es decir los archivos creados ya no existen.

Lo que haremos es crearlos nuevamente:



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> <nombreNuevaImagen>**

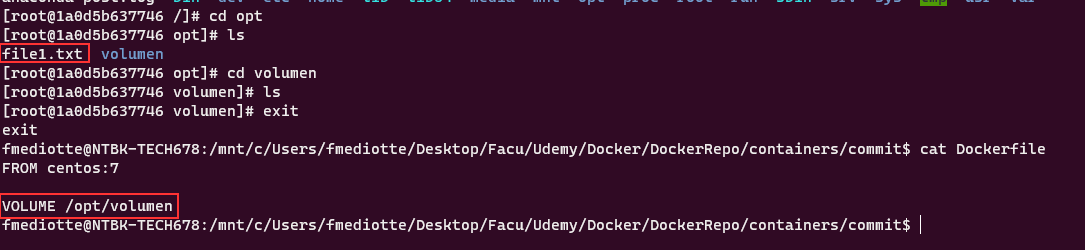
Al ejecutar esto se crea una nueva imagen con el estado actual del contenedor:

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:



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.

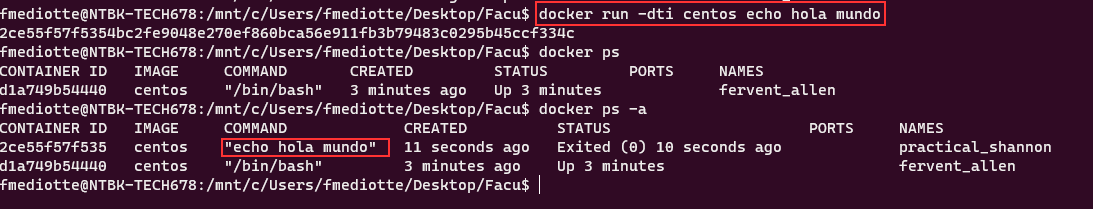
### **Sobrescribe el CMD de una imagen sin un Dockerfile**

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:

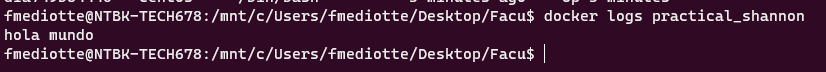


Si queremos sobrescribir este comando desde la terminal debemos agregar un argumento más al **docker run** que ya venimos utilizando, el mismo se agrega posterior al nombre de la imagen que queremos que quede corriendo en segundo plano en el contenedor, por ejemplo, si escribiésemos luego de la imagen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

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

Nuestro contenedor tomaría el echo hola mundo como comando:

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



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



### **Aprende a destruir contenedores automáticamente**

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

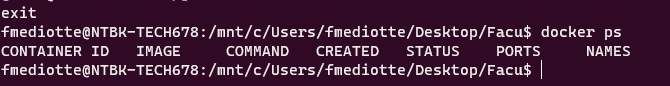
Para lograr esto debemos agregar al comando docker run el argumento **--rm** el cuál instruye a docker que el contenedor que estas creando es temporal y que una vez el contenedor se salga o que te salgas de la sesión el contenedor debería finalizar.

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:

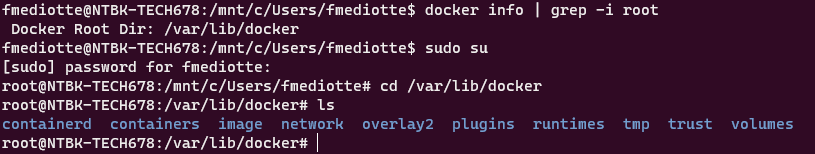


### **Cambiar el Document Root de Docker**

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:

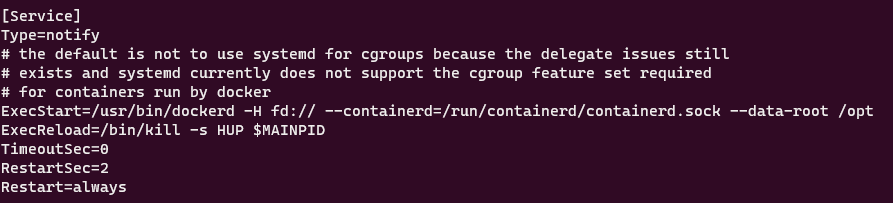
**docker info | grep -i root**

****

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 encuenta 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**

****

## 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 mantenedor 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:

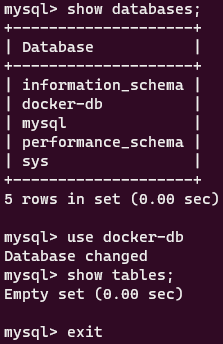
* **Host:** se almacenan en nuestro docker host, y viven dentro de una carpeta de file system que nosotros definimos.
* **Anonymus:** no definimos una carpeta, sino que docker genera una carpeta random y dentro de la misma persiste la información.
* **Named Volumes:** son volúmenes que nosotros creamos, 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.



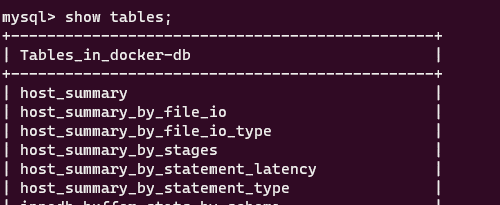
Hacemos un dump de la base sys con el comando:

**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

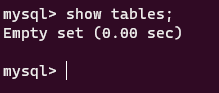
Una vez realizado accedemos nuevamente a nuestra base de datos docker-db y ejecutamos el comando show tables, el cual nos mostrará las tablas nuevas importadas del dump.sql.



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:

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:



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.

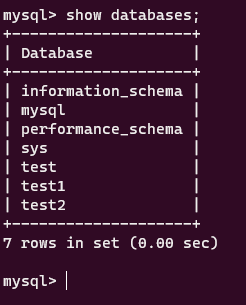
Podríamos crear bases de datos de ejemplo como test, test1 y test2 y posteriormente borrar el contenedor:



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

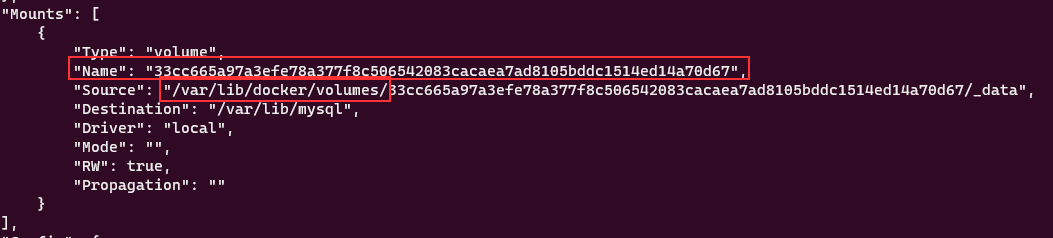
Docker lo que hará es obtener la información del volumen persistente **/opt/mysql** que guardo información del contenedor anterior, por lo que al acceder nuevamente al nuevo contenedor veremos que las bases de datos siguen creadas:



### **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:

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

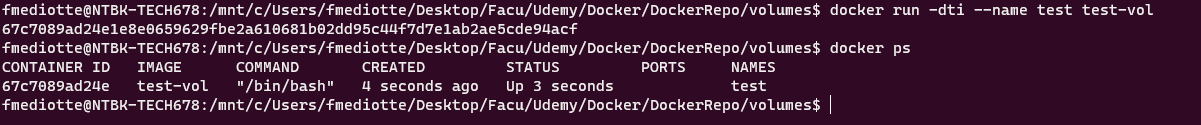
VOLUME /opt/

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

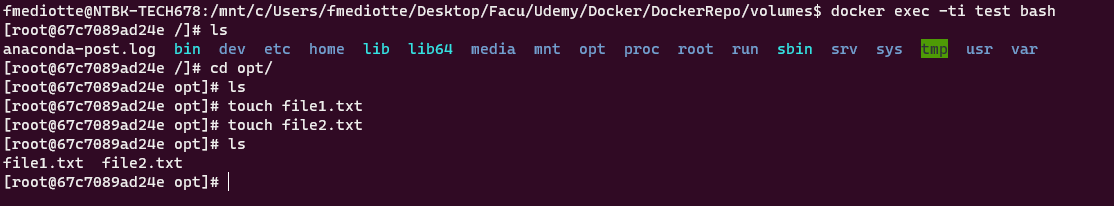
Para ver los volumes persistentes que tenemos en nuestra máquina podemos hacer uso del comando:

**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**

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

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

Si accedemos al volumen veremos los dos archivos creados.

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

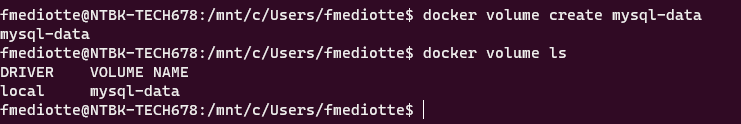
### **Volúmenes nombrados – Caso práctico MySQL**

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:

****

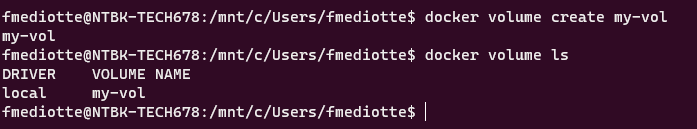
Para eliminar el mismo se utiliza el comando:

**docker volumen rm <volumeName>**

****

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

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

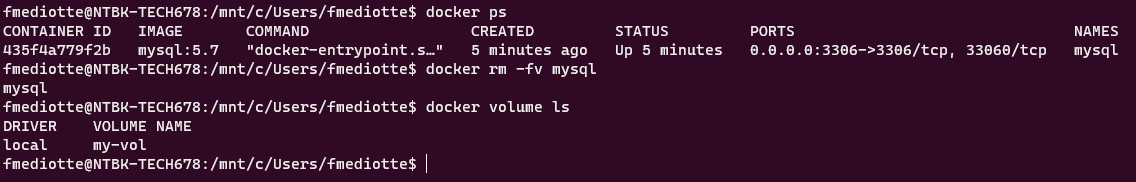


Y en la instrucción docker run hacemos uso del argumento -v para realizar el mapeo de los volúmenes local y del contenedor, de la siguiente forma:

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:



**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:*

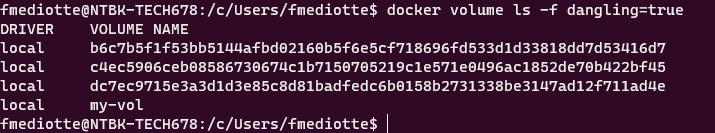
chroot /host

### **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**



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**

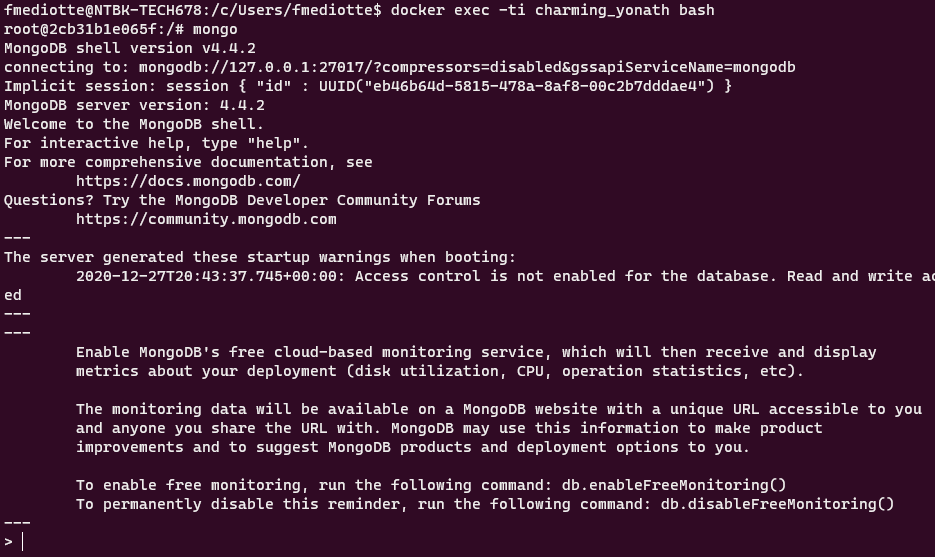
Creamos una carpeta llamada mongo en la ruta opt que vamos a utilizar de volumen, por lo que nos quedará algo así:



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**

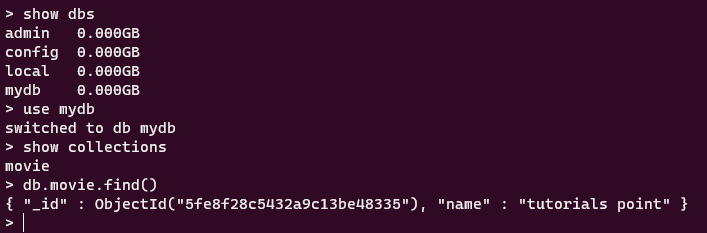
Una vez creado nuestro contenedor, accedemos al mismo mediante el comando **docker exec -ti <containerName> bash** y tipeamos la palabra mongo para poder acceder a nuestro motor de base de datos.

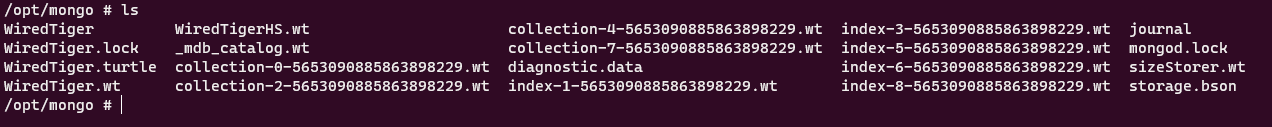


Insertamos un registro en la base de datos siguiendo el siguiente tutorial:

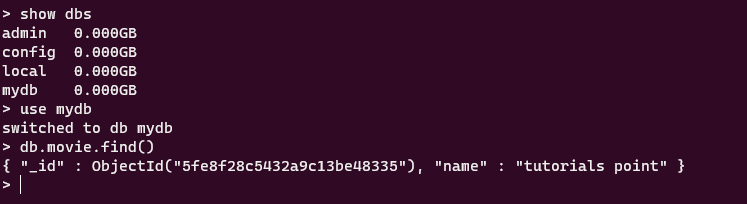
<https://www.tutorialspoint.com/mongodb/mongodb_create_database.htm>

Veremos que ahora nuestra base de datos contiene datos:



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

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:



### **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:

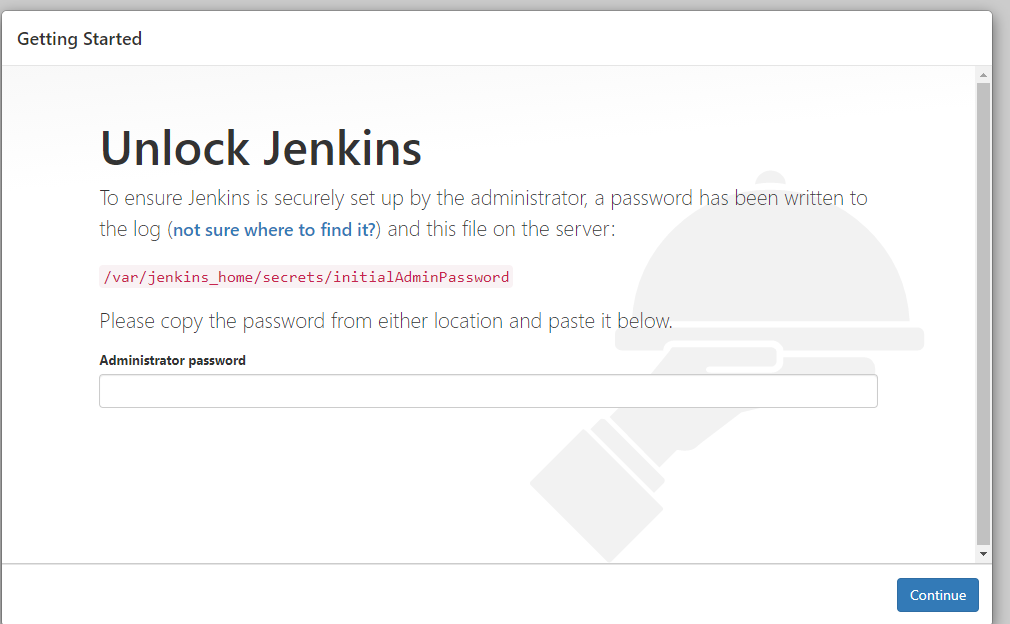


Creamos un contenedor con una imagen de jenkins mapeada al puerto 8080, y mapeamos el volumen de nuestra path local con el volumen persistente de jenkins, el cual podemos consultar en la documentación oficial de jenkins en docker hub. En el caso de jenkins la misma se guarda en la carpeta **/var/jenkins\_home**,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

**docker run -d --name jenkins -p 8080:8080 -v /opt/jenkins/:/var/jenkins\_home jenkins**

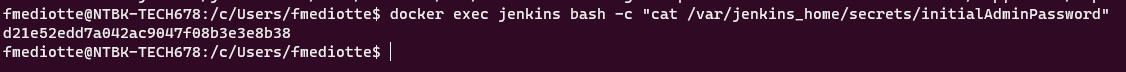
(En el video esta explicado así, pero la imagen de jenkins fue modificada y no funciona el volumen /opt/jenkins por lo que vamos a utilizar jenkins-data que va a estar ubicado en /var/lib/docker/volumes)

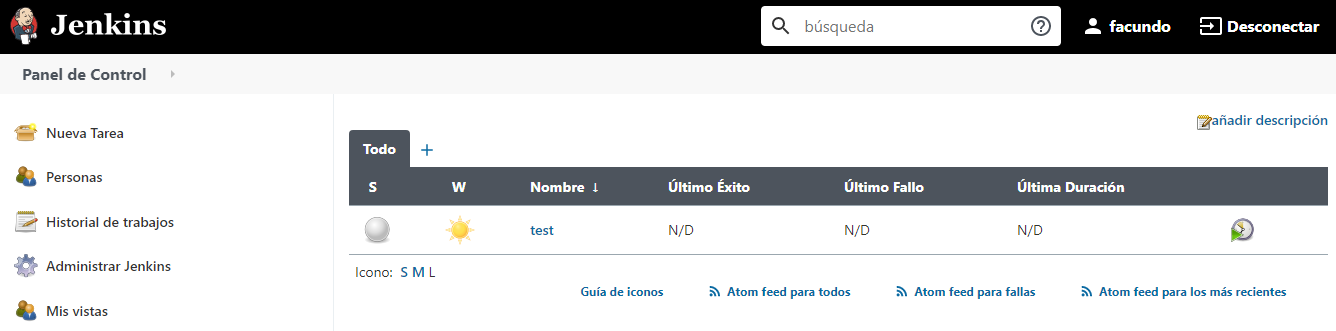
Cómo siempre Jenkins nos solicitara la password que se encuentra dentro del contenedor en el archivo: /var/jenkins\_home/secrets/initialAdminPassword

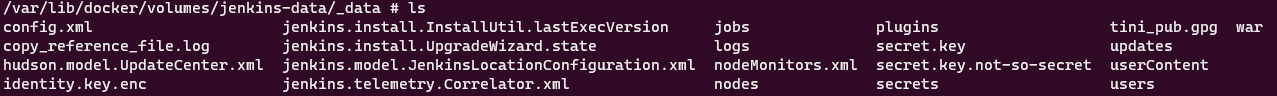


Podríamos acceder con docker exec -ti pero vamos a aprender una forma de que no haga falta acceder al contenedor sino que podemos obtener el contenido de un archivo desde fuera del mismo, esto se logra quitando el argumento -ti (terminal interactive) y agregando el argumento -c (command) en el comando 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”**

****

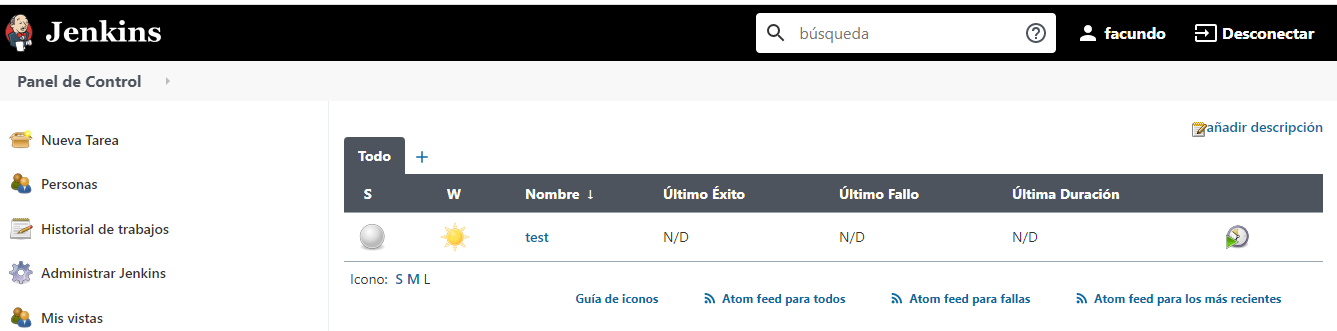
Creamos una tarea automatizada en Jenkins:

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:

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 solicitara 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:



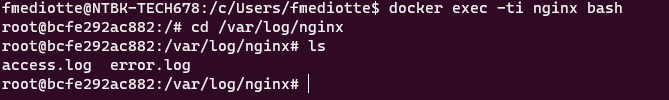
De esta manera comprobamos que nuestro volumen persistente funciono correctamente y no perdimos toda la información útil que ya teníamos en el volumen.

### **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:

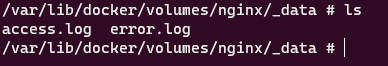


Los mismos como se muestra en la imagen se guardan en /var/log/nginx. De esta forma averiguamos donde se están guardando dentro del contenedor los logs, por lo que borramos el contenedor.

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:



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

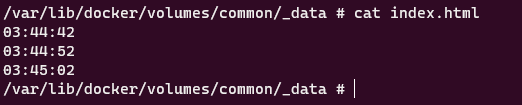
done

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**

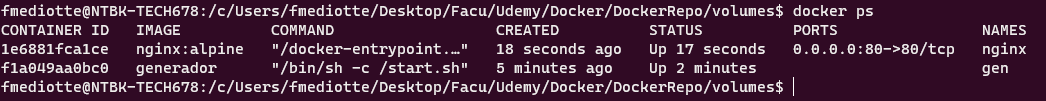
Luego si inspeccionamos el archivo index.html veremos que se esta escribiendo la hora cada 10 segundos:



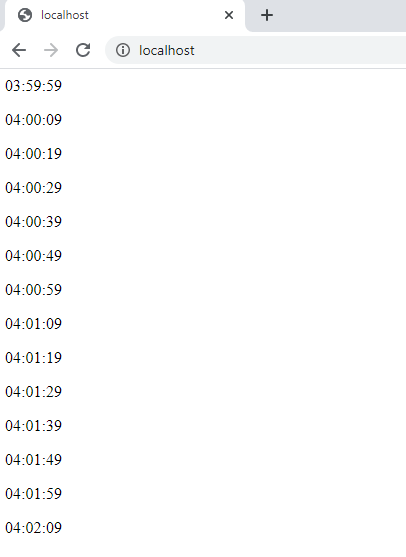
Ahora vamos a crear un contenedor de nginx con el comando:

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

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



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

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

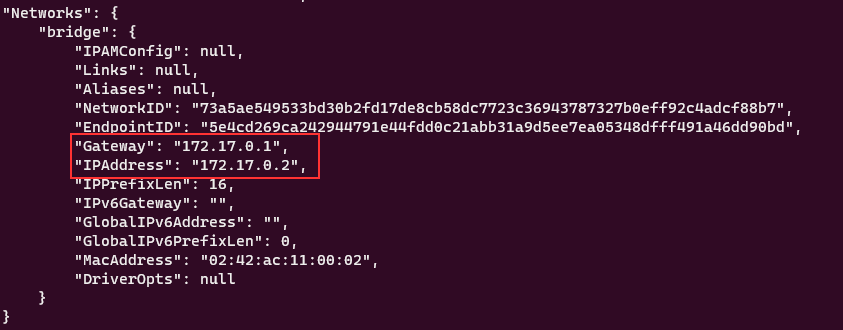
En esta sección vamos a ver los siguientes tópicos:

* Crear / Eliminar Redes
* Tipos de redes:
  + Brigde
  + 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:

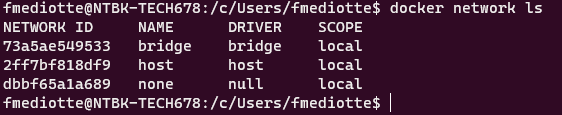


**There is no docker0 bridge on Windows**

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**

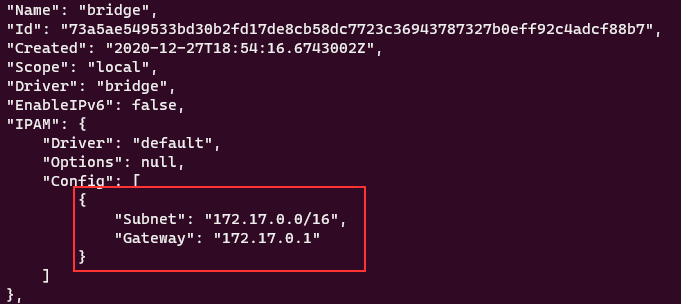


Y si queremos filtrar por la red **bridge (red por defecto de Docker)** que es la que nos interesa en este momento le agregamos un grep brigde de la siguiente forma:

**docker network ls | grep bridge**

****

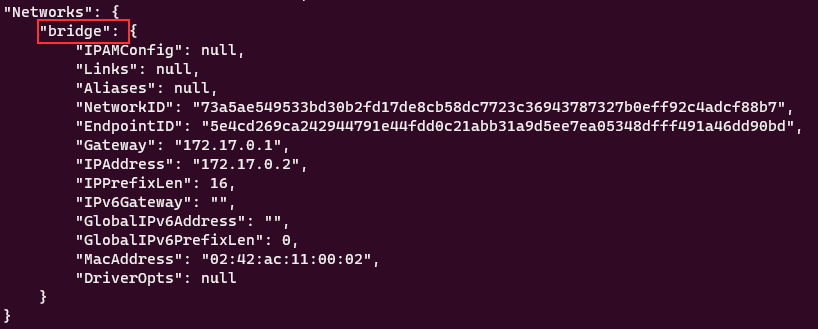
Hacemos un **docker network inspect bridge** para ver la configuración de red de la misma:



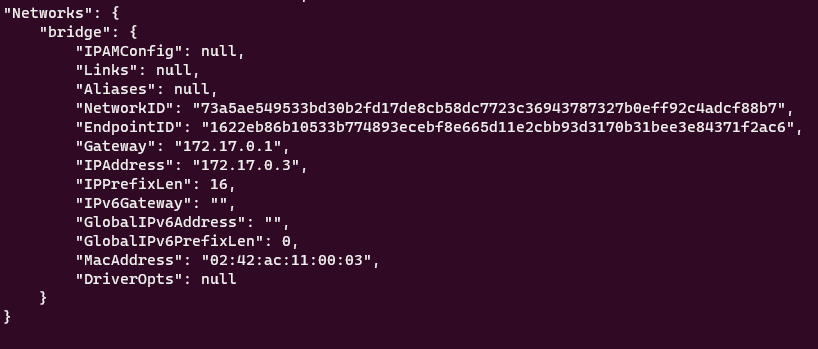
Si hacemos un docker inspect del contenedor ya creado de la siguiente forma:

docker inspect <containerName>

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

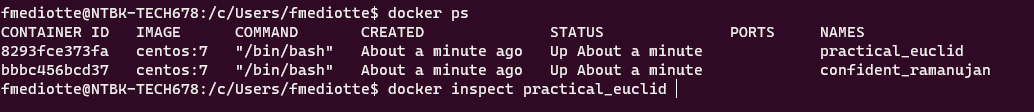


La red por defecto que trae docker es **bridge**, es decir que si creamos un nuevo contenedor si no definimos que trabaje con otra red va a asignar siempre el mismo a la red bridge:

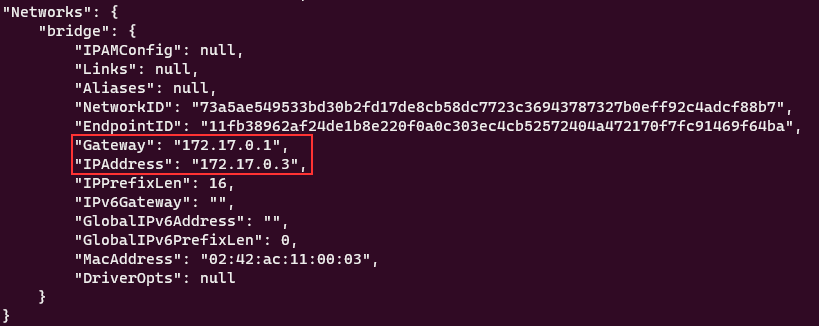


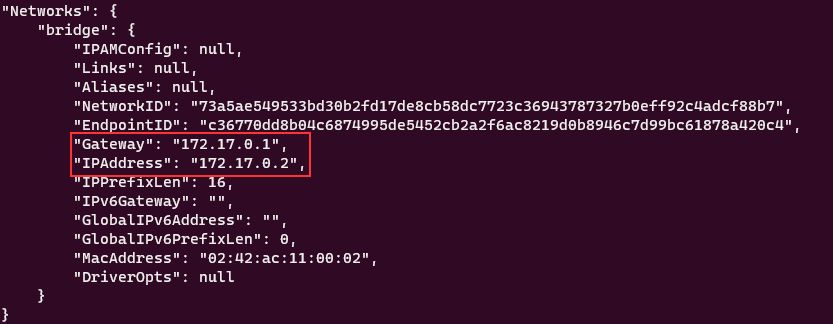
También se puede utilizar el comando ping desde un contenedor a otro, debido a que pertenecen a la misma red.

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



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

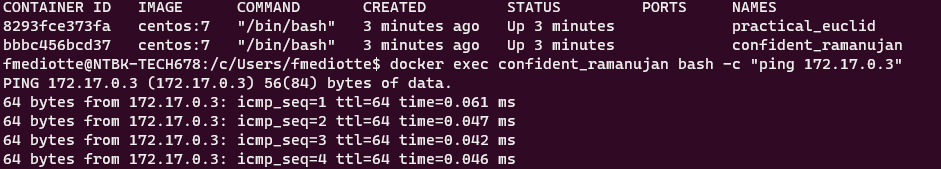




Por lo que podemos utilizar el comando:

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

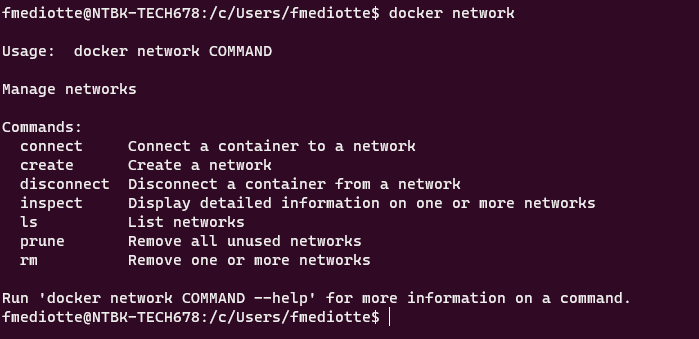
En este ejemplo:



### **Crear una red definida por el usuario**

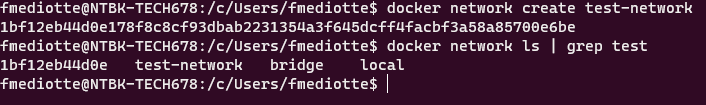
Para poder crear una red necesitamos hacer uso del driver bridge, que es el driver por defecto que trae la red de docker, que funciona como una mini red virtual donde podemos agregar contenedores.

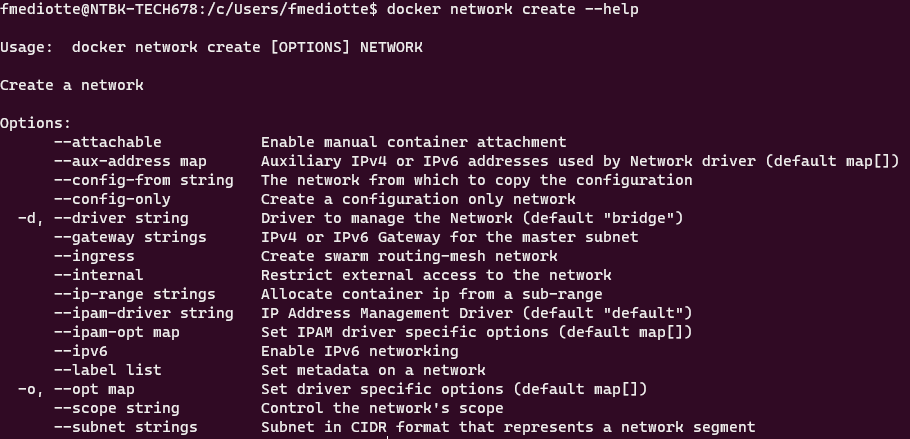
Comandos disponibles para trabajar con redes:



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

**docker network create <networkName>**

****

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:

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



**Docker inspect de la red creada:**

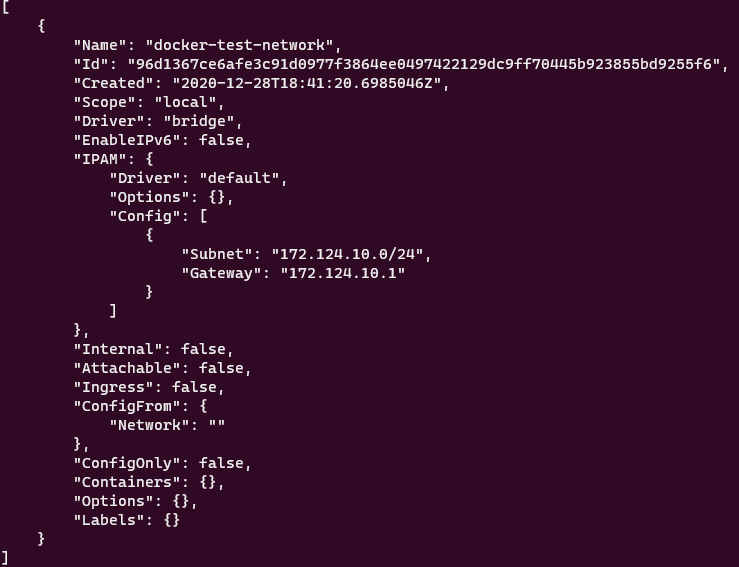
****

### **Inspeccionar Redes**

Para inspeccionar redes se hace uso del comando:

**docker network inspect <netName>**

Dentro de la misma encontraremos el siguiente formato:



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**.

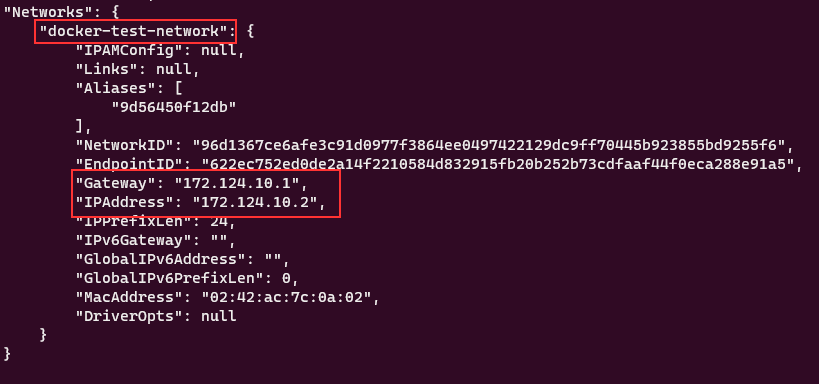
Si queremos que nuestro contenedor se conecte a un red distinta a la por defecto, por ejemplo, a la creada previamente llamada docker-test-network, debemos utilizar el argumento --network en el comando docker run de la siguiente manera:

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

En el ejemplo:

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

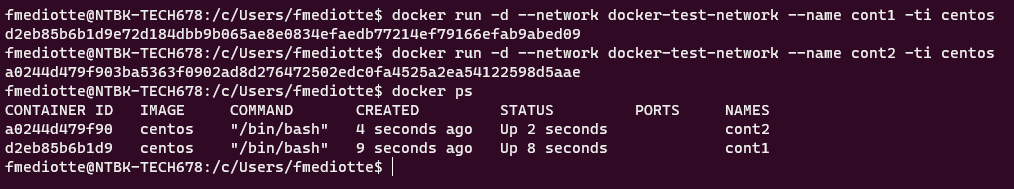
Y si inspeccionamos el contenedor creado veremos que se conecto con la red que pasamos como parámetro heredando las configuraciones que habíamos definido para la red:

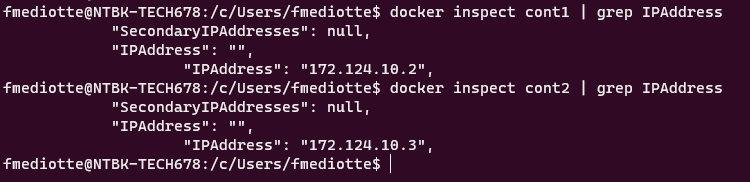


De esta manera creamos un contenedor a una red distinta a la red de docker.

### **Conectar contenedores en la misma red**

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



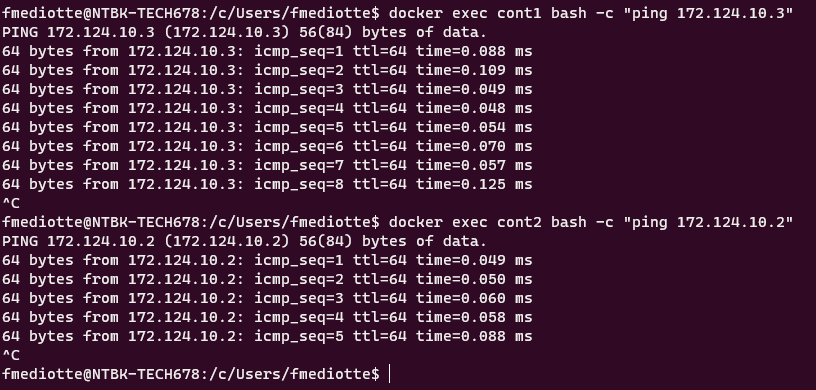


Si por ejemplo mi contenedor 1 es mi WebServer y mi contenedor 2 es mi base de datos.

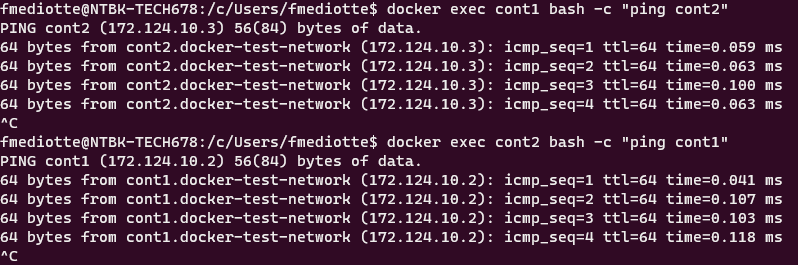
El cont1 tiene la ip: 172.124.10.2

El cont2 tiene la ip: 172.124.10.3

Verifico que se vean entre ellos con el comando ping:



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:

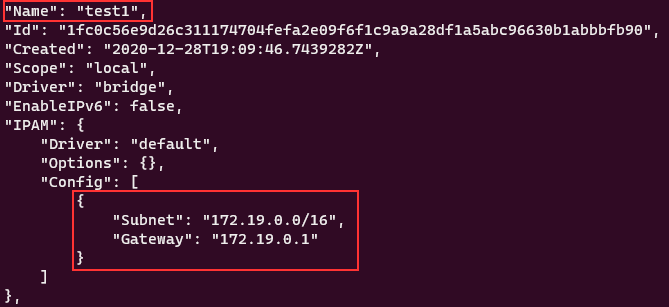


### **Conectar contenedores en distintas redes**

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



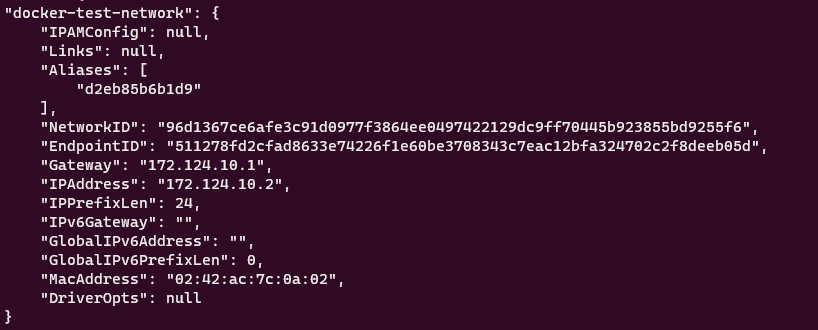




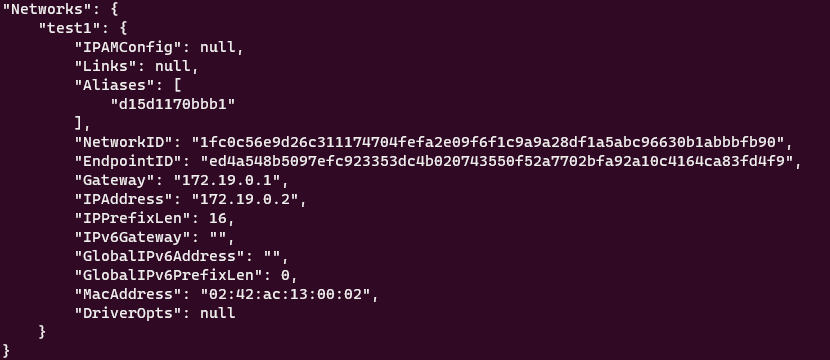
Como vemos ambas tienen configuraciones de red.

Vamos a tener dos contenedores uno en cada red:

**Cont1**

****

**Cont3**

****

Para poder conectar ambos contenedores vamos a utilizar el comando:

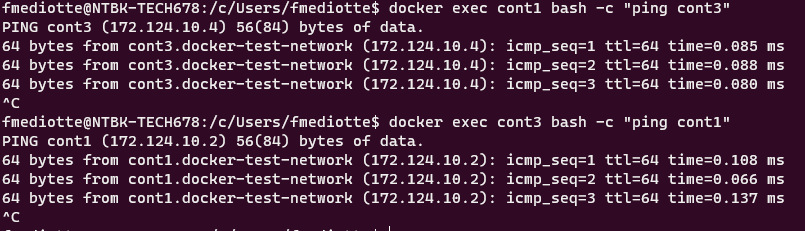
**docker network connect <netNameForConnect> <nameContainerToConnect>**

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:



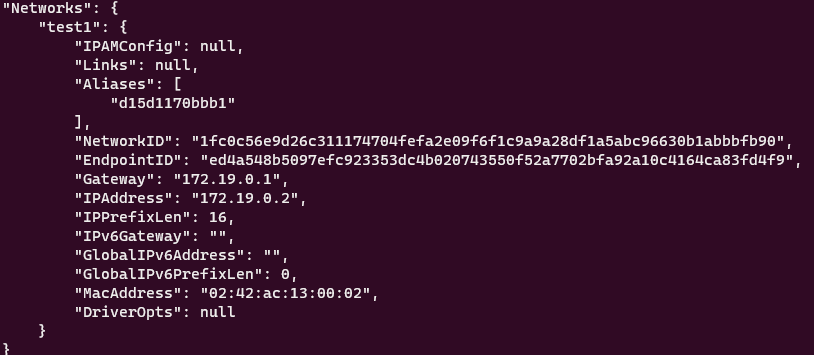
De esta manera lo que se hizo fue conectar el cont 3 a la red, pero si ahora lo queremos desconectar debemos utilizar el comando:

**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.



### **Eliminar redes**

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

**docker network rm <netName>**

### **Asignar IP a un contenedor**

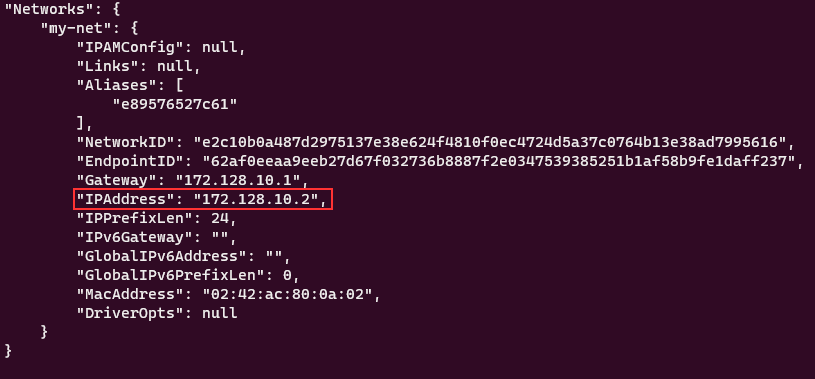
Primero creamos una red de la siguiente forma:

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

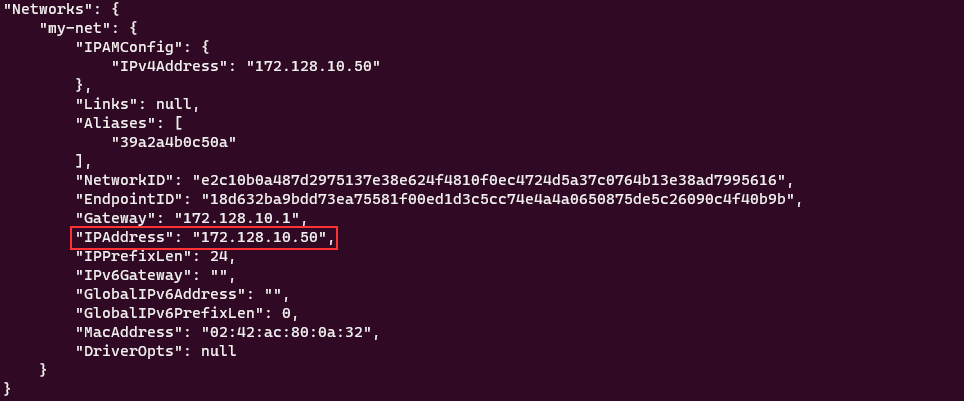
Al inspeccionarlo veremos que se creo con una ip al azar:



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:

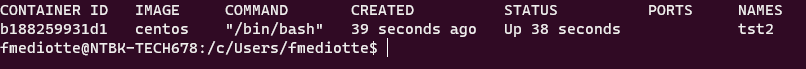


### **La red host**

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



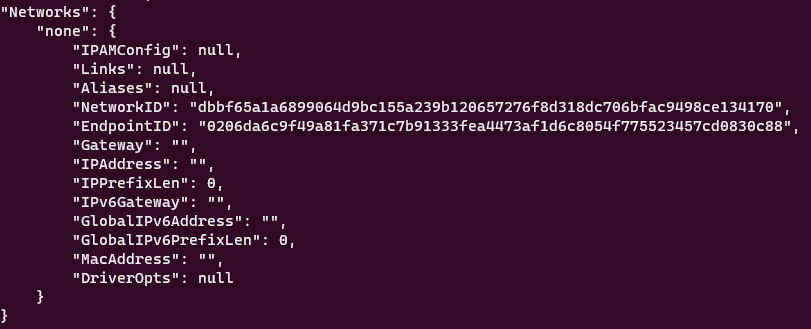
### **La red none**

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:



## 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.

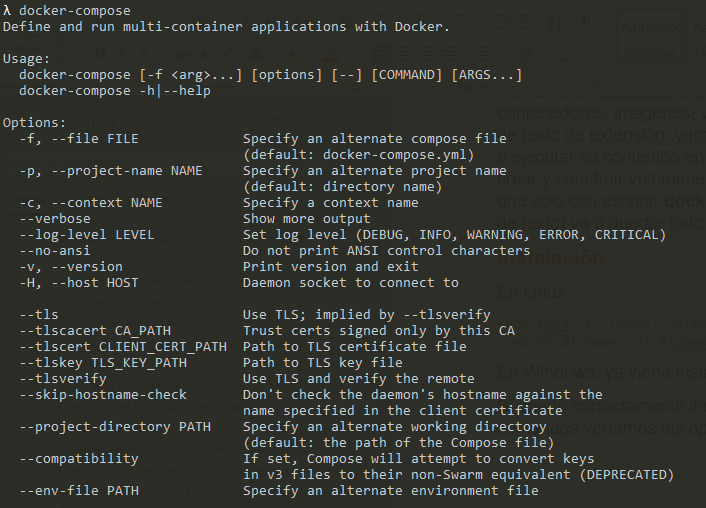
### **Instalación**

**En Linux:**

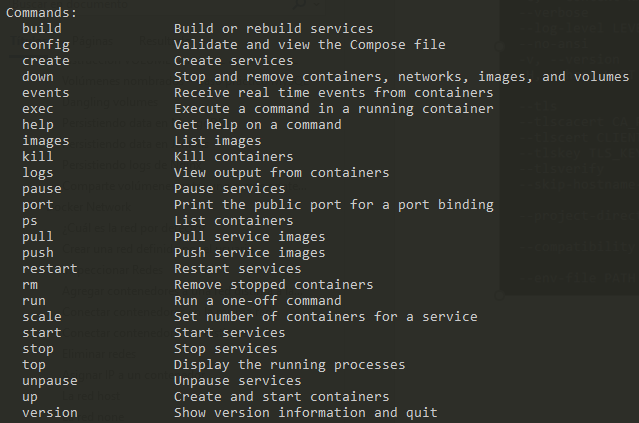
sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

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:



### **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:

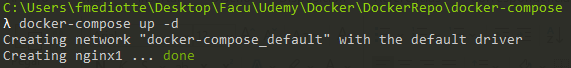
      - "8080:80"

    image: nginx

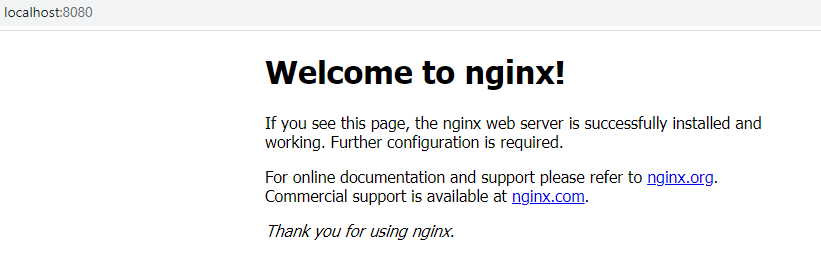
Para poder utilizarlo debemos ir a la terminal y escribir el siguiente comando:

**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:



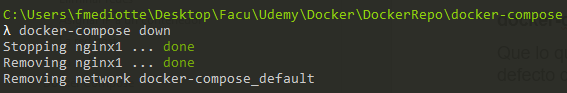
Si accedemos a nuestro localhost:8080 veremos nginx corriendo con docker-compose:



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:



### **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

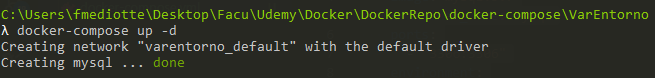
    ports:

      - "3306:3306"

    environment:

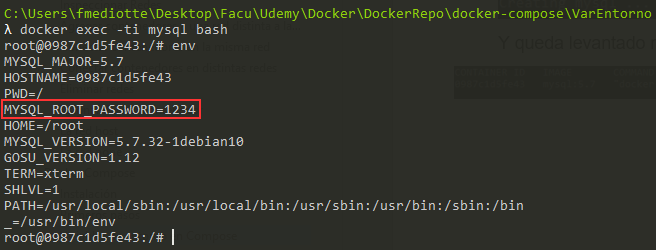
      - "MYSQL\_ROOT\_PASSWORD=1234"

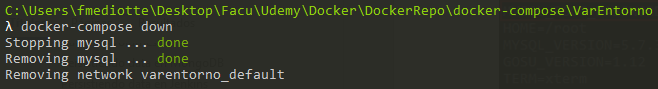
Ejecutamos el docker-compose up



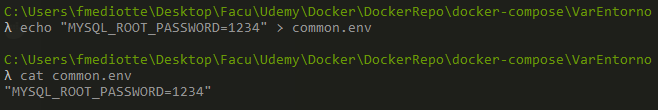
Y queda levantado nuestro contenedor en el puerto 3306:

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





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:



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

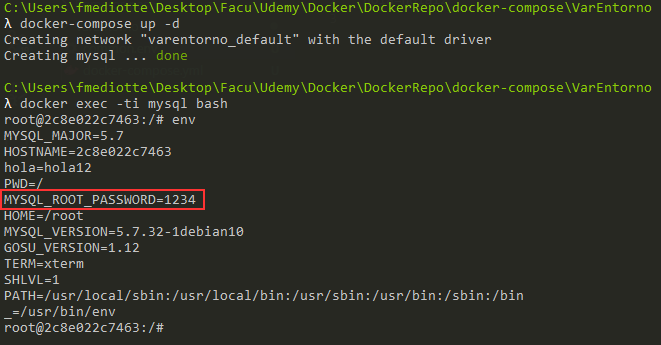
    container\_name: mysql

    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:



### **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:

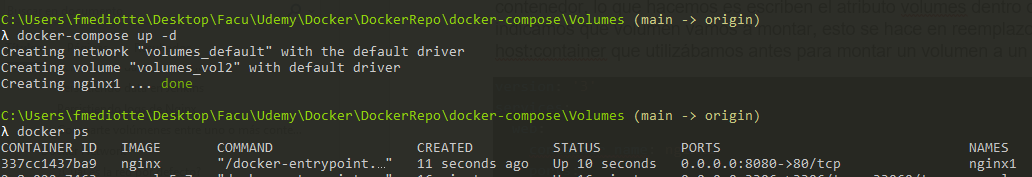
      - "vol2:/usr/share/nginx/html"

    image: nginx

volumes:

  vol2:

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



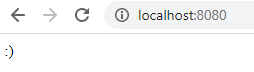
Para validar que funciona nuestro volumen nos iremos al document root obtenido con el comando:

docker info | grep -i root



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:





#### **¿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:



Y utilizaremos el mismo para montar el volumen:

version: '3'

services:

  web:

    container\_name: nginx2

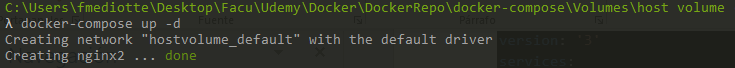
    ports:

      - "8081:80"

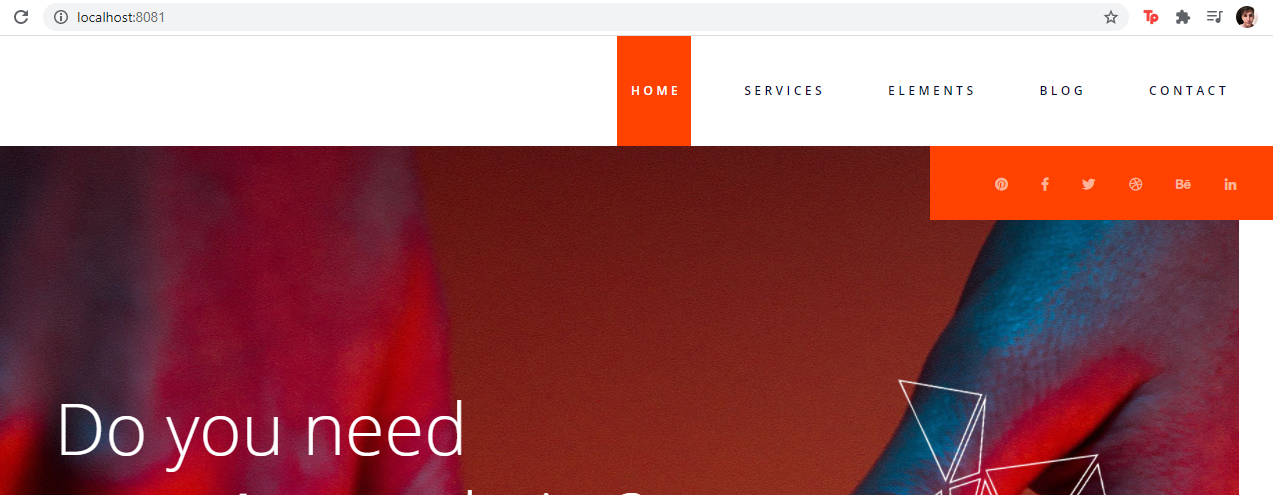
    volumes:

      - "/c/Users/fmediotte/Desktop/Facu/Udemy/Docker/DockerRepo/docker-compose/Volumes/host volume/html:/usr/share/nginx/html"

    image: nginx



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>**



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

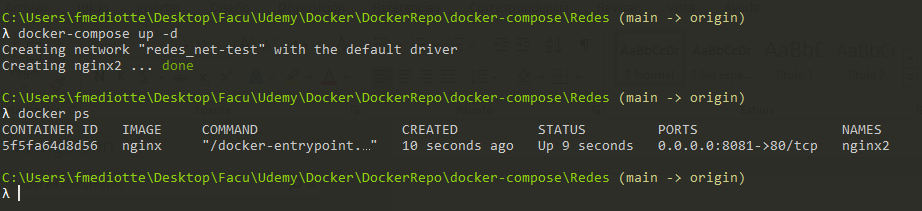
    networks:

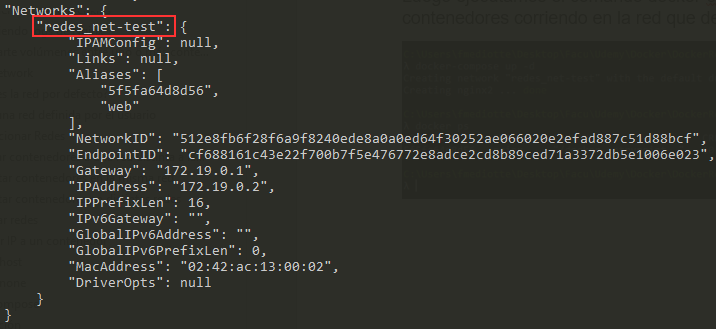
      - net-test

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:





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 ngin2 bash -c “ping nginx3”**

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

**docker exec -ti ngin2 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

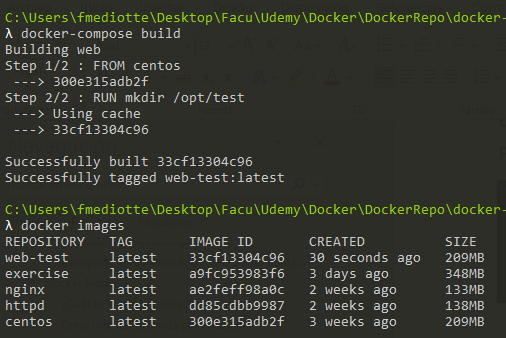
    build: .

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:



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

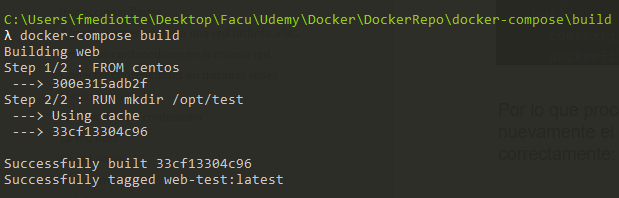
    image: web-test

    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:



### **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:

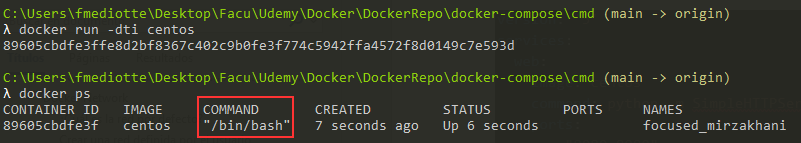
    image: centos

    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**:



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

### **Limitar recursos en contenedores (Compose v2)**