# 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.

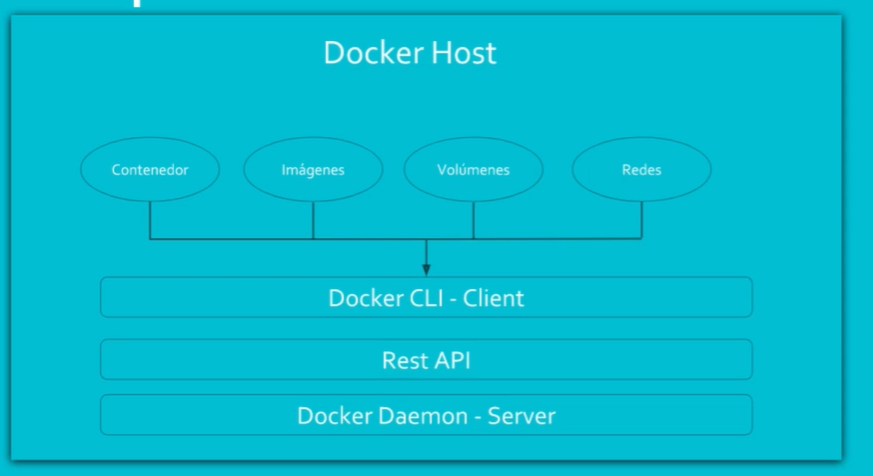
Las imágenes son muy portables y te permiten desplegar y escalar aplicaciones, como así también destruir y recrear imágenes de una manera muy fácil y rápido.

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.

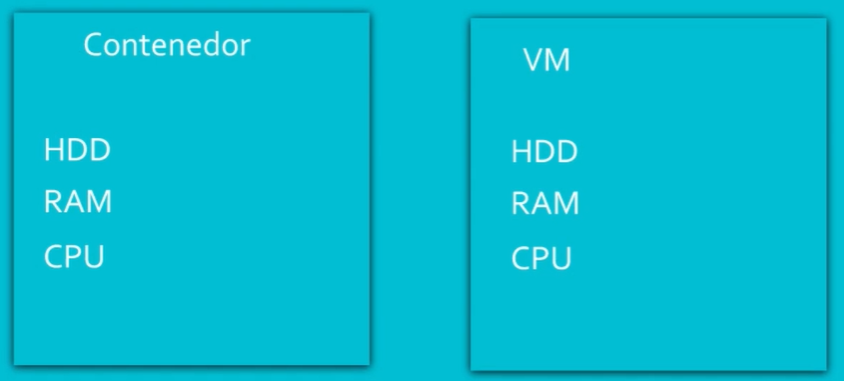
En resumen, una imagen es un paquete con las configuración y un contenedor es una capa adicional de ejecución que inicia todo lo que está definido en la imagen.



## 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

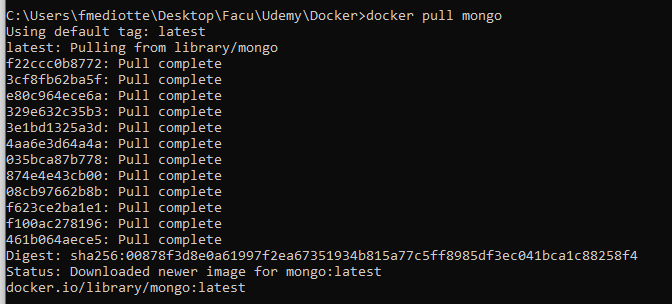
### **Imágenes oficiales**

**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.

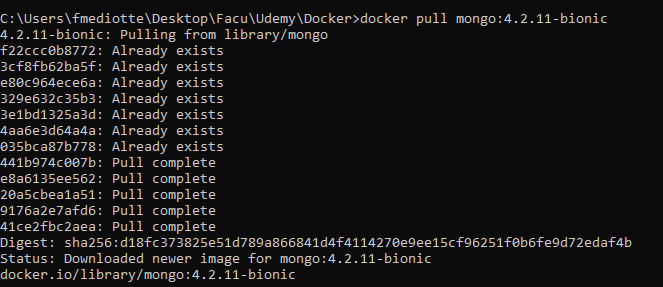
Si realizamos un Docker pull mongo por ejemplo estaremos descargando la imagen de mongo de Docker hub a nuestra máquina local, por defecto si no le especificamos ningún tag se va a descargar la última imagen que mongo haya subido a docker hub.



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

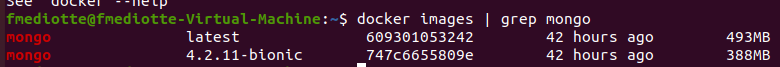
(Como ya habíamos descargado mongo previamente Docker detecta que ya tiene varias capas que ya existen localmente, por lo que solo modifica las partes que son diferentes, esto se llama **Copy on Write**)



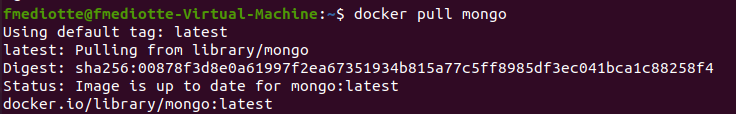
Estos tags los obtenemos de la página oficial de docker hub. En caso de que la versión que queremos descargar no está disponible podríamos nosotros crear una imagen personalizada.

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.

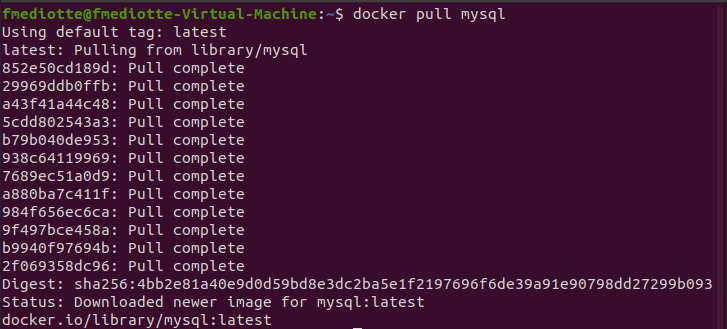


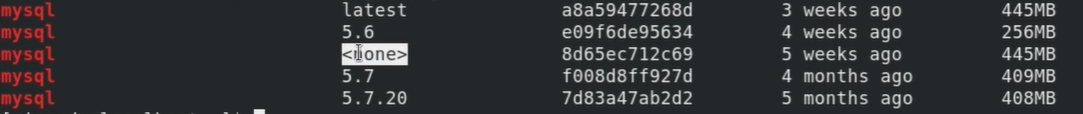
Si intentamos descargar nuevamente la imagen de mongo, lo que va a pasar es que nos va a aparecer un mensaje por consola avisándonos de que ya tenemos dicha imagen descargada y que la misma no presenta cambios para descargar.



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**

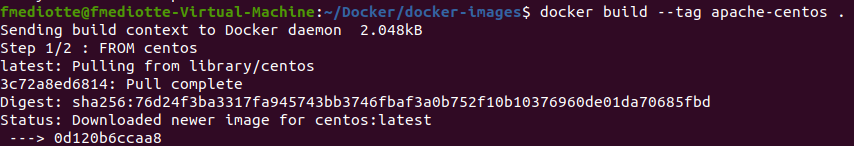
Para construir nuestra imagen en base al Docker file que generamos, se debe ejecutar el comando docker build con un nombre de tag que indica el nombre de la imagen resultante.

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

Ejemplo:

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

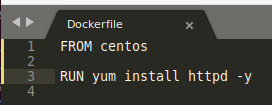
El proceso como tenemos el docker file actualmente vamos a empezar a correr los pasos que definimos en el docker file, por ejemplo:

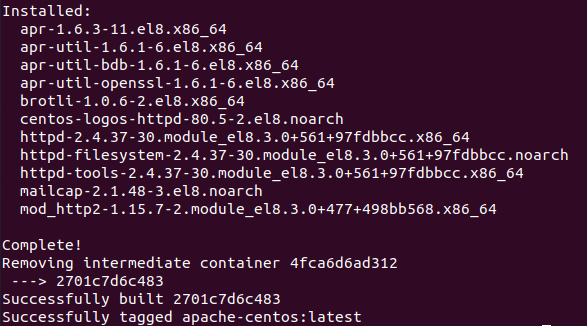


La primer capa Step 1/2 instala la imagen de centos por medio de un docker pull centos.

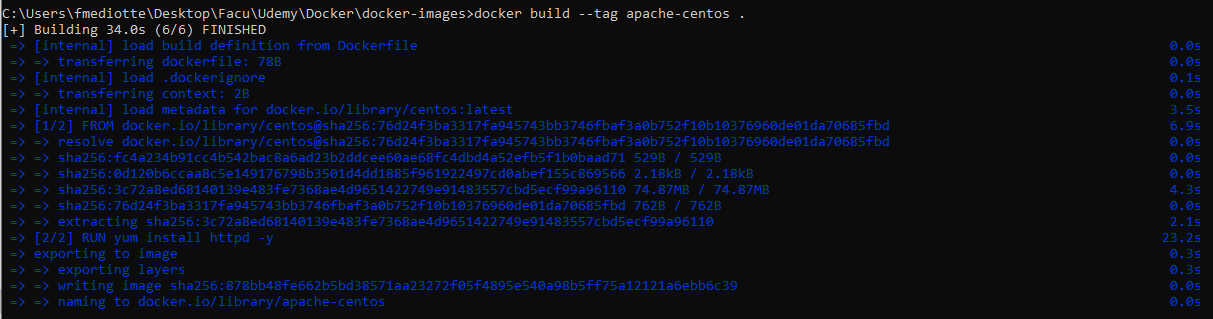
Lo que sucede en el paso 2 es tratar de descargar apache, pero desde el SO de centos y aquí va a fallar porque todo lo que creamos en docker debe ser lo más automatizado posible debido a que docker file no debería tener interacción con nosotros sino simplemente ejecutar comandos, por lo que debemos incluir 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:



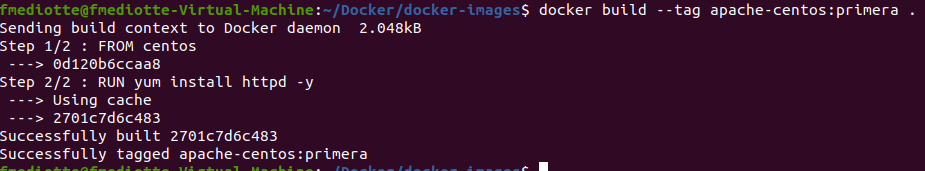
Una vez ejecutado nuevamente el comando docker build, se descargarán los paquetes necesarios para instalar apache dando como resultado un complete successfully, listando todo lo instalado:

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:

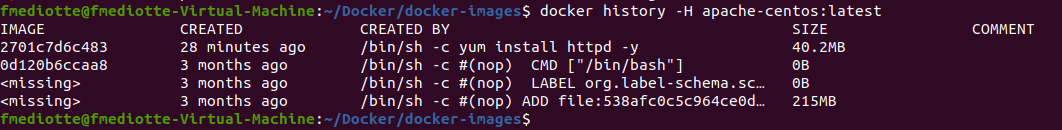
Si realizamos un docker images nuevamente veremos que tenemos 2 imágenes que en realidad son la misma imagen pero que tienen una etiqueta diferente pero que fueron creadas desde el mismo dockerfile:



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.

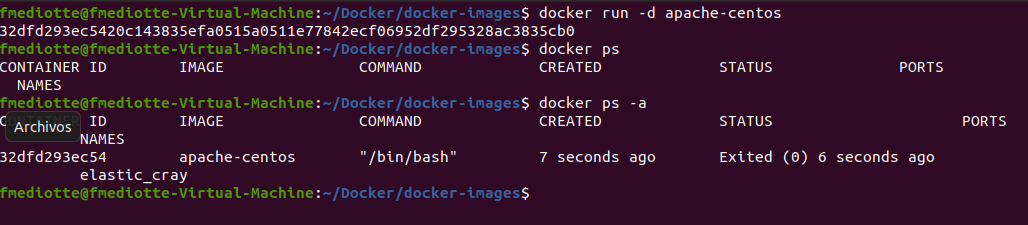
### **Creando un contenedor con nuestra imagen**

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:

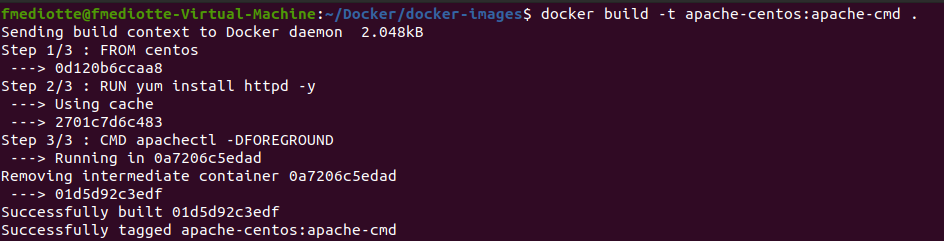
**CMD apachectl -DFOREGROUND**

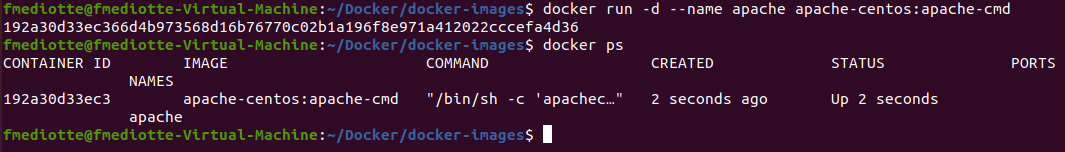
Este comando lo que hace es ejecutar el servicio de apache en primer plano.

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:

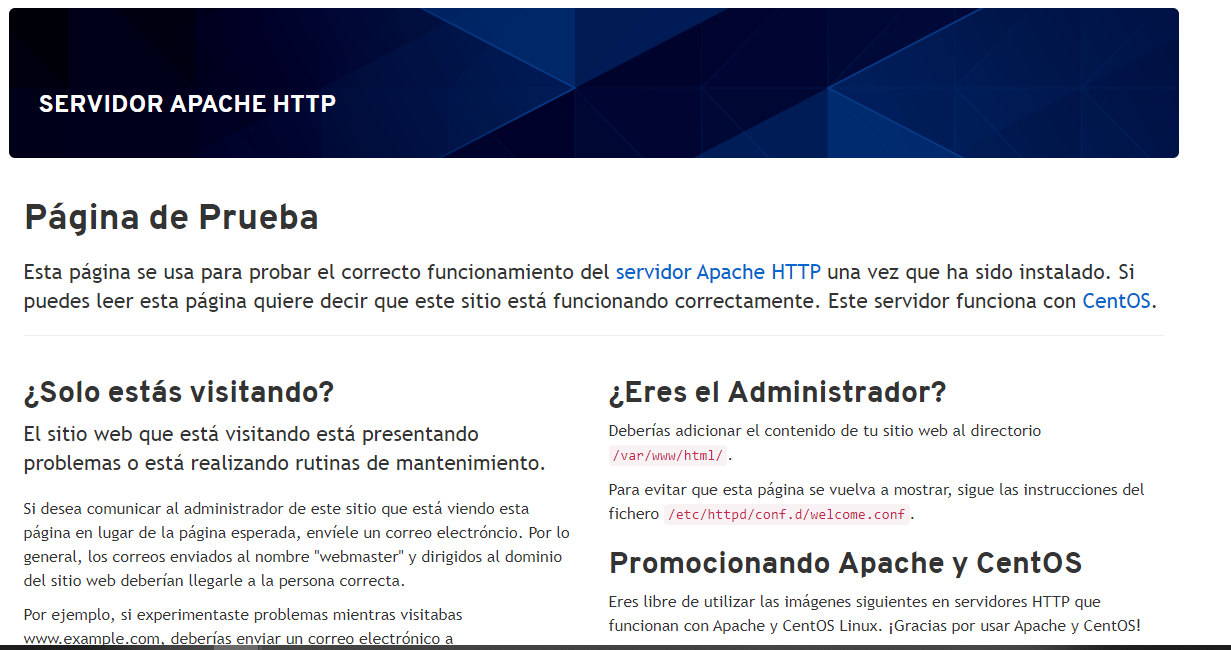
**docker rm -fv <nombre-contenedor>**

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”

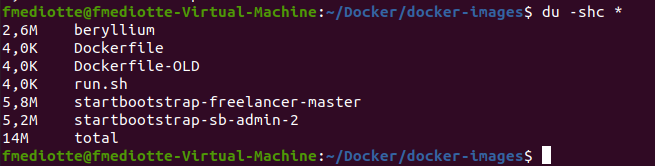
apachectl -DFOREGROUND

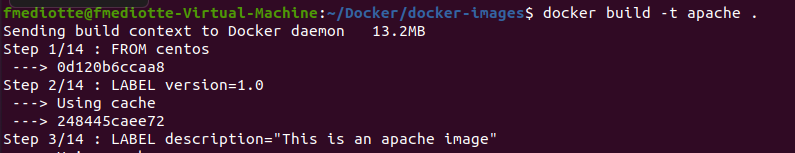
y en nuestro Dockerfile llamarlo de la siguiente manera:

COPY run.sh /run.sh

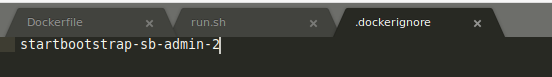
CMD sh /run.sh

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.

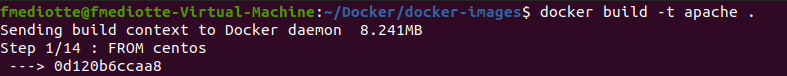




Si queremos por ejemplo ignorar uno de los archivos de dicho directorio porque no lo queremos incluir en nuestra imagen lo que se hace es poner el nombre del mismo en un archivo .dockerignore:



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**

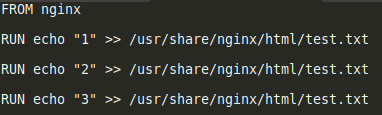
* La imagen o el servicio que está instalado debe ser efímero, es decir que se debe poder destruir con gran facilidad.
* Debería haber un solo servicio por contenedor o un solo servicio instalado por imagen.
* Si queremos excluir archivos que no queremos que estén en el contexto de Docker, cuando vayamos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

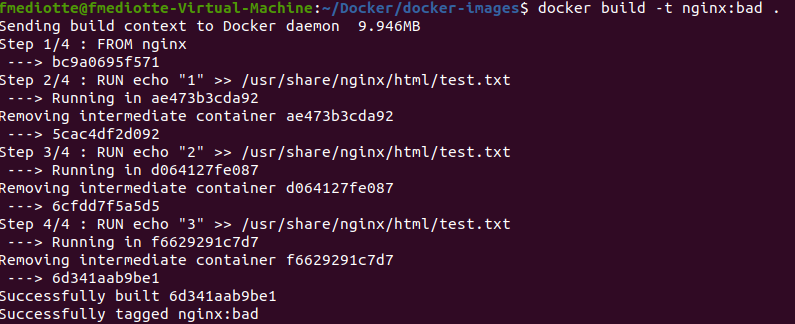
Para reducir la cantidad de capas que tiene la imagen se puede hacer la misma instrucción en una sola línea concatenando los comandos en el caso de que sea posible, por ejemplo:

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

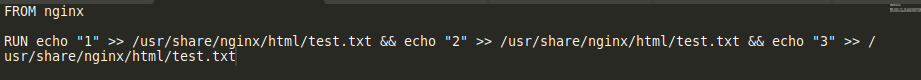
**Dockerfile**



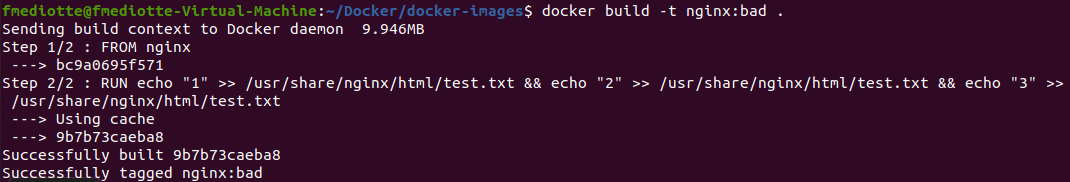
**Docker build**



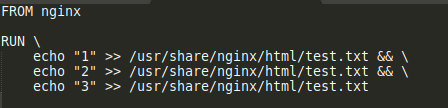
Podemos reducir esa cantidad de capas si concatenamos en el Dockerfile las directivas RUN de la siguiente manera:



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



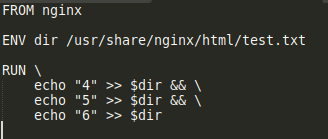
Esto se puede mejorar aún más utilizando buenas prácticas separando las líneas por un *escape* (“\”), dejando el Dockerfile de la siguiente manera:



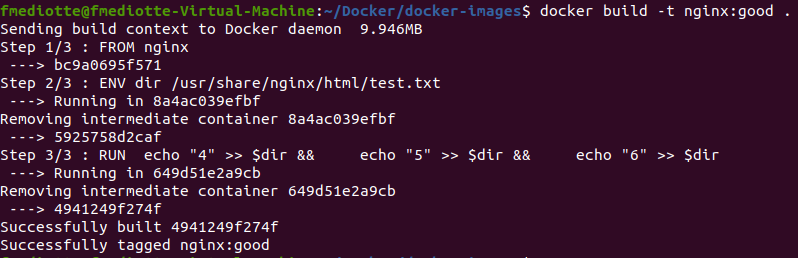
De esta forma lo que se hace es mejorar el ámbito visual, todas las tareas quedan dentro de la misma capa, el backslash significa que continua todo en la misma línea.

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

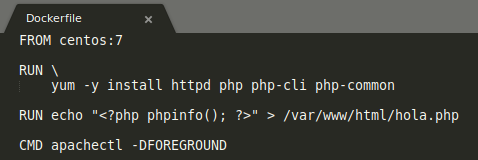
**Dockerfile:**



**Docker build:**



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



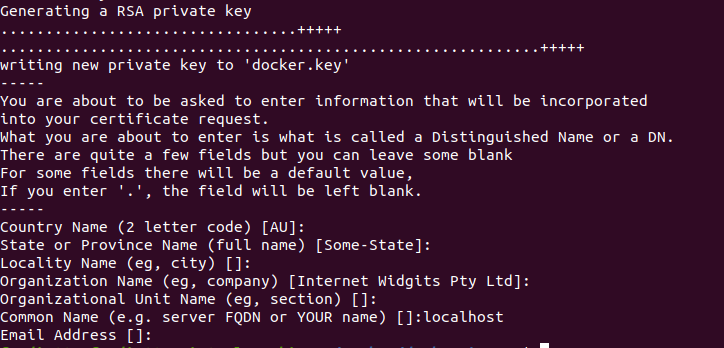
El comando yum se encarga de instalar httpd y el php ya que soporta varios argumentos de entrada. Una vez corrido el docker build con su tag correspondiente yum también se encarga de instalar las dependencias necesarias que necesite la imagen.

**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.



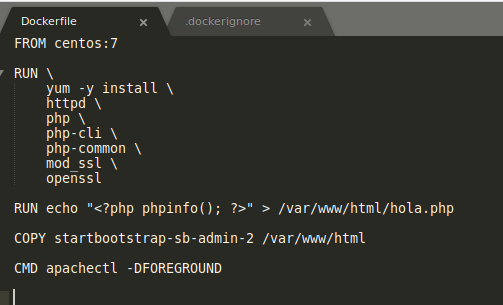
Este comando nos va a generar un par de archivos que son:

* Docker.crt
* Docker.key

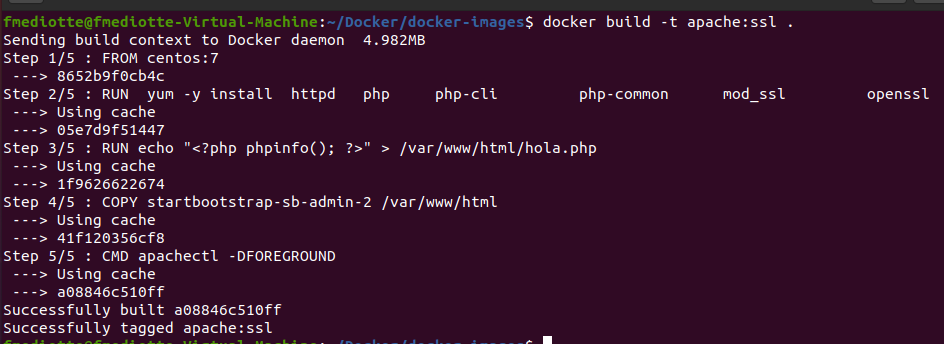
Estos archivos son los certificados SSL que necesitamos instalar en nuestro web server.

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**

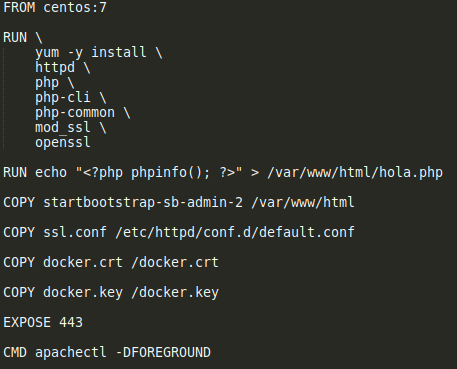
SSLCertificateKeyFile **/docker.key**

</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:

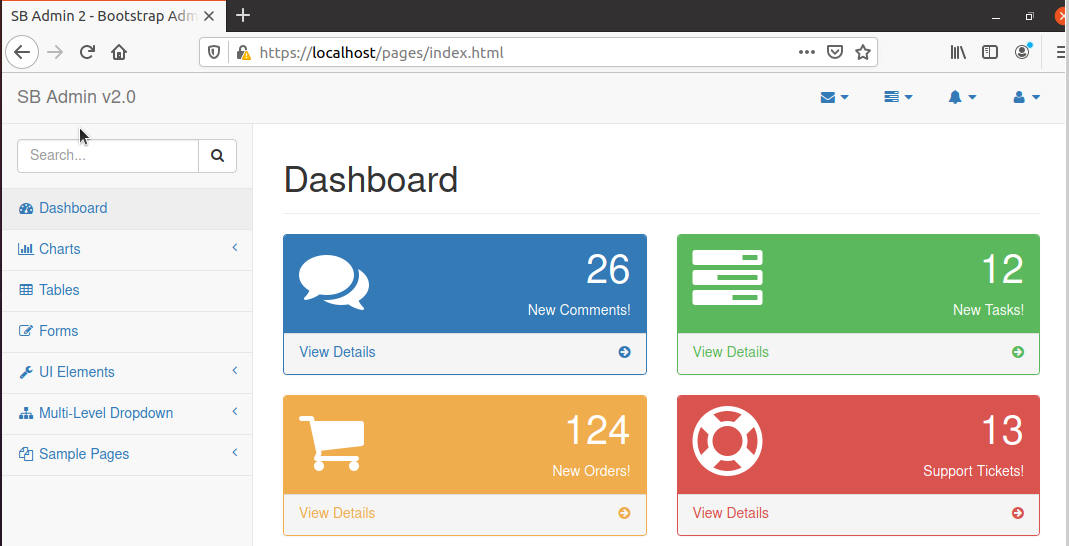


Luego solo resta correr los comandos docker build y docker run para poder acceder a nuestro sitio de forma segura.

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:

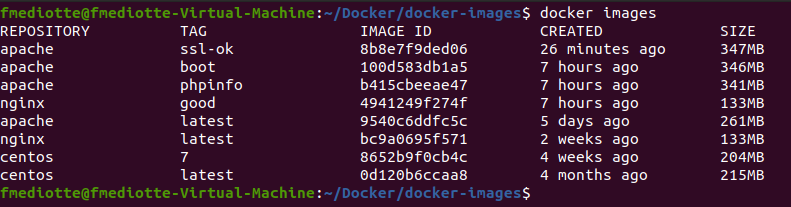
**docker run -d -p 443:443 apache:ssl-ok**

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



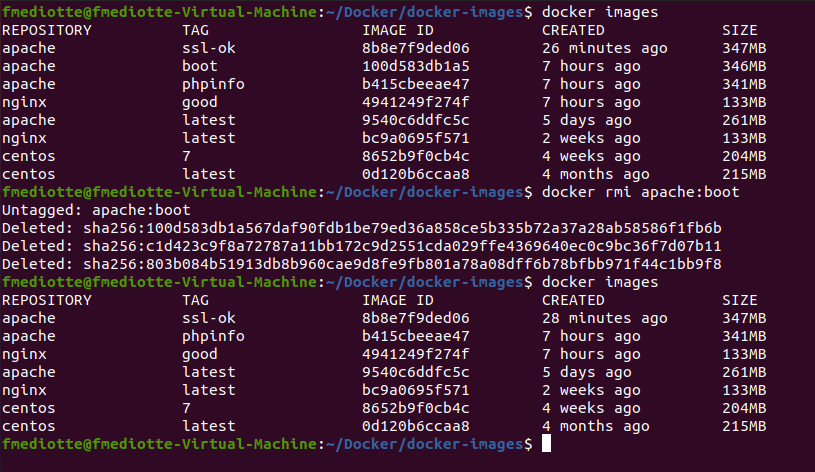
### **Eliminar imágenes**

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

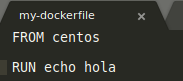
**Docker rmi argumento**, donde argumento puede ser el id de la imagen o el nombre + el tag, por ejemplo queremos eliminar la imagen apache:boot, por lo que corremos el comando **docker rmi apache:boot** y si listamos nuevamente las imágenes ya no va a estar disponible la misma para su uso.

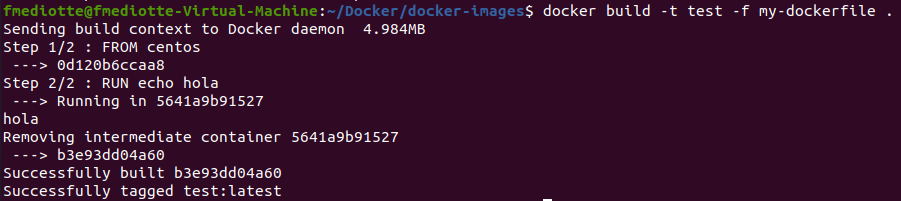


### **Cambiar el nombre del Dockerfile**

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:

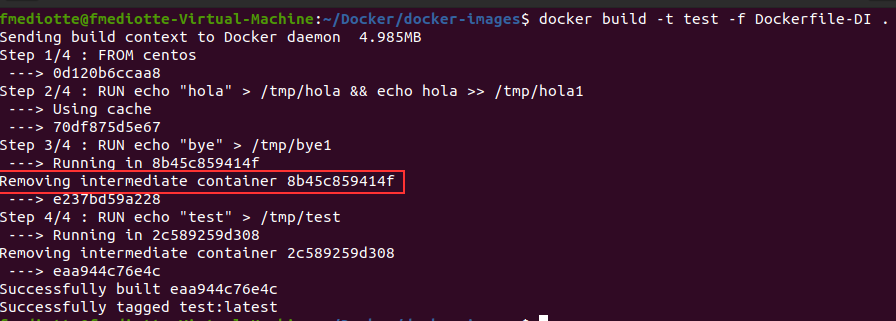
**docker build -t test -f my-dockerfile .**

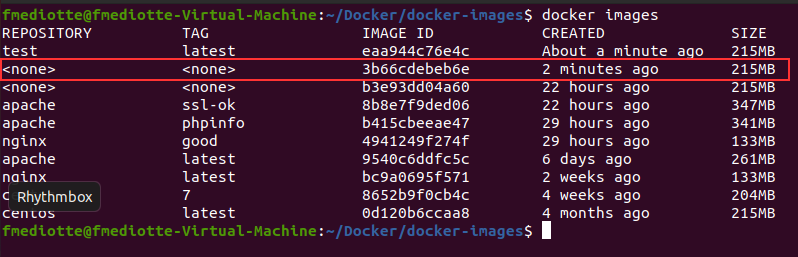
****



### **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:

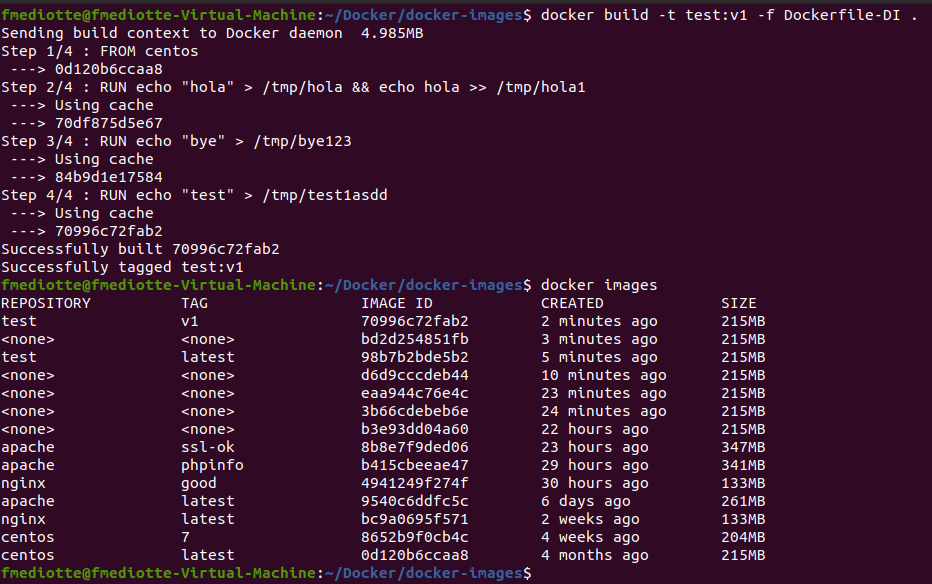


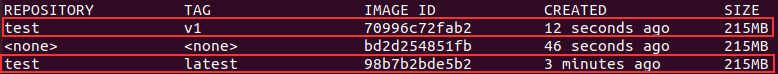


Esto sucede porque las capas de la imagen son de **SOLO LECTURA**, por lo tanto, las capas no pueden modificarse por lo que al modificar un Dockerfile, docker crea otra imagen totalmente nueva y le quita la referencia a la imagen anterior.

**¿Cómo podríamos evitar este problema?**

Definiendo tags en las imágenes:





Podríamos eliminar estas imágenes huérfanas mediante los siguiente comandos:

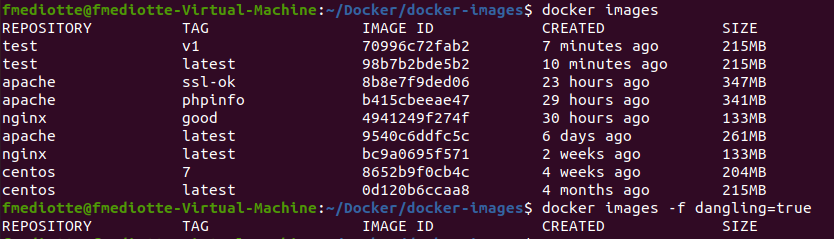
**docker images -f dangling = true**

**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)

