Iniciacion en Docker

Ernest Henry Shackleton

October 4, 2020

Contents

Figure 1: Logo of Docker, a Linux container engine.

1 Que es un contendor?

La idea de Docker, es la de crear aplicaciones/servicios que sean independientes y portables. Esto es, no importa que sistema operativo utilices o con que hardware cuentas, si puedes instalar docker, entonces podras correr tus contenedores en él. Entre las ventajas de usar docker, se encuentra la de olvidarte de instalar dependencias (ejemplo nodejs, java, python, ruby, etc) dentro de tu host o servidor y sin utilizar máquinas virtuales

Docker posibilita ir de tu maquina local a producción, con tu aplicació lista. ya no es necesario instalar en cada servidor dependencias o depender del sistema operativo. solo basta con tener docker instalado.

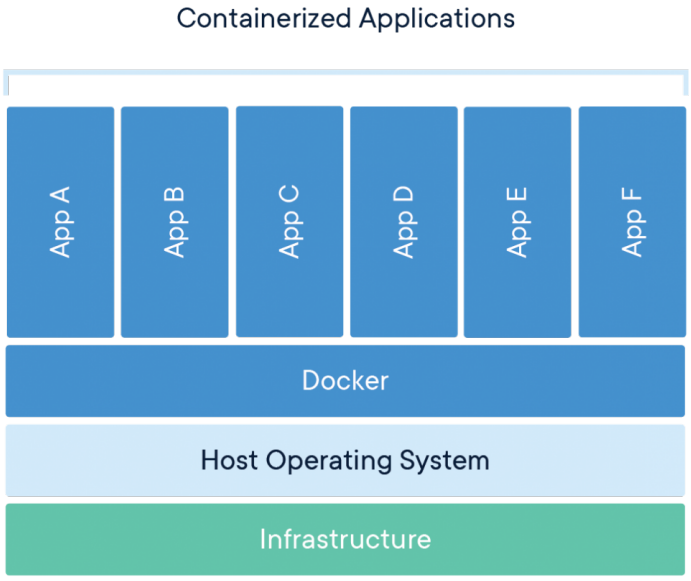

Figure 2: Arquitectura de Containers

2 Instalacion de docker

2.1 https://docs.docker.com/engine/install/

3 Conceptos importantes:

Docker en 4 minutos: https://www.youtube.com/watch?v=LYbLLKlaJ7g&ab_channel=DostorLinux

Tutorial Docker: https://www.youtube.com/watch?v=0qfxu6RiSjA&ab_channel=DostorLinux

3.1 Docker (daemon)

3.2 Docker-Machine(client)

3.3 Docker Images

3.4 Docker Containers

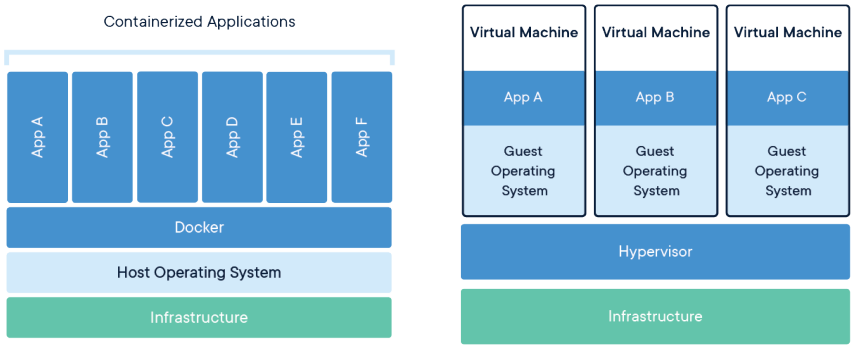

Figure 3: Comparacion Docker VS Virtual Machine

4 Introducción

La idea de Docker, es la de crear aplicaciones/servicios que sean independientes y portables. Esto es, no importa que sistema operativo utilices o con que hardware cuentas, si puedes instalar docker, entonces podras correr tus contenedores en él. Entre las ventajas de usar docker, se encuentra la de olvidarte de instalar dependencias (ejemplo nodejs, java, python, ruby, etc) dentro de tu host o servidor y sin utilizar máquinas virtuales. A lo largo

de este documento, iremos viendo el ecosistema de Docker y como todo se relaciona.

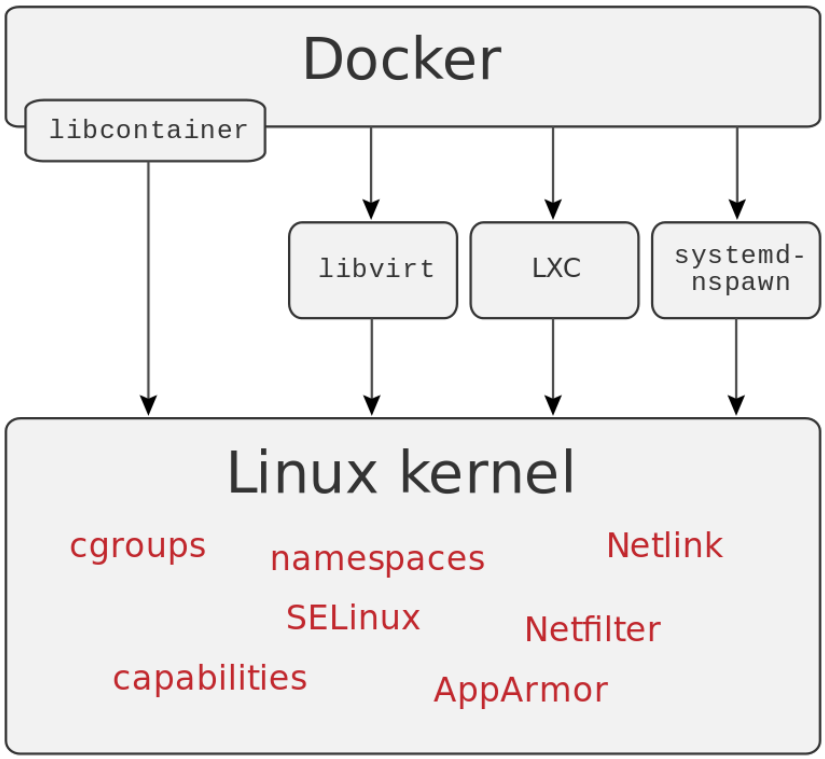

Figure 4: Docker can use different interfaces to access virtualization features of the Linux kernel.

5 Instalación

La instalación puede seguirse desde la Documentacion Oficial `https://docs.docker.com/engine/installation/`

6 Empezando

6.1 images

Muestra las imagenes locales disponibles.

```
$ docker images
```

Podemos eliminarlas con 'docker rmi <image-name>', siempre y cuando no tenga contenedores asociados (corriendo o no).

6.2 events

Podemos ver en tiempo real, los eventos que Docker lanza en nuestro servidor, solo basta con:

```
$ docker events
```

6.3 run

Corremos un contenedor con la imagen base 'busybox', que ejecuta el comando 'echo hello world' dentro. Luego de esto, el contenedor se detendrá, porque de esta forma funciona como un "job".

```
$ docker run busybox echo hello world
```

6.4 ps

Muestra los contenedores en ejecución (running).

```
$ docker ps
```

Si lo ejecutamos luego del comando anterior, no mostrará nada porque lo anterior era solo un 'build, run, die'.

Para ver el historial de ejecución y los contenedores actualmente creados (corriendo o no), ejecutamos:

```
$ docker ps -a
```

Esto nos mostrará todos los contenedores que se encuentran creados hasta el momento. Podemos borrarlos si queremos ejecutando

```
docker rm <container-id>
```

6.5 run avanzado

Podemos pasar muchas opciones al comando 'run', las cuales podemos ver con:

```
$ docker run --help
```

6.6 run interactivo

Probemos de ejecutar y usar una terminal en el contenedor:

```
$ docker run -t -i ubuntu:14.04 /bin/bash
```

- '-t': Asigna una tty
- '-i': Nos comunicamos con el contenedor de modo interactivo.

NOTA: Al salir del modo interactivo el contenedor se detendrá.

6.7 run Detached Mode

Problema: Ya sabemos cómo correr un contenedor de manera interactiva, pero el problema es que el mismo al terminar de ejecutar la tarea, finaliza. Si se quieren hacer contenedores que corran servicios (por ejemplo, un servidor web) el comando es el siguiente:

```
$ docker run -d -p 1234:1234 python:2.7 python -m SimpleHTTPServer 1234
```

Esto ejecuta un servidor Python (SimpleHTTPServer module), en el puerto '1234'. El argumento '-p 1234:1234' le indica a Docker que tiene que hacer un **port forwarding** del contenedor hacia el puerto '1234' de la máquina host.

Ahora podemos abrir un browser en la dirección '`http://localhost:1234`'.

Algo más

La opción '-d' hace que el contenedor corra en segundo plano. Esto nos permite ejecutar comandos sobre el mismo en cualquier momento mientras esté en ejecución. Por ejemplo:

```
$ docker exec -ti <container-id> /bin/bash
```

Aquí simplemente se abre una 'tty' en modo 'interativo'. Podrían hacerse otras cosas como cambiar el working directory, setear variables de entorno, etc. La lista completa puede verse [acá](`https://docs.docker.com/reference/run/`)

6.8 Ciclo de vida de un contenedor

Hasta ahora vimos cómo ejecutar un contenedor tanto en foreground como en background (detached). Ahora veremos cómo manejar el ciclo completo de vida de un contenedor. Docker provee de comandos como 'create' , 'start', 'stop', 'kill' , y 'rm'. En todos ellos podría pasarse el argumento '-h' para ver las opciones disponibles. Ejemplo:

```
docker create -h
```

Más arriba vimos cómo correr un contenedor en segundo plano (detached). Ahora veremos en el mismo ejemplo, pero con el comando 'create'. La única diferencia que esta vez no especificaremos la opción '-d'. Una vez preparado, necesitaremos lanzar el contenedor con 'docker start'.

Ejemplo:

```
$ docker create -P --expose=8001 python:2.7 python -m SimpleHTTPServer 8001
  a842945e2414132011ae704b0c4a4184acc4016d199dfd4e7181c9b89092de13

$ docker ps -a
  CONTAINER ID IMAGE      COMMAND               CREATED       ... NAMES
  a842945e2414 python:2.7 "python -m SimpleHTT 8 seconds ago ... fervent_hodgkin

$ docker start a842945e2414
  a842945e2414

$ docker ps
  CONTAINER ID IMAGE      COMMAND               ... NAMES
  a842945e2414 python:2.7 "python -m SimpleHTT ... fervent_hodgkin
```

Siguiendo el ejemplo, para detener el contenedor se puede ejecutar cualquiera de los siguientes comandos:

```
docker kill a842945e2414 # (envía SIGKILL)

docker stop a842945e2414 # (envía SIGTERM).
```

Así mismo, pueden reiniciarse:

```
docker restart a842945e2414
```

o destruirse:

```
docker rm a842945e2414
```

6.9 Crear una imagen Docker con un Dockerfile

Problema:

Ya entendemos cómo se descargan las imágenes del Docker Registry. ¿Qué pasa si ahora quisiéramos armar nuestras propias imagenes?

Solución:

Usando un [Dockerfile](`https://docs.docker.com/engine/reference/builder/`). Un 'Dockerfile' es un archivo de texto, que describe los pasos (secuenciales) a seguir para preparar una imagen Docker. Esto incluye instalación de paquetes, creación de directorios, definición de variables de entorno, ETC. Toda imagen que creemos, parte de una base image. Como en otro de los ejemplos, usaremos la imagen [busybox](`https://busybox.net/about.html`) la cual combina utilidades UNIX en un único y simple ejecutable.

Comenzando:

Crearemos nuestra propia imagen con la imagen base busybox y seteare-mos sólo una variable de entorno para mostrar el funcionamiento.

El siguiente comando crea el directorio cuatrolibertades y se posiciona dentro de él:

```
$ mkdir cuatrolibertades && cd cuatrolibertades
```

Creamos el Dockerfile:

```
$ touch Dockerfile
```

Escribimos las siguientes líneas dentro del 'Dockerfile':

```
FROM busybox
ENV foo=bar
```

Hecho esto, haremos un 'build' de la imagen con el nombre 'my-busybox-4lib':

```
$ docker build -t my-busybox-4lib . #el . (punto) es importante
```

Si todo salió bien al hacer

```
$ docker images
```

deberíamos encontrar nuestra imagen.

Ejemplo Real: Wordpress Dockerizado. Para esto usaremos MySql y HTTPD (apache o nginx).

Problema:

Como Docker ejecuta procesos en foreground, necesitamos encontrar la forma de ejecutar varios de estos simultáneamente. La directiva 'CMD' que veremos más adelante, sólo ejecutará una instrucción. Es decir, si tenemos varios 'CMD' dentro de un Dockerfile, ejecutará sólo el último.

Solución:

Usando [Supervisor](`http://supervisord.org/index.html`) para monitorear y ejecutar MySql y HTTPD. Supervisor se encarga de controlar varios procesos y se ejecuta como cualquier otro programa.

Veremos diferentes formas de hacer esto. En principio crearemos todo dentro de un único contenedor, pero luego explotaremos al máximo los principios y características de Docker para hacerlo, por ejemplo separar servicios en diferentes contenedores y linkearlos.

6.10 Usando Supervisor y en un único contenedor

Creamos el Dockerfile, con este contenido:

```
# Imagen Base
FROM ubuntu:14.04

# Instalamos dependencias
  # apache2: Servidor Web
  # php5: Lenguaje de programacion PHP
  # php5-mysql: Driver de MySql para PHP
  # supervisor: Lanzadaror y Monitor de procesos
  # wget: Utilidad para obtener archivos via HTTP
RUN apt-get update && apt-get -y install \
  apache2 \
  php5 \
  php5-mysql \
  supervisor \
  wget

# mysql-server se instala con intervención del usuario,
# pero como no es modo interactivo lo que hacemos es setearle las variables
# con un valor.
# Para simplificar hemos usado como usuario y contraseña de mysql 'root'
RUN echo 'mysql-server mysql-server/root_password password root' | \
  debconf-set-selections && \
  echo 'mysql-server mysql-server/root_password_again password root' | \
```

```
    debconf-set-selections

  # Procedemos ahora sí, a instalar mysql-server
  RUN apt-get install -qqy mysql-server

  # Preparamos Wordpress
    # Obtenemos la última versión
    # Descomprimimos
    # Copiamos el contenido dentro del root del servidor
    # Removemos el viejo index.html (mensaje de bienvenida de apache)
  RUN wget http://wordpress.org/latest.tar.gz && \
    tar xzvf latest.tar.gz && \
    cp -R ./wordpress/* /var/www/html && \
    rm /var/www/html/index.html

  # De esto se encargaría supervisor, pero como necesitamos crear la base de datos
  # ejecutamos a mysql en background y creamos la base de datos llamada wordpress
  RUN (/usr/bin/mysqld_safe &); sleep 5; mysqladmin -u root -proot create wordpress

  # Reemplazamos el archivo wp-config.php (más abajo lo creamos) a la carpeta de wordp
  # Este archivo contiene la configuración de nuestro sitio
  COPY wp-config.php /var/www/html/wp-config.php

  # Copiamos el archivo de configuración de supervisor (más abajo lo creamos)
  COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

  # Le decimos al contenedor que tiene que hacer accesible al puerto 80 (en el que cor
  # para así nosotros poder acceder al mismo desde fuera
  EXPOSE 80

  # Lanzamos Supervisor como proceso Foreground de Docker
  # Este se encargará de lanzar simultaneamente los demás :D
  CMD ["/usr/bin/supervisord"]
```

Creamos el archivo 'supervisord.conf' con este contenido:

```
[supervisord]
nodaemon=true

[program:mysqld]
```

```
command=/usr/bin/mysqld_safe
autostart=true
autorestart=true
user=root

[program:httpd]
command=/bin/bash -c "rm -rf /run/httpd/* && /usr/sbin/apachectl -D FOREGROUND"
```

Creamos el archivo 'wp-config.php' con este contenido:

```
<?php
/**
 * The base configurations of the WordPress.
 *
 * This file has the following configurations: MySQL settings, Table Prefix,
 * Secret Keys, and ABSPATH. You can find more information by visiting
 * {@link http://codex.wordpress.org/Editing_wp-config.php Editing wp-config.php}
 * Codex page. You can get the MySQL settings from your web host.
 *
 * This file is used by the wp-config.php creation script during the
 * installation. You don't have to use the web site, you can just copy this file
 * to "wp-config.php" and fill in the values.
 *
 * @package WordPress
 */

// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'root');

/** MySQL database password */
define('DB_PASSWORD', 'root');

/** MySQL hostname */
define('DB_HOST', 'localhost');

/** Database Charset to use in creating database tables. */
```

```
define('DB_CHARSET', 'utf8');

/** The Database Collate type. Don't change this if in doubt. */
define('DB_COLLATE', '');

/**#@+
 * Authentication Unique Keys and Salts.
 *
 * Change these to different unique phrases!
 * You can generate these using the {@link https://api.wordpress.org/secret-key/1.1/
 * You can change these at any point in time to invalidate all existing cookies. This
 *
 * @since 2.6.0
 */
define('AUTH_KEY',         'put your unique phrase here');
define('SECURE_AUTH_KEY',  'put your unique phrase here');
define('LOGGED_IN_KEY',    'put your unique phrase here');
define('NONCE_KEY',        'put your unique phrase here');
define('AUTH_SALT',        'put your unique phrase here');
define('SECURE_AUTH_SALT', 'put your unique phrase here');
define('LOGGED_IN_SALT',   'put your unique phrase here');
define('NONCE_SALT',       'put your unique phrase here');

/**#@-*/

/**
 * WordPress Database Table prefix.
 *
 * You can have multiple installations in one database if you give each a unique
 * prefix. Only numbers, letters, and underscores please!
 */
$table_prefix  = 'wp_';

/**
 * For developers: WordPress debugging mode.
 *
 * Change this to true to enable the display of notices during development.
 * It is strongly recommended that plugin and theme developers use WP_DEBUG
 * in their development environments.
 */
```

```
define('WP_DEBUG', false);

/* That's all, stop editing! Happy blogging. */

/** Absolute path to the WordPress directory. */
if ( !defined('ABSPATH') )
 define('ABSPATH', dirname(__FILE__) . '/');

/** Sets up WordPress vars and included files. */
require_once(ABSPATH . 'wp-settings.php');
```

Ahora sólo queda realizar el build de nuestra imagen y luego ejecutar un contenedor

```
$ docker build -t wordpress .
$ docker run -d -p 80:80 wordpress
```

Una vez funcionando, ingresando en '`http://`<IP OF DOCKER HOST>' deberíamos visualizar la página de instalación de wordpress.

Nota:

Usar Supervisor para ejecutar varios servicios dentro del mismo contenedor, podría trabajar perfectamente, pero es mejor usar múltiples contenedores. Estos proveen del aislamiento (isolation) entre otras bondades de Docker, y nos ayuda además a crear una aplicación basada en [microservicios](`http://bit.ly/building-microservices`). Por último, también esto nos ayuda a escalar y a recuperarnos de posibles fallas.

6.11 Corriendo Wordpress usando 2 contenedores linkeados.

Problema:

Hasta ahora ejecutamos una instancia de wordpress con su servidor y su base de datos, en un mismo contenedor. El problema es que no explotamos al máximo a Docker, y no mantenemos tampoco el concepto de Separation of concerns. Necesitamos desacoplar el contenedor lo más fino posible.

Solución:

Usar 2 contenedores. Uno para Wordpress y otro para MySql. Luego se interconectarán mediante la opción de docker '–link'.

Manos a la obra:

Para este ejemplo usaremos las imágenes docker oficiales de wordpress y mysql.

```
$ docker pull wordpress:latest
$ docker pull mysql:latest
```

Ejecutamos un contenedor MySql

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker \
                            -e MYSQL_DATABASE=wordpress \
                            -e MYSQL_USER=wordpress \
                            -e MYSQL_PASSWORD=wordpresspwd \
                            -v /db/mysql:/var/lib/mysql \
                            -d mysql
```

NOTA: Aquí hay nuevas opciones:

- '-e' es para setear variables de entorno. Esas variables están definidas dentro del Dockerfile de MySql, por lo que nosotros le damos valor, para que el contenedor a ejecutar, use esos datos.
- '-v' es para montar un volumen entre el host y el contenedor. En este caso en el host se populará el volumen '/db/mysql/' con la info de '/var/lib/mysql'.
  - Los volúmenes tienen diferentes usos:
    - Se crean cuando se inicializa el contenedor
    - Compartir información entre diferentes contenedores
    - Mantener la info luego de haber borrado el contenedor
    - Cambios en los volúmenes son directamente aplicados (no hay que hacer nada adicional con el contenedor para actualizar)
    - Los cambios de un volumen no se incluirán en la actualización de la imagen

Ejecutamos y linkeamos a wordpress

```
$ docker run --name wordpress --link mysqlwp:mysql -p 80:80 \
                              -e WORDPRESS_DB_NAME=wordpress \
                              -e WORDPRESS_DB_USER=wordpress \
                              -e WORDPRESS_DB_PASSWORD=wordpresspwd \
                              -d wordpress
```

NOTA: La imagen de wordpress, expone el puerto 80 y lo que hacemos es mapearlo con el 80 del nuestro Host. Como en la imagen de MySql, en wordpress tambien contamos con algunas variables de entorno, éstas para la configuración del mismo. Básicamente seteamos las credenciales de la base de datos anteriormente creada, para que wordpress use las mismas.

6.12 Haciendo backups de la base de datos de un contenedor

Problema:

Tenemos un contenedor de mysql ejecutando, pero necesitamos hacer un backup de la base de datos que se ejecuta dentro del contenedor.

Solucion:

Usar el comando 'docker exec' para ejecutar en el contenedor MySql el comando 'mysqldump'

Chequeamos en nuestro host que existe la carpeta '/db/mysql'

```
$ ls /db/mysql
```

Ahora, para hacer un backup de la base de datos de ese contenedor ejecutamos:

```
docker exec mysqlwp mysqldump --all-databases --password=wordpressdocker > wordpress.ba
```

Ahora ejecutamos

```
$ ls
```

y veremos el archivo wordpress.backup

6.13 Compartir información entre el Docker Host y los contenedores

Problema:

- Tenemos información local, que queremos que este disponible en un contenedor (por ejemplo, en desarrollo, el codigo de tu aplicación).
- Tenemos informacion del contenedor que necesitamos guardar en el host (por ejemplo una base de datos)

Solución:

Usando volúmenes (opción '-v' antes vista) para montar uno entre el host y el contenedor. Por ejemplo, si queremos compartir nuestro directorio de trabajo, con un directorio particular del contenedor podríamos hacer:

```
docker run -ti -v "$PWD":/pepe ubuntu:14.04 /bin/bash
```

Lo que hicimos con ese comando, fue montar como volumen nuestro directorio actual con el directorio '/cuatrolibertades' en el contenedor (OJO, '/' referencia al root del filesystem). Además, como vimos antes con las opciones '-ti' levantamos un tty y de modo interativo ejecutamos una instancia de bash.

Algo más:

Docker provee de un comando

```
$ docker inspect
```

que sirve para observar la información de un contenedor

```
docker inspect -f {{.Mounts}} <container-id>
```

Con el comando anterior, filtramos de toda la información, solo los puntos de montaje. Como salida obtendremos algo como: '[{ /path/to/pwd /cuatrolibertades true}]'

6.14 Disclaimer

Docker tiene un [apartado sobre volúmenes](`https://docs.docker.com/storage/`) que es muy importante conocer. Cada tipo de volumen (bind, tmpfs, named) tienen usos distintos, y hacer uso del equivocado podria traernos problemas de performance o aun peor, perdida de datos si no usamos los comandos adecuadamente. Si te encontras con problemas de performance en OSX, probablemente esto te sea util [docker sync](`http://docker-sync.io/`)

6.15 Compartir información entre contenedores

Problema:

Ya sabemos cómo montar un volumen de nuestro Host en un contenedor. Pero ahora quisiéramos compartir ese volumen definido en el contenedor con otros contenedores.

Solución:

Usando data containers. Cuando queremos montar un volúmen en un contenedor lo que hacemos es con el argumento '-v' decirle el directorio X del host que debe montarse en el el path Y del contenedor. El volúmen especificado se crea como de lectura-escritura dentro del contenedor y no como las capas de sólo lectura usadas para crear el contenedor, pudiéndose modificar también desde la máquina host.

```
$ docker run -ti --name=cont1 -v /pepe ubuntu:14.04 /bin/bash
root@cont1:/# touch /pepe/foobar
root@cont1:/# ls pepe/
foobar
root@cont1:/# exit
exit
bash-4.3$ docker inspect -f {{.Mounts}} cont1
[{dbba7caf8d07b862b61b39... /var/lib/docker/volumes/dbba7caf8d07b862b61b39... \
/_data /pepe local true}]
$ sudo ls /var/lib/docker/volumes/dbba7caf8d07b862b61b39...
foobar
```

Y ahora ejecutamos otro contenedor con el volumen anteriormente creado.

```
$ docker run --volumes-from=cont1 --name=cont2 ubuntu:14.04
$ docker inspect -f {{.Mounts}} cont2
[{4ee1d9e3d453e843819c6ff... /var/lib/docker/volumes/4ee1d9e3d453e843819c6ff... \
/_data /pepe local true]
```

6.16 Copiando datos entre el host desde y para los contenedores

Problema:

Tenemos un contenedor que no tiene volúmenes configurados, y queremos copiar archivos desde y en el contenedor.

Solución:

Usando

```
$ docker cp
```

para pasar información desde y para un contenedor en ejecución Podemos ver más opciones con 'docker cp –help' o sólo 'docker cp'.

Por ejemplo, para pasar archivos desde el docker host hacia el contenedor:

```
$ docker run -d --name testcopy ubuntu:14.04 sleep 360
$ touch pepe.txt
$ docker cp pepe.txt testcopy:/root/file.txt
```

Y pasando del contenedor hacia el docker host:

```
$ docker cp testcopy:/root/file.txt pepe.txt
$ ls
pepe.txt
```

6.17 Crear y compartir Docker Images

Después de crear varios contenedores, tal vez quisiéramos crear nuestras propias imágenes también. Cuando iniciamos un contenedor, al mismo lo iniciamos desde una imagen base. Una vez con el contenedor en ejecución nosotros podríamos hacer cambios, por ejemplo, instalarle ciertas librerías o dependencias (ejemplo, correr 'apt install htop vim git' dentro de un contenedor que tiene de imagen base, 'ubuntu'). Luego de haber ejecutado este comando, el contenedor ha modificado su filesystem. Nosotros a futuro tal vez quisiéramos ejecutar contenedores iguales al anterior, por lo que Docker nos provee del comando 'commit' para, a partir de un contenedor, crear una imagen. Docker mantiene las diferencias entre la imagen base y la que se quiere crear, creando una nueva layer usando [UnionFS](`https://es.wikipedia.org/wiki/UnionFS`). Similar a git.

Crearemos un contenedor de ubuntu, y al mismo le actualizaremos la lista de repositorios. Luego de ello, haremos un 'docker commit', para definir la nueva imagen para mantener una imagen mas actualizada.

```
$ docker run -t -i --name=contenedorPrueba ubuntu:14.04 /bin/bash
root@69079aaaaab1:/# apt update
```

Cuando salgamos de este contenedor, el mismo se detendrá, pero seguirá estando disponibles a menos que lo eliminemos explícitamente con 'docker rm'. Ahora commitiemos el contenedor, para crear una nueva imagen.

```
$ docker commit contenedorPrueba ubuntu:update
13132d42da3cc40e8d8b4601a7e2f4dbf198e9d72e37e19ee1986c280ffcb97c
$ docker images
REPOSITORY    TAG     IMAGE ID      CREATED          VIRTUAL SIZE
ubuntu        update  13132d42da3c  5 days ago  ...  213 MB
```

NOTA: Esto 'ubuntu:update' especifica '$<\text{nombre}_{\text{imagen}}>:<\text{tag}_{\text{del commit}}>$'.

Luego ya podremos lanzar contenedores basados en la nueva imagen 'ubuntu:update'.

ADICIONAL

Podemos chequear las diferencias con 'docker diff'.

```
$ docker diff contenedorPrueba
C /root
A /root/.bash_history
C /tmp
```

```
C /var
C /var/cache
C /var/cache/apt
D /var/cache/apt/pkgcache.bin
D /var/cache/apt/srcpkgcache.bin
C /var/lib
C /var/lib/apt
C /var/lib/apt/lists
...
```

6.18 Guardando Images y Containers como archivos .tar para compartir

Problema: Tenemos creados imagenes o tenemos contenedores que queremos mantener y nos gustaría compartirlo con nuestros colaboradores.

Solución:

- Para las ‘images‘: Usar los comandos ‘save‘ y ‘load‘ para crear el archivo comprimido de la imagen anteriormente creada.
- Para los ‘containers‘: Usar los comandos ‘import‘ y ‘export‘.

Comencemos con un ‘container‘ creado y exportándolo en un archivo ‘.tar‘ (tarball).

```
$ docker ps -a
CONTAINER ID  IMAGE         COMMAND       CREATED         ...   NAMES
77d9619a7a71  ubuntu:14.04  "/bin/bash"   10 seconds ago  ...   high_shockley
$ docker export 77d9619a7a71 > update.tar
$ ls
update.tar
```

Se puede hacer ‘commit‘ de este contenedor como una nueva imagen local, pero tambien se podría usar el comando ‘import‘:

```
$ docker import - update < update.tar
157bcbb5fdfce0e7c10ef67ebdba737a491214708a5f266a3c74aa6b0cfde078
$ docker images
REPOSITORY  TAG     IMAGE ID      ...   VIRTUAL SIZE
update      latest  157bcbb5fdfc  ...   188.1 MB
```

Si se quiere compartir esta imagen con uno de sus colaboradores, podría subirse el tarball a un webserver y decirle al colaborar que descarga tal, y use el comando 'import' en su Docker Host. Si se prefiere usar imagenes que ya se han comitiado, se puede usar los comandos 'load' y 'save' mencionados anteriormente.

Entonces, ¿Cuál es la diferencia?

Los 2 métodos son similares; La diferencia está en que guardando una imagen mantenemos el historial de cambios, y exportándola como contenedor NO.

A mi punto de vista, tal vez lo mejor sería sólo mantener los cambios cuando ya es algo en producción y deseamos hacer actualización de software. Por ejemplo del SO o de APACHE/NGINX, donde si ocurre una falla o incompatibilidad, podría volverse atrás. En cambio mientras estamos haciendo el desarrollo, mantener los cambios tal vez no sea tan importante.

6.19 Escribiendo nuestro primer DockerFile

Problema:

Ejecutar contenedores en modo interactivo, hacer algunos cambios y para luego comitear estos en una nueva imagen, funciona bien. Pero en la mayoría de los casos, tal vez quieras automatizar este proceso de creación de nuestra propia imagen y compartir estos pasos con otros.

Solución:

Para automatizar el proceso de creación de imágenes Docker, prepararemos tales paso en un archivo de manifiesto, llamado **Dockerfile**. Este archivo de texto está compuesto por una serie de instrucciones que describe cuál es la imagen base de la que el nuevo contenedor se basará, cuáles pasos necesitan llevarse a cabo para instalar las dependencias de la aplicación, cuáles archivos necesitan estar presentes en la imagen, cuáles puertos serán expuestos por el contenedor y cuáles comando ejecutar cuando se ejecuta el contenedor, entre otras cosas.

Para ilustrar esto, crearemos un simple Dockerfile. La **imagen** resultante nos permitirá crear un contenedor que ejecuta el comando '/bin/echo'.

```
FROM ubuntu:14.04

ENTRYPOINT ["/bin/echo"]
```

La instrucción 'FROM' dice de cuál **imagen base** partimos para crear la nuestra. En este caso 'ubuntu:14.04', que la primera vez será descargada del repositorio del Docker Hub.

La instrucción 'ENTRYPOINT' dice cuál es el comando a ejecutar cuando el contenedor basado en esta imagen, sea ejecutado.

Para hacer 'build' de esta imagen, ejecutamos 'docker build .'

Hecho el build, ejecutamos un nuevo contenedor a partir de esta imagen:

```
docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUA
<none>              <none>              99fac58824c2        5 minutes ago       187.9 
```

```
docker run 99fac58824c2 Hi Docker!
Hi Docker !
```

Lo que hemos hecho es ejecutar un contenedor a partir de la imagen previamente creada, pasándole como argumento 'Hi Docker¡. El contenedor al ejecutarse, corrió el comando definido por el 'ENTRYPOINT', seguido por el argumento anteriormente mencionado. Una vez que el comando ha **finalizado** (la tarea finaliza), el contenedor es finalizado también.

Tambien podemos usar la instrucción 'CMD' en un Dockerfile. Esta tiene la ventaja que se puede sobreescribir cuando este se ejecuta, pasándolo como argumento. Por ejemplo:

```
FROM ubuntu:14.04

CMD ["/bin/echo" , "Hi Docker !"]
```

Construimos la nueva imagen:

```
$ docker build .
```

Ejecutamos un contenedor a partir de esta:

```
docker run 99fac58824c2
Hi Docker!
```

Y ahora sobreescribiendo el comando:

```
docker run 99fac58824c2 /bin/date
Thu Mar 17 00:14:00 UTC 2016
```

Si el Dockerfile utiliza la instrucción 'ENTRYPOINT' y necesitamos hacer override, se le puede pasar la opción '–entrypoint' al 'docker run'.

Tenemos una imagen creada, pero como verán no tiene un 'tag' y siempre nos referimos a ella por su 'IMAGE ID'. Para esto podemos hacer un rebuild usando la opción '-t'.

```
$ docker build -t ubuntu-echo:1.0.0 .
...
...
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL
ubuntu-echo         1.0.0           99fac58824c2        About an hour ago   187.9 MB
...
```

Podemos colocarle el nombre que querramos, pero siempre es mejor seguir las convenciones

'<name-of-recipe>:<version-of-recipe>'

El comando 'build' tiene una serie de opciones configurables y pueden verse con la opcion -h

```
$ docker build -h

Usage: docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

  --build-arg=[]                  Set build-time variables
  --cpu-shares=0                  CPU shares (relative weight)
  --cgroup-parent=                Optional parent cgroup for the container
  --cpu-period=0                  Limit the CPU CFS (Completely Fair Scheduler) period
  --cpu-quota=0                   Limit the CPU CFS (Completely Fair Scheduler) quota
  --cpuset-cpus=                  CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems=                  MEMs in which to allow execution (0-3, 0,1)
  --disable-content-trust=true    Skip image verification
  -f, --file=                     Name of the Dockerfile (Default is 'PATH/Dockerfile'
  --force-rm=false                Always remove intermediate containers
  --help=false                    Print usage
  -m, --memory=                   Memory limit
  --memory-swap=                  Total memory (memory + swap), '-1' to disable swap
  --no-cache=false                Do not use cache when building the image
  --pull=false                    Always attempt to pull a newer version of the image
```

```
  -q, --quiet=false               Suppress the verbose output generated by the contain
  --rm=true                       Remove intermediate containers after a successful bu
  -t, --tag=                      Repository name (and optionally a tag) for the image
  --ulimit=[]                     Ulimit options
```

6.20 Empaquetando una aplicación Flask en un contenedor

Problema

Tenemos una aplicación web buildeada en **Flask** corriendo en nuestro Ubuntu 14.04 y queremos correrla en un contenedor.

Solución

Como un ejemplo, vamos a usar una simple aplicacion [Flask Hello World](`http://flask.pocoo.org/`)

Para instalar el modulo Flask simplemente corremos este comando

```
$ pip install Flask

#!/usr/bin/env python

from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!"

if __name__ == "__main__":
  app.run(host='0.0.0.0', port=5000)
```

Para tener esta aplicación corriendo en un contenedor Docker, necesitamos escribir un 'Dockerfile' que instale las dependencias de este framework (comando 'RUN'), y poder correr nuesta app. También necesitamos exponer el puerto del contenedor (comando 'EXPOSE'). También necesitamos mover nuestra aplicación al Filesystem del contenedor (comando 'ADD'). El Dockerfile quedaría de la siguiente forma:

```
  FROM ubuntu:14.04

  # Actualizamos repositorios e instalamos dependencias.
  RUN apt-get update
```

```
RUN apt-get install -y python python-pip
RUN apt clean all
RUN pip install flask

# Agregamos nuestra aplicación al Filesystem del contenedor.
ADD hello.py /tmp/hello.py

# Exponemos el puerto del contenedor
EXPOSE 5000

# Comando por default que se ejecuta cuando se corre el contenedor
CMD ["python","/tmp/hello.py"]
```

Nota: Este Dockerfile no está optimizado, intencionalmente. Para optimizarlo lo veremos más adelante, pero esto sólo es para entender lo básico.

El comando 'RUN' permite ejecutar comandos específicos durante el build de la imagen del contenedor. Para copiar nuestra aplicación dentro de la imagen del contenedor, usamos el comando 'ADD'. En nuestro caso, copia el archivo 'hello.py' al directorio '/tmp' de la imagen del contenedor. La aplicación usa el puerto '5000', y tenemos que exponer este puerto al Docker Host. Finalmente, el comando 'CMD' especifica que el contenedor debe ejecutar 'python /tmp/hello.py' cuando se ejecute.

Procedemos a hacer build de la imagen.

```
$ docker build -t flask .
```

Esto creó una imagen Docker flask:

```
$ docker images
REPOSITORY    TAG       IMAGE ID        CREATED         VIRTUAL SIZE
flask         latest    d381310506ed    3 seconds ago   354.6 MB
...
```

Para correr esta aplicación usaremos la opción '-d', la cual daemonizará el contenedor. Tambien pasaremos el argumento '-P' para decirle a Docker que elija un puerto en el Docker Host para forwardear al puerto expuesto por el contenedor.

```
$ docker run -d -P flask
5ac72ed12a72f0e2bec0001b3e78f11660905d20f40e670d42aee292263cb890
```

```
$ docker ps
CONTAINER ID    IMAGE           COMMAND                  ...   PORTS
5ac72ed12a72    flask:latest    "python /tmp/hello.py    ...   0.0.0.0:49153->5000/t
```

El contenedor retornado, está daemonizado y no con nosotros logueados en una shell interativa dentro. La sección PORTS nos muestra el mapeo de puertos del contenedor en cuestión. En este caso mapea el puerto 49153 del **Docker Host** al puerto 5000 del **contenedor**. Si ahora ingresamos en http://localhost:49153, deberíamos ver el mensaje 'hello world¡.

Nota: Notar que no se le pasó un comando a ejecutar en el comando 'run', esto se debe a que ejecutará el 'CMD' definido en el Dockerfile. También podriamos sobreescribir el comando, por ejemplo:

```
$ docker run -t -i -P flask /bin/bash
root@fc1514ced93e:/# ls -l /tmp
total 4
-rw-r--r-- 1 root root 194 Dec 8 13:41 hello.py
root@fc1514ced93e:/#
```

6.21 Optimizando el Dockerfile siguiendo buenas prácticas

Problema

Se quiere seguir las buenas prácticas para crear Dockerfiles y optimizar las imágenes Docker.

Solución

Docker expone en su documentación [una sección de buenas prácticas](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) para ecribir Dockerfiles. Estas prácticas nos ayudarán a crear imágenes de forma más eficiente, modulares y con menor esfuerzo.

Estas son algunas instrucciones para crear buenas 'Docker Images'.

1. Ejecutar un único proceso por contenedor. De todas formas podríamos correr multiples procesos por contenedor, como se vió cuando usamos 'supervisor'. En este caso, 'supervisor' es el único proceso de cara al contenedor, pero éste levanta internamente otros procesos. Seguir la práctica de un único proceso por contenedor, nos permite hacer aplicaciones desacopladas que podrían escalar. Esto nos permite además usar container links u otras técnicas de container networking que veremos más adelante.

2. No asumir que nuestros contenedores estarán siempre corriendo; Estos son efímeros y serán parados y reiniciados. Se debería tratarlos como entidades inmutables, lo que significa que no deberíamos modificarlos mientras están en ejecución, sino modificar el Dockerfile reconstruir la imagen y levantar un contenedor con esa imagen actualizada. Por lo tanto, se recomienda manejar datos y configuraciones de ejecución fuera del contenedor y por lo tanto de su imagen. Para esto, usamos 'Docker Volumes'.

3. Usar un archivo '.dockerignore'. Cuando creamos imágenes, Docker copiará el contenido del working directory donde se encuentra el Dockerfile, dentro de la imagen. Con los archivos '.dockerignore' obtenemos un funcionamiento como el '.gitignore' y básicamente lo que logramos es excluir archivos (basura o sensibles) que no queremos que estén dentro de la imagen. El uso del '.dockerignore' es opcional, pero si no lo usamos, aseguremonos de copiar lo mínimo y necesario. Podemos chequear la sintaxis del mismo en este [link](`https://docs.docker.com/engine/reference/builder/#dockerignore-file`).

4. Usar imágenes oficiales del Docker Hub, en lugar de escribir las nuestras desde cero. Estas imágenes están mantenidas por quienes son las empresas autoras de ese software. También podemos usar 'ONBUILD images', para simplicar el proceso de creación de nuestras imágenes.

5. Finalmente, y de los más importantes, minimizar el número de capas de nuestras imágenes usando la caché de imagen. Docker usa [union filesystems](`https://es.wikipedia.org/wiki/UnionFS`) para almacenar las imágenes. Esto quiere decir que cada imagen se hace a partir de una imagen base más una colección de diffs que agregan los cambios requeridos. Cada diff representa una capa adicional en una imagen. Esto tiene un impacto directo en como nosotros escribimos nuestro Dockerfile y las directivas que usamos. En la sección siguiente veremos este punto.

Con estos puntos, haremos unos pequeños cambios en la imagen creada en la sección anterior:

Tenemos el Dockerfle de esta forma:

```
FROM ubuntu:14.04

# Actualizamos repositorios e instalamos dependencias.
```

```
RUN apt-get update
RUN apt-get install -y python python-pip
RUN apt clean all
RUN pip install flask

# Agregamos nuestra aplicación al Filesystem del contenedor.
ADD hello.py /tmp/hello.py

# Exponemos el puerto del contenedor
EXPOSE 5000

# Comando por default que se ejecuta cuando se corre el contenedor
CMD ["python","/tmp/hello.py"]
```

Aplicamos unos cambios:

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y \
  python
  python-pip

RUN pip install flask

COPY hello.py /tmp/hello.py

EXPOSE 5000

CMD ["python","/tmp/hello.py"]
```

Usar múltiples comandos 'RUN' es una mala práctica, ya que genera una nueva capa por cada uno. También cambiamos el comando 'ADD' por 'COPY' ya que 'ADD' es para operaciones de copiado más complejas, y nosotros sólo copiamos de manera simple.

Aun así podríamos aplicar más optimizaciones como la siguiente:

```
FROM python:2.7.10

RUN pip install flask
```

```
COPY hello.py /tmp/hello.py

EXPOSE 5000

CMD ["python","/tmp/hello.py"]
```

Entre los cambios, se puede ver que cambiamos a ‘ubuntu‘ por ‘python‘ como imagen base (aplicando el punto ‘2.‘ de optimizaciones). Eliminando toda la instalación de dependencias para python. Estas optimizaciones aún podrían ser más optimizables como, por ejemplo, usar la imagen base de ‘Flask‘, pero la idea es que se note la diferencia entre un ‘Dockerfile‘ y otro optimizado.

6.22 Configuración avanzada de red

Con todo lo descrito anteriormente ya se puede empezar a trabajar con Docker y tener una visión de como funciona. A partir de aquí es interesante conocer como funciona la configuración avanzada de red en la cual podemos, entre otras cosas, especificar un bridge de conexión para el contenedor, activar la comunicación entre contenedores, configuración de IPTABLES, DNS, IP, etc. Estos son algunos de los parámetros disponibles a la hora de arrancar el contenedor que establecen estas configuraciones:

Configuración de bridges:

```
--bridge=BRIDGE
```

Activar la comunicación entre contenedores:

```
--icc=true|false
```

Especificar la IP a la que responderá el contenedor:

```
--ip=IP_ADDRESS
```

Habilitar IP Forwarding:

```
--ip-forward=true|false
```

Habilitar iptables:

```
--iptables=true|false
```

Especificar DNS y dominio de búsqueda DNS:

```
--dns=IP_ADDRESS
--dns-search=DOMAIN
```