

# Práctica Docker

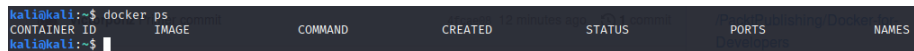
## Primer contenedor en Docker

NOTA. ESTA PRÁCTICA LA HAREMOS EN CLASE

Vamos a usar un repositorio de GitHub que ya tiene todos los componentes contenedorizados. El repositorio se encuentra en <https://github.com/victorponz/docker/tree/master/P1>.

Es una pequeña aplicación Apache+PHP. Hay una serie de archivos `sh` para ahorrarnos el trabajo de escribir comandos pero estaría bien que echaras un vistazo a los mismos.

Antes de empezar a trabajar, comprobaremos si Docker está instalado correctamente, mediante el comando `docker ps` que muestra esta salida:



```
kali@kali:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
kali@kali:~$
```

Figure 1: image-20210109112248195

La salida muestra que no hay corriendo ningún contenedor Docker

Si el comando no existe hay que instalar siguiendo el proceso de instalación que para Ubuntu se encuentran en <https://docs.docker.com/engine/install/ubuntu/>

Los scripts `sh` son los siguientes:

- `./build.sh`: Crea el container a partir de la definición descrita en Dockerfile. Se debe lanzar cada vez que este fichero cambie.
- `./debug.sh`: Corre el container en primer plano de tal forma que se puede parar con `^C`
- `./run.sh`: Corre el contenedor como un demonio (daemon). Se usa para probar el contenedor en loca pero como si fuera producción.
- `./stop.sh`: Cuando el contenedor corre en background, este es el script para pararlo.
- `./shell.sh`: A veces las cosas no funcionan según lo esperado y mediante este script se abre un shell en el contenedor una vez arrancado, de tal forma que podamos hacer diagnósticos.

Contenido de Dockerfile:

```
# we will inherit from the Debian image on DockerHub
FROM debian

# set timezone so files' timestamps are correct
ENV TZ=Europe/Madrid

# install apache and php 7.3
# we include procs and telnet so you can use these with shell.sh prompt
RUN apt-get update -qq >/dev/null && apt-get install -y -qq procs telnet apache2 php7.3 -q

# HTML server directory
WORKDIR /var/www/html
COPY . /var/www/html/

# The PHP app is going to save its state in /data so we make a /data inside the container
RUN mkdir /data && chown -R www-data /data && chmod 755 /data & chmod 755 -R /var/www/html/

# we need custom php configuration file to enable userdirs
COPY php.conf /etc/apache2/mods-available/php7.3.conf

# enable userdir and php
RUN a2enmod php7.3

# we run a script to start the server; the array syntax makes it so ^C will work as we want
CMD ["/entrypoint.sh"]
```

Vamos a ver qué hace este fichero Dockerfile, paso a paso:

1. El Dockerfile hereda de la imagen de Debian en Docker Hub.
2. Configuramos la zona horaria del contenedor para que coincida con la zona horaria del host; en otras palabras, asegúrese de que las marcas de tiempo de los archivos dentro del contenedor y en el host coincidan. Esto es importante al mapear directorios de host al sistema de archivos del contenedor. Para saber la zona horaria escribe en una shell:

```
Timedatectl | grep "Time zone"
Time zone: Europe/Madrid (CET, +0100)
```

3. Luego instalamos Apache y PHP 7.3. Estos se instalan en el sistema de archivos del contenedor y no en el sistema de archivos del host. Hemos evitado el problema de la contaminación de tener una versión de ambos instalados en el host que luego quedan sin usar cuando no se trabaja en este proyecto.

4. También instalamos algunas utilidades de línea de comandos que nos permiten examinar el estado del contenedor construido desde un shell Bash que se ejecuta dentro del contenedor.
5. De forma predeterminada, el usuario y el grupo que ejecutarán el proyecto en el contenedor es root. Para proporcionar una seguridad típica de Unix / Linux, queremos ejecutar como un usuario real; en nuestro caso, el nombre de usuario es app. Entonces agregamos al usuario al entorno del contenedor con useradd.
6. Vamos a poner nuestros scripts PHP en `/var/www/html`.
7. Nuestra aplicación de demostración escribe su estado en `/data`, por lo que debemos crearla y asegurarnos de que el script PHP que se ejecuta como aplicación de usuario pueda leer y escribir archivos allí.
8. Creamos un archivo de configuración PHP personalizado que queremos usar dentro del contenedor, así que lo copiamos al contenedor en la ubicación correcta en el sistema de archivos.
9. Necesitamos habilitar el módulo `php7.3`. Esto nos permite ejecutar scripts PHP desde Apache.
10. Cuando se inicia el contenedor, necesita ejecutar algún programa o script dentro del contenedor. Usamos un script `sh` llamado `entrypoint.sh` en el directorio `/var/www/html` para iniciar la aplicación. Podemos editar este archivo para satisfacer nuestras necesidades durante el desarrollo.

Una vez entendido `Dockerfile`, necesitamos `build.sh` para construir el contenedor.

```
#!/bin/sh
```

```
# build.sh
```

```
# we use the "docker build" command to build a container named "chapter2" from . (current d-  
# Dockerfile is found in the current directory, and determines how the container is built.
```

```
docker build -t chapter2 .
```

La salida del comando es la siguiente y puedes comprobar que cada paso se imprime en la salida. Por ejemplo:

Step 1/11 : **FROM** debian

El contenedor se crea de forma incremental, como se describe en `Dockerfile`. Cada paso se construye en una capa de imagen indicada con un valor `hash`; esos que se muestran son los valores hash hexadecimales impresos. Cuando se vuelva a construir el contenedor, Docker puede comenzar desde el estado de cualquiera de los valores hash de esas capas, lo que reduce la necesidad de reconstruir constantemente el contenedor desde cero. Cada capa es simplemente una diferencia (`diff`) entre los requisitos de la capa actual y el estado de la capa anterior.

Una vez construido, lo más habitual es lanzar el script `debug.sh` en desarrollo

```
#!/usr/bin/env bash

# debug.sh

# run container without making it a daemon - useful to see logging output

docker run \
  --rm \
  -p8086:80 \
  --name="chapter2" \
  -v `pwd`: /home/app \
  chapter2
```

El comando `docker run` toma muchos argumentos opcionales que son demasiado numerosos para detallarlos aquí. Para obtener información más completa sobre todos los posibles argumentos de la línea de comandos para ejecutar Docker, consulta la documentación de `Docker Run` en el sitio de Docker: <https://docs.docker.com/engine/reference/run/>. Solo cubriremos los que se usan en nuestros scripts:

- Aquí, usamos `-rm`, que le dice a Docker que realice una limpieza cuando el contenedor sale, eliminando el contenedor y el sistema de archivos del contenedor.
- La bandera `-p` le dice a Docker que asigne el puerto 80 desde el contenedor (HTTP) al puerto 8086 en el host; se puede acceder al servidor HTTP en el contenedor utilizando el puerto 8086 en el host.
- El argumento `-name` nombra el contenedor en ejecución; si no proporcionas un nombre, tendrás que usar `docker ps` para obtener el hash que identifica el contenedor para detenerlo usando `docker stop`.
- El modificador `-v` monta volúmenes en el contenedor. Un volumen puede ser un directorio de un archivo en el host, un volumen con nombre que Docker administra por ti. Si deseas de-

tener y reiniciar el contenedor y retener los datos que el contenedor escribe en el sistema de archivos, debes montar un volumen y el contenedor debe escribir en este volumen. Puedes montar varios volúmenes, si lo deseas. En nuestro script `debug.sh`, montamos el directorio actual con las fuentes sobre `/home/app`, por lo que podemos modificar las fuentes y los programas contenedores ven que los archivos se cambian (porque las marcas de tiempo de los archivos son más recientes) como si estuvieran dentro el contenedor también. Para esta demostración, puedes editar el script `index.php` y volver a cargar la página, y verás el cambio en acción.

- El último argumento para `docker run` es el nombre del contenedor para empezar; en nuestro caso, es el `chapter2`, la imagen del contenedor que creamos usando el script `build.sh`.

**Nota.** No persistimos `/data` en el contenedor. Podemos hacer esto agregando el modificador `-v` para mapear un volumen Docker a `/data`, lo que haremos en el script `persist.sh`.

## Ejecutar el contenedor

Hemos lanzado `build.sh` y todo fue bien. Ahora usaremos `debug.sh` para lanzar el contenedor en modo `debug/foreground`

Una vez visitemos la página varias veces veremos que el contador de visitas aumenta:

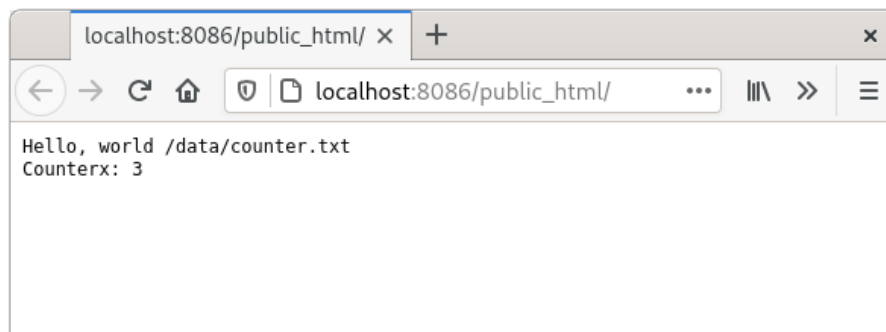


Figure 2: image-20210110210411340

Podemos comprobar que el contenedor está corriendo ejecutando `docker ps`

Para reiniciar el contenedor podemos lanzar `docker restart chapter2` y veremos que el contador continúa donde se quedó.

```
[victorponz@localhost ~]$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
98eb590bf396	chapter2	"./entrypoint.sh"	9 minutes ago	Up 9 minutes	0.0.0.0:8086->80/tcp	chapter2

Figure 3: docker ps

Sin embargo, si lo paramos con `docker stop chapter2` o `stop.sh` y luego lo volvemos a lanzar con `run.sh`, por ejemplo, veremos que el contador empieza en 1. Esto es así porque estamos escribiendo en el sistema de archivos del contenedor, así que desaparece cuando el contenedor se para.

Para que persista entre reinicios en `data/container.txt` debemos:

- Crear un fichero llamado `container.txt` en el anfitrión y que sea `/data/container.txt` en el huésped
- Montar un directorio en el anfitrión como `data` en el huésped
- Hacer que Docker cree y mantenga un volumen por nosotros.

Para ello creamos un volumen con nombre usando la opción `-v` en `docker run` con justamente el nombre del directorio el en huésped; por ejemplo, `-v name:/data`. El script `persist.sh` hace justamente eso. Es como `debug.sh` pero con la opción `-v`

```
#!/usr/bin/env bash

# persist.sh

# run container without making it a daemon - useful to see logging output
# we are adding a named volume for /data in the container so the
# counter persists between runs.

docker run \
  --rm \
  -p8086:80 \
  --name="chapter2" \
  -v `pwd`:~/home/app \
  -v name:/data \
  chapter2
```

Cuando lo ejecutamos y apuntamos nuestro navegador a `http://localhost:8086/public_html/`, vemos que el contador funciona, incluso si paramos y reiniciamos el contenedor.

El script `run.sh` ejecuta el contenedor en modo demonio; no podrás ver la salida de la aplicación sin usar el comando de registro de docker. Tampoco monta el directorio de host como un volumen en el contenedor. Esto simula el entorno de producción:

```
#!/usr/bin/env bash

# run.sh

# run the container in the background
# /data is persisted using a named container

docker run \
    --detach \
    --rm \
    -p8086:80 \
    -v name:/data \
    --name="chapter2" \
    chapter2
```

- La bandera `-detach` le dice a `docker run` que el contenedor se ejecute en segundo plano.
- Se usa el volumen nombrado, por lo que los datos se conservan entre el inicio y la detención del contenedor. Estos datos se almacenan dentro de `/var/lib/docker/volumes/`
- El directorio de trabajo de desarrollo está montado en `/home/app` dentro del contenedor.
- La opción `-restart` le dice a Docker que reinicie el contenedor cuando se reinicia el sistema.

El script `shell.sh` ejecuta el contenedor e inicia el shell Bash para que pueda usar programas de línea de comandos para diagnosticar problemas con el contenedor.

**!Ya hemos creado nuestro primer contenedor!**