



KALEIDO | COWORKING
CENTER

opsou

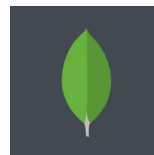
pedrofigueras



Altia **Senior Developer**
Disfrutando del desarrollo web
desde 1998.

 @rolando_caldas

<https://rolandocaldas.com>



Meetups

 PHP VIGO

Instalar Docker

- Ubuntu: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- Windows: <https://store.docker.com/editions/community/docker-ce-desktop-windows>
- Mac: <https://store.docker.com/editions/community/docker-ce-desktop-mac>
- Post-Install:
 - Ubuntu: Lanzar docker sin ser root + Activar acceso remoto (IDE)
 - Windows: Activar acceso remoto + Permitir conexiones no seguras
 - Mac: Activar acceso remoto

Ejecutar docker sin ser root

- Es necesario crear el grupo docker y asociarlo al usuario con el que se va a utilizar docker:

```
$ sudo groupadd docker  
$ sudo usermod -aG docker $USER
```

- Logout/Login
- GO GO GO

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.
- Como en programación, un namespace es un espacio aislado.

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.
- Como en programación, un namespace es un espacio aislado.
- Cgroups: Es una funcionalidad de linux para limitar los recursos que puede usar una aplicación.

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.
- Como en programación, un namespace es un espacio aislado.
- Cgroups: Es una funcionalidad de linux para limitar los recursos que puede usar una aplicación.
- Cuando lanzas un contenedor de Docker, éste crea un namespace en linux y crea una capa de aislamiento para el contenedor.

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.
- Como en programación, un namespace es un espacio aislado.
- Cgroups: Es una funcionalidad de linux para limitar los recursos que puede usar una aplicación.
- Cuando lanzas un contenedor de Docker, éste crea un namespace en linux y crea una capa de aislamiento para el contenedor.
- Todo lo que ocurra en el contenedor está limitado a su namespace.

Docker no es un entorno virtual

- Docker utiliza el Namespace y el Cgroups de Linux.
- Como en programación, un namespace es un espacio aislado.
- Cgroups: Es una funcionalidad de linux para limitar los recursos que puede usar una aplicación.
- Cuando lanzas un contenedor de Docker, éste crea un namespace en linux y crea una capa de aislamiento para el contenedor.
- Todo lo que ocurra en el contenedor está limitado a su namespace.
- Docker utiliza los Cgroups para limitar los recursos que pueden consumir los contenedores.

Docker no es un entorno virtual

En resumen: Lo que se ejecuta en el contenedor ...

Docker no es un entorno virtual

En resumen: Lo que se ejecuta en el contenedor ...
... se ejecuta en nuestro sistema, no en una máquina virtual.

Comandos básicos

- `$ docker pull` Descarga una imagen de la red
- `$ docker build` Crea una imagen
- `$ docker run` Lanza un contenedor
- `$ docker stop` Para un contenedor
- `$ docker rm` Elimina un contenedor
- `$ docker rmi` Elimina una imagen
- `$ docker ps` Contenedores activos
- `$ docker ps -a` Contenedores activos e inactivos
- `$ docker images` Imágenes descargadas

Corriendo nuestro primer contenedor

```
$ cd $HOME && mkdir dockerWorkshop && cd dockerWorkshop  
$ docker run hello-world
```

- Docker intenta ejecutar la imagen hello-world
- Como no existe, porque no la hemos descargado (sería haciendo el pull) va a buscarla a l red
- La busca, por defecto, en el docker hub. Como la encuentra, la descarga (hace el pull por nosotros) y tras ello, hace el run.
- Si lanzamos `docker ps` no sale nada ¿por que? Porque el contenedor se lanzó, se ejecutó y, al no tener ningún proceso o servicio que lo mantenga abierto, se cerró.
- Si lanzamos `docker ps -a` veremos nuestro contenedor.
- Si lanzamos `docker image` veremos la imagen descargada y su huella.

Corriendo nuestro primer contenedor

- `$ docker run ubuntu`
- `$ docker ps`
- `$ docker ps -a`
- `$ docker images`
- `$ docker run -it ubuntu`
 - `-i`: Mantiene abierta la entrada estándar (STDIN) para poder interactuar desde fuera.
 - `-t`: Abre una terminal (TTY)
 - ¿por qué una consola bash? => Porque es el CMD de la imagen ubuntu
- `$ docker run -it node`
 - Consola de node porque el CMD de la imagen es NODE, no BASH
- `$ docker ps`
 - Ya tenemos cosas!!!

Recordatorio: Docker no es una VM

Con nuestro contenedor de ubuntu corriendo, vamos a su shell y escribimos:

```
root@f709b1535b47:/# top
```

Vamos a un terminal de nuestro linux (no del contenedor):

```
$ ps afux
```


Recordatorio: Docker no es una VM

Con nuestro contenedor de ubuntu corriendo, vamos a su shell y escribimos:

```
root@f709b1535b47:/# top
```

Vamos a un terminal de nuestro linux (no del contenedor):

```
$ ps afux
```

```
root      3322 /usr/bin/dockerd -H fd://
root      3485 \_ docker-containerd --config /var/run/docker/containerd/containerd.toml
root      14708 \_ docker-containerd-shim -namespace moby -workdir /var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/f709b1535b477
root      14726 \_ bash
root      16877 \_ top
root      3333 /bin/sh -c snap /nextcloud/8971/bin/renew-ccsts
```

Dockerizando PHP

En el hub de docker existen múltiples imágenes oficiales, PHP es una de ellas:

- Entramos en <https://hub.docker.com/>
 - Buscamos PHP: https://hub.docker.com/_/php/
 - Tenemos múltiples imágenes de PHP disponibles, tanto en base a la versión como al SO base, PHP + Apache, PHP-CLI, PHP-FPM, ZTS...
 - Vamos a utilizar la versión 7.2-fpm
- ```
$ docker run --name=php-fpm php:7.2-fpm
```
- con `--name=php-fpm` forzamos que nuestro contenedor se llame php-fpm... esto será muy útil cuando necesitemos interactuar con el contenedor.

# PHP Dockerizado running!

Podemos ejecutar comandos en nuestros contenedores desde fuera con:

```
docker exec [container] [command]
```

Para ver qué versión de PHP está en nuestro container:

```
$ docker exec php-fpm php -v
```

# PHP Dockerizado running!

Podemos ejecutar comandos en nuestros contenedores desde fuera con:

```
docker exec [container] [command]
```

Para ver qué versión de PHP está en nuestro container:

```
$ docker exec php-fpm php -v
```

```
rolando@rolando-Lenovo-Z50-70:~$ docker exec php-fpm php -v
PHP 7.2.10 (cli) (built: Sep 15 2018 02:33:49) (NTS)
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
```

PHP 7.2.10 CLI **funcionando.**

# Versiones dockerizadas de PHP

Los nombres de las imágenes siguen el patrón: versionPHP-tipo-SO

- versionPHP:
  - 5.6.38 (5.6 y 5 como alias)
  - 7.0.32 (7.0 como alias)
  - 7.1.22 (7.1 como alias)
  - 7.2.10 (7.2 como alias)
  - 7.3.0RC2 (7.3-rc y rc como alias)
- tipo:
  - cli: la imagen sólo contiene la versión de PHP para línea de comandos
  - fpm: la imagen contiene tanto php-cli como el gestor de procesos de PHP (FastCGI Process Manager)
  - zts: PHP CLI con ZTS habilitado, requerido para utilizar pthreads.
  - apache: Incluye el servidor web Apache, PHP y el mod\_php.

# Versiones dockerizadas de PHP

- S0:
  - stretch: Debian stretch
  - jessie: Debian jessie
  - alpine3.X: Minimal distro Alpine

# Docker PHP + Apache

Aunque pueda ser útil normalmente de poco o nada sirve tener un contenedor con PHP... necesitamos al menos un servidor web para acceder por HTTP.

De forma oficial, tenemos la imagen con Apache2:

```
$ docker run -p 8080:80 -it --name=php-apache php:7.2-apache
```

# Docker PHP + Apache

Docker ejecuta los contenedores de forma aislada, por lo que el Apache del contenedor no estará accesible desde el exterior. Necesitamos solicitar a Docker que puentee un puerto de nuestra máquina con el contenedor:

- `-p [puerto-local]:[puerto-contenedor]`

Si accedemos con nuestro navegador a `http://localhost:8080` veremos un error de Apache, porque aunque el servidor web funciona y la conexión entre puertos también, el contenedor no tiene contenido a mostrar.




# Docker PHP + Apache

- `$ docker exec -it php-apache bash`
- `root@be8415f43808:/var/www/html# cat > index.php`
- escribimos el código para el phpinfo: `<?php phpinfo();`
- salimos con control+C
- Ahora si accedemos a <http://localhost:8080> ...

# Docker PHP + Apache

- `$ docker exec -it php-apache bash`
- `root@be8415f43808:/var/www/html# cat > index.php`
- escribimos el código para el phpinfo: `<?php phpinfo();`
- salimos con control+C
- Ahora si accedemos a <http://localhost:8080> ...

|                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                     |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| PHP Version 7.2.10        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |  |
| System                    | Linux be8415f43808 4.15.0-36-generic #39-Ubuntu SMP Mon Sep 24 16:19:09 UTC 2018 x86_64                                                                                                                                                                                                                                                                                                                                                                              |                                                                                     |
| Build Date                | Sep 15 2018 02:27:13                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                     |
| Configure Command         | './configure' '--build=x86_64-linux-gnu' '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--enable-option-checking=fatal' '--with-mhash' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with-password-argon2' '--with-sodium=shared' '--with-curl' '--with-libedit' '--with-openssl' '--with-zlib' '--with-libdir=lib/x86_64-linux-gnu' '--with-apxs2' '--disable-cgi' 'build_alias=x86_64-linux-gnu' |                                                                                     |
| Server API                | Apache 2.0 Handler                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                     |
| Virtual Directory Support | disabled                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                     |

## Levantar contenedores existentes

Cuando paramos un contenedor ya sea con docker stop o con “CTRL+C” el contenedor no se elimina sino que se para. Si hacemos un run:

```
$ docker run -it --name=php-fpm php:7.2-fpm
```

Docker creará el contenedor y lo levantará. Como le hemos especificado el nombre del contenedor y éste ya existe tenemos un error.

En estos casos lo que necesitamos es levantar el contenedor, no crear uno nuevo:

```
$ docker start php-fpm
```

# Entender un poco mejor docker ps

```
$ docker ps
```

En estos momentos hay dos contenedores levantados, por lo que vemos su información:

- CONTAINER ID: El identificador del contenedor
- IMAGE: La imagen del contenedor
- PORTS: Los puertos expuestos del contenedor y su vínculo con un puerto local, de existir.
  - 0.0.0.0:8080->80/tcp
  - 9000/tcp El contenedor tiene expuesto el puerto 9000, pero sólo está disponible en su red, al no estar vinculado con un puerto local.
- NAMES: El nombre del contenedor

# Añadiendo extensiones PHP y utilidades

- Las imágenes de PHP, como todas las imágenes oficiales en Docker Hub, van con lo mínimo.
- Programas habituales como vi, vim, nano, ping, etc... NO están en nuestro contenedor.
- Podríamos entrar al contenedor por bash y tirar de apt-get etc etc, pero NO debemos hacer eso.
- La idea de un contenedor es que cuando se ejecuta con el run, realice todas las acciones necesarias y sólo las justas minimizando el espacio que necesita.
- Para realizar todas estas acciones tenemos los Dockerfile.

# Añadiendo extensiones PHP y utilidades

Para un entorno base, válido para desarrollo y producción deberíamos contar con:

- **php-intl**: Funciones de internacionalización.
- **php-gd**: Graphics draw. Funciones de tratamiento de imágenes
- **php-mbstring**: Cadenas de caracteres multibyte
- **composer**: Gestor de paquetes... el APT de PHP.

Para un entorno de desarrollo:

- **xDebug**: Debugger y Profiler para PHP
- **phpUnit**: Test unitarios en PHP + Cobertura de código (xDebug requerido)
- **BlackFire**: Análisis de rendimiento (recomendable sin xDebug)

# Añadiendo extensiones PHP y utilidades

Gracias a utilizar imágenes base de Docker oficiales, tendremos extras interesantes. En el caso de PHP tenemos:

- **docker-php-ext-configure**: Permite configurar fácilmente extensiones de PHP
- **docker-php-ext-install**: Permite instalar fácilmente extensiones de PHP

# Personalizar la imagen de PHP: Dockerfile

El Dockerfile es un fichero utilizado por Docker para crear imágenes, a partir de “nada” o de otras imágenes existentes:

- Se basa en una ejecución secuencial de comandos.
- Cada comando ejecutado sería como realizar un commit en un repositorio.
- Cada “commit” se guarda en caché, para no ejecutarlo nuevamente al reconstruir la imagen.
- Siempre usará la caché, hasta que exista un cambio en un comando, en este caso, usa la caché hasta ese comando y, a partir de ahí, no.



# Personalizar la imagen de PHP: Dockerfile

Creamos un directorio php-base:

```
$ mkdir php-base && cd php-base
```

Creamos un fichero Dockerfile:

```
$ touch Dockerfile
```

Lo abrimos con nuestro editor favorito:

```
$ vim Dockerfile
```

# Personalizar la imagen de PHP: Dockerfile

```
FROM php:7.2-fpm
```

```
WORKDIR "/application"
```

```
RUN apt-get update \
 && apt-get install -y libicu-dev libpng-dev libjpeg-dev libpq-dev \
 && apt-get clean; rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
/usr/share/doc/* \
 && docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr \
 && docker-php-ext-install intl gd mbstring
```

```
RUN curl -sS https://getcomposer.org/installer | php --
--install-dir=/usr/local/bin --filename=composer
```

# Personalizar la imagen de PHP: Dockerfile

Definimos cuál será la imagen base de nuestro contenedor:

```
FROM php:7.2-fpm
```

Establecemos el directorio de trabajo:

```
WORKDIR "/application"
```

Una vez levantado el contenedor, si nos conectamos por bash accederemos al directorio /application en lugar de a /

# Personalizar la imagen de PHP: Dockerfile

```
RUN apt-get update \
 && apt-get install -y libicu-dev libpng-dev libjpeg-dev libpq-dev \
 && apt-get clean; rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
/usr/share/doc/* \
 && docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr \
 && docker-php-ext-install intl gd mbstring
```

- **apt-get update:** Para actualizar las fuentes de APT
- **apt-get install -y:** Para instalar sin necesidad de confirmar por consola la acción
- **apt-get clean && rm:** Se elimina la caché y posibles ficheros temporales de la instalación. Recordemos la importancia de mantener nuestra imagen lo más liviana posible.
- **docker-php-ext-configure:** Agrega los parámetros necesarios a la configuración de PHP
- **docker-php-ext-install:** Se instalan las extensiones básicas de PHP

# Personalizar la imagen de PHP: Dockerfile

```
RUN curl -sS https://getcomposer.org/installer | php --
--install-dir=/usr/local/bin --filename=composer
```

Podríamos utilizar wget, pero tendríamos que instalarlo por APT y desinstalarlo posteriormente, para mantener la idea de “minimal installation”

Con este comando, lo que hacemos es instalar composer en /usr/local/bin bajo el nombre composer.

# Personalizar la imagen de PHP: Dockerfile

- Guardamos el fichero Dockerfile y volvemos a la consola, al directorio que contiene el Dockerfile.
- Hacemos el build y el run de nuestra imagen de docker customizada:

```
$ docker build -t "php-fpm-base" .
```

  - **-t** : El nombre que le damos a nuestra imagen
  - **.** : El directorio dónde está el Dockerfile a utilizar a la hora de construir la imagen.
- Ya tenemos la imagen creada. Ahora si volvemos a lanzar el comando, veremos que no tarda nada en crear la imagen, ésto es porque está utilizando la caché.

# Personalizar la imagen de PHP: Dockerfile

Con la imagen construida, la ejecutamos:

```
$ docker run --name="php-fpm-base" php-fpm-base
```

Ejecutamos un comando de PHP en el contenedor para ver el phpinfo:

```
$ docker exec php-fpm-base php -i
```

Y hacemos lo mismo para ver que tenemos composer disponible:

```
$ docker exec php-fpm-base composer -V
```

# Personalizar la imagen de PHP: Dockerfile

Creamos un directorio php-xdebug:

```
$ mkdir php-xdebug && cd php-xdebug
```

Creamos un fichero Dockerfile:

```
$ touch Dockerfile
```

Lo abrimos con nuestro editor favorito:

```
$ vim Dockerfile
```



# Personalizar la imagen de PHP: Dockerfile

Definimos cuál será la imagen base de nuestro contenedor, que será la que acabamos de crear:

```
FROM php-fpm-base
```

Instalamos xdebug:

```
install xdebug
RUN pecl install xdebug \
 && docker-php-ext-enable xdebug
```

# Personalizar la imagen de PHP: Dockerfile

- Guardamos el fichero Dockerfile y volvemos a la consola, al directorio que contiene el Dockerfile.
- Hacemos el build y el run de nuestra imagen de docker customizada:

```
$ docker build -t "php-fpm-xdebug" .
$ docker run --name="php-fpm-xdebug" php-fpm-xdebug
```

- Comprobamos que el xDebug está ejecutándose:

```
$ docker exec php-fpm-xdebug php -v
```

# Personalizar la imagen de PHP: Dockerfile

Creamos un directorio php-dev:

```
$ cd .. && mkdir php-dev && cd php-dev
```

Creamos un fichero Dockerfile:

```
$ touch Dockerfile
```

Lo abrimos con nuestro editor favorito:

```
$ vim Dockerfile
```

# Personalizar la imagen de PHP: Dockerfile

Definimos cuál será la imagen base de nuestro contenedor, que será la que acabamos de crear:

```
FROM php-fpm-base
```

Instalamos xdebug + las dependencias para phpUnit

```
RUN pecl install xdebug && docker-php-ext-enable xdebug
```

```
RUN apt-get update \
 && apt-get -y install unzip zlib1g-dev \
 && apt-get clean; rm -rf /var/lib/apt/lists/* /tmp/*
/var/tmp/* /usr/share/doc/* \
 && docker-php-ext-install zip
```

# Personalizar la imagen de PHP: Dockerfile

- Guardamos el fichero Dockerfile y volvemos a la consola, al directorio que contiene el Dockerfile.
- Hacemos el build y el run de nuestra imagen de docker customizada:

```
$ docker build -t "php-fpm-dev" .
$ docker run --name="php-fpm-dev" php-fpm-dev
```

# Personalizar la imagen de PHP: Dockerfile

Creamos un directorio php-dev-mysql:

```
$ cd .. && mkdir php-dev-mysql && cd php-dev-mysql
```

Creamos un fichero Dockerfile:

```
$ touch Dockerfile
```

Lo abrimos con nuestro editor favorito:

```
$ vim Dockerfile
```

# Personalizar la imagen de PHP: Dockerfile

Definimos cuál será la imagen base de nuestro contenedor, que será la que acabamos de crear:

```
FROM php-fpm-dev
```

Instalamos la extensión PDO para MySQL de PHP y sus dependencias

```
Install mysql
RUN apt-get update && apt-get install -y mysql-client \
 && apt-get clean; rm -rf /var/lib/apt/lists/* /tmp/* \
/var/tmp/* /usr/share/doc/* \
 && docker-php-ext-install mysqli pdo pdo_mysql
```

# Personalizar la imagen de PHP: Dockerfile

- Guardamos el fichero Dockerfile y volvemos a la consola, al directorio que contiene el Dockerfile.
- Hacemos el build y el run de nuestra imagen de docker customizada:

```
$ docker build -t "php-fpm-dev-mysql" .
$ docker run --name="php-fpm-dev-mysql" php-fpm-dev-mysql
```



# Llevando nuestras imágenes a GitHub

- Ya tenemos nuestras imágenes de PHP creadas en local.
- Aunque podemos trabajar en local... no deja de ser algo limitado.
- Lo interesante de Docker es poder llevarnos nuestras imágenes a cualquier lado.

# Llevando nuestras imágenes a GitHub

So...

# Llevando nuestras imágenes a GitHub

- Como primer paso, crearemos un repositorio en GitHub.
- En nuestro repositorio guardaremos nuestra configuración.
- Creamos el repositorio “docker-php” desde la interfaz de Github
- Vamos a transformar nuestro directorio de trabajo en el repositorio de docker-php

```
$ cd ..
```

# Llevando nuestras imágenes a GitHub

```
$ git init
$ git config --global user.email "rolando.caldas@gmail.com"
$ git config --global user.name "Rolando Caldas"
$ git add .
$ git commit -m "My php docker containers configuration for development"
$ git remote add origin https://github.com/rolando-caldas/docker-php.git
$ git push -u origin master
```

# Llevando nuestras imágenes a GitHub

- Ya tenemos nuestras imágenes de PHP en GitHub
- A partir de ahora podremos hacer un git clone en cualquier equipo y hacer el docker build en él.
- También podremos hacer commits, push y pull... para mantener actualizados nuestros Dockerfiles

# Llevando nuestras imágenes a GitHub

...pero...

# Llevando nuestras imágenes a GitHub

- ¿por qué tener que estar haciendo constantemente los build?
- ¿no sería mejor tener ya las imágenes construidas en alguna parte?
- Let me introduce the docker hub

# Construyendo nuestras imágenes en Docker Hub

- Vinculamos Docker Hub con GitHub: Vamos a “Settings” => “Linked Accounts & Services” y clicamos en “Link Github”.
- Aunque estamos usando GitHub, podremos hacer lo mismo con Bitbucket.
- Cualquier repositorio de Docker Hub que creemos público podremos tenerlo sin pagar nada. Para tener builds privados, tendremos que actualizarnos a la versión de pago.
- Ya con la cuenta vinculada, podemos crear nuestro repositorio:
- “Create” => “Create Automated Build” => “Create Auto-Build Github”
- Filtramos por nuestro repo de github “Docker-php”



# Construyendo nuestras imágenes en Docker Hub

- Escribimos la descripción
- Agregamos nuestras diferentes versiones de las imágenes a construir:
  - Branch => master => /php-base => latest
  - Branch => master => /php-base => base
  - Branch => master => /php-xdebug => xdebug
  - Branch => master => /php-dev => dev
  - Branch => master => /php-dev-mysql => dev-mysql
- Creamos nuestro repository.
- Vamos a “Build Settings” y clicamos en “Trigger” para el Docker Tag base.
- Esperamos (mucho) a que se construya la imagen... mientras tanto actualizaremos nuestros Dockerfiles.

# Actualizado los Dockerfiles de nuestro repo

- Hasta ahora, los Dockerfiles de debug, dev y dev-mysql utilizaban imágenes de Docker creadas de forma local.
- Al tener ya nuestra php-base en el Docker Hub es el momento de hacer que la imagen base sea la del Docker Hub.
- Para los Dockerfiles de php-xdebug y php-dev cambiamos el FROM por:  
`FROM rolandocaldas/php:base`
- Para el Dockerfile de php-dev-mysql cambiamos el FROM por:  
`FROM rolandocaldas/php:dev`
- Haremos un commit con todos los cambios.
- Ejecutamos el push.
- Puede que algún build falle, en cuyo caso se lanzará manualmente

# Docker Hub: Bye bye builds locales

- Al tener nuestras imágenes de Docker subidas en el Docker Hub, podemos lanzar la que necesitamos como si fuese una de las imágenes oficiales:  

```
$ docker run rolandocaldas/php:dev-mysql
```
- Además, el contenedor más completo tiene un peso de 190 MB.

Ahora bien, tenemos un contenedor con PHP-FPM, pero no deja de ser algo “cojo”...  
... no tenemos servidor web ...  
... y tampoco tenemos un servidor MySQL ...

# Un contenedor por servicio

Una idea latente en Docker es utilizar un contenedor por servicio. Esto significa que para una aplicación web PHP necesitaríamos:

- Un contenedor con PHP-FPM: Hecho!
- Un contenedor con un servidor web... por ejemplo Nginx. Pendiente :[
- Otro contenedor con MySQL. Pendiente :[

Problema: Los contenedores por defecto no estarán en la misma red, por lo que entre ellos ni se ven, ni se escuchan... Nginx no podrá enviar la petición a PHP-FPM y éste no podrá conectarse al MySQL.

Solución: Crear en docker un network y correr todos los contenedores asociandolos al network creado.

# *Un contenedor por servicio*

La cosa empieza a complicarse... afortunadamente tenemos...

*Un contenedor por servicio*

docker-compose

# *docker-compose*

docker-compose es una capa sobre docker que nos permite montar entornos complejos a partir de un simple yaml.

Todo lo que hacemos a través del docker-compose se puede hacer directamente con el comando de docker, aunque con un nivel de complejidad muy superior a utilizar docker-compose.

Para ejecutar el entorno que configuramos vía YAML sólo hay que lanzar:

```
$ docker-compose up -d
```

- -d: Lanza todos los contenedores en segundo plano, sin bloquear nuestro terminal.

Con `docker-compose stop` detendremos todos los contenedores asociados.

## *docker-compose: Fichero .yaml*

- Creamos un fichero docker-compose.yml y lo abrimos con nuestro editor favorito:

```
$ cd .. && mkdir infrastructure && cd infrastructure/
$ touch docker-compose.yml && vim docker-compose.yml
version: "3.5"
services:
 php-fpm:
 image: rolandocaldas/php:dev-mysql
```

- services: Listado de los contenedores docker a levantar
  - php-fpm: Nombre del servicio
  - image: La imagen a utilizar.
- Guardamos el fichero y para lanzar todo:  

```
$ docker-compose up -d
```



## *docker-compose: Fichero .yml*

- Necesitamos agregar el servidor web al docker-compose. Usamos la imagen oficial de Nginx:

```
version: "3.5"
services:
 php-fpm:
 image: rolandocaldas/php:dev-mysql
 webserver:
 image: nginx:alpine
 ports:
 - "8080:80"
```

- ports: Vinculamos el puerto local 8080 con el 80 del docker webserver. Al entrar en <http://localhost:8080> cargará el nginx.

## *docker-compose: Fichero .yaml*

Actualmente tenemos dos contenedores de docker ejecutándose y uno de ellos con el puerto 80 asociado al 8080 local. Todo ello gracias a docker-compose.

Si accedemos a `http://localhost` tenemos: Welcome to nginx!

Para comprobar que los contenedores se escuchan, ejecutamos un ping en webserver a php-fpm:

```
$ docker-compose exec webserver ping php-fpm
```

Incorporamos a docker-compose.yml el servicio para MySQL

# *docker-compose: Fichero .yml*

```
version: "3.5"
services:
 php-fpm:
 image: rolandocaldas/php:dev-mysql
 webserver:
 image: nginx:alpine
 ports:
 - "8080:80"
 mysql:
 image: mysql:5.7
 environment:
 - MYSQL_ROOT_PASSWORD=root
 - MYSQL_DATABASE=database
 - MYSQL_USER=user
 - MYSQL_PASSWORD=pwd
```

## *docker-compose: Fichero .yml*

En nuestro servicio PHP-FPM tenemos el mysql-client instalado, por lo que vía docker-compose nos conectamos al MySQL del servicio mysql a través del servicio PHP-FPM:

```
$ docker-compose exec php-fpm mysql -h mysql -uuser -p
MySQL [(none)]> use database;
Database changed
MySQL [database]> quit
Bye
```

Tenemos los tres servicios funcionando, ahora sólo queda hacer que carguen una aplicación PHP concreta.

## *docker-compose: Fichero .yml*

Como broche de contenedores para nuestro entorno, vamos a agregar un contenedor que corre un phpMyAdmin y vamos a asociarlo con la base de datos del contenedor MySQL.

Para ello debemos añadir a nuestro docker-compose.yml:

```
phpmyadmin:
 image: phpmyadmin/phpmyadmin
 ports:
 - "8081:80"
 environment:
 - PMA_USER=user
 - PMA_PASSWORD=pwd
 - PMA_HOST=mysql
```

# *docker-compose: Fichero .yml*

Detenemos los contenedores y los lanzamos de nuevo:

```
$ docker-compose stop && docker-compose up -d
```

Si accedemos a <http://localhost:8081> entraremos al phpMyAdmin conectados con el usuario y al host indicados en el docker-compose.yml

## *docker-compose: Fichero .yaml - depends\_on*

Los contenedores que estamos lanzando, están relacionados entre sí:

- El phpMyAdmin necesita un MySQL al que conectarse.
- El webserver necesita el servicio php-fpm para servir las páginas en PHP
- Nuestro PHP se conectará a una base de datos, a nuestro contenedor MySQL.

Podríamos decir que:

- php-fpm depende de mysql
- webserver depende de php-fpm
- phpmyadmin depende de mysql

Podemos explicitarlo en nuestro docker-compose con depends\_on:

## *docker-compose: Fichero .yml - depends\_on*

```
version: "3.5"
services:
 php-fpm:
 image: rolandocaldas/php:dev-mysql
 depends_on:
 - mysql
 webserver:
 image: nginx:alpine
 ports:
 - "8080:80"
 depends_on:
 - php-fpm
```

[...]



# *docker-compose: Fichero .yml - depends\_on*

```
[...]
mysql:
 image: mysql:5.7
 environment:
 - MYSQL_ROOT_PASSWORD=root
 - MYSQL_DATABASE=database
 - MYSQL_USER=user
 - MYSQL_PASSWORD=pwd
[...]
```

# *docker-compose: Fichero .yml - depends\_on*

```
[...]
 phpmyadmin:
 image: phpmyadmin/phpmyadmin
 ports:
 - "8081:80"
 environment:
 - PMA_USER=user
 - PMA_PASSWORD=pwd
 - PMA_HOST=mysql
 depends_on:
 - mysql
```

## *docker-compose: Fichero .yml - depends\_on*

- Cuando establecemos la relación `depends_on`, los contenedores se levantarán en orden para respetar la dependencia.
- Si levantamos un sólo contenedor, se levantarán previamente todos aquellos de los que dependa:

```
$ docker-compose stop
$ docker-compose up webserver
```

- Para levantar `webserver`, previamente se levantará `php-fpm`
- Para levantar `php-fpm`, previamente se levantará `mysql`
- Por lo tanto, al levantar `webserver`, también se levantan `mysql` y `php-fpm`. Sólo `phpyadmin` queda parado.

# *Persistencia en Docker: Volúmenes*

Hasta ahora hemos levantado diferentes contenedores.

Lo malo de los contenedores de Docker es que la información se almacena en el interior de los contenedores... si eliminamos un contenedor se pierden todos los cambios realizados.

Ésto nos genera muchas limitaciones.

# *Persistencia en Docker: Volúmenes*

Afortunadamente, docker permite el uso de volúmenes.

Un volumen en Docker es como un enlace simbólico, vinculamos un directorio o un fichero de nuestro equipo, con un directorio o fichero dentro de contenedor.

De esta forma, todo o que se almacene en un directorio del contenedor vinculado con local por un volumen.... o todo fichero que se modifique que esté vinculado de igual manera no se perderá al eliminar el contenedor, porque es información persistida en nuestro equipo local.

# *Persistencia en Docker: Volúmenes*

Los volúmenes nos permiten:

- Hacer que el directorio `/var/lib/mysql` se almacene en local: No perderemos nuestra base de datos.
- Hacer que el directorio de nuestra aplicación PHP se almacene en local: Podremos montar nuestra aplicación en el contenedor y trabajar con ella desde nuestro IDE, etc. Y ver los cambios en tiempo real sin tener que reconstruir la imagen.
- Tener las configuraciones de PHP y de los virtual host de nginx editables en tiempo real.

# Persistencia en Docker: Volúmenes

Creamos un directorio database, en cuyo interior se guardarán los ficheros del mysql:

```
$ mkdir database
```

Debemos editar nuestro docker-compose.yml y agregaremos el volumen:

```
volumes:
 - "directorioLocal:directorioDocker"
```

# *Persistencia en Docker: Volúmenes*

```
[...]
 mysql:
 image: mysql:5.7
 volumes:
 - "./database:/var/lib/mysql"
 environment:
 - MYSQL_ROOT_PASSWORD=root
 - MYSQL_DATABASE=database
 - MYSQL_USER=user
 - MYSQL_PASSWORD=pwd
[...]
```



# Persistencia en Docker: Volúmenes

Lanzamos docker-compose forzando que haga de nuevo el build:

```
$ docker-compose stop && docker-compose up -d --build
$ ls -lha ./database
```

Si estamos en linux veremos que el directorio database tiene contenido, pero lo está asociado a un uid y gid 999. Esto lo hace así porque es el uid y gid del usuario mysql del contenedor:

```
$ docker-compose exec mysql id mysql
uid=999(mysql) gid=999(mysql) groups=999(mysql)
```

# *Persistencia en Docker: Volúmenes*

PROBLEMÓN!

# Persistencia en Docker: Volúmenes

Para solucionarlo necesitamos que el uid y gid de mysql en el container sea el uid y gid de nuestro usuario en linux... so...

```
$ id
uid=1000(rolando) gid=1000(rolando)
```

En mi caso el uid y gid de mysql debe ser 1000. Para ello debemos hacer que el docker-compose haga un build a partir de la imagen de mysql, porque este cambio lo haremos desde el Dockerfile.

## Persistencia en Docker: Volúmenes

```
$ mkdir mysql && cd mysql
$ touch Dockerfile && vim Dockerfile
FROM mysql:5.7
RUN usermod -u 1000 mysql && groupmod -g 1000 mysql
```

En el apartado del servicio mysql, cambiamos el image: mysql:5.7 por nuestro custom build.

```
$ cd .. && vim docker-compose.yml
 build:
 context: .
 dockerfile: mysql/Dockerfile
```

# Persistencia en Docker: Volúmenes

Ahora forzamos el rebuild:

```
$ docker-compose stop && docker-compose up -d --build
```

Y comprobamos permisos:

```
$ docker-compose exec mysql id mysql
uid=1000(mysql) gid=1000(mysql) groups=1000(mysql)
$ ls -la
drwxr-xr-x 6 rolando rolando 4,0K oct 10 01:20 database
```

## *Persistencia en Docker: Volúmenes*

Vamos a agregar nuevos volúmenes al resto de servicios

## *Persistencia en Docker: Volúmenes*

```
[...]
 php-fpm:
 image: rolandocaldas/php:dev-mysql
 depends_on:
 - mysql
 volumes:
 - ../application:/application
 -
 ./php-fpm/php-ini-overrides.ini:/usr/local/etc/p
hp/conf.d/infrastructure-overrides.ini
[...]
```

## *Persistencia en Docker: Volúmenes*

```
[...]
 webserver:
 image: nginx:alpine
 ports:
 - "8080:80"
 depends_on:
 - php-fpm
 volumes:
 -
 ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf
 - ../application:/application
[...]
```



# Persistencia en Docker: Volúmenes

- **./php-fpm/php-ini-overrides.ini** será el fichero donde modificaremos en tiempo real la configuración de PHP:

```
$ mkdir php-fpm && cd php-fpm
$ touch php-ini-overrides.ini && vim php-ini-overrides.ini
upload_max_filesize = 100M
post_max_size = 108M
```

- **./nginx/nginx.conf** será el fichero dónde configuraremos el virtualHost de nuestra aplicación

```
$ cd .. && mkdir nginx && cd nginx
$ touch nginx.conf && vim nginx.conf
```

# Persistencia en Docker: Volúmenes

```
server {
 listen 80 default;

 client_max_body_size 108M;

 access_log /var/log/nginx/application_access.log;
 error_log /var/log/nginx/application_error.log;

 root /application/public;
 index index.php;

 [...]
```

# Persistencia en Docker: Volúmenes

```
[...]
 if (!-e $request_filename) {
 rewrite ^.*$ /index.php last;
 }

 location ~ \.php$ {
 fastcgi_pass php-fpm:9000;
 fastcgi_index index.php;
 fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
 fastcgi_buffers 16 16k;
 fastcgi_buffer_size 32k;
 include fastcgi_params;
 }
}
```

## Persistencia en Docker: Volúmenes

- **../application:/application** Va a ser el acceso al código fuente de nuestra aplicación... por ahora es algo de lo que no nos preocupamos, salvo de crear el directorio.
- Vamos a tener con PHP-FPM el mismo problema de permisos que tuvimos con database en MySQL por lo que crearemos un Dockerfile para construir una imagen de php-fpm con el uid y gid cambiados a los de nuestro usuario y grupo en la máquina local:

```
$ cd .. && mkdir ../application && cd php-fpm/
$ touch Dockerfile
$ vim Dockerfile
FROM rolandocaldas/php:dev-mysql
RUN usermod -u 1000 www-data && groupmod -g 1000
www-data
```

# *Persistencia en Docker: Volúmenes*

```
$ cd ../ && vim docker-compose.yml
```

```
version: "3.5"
```

```
services:
```

```
 php-fpm:
```

```
 build:
```

```
 context: .
```

```
 dockerfile: php-fpm/Dockerfile
```

```
 depends_on:
```

```
 - mysql
```

```
 volumes:
```

```
[...]
```

## Persistencia en Docker: Volúmenes

```
$ docker-compose stop && docker-compose up -d --build
```

Si ahora accedemos por el navegador a <http://localhost:8080> en lugar de welcome de nginx tendremos un “File not found.”, porque en application no tenemos nada.

Pero antes de seguir, en docker-compose y en los Dockerfiles podemos usar variables de entorno que podemos configurar en un .env situado al mismo nivel que docker-compose.yml

```
$ touch .env
$ vim .env
```

# *Persistencia en Docker: Volúmenes*

```
Symfony
SYMFONY_APP_PATH=../application
```

```
MySQL
MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=db
MYSQL_USER=user
MYSQL_PASSWORD=pwd
```

```
Local
LOCAL_UID=1000
LOCAL_GID=1000
```

## *Persistencia en Docker: Volúmenes*

Editamos el `docker-compose.yml` para incluir las variables de entorno en nuestros servicios



# Persistencia en Docker: Volúmenes

```
$ vim docker-compose.yml
version: "3.5"
services:
 php-fpm:
 build:
 context: .
 dockerfile: php-fpm/Dockerfile
 args:
 - LOCAL_UID=${LOCAL_UID}
 - LOCAL_GID=${LOCAL_GID}
[...]
```

# Persistencia en Docker: Volúmenes

```
[...]
 depends_on:
 - mysql
 volumes:
 - ${SYMFONY_APP_PATH}:/application
 -
./php-fpm/php-ini-overrides.ini:/usr/local/etc/php/conf.d/in
frastructure-overrides.ini
 webserver:
 image: nginx:alpine
 ports:
 - "8080:80"
[...]
```

# Persistencia en Docker: Volúmenes

```
[...]
 depends_on:
 - php-fpm
 volumes:
 -
 ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf
 - ${SYMFONY_APP_PATH}:/application
 mysql:
 build:
 context: .
 dockerfile: mysql/Dockerfile
 args:
 - LOCAL_UID=${LOCAL_UID}
 - LOCAL_GID=${LOCAL_GID}
[...]
```

# Persistencia en Docker: Volúmenes

```
[...]
 volumes:
 - "./database:/var/lib/mysql"
 environment:
 - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
 - MYSQL_DATABASE=${MYSQL_DATABASE}
 - MYSQL_USER=${MYSQL_USER}
 - MYSQL_PASSWORD=${MYSQL_PASSWORD}
 phpmyadmin:
 image: phpmyadmin/phpmyadmin
[...]
```

# Persistencia en Docker: Volúmenes

```
[...]
ports:
 - "8081:80"
environment:
 - PMA_USER=${MYSQL_USER}
 - PMA_PASSWORD=${MYSQL_PASSWORD}
 - PMA_HOST=mysql
depends_on:
 - mysql
```

## *Persistencia en Docker: Volúmenes*

Lo siguiente es editar los Dockerfile de PHP-FPM y MySQL para incluir las variables de entorno de UID y GID

# Persistencia en Docker: Volúmenes

```
$ vim php-fpm/Dockerfile
FROM rolandocaldas/php:dev-mysql
ARG LOCAL_UID
ARG LOCAL_GID
RUN usermod -u ${LOCAL_UID:-33} www-data && groupmod -g
${LOCAL_GID:-33} www-data
```

```
$vim mysql/Dockerfile
FROM mysql:5.7
ARG LOCAL_UID
ARG LOCAL_GID
RUN usermod -u ${LOCAL_UID:-999} mysql && groupmod -g
${LOCAL_GID:-999} mysql
```

## Persistencia en Docker: Volúmenes

Con estos cambios, ya tenemos parametrizados nuestros contenedores y sólo deberemos tocar nuestro `.env` para modificar los valores. Relanzamos los contenedores:

```
$ docker-compose stop && docker-compose up -d --build
```

Y el siguiente paso es descargarnos el código de nuestra aplicación del repositorio:

```
$ cd ../application
$ git clone https://github.com/phpvigo/workshop-symfony4.git ./
```

Si ahora accedemos por el navegador a `http://localhost:8080` en lugar de “File not found.” tendremos:



# *Persistencia en Docker: Volúmenes*

Warning: require(/application/public/../vendor/autoload.php): failed to open stream: No such file or directory in /application/public/index.php on line 8

# Persistencia en Docker: Volúmenes

Warning: require(/application/public/../../vendor/autoload.php): failed to open stream: No such file or directory in /application/public/index.php on line 8

Esto es bueno, está cargando nuestra aplicación, sólo que como acabamos de descargarlo del repositorio, nos faltan todos los vendor:

```
$ docker-compose exec --user www-data php-fpm composer update
```

Si volvemos a acceder a <http://localhost:8080> ya tenemos la página de error de Symfony al existir un error de conexión a la base de datos, porque tenemos que configurar doctrine.

# Persistencia en Docker: Volúmenes

```
$ cd ../application/
$ vim .env
```

Buscamos:

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

Y lo reemplazamos por:

```
DATABASE_URL=mysql://user:pwd@mysql:3306/database
```

Si recargamos <http://localhost:8080> tendremos un error diferente, nuestra aplicación ya es capaz de conectarse al servidor MySQL, ahora sólo falta cargar las migrations.

## Persistencia en Docker: Volúmenes

```
$ docker-compose exec --user www-data php-fpm php bin/console
make:migration
```

```
$ docker-compose exec --user www-data php-fpm php bin/console
doctrine:migrations:migrate
```

- Ahora ya sí... recargamos <http://localhost:8080> y carga la aplicación.
- Además, si entramos en <http://localhost:8081> tendremos acceso a la base de datos db, en cuyo interior están todas las tablas de nuestra aplicación.



**KALEIDO** | COWORKING  
CENTER

**opsou**

[www.opsou.com](http://www.opsou.com)

**pedrofigueras**

[www.pedrofigueras.com](http://www.pedrofigueras.com)