

# docker

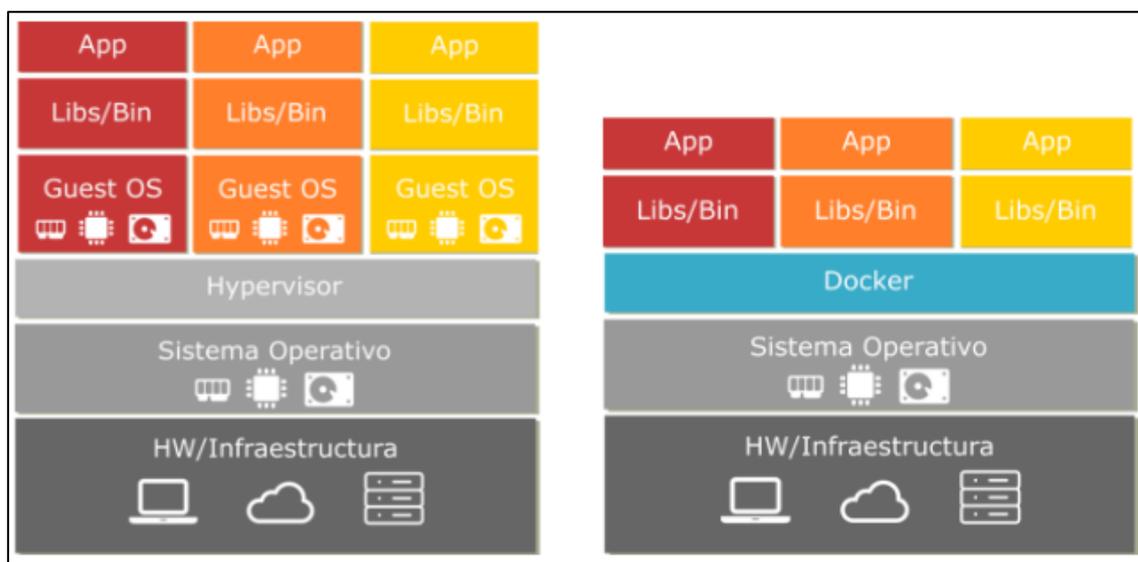
Build, Ship, Run and Manage  
applications

Pello Xabier Altadill Izura  
Cuatrovientos Centro Integrado

## 1 Docker

### 1.1 Introducción

Docker es una aplicación cliente servidor que nos permite crear aplicaciones de una forma empaquetada, optimizada y estándar para provisionar y ejecutarla en un entorno aislado. En cierto modo Docker permite la virtualización de aplicaciones pero de una forma mucho más óptima que si utilizáramos una máquina virtual completa.



Docker nació como un proyecto interno de una empresa de cloud computing; se trataba de una herramienta programada en lenguaje Go y que utilizaban para precisamente para provisionar sus infraestructuras. Su rápido crecimiento, la absorción de muchas otras herramientas y sobre todo su naturaleza de estándar ha sido determinante para convertirse en un éxito.

Hoy en día se utiliza ampliamente como herramienta de empaquetado y ejecución de aplicaciones. Ha encajado perfectamente en los procesos de integración continua de los desarrolladores y los proveedores de computación en la nube ofrecen posibilidad de ejecutar instancias Docker.

### 1.2 Imágenes y Containers

Docker gestiona imágenes y containers, y es fundamental distinguir ambos.

#### Imágenes

Las imágenes son una plantilla de solo lectura que define cómo debe ser la aplicación.Cuál es su base, que configuración tiene, qué ficheros iniciales tiene, etc. Para generar una imagen se utiliza el comando *build* y este se basa en las especificaciones de un fichero llamado Dockerfile.

#### Containers

Un container es una instancia ejecutable de una imagen. En principio son aplicaciones aisladas pero pueden usarse dentro de conjuntos de composer o comunicarse con otros containers independientes. Los datos que se guarden pueden hacerse persistentes o no, según la configuración. Para generar el container utilizamos el comando *docker run*, y a partir de ahí podemos para, reiniciar, eliminar el container como queramos.

## 1.3 Ecosistema Docker

Docker puede ejecutarse de dos maneras:

1. De forma nativa, tendremos un **host Docker** en sistemas:
  - a. Linux
  - b. Windows 10 Enterprise o Windows 2016 Server en adelante
  - c. Mac
2. De forma virtual en el resto de sistemas, Windows o Mac principalmente

Cuando se ejecuta de forma virtual lo que se utiliza es una máquina virtual Linux que hace la función de host Docker.

Docker consta básicamente de:

- Un servidor o *daemon* que es quien ejecuta los containers
- Un cliente, el comando docker, que es quien nos permite gestionar imágenes y containers.

Además de eso, alrededor de los servicios básicos y esos comandos también tenemos:

- Docker Machine
- Docker composer
- Docker Swarm
- Docker Registry

## 2 Instalación

### 2.1 Linux

En los sistemas Linux de 64 bits no deberíamos tener problemas para instalar docker. Si bien la instalación no es tan simple como hacer un install con el gestor de paquetes correspondiente, según la distribución necesitaremos:

- Tener instalados una serie de paquetes
- Añadir un repositorio de paquetes
- Instalar el paquete Docker desde ese repositorio.

Por ejemplo, estos serían los pasos en para Ubuntu:

<https://docs.docker.com/engine/installation/linux/ubuntu/>

En cualquier caso, si nuestra distribución no dispone de imagen, podemos instalar Docker como un binario. Las instrucciones se encuentran en:

<https://docs.docker.com/engine/installation/binaries/>

### 2.2 Windows 2016 Server

Desde la web de docker.com podemos descargar el Docker for Windows, que consiste en un paquete de instalación muy simple.

<https://download.docker.com/win/stable/InstallDocker.msi>

Si no tenemos un sistema Windows compatible el instalador nos dará un aviso. En principio lo podemos instalar en un Windows 10 Enterprise o en servidores Windows a partir de la versión 2016.

## 2.3 Otros sistemas

En otros sistemas donde no podamos tener un host Docker nativo tendremos que emularlo. Para eso tenemos el Docker Toolbox, tanto en Window:

[https://docs.docker.com/toolbox/toolbox\\_install\\_windows/](https://docs.docker.com/toolbox/toolbox_install_windows/)

Como para Mac. En el en caso de Mac sí podemos tener el modo nativo o la opción del Docker Toolbox

<https://docs.docker.com/docker-for-mac/>

## 3 Hello Docker

### 3.1 Introducción

En el DockerHub hay una imagen de una aplicación muy sencilla: hello-world. Esta nos permite ejecutar esa imagen y convertirla en un container que es ejecutado por Docker.

Conectate a tu máquina Docker o abre el Docker Quickstart Terminal .

Luego puedes teclear simplemente:

```
docker run hello-world
```

Y esto es lo que verás:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://cloud.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

## 4 Herramientas Docker

### 4.1 Docker Machine

Docker machine es la herramienta que nos permite crear hosts Docker, tanto de forma local como en proveedores de computación en la nube.



```
docker-machine create -d virtualbox dev
```

Luego la iniciamos:

```
docker-machine start dev
```

Y cambiamos a su Shell:

```
eval $(docker-machine env dev)
```

Para sacar la IP de tu docker-machine puedes utilizar:

```
docker-machine ip nombre_host
```

## 4.2 Docker

### 4.2.1 Introducción

El comando docker es quien nos permite construir imágenes, descargarlas y lo más importante, ejecutar, detener y en definitiva gestionar los containers.

### 4.2.2 Comandos importantes

Los comandos aquí listados siguen una secuencia que en cierto modo puede seguirse como el ciclo de vida de una aplicación.

#### docker images

Este comando nos muestra aquellas imágenes que tenemos instaladas en el sistema.

```
docker images
```

Nos muestra tanto el TAG, como el identificador de imagen, tamaño, etc.

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
pello/mean-aws-sample  latest      c8fde86f3732     11 days ago     667.1 MB
<none>              <none>      52adcca8e826     11 days ago     667.1 MB
node                 boron       06b984afb149     2 weeks ago     655.5 MB
itzg/minecraft-server latest      63d77dd5e40e     3 months ago    774.8 MB
```

#### docker search

Nos permite buscar imágenes de docker en el dockerhub

```
docker search nombre
```

Por ejemplo, podemos buscar la imagen oficial de MySQL y de su servidor:

```
$ docker search mysql
NAME                DESCRIPTION                STARS   OFFICIAL   AUTOMATED
mysql               MySQL is a widely used, open-source relati...  3756   [OK]
mysql/mysql-server  Optimized MySQL Server Docker images. Crea...  247    [OK]
centurylink/mysql   Image containing mysql. Optimized to be li...  48     [OK]
sameersbn/mysql     sameersbn/mysql            41     [OK]
```

En el listado podemos verificar de forma obvia cuál de ellas es la imagen oficial.

A a continuación podemos instalar esa imagen para tenerla disponible localmente.

## docker pull

Este comando nos permite simplemente descargar una imagen de dockerhub

```
docker pull nombreimagen
```

Vamos a descargar por ejemplo una imagen de nginx, un proxy inverso que nos permite acelerar contenido web estático.

```
docker pull nginx
```

Cada imagen está identificada con un tag. Podemos indicar el nombre sin más, por lo que docker aplicará el tag "latest". Si queremos alguna otra versión de la imagen tendremos que indicar el tag.

Por ejemplo, para descargar una versión específica de Ubuntu

```
docker pull ubuntu:12.04
```

En el caso de estar detrás de un proxy, debemos definir una variable de entorno específica. En el caso de Windows, debemos aplicar la siguiente configuración en la consola, donde 172.30.0.1 es el servidor proxy y 80 su puerto

```
set proxy=http://172.30.0.1:80  
set HTTP_PROXY=%proxy%  
set HTTPS_PROXY=%proxy%
```

## docker run

Este es el comando que ejecuta las aplicaciones. Lo que hace es coger la imagen y convertirla en un container. Si la imagen no está ya descargada docker la descarga en el momento.

La forma más simple de hacerlos sería:

```
docker run nombre_imagen
```

Opciones básicas de docker run

- --rm : elimina el container una vez que ha terminado de ejecutarse
- -d : ejecutar el container en background y devolver el id
- -name: dar un nombre al container
- -p puertohost:puertocontainer establece el mapeo de puertos en caso de aplicaciones que utilicen sockets.
- -i : interactivo, mantener abierto el STDIN
- -t : abrir un terminal.
- -h : establecer hostname para la máquina

Podemos indicar otras opciones como -p para mapear los puertos TCP en caso de aplicaciones de redes

```
docker run -d -name nombre_container -p 8080:80 nombre_imagen
```

Otras opciones que se suelen indicar

### docker ps

Nos muestra los containers docker que tenemos en ejecución.

Puede que no salga nada. Mediante la opción `-a` podemos ver los últimos containers que estaban en ejecución anteriormente

### docker logs

Si queremos saber algo sobre la puesta en marcha del *container*, en especial en caso de fallo, podemos usar `docker logs` para saber qué ha pasado dentro del sistema.

```
docker logs nombre_imagen
```

### docker inspect

Esta opción nos muestra detalles sobre los *containers* en ejecución. Basta con indicar el identificador del mismo y obtendremos mucha información útil

```
docker inspect <container-id>
```

### docker exec

Esta opción nos permite ejecutar comandos sobre un *container* en ejecución. Algo especialmente útil si queremos iniciar un Shell y *entrar* en ese *container* para administrarlo.

```
docker exec -it <container-id> /bin/bash
```

Esto es lo que podríamos ver, por ejemplo, en un container nginx: los procesos mínimos y un sistema de ficheros Linux:

```
$ docker exec -it 5761 /bin/bash
root@57613cab3266:/# ps -axf
  PID TTY          STAT       TIME COMMAND
   18 ?           Ss         0:00 /bin/bash
   22 ?           R+         0:00 \_ ps -axf
    1 ?           Ss         0:00 nginx: master process nginx -g daemon off;
    5 ?           S          0:00 nginx: worker process
root@57613cab3266:/# ls /
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@57613cab3266:/#
```

Podemos también, ejecutar un simple comando, siempre identificando el container:

```
docker exec <containter-id> ls /
```

### docker cp

`cp` nos permite copiar ficheros desde nuestro sistema al container y viceversa. Como todo comando `cp`, indicaremos un origen y un destino. Por ejemplo, aquí copiamos el fichero `default.conf` del container a nuestra carpeta actual:

```
docker cp container:/etc/apache2/default.conf .
```

### docker stop

Esto nos permite detener un container. Basta con indicar el nombre

```
docker stop nombreapp
```

Donde nombreapp debe ser el nombre que le habíamos dado a la aplicación.

### docker rm

Elimina un **container**, de forma independiente a la imagen.

```
docker rm nombre_app
```

Puede utilizarse -f para forzar el borrado.

En caso de necesitar detener o **borrar todos** puede usarse esta variante

```
docker rm $(docker ps -a -q)
```

### docker rmi

Elimina **imágenes** que tenemos instaladas. Basta con indicar el nombre de la imagen o bin su id.

```
docker rmi nombre_images
```

Puede utilizarse -f para forzar el borrado en caso de haber alguna pega.

### docker volume

Nos permite crear volúmenes, una especie de discos virtuales que podemos montar en los containers, que podemos gestionar al margen de estos y reutilizarlos en distintos containers.

Usando esta opción podremos ver todos los volúmenes ya existentes en nuestro sistema:

```
docker volumen ls
```

Lo primero que se hace es crear un volumen, que no deja de ser un directorio de nuestro host:

```
docker volume create --name mydatastore
```

Luego podemos sacar los detalles mediante:

```
docker volume inspect mydatastore
```

Para utilizar este volumen en un container, podemos pasarle en el momento de ejecutarlo. En el siguiente ejemplo, creamos un container del SGBD mongo y hacemos que el directorio de datos esté en ese volumen:

```
docker run -d --name mongo -v mydatastore:/data mongo
```

### 4.3 Docker Compose

Docker compose es una maravilla que nos permite poner en marcha múltiples containers independientes que colaboran entre ellos.

Para crear este tipo de aplicaciones se utiliza un fichero llamado `docker-compose.yml` donde se define cada uno de los componentes y en el momento de generarlo, `docker-compose` genera una imagen de cada uno y ejecuta los containers según las especificaciones indicadas.

Más adelante se muestran algunos escenarios de compose y el formato del fichero `docker-compose`.

Una vez configurado, dentro del propio directorio donde se encuentre eses fichero `docker-compose.yml` podemos ejecutar:

```
docker-compose build
```

Lo cual generará las imágenes implicadas Si además queremos ejecutarlas en modo detach o como daemon, hacemos:

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

A partir de ahí, si queremos parar, iniciar, reiniciar podemos hacer

```
docker-compose start/stop/restart
```

Y si queremos eliminar para volver a hacer un build nuevo, hacemos

```
docker-compose rm -f
```

### 4.4 Docker Swarm

Docker Swarm es una herramienta que nos permite crear clusters de hosts Docker, de tal manera que nuestras aplicaciones pueden escalarse de forma más fácil. Básicamente se trata de un conjunto de `docker-machines` que pueden trabajar juntas.

Vamos a crear un cluster creando este host de containers al que llamaremos **local**:

```
docker-machine create -d virtualbox local
```

A continuación nos vamos a poner nuestra terminal en ese nuevo host Docker.

```
eval "$(docker-machine env local)"
```

Ahora vamos crear algo llamado “Discovery Token”, un dato que cada componente de nuestro cluster utilizará para formar el Swarm.

```
docker run swarm create
```

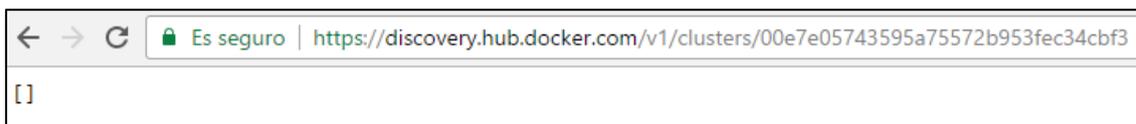
Se verá algo así:

```
$ docker run swarm create
Unable to find image 'swarm:latest' locally
latest: Pulling from library/swarm
ebe0176dcf9a: Pull complete
19f771faa982: Pull complete
902eedf931a: Pull complete
Digest: sha256:815fc8fd4617d866e1256999c2c0a55cc8f377f3dade26c3edde3f0543a70c04
Status: Downloaded newer image for swarm:latest
00e7e05743595a75572b953fec34cbf3
```

La última línea que genera ese comando es el token que necesitamos. Podemos verificar la información relativa al token en:

<https://discovery.hub.docker.com/v1/clusters/TOKEN>

Que en principio mostrará un array vacío:



The screenshot shows a browser address bar with the URL `https://discovery.hub.docker.com/v1/clusters/00e7e05743595a75572b953fec34cbf3`. The page content displays an empty array: `[]`.

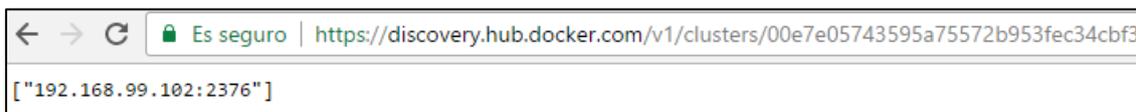
Ahora podemos crear nodos para nuestro cluster, empezando por el nodo maestro o Swarm Master:

```
docker-machine create -d virtualbox --swarm --swarm-master --
swarm-discovery token://TUTOKENAQUÍ swarm-master
```

Comprobamos que la master está ahí:

```
docker-machine.exe env swarm-master
```

Y ya se ve un cambio en la url remota, lo cual servirá al resto de nodos:



The screenshot shows the same browser address bar as before. The page content now displays an array with an IP address: `["192.168.99.102:2376"]`.

Y a continuación creamos varios nodos:

```
docker-machine create -d virtualbox --swarm --swarm-discovery
token://TUTOKENAQUÍ swarm-nodo-00
docker-machine create -d virtualbox --swarm --swarm-discovery
token://TUTOKENAQUÍ swarm-nodo-01
```

También se identifican online:

```
← → ↻ Es seguro | https://discovery.hub.docker.com/v1/clusters/00e7e05743595a75572b953fec34cbf3  
["192.168.99.104:2376", "192.168.99.103:2376", "192.168.99.102:2376"]
```

Ahora ya podemos borrar la docker-machine llamada local creada antes y pasarnos al master.

```
eval "$(docker-machine env swarm-master)"  
docker-machine rm local
```

Mediante docker info puedes verificar el estado del cluster

```
docker info
```

Y a partir de ahí, puedes ejecutar aplicaciones y verás cómo utilizan distintos nodos:

```
docker ps
```

## 5 Recetario Builds Docker

### 5.1 El fichero Dockerfile

El fichero Dockerfile contiene las instrucciones para crear una imagen de aplicación para que docker la pueda convertir en un container ejecutable. Se trata de un fichero de texto que tiene estos elementos principales, entre otros:

#### **FROM**

Indica la imagen en que se basa este build

#### **MAINTAINER**

Nos permite indicar el nombre de autor y el email de quien mantiene este fichero

#### **ARG**

Se trata de argumentos como los que se pasan en el comando docker build, que en este caso lo podemos poner en el Dockerfile

#### **RUN**

Ejecuta los comandos unix indicados para compilar la imagen

#### **ENV**

Establece las variables de entorno que nuestra aplicación puede necesitar.

#### **CMD**

Nos permite introducir un comando en el inicio del container. Estos comandos pueden ser sobrescritos a través del comando run de docker

#### **ENTRYPOINT**

Los ficheros Dockerfile pueden tener un CMD o un ENTRYPOINT. Este último se utiliza cuando tenemos un container que es directamente ejecutable.

En caso de tener un ENTRYPOINT, CMD puede usarse para pasar argumentos.

### ADD

A través de esta instrucción podemos indicar que se copien los nuevos ficheros/directorios dentro de un directorio específico del sistema del container docker.

### EXPOSE

A través de esta instrucción podemos This instruction exposes specified port to the host machine.

## 5.2 Un script Shell

Vamos a empezar con el ejemplo más simple posible, una aplicación que consiste en un script de Shell que simplemente muestra por pantalla un saludo.

### Dockerfile

```
FROM bash:4.4
COPY hello.sh /
CMD ["bash", "hello.sh"]
```

### Build y Ejecución

Hacemos el build y la ejecución con

```
docker build -t my-bash-app .
docker run -it --rm --name my-running-app my-bash-app
```

También podemos ejecutar un container que consista en el Shell BASH

```
docker run -it --rm bash:4.4
```

## 5.3 Una aplicación Java

Supongamos que tenemos una sencilla aplicación java que simplemente muestra un mensaje por la pantalla. La aplicación es un único fichero organizado de la siguiente manera:

- El programa está en el fichero Main.java
- El fichero está dentro de src/io/pello/docker
- La clase java está en el paquete io.pello.docker
- Al compilar, guardaremos el programa en la carpeta bin

Para ejecutar un programa Java utilizaremos una imagen basada en openjdk, la más directa y estándar para aplicaciones Java de consola.

Presta atención a cómo se deben pasar los parámetros en la configuración CMD: es un array con cada una de las opciones.

### Dockerfile

```
FROM openjdk:alpine
COPY ./usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac -d bin src/io/pello/docker/Main.java
CMD ["java", "-cp", "bin", "io.pello.docker.Main"]
```

### Build y Ejecución

```
docker build -t my-java-app .
docker run -it --rm --name my-running-app my-java-app
```

## 5.4 Aplicación Web estática

En este escenario ya vamos a utilizar un container con un servidor que utiliza un puerto web. Se trata de una imagen basada en el archiconocido servidor web apache, y crearemos un container con una web puramente estática. El contenido se encuentra en una carpeta llamada public-html y dentro no hay más que un index y las carpetas css y js donde se aplica bootstrap.

### Dockerfile

El fichero es muy sencillo, basta con indicar el directorio público y copiarlo a un punto concreto:

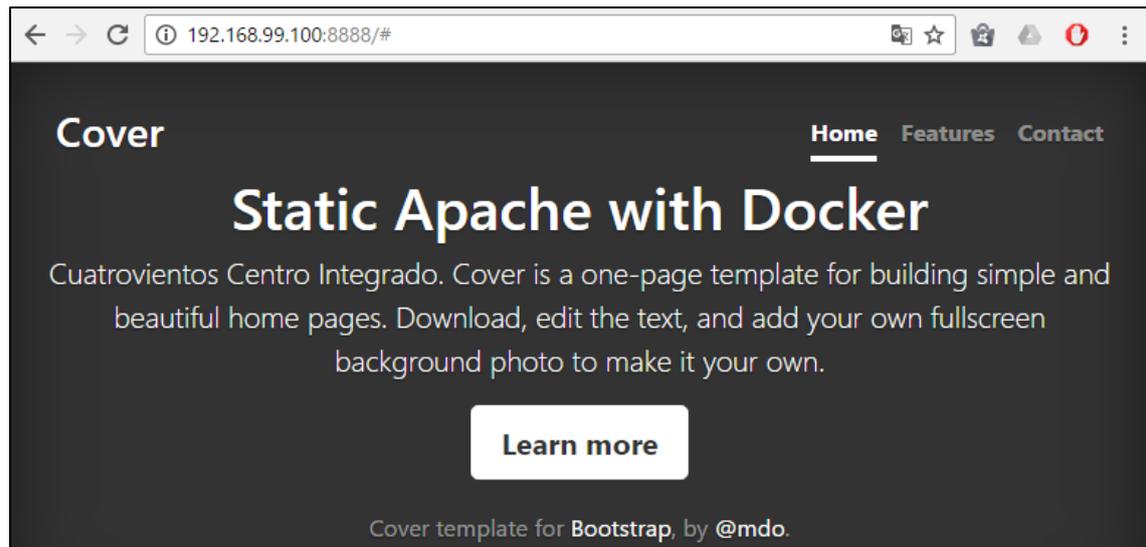
```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
```

### Ejecución

En este caso el container quedará en ejecución como demonio y le mapearemos el puerto del 8888 al 80.

```
$ docker build -t my-apache2 .
$ docker run -dit --name my-running-app -p 8888:80 my-apache2
```

¡OJO! Estamos mapeando el puerto de nuestro host del 8888 al puerto 80 del container. Para probarlo tendremos que poner en el navegador la ip de nuestra docker-machine (default) además del puerto 8888:



## 5.5 Aplicación Node.js

Node.js es un entorno de ejecución de JavaScript para servidores relativamente reciente y que posibilita principalmente el desarrollo de aplicaciones Web fullstack, es decir, aplicaciones desarrolladas íntegramente en JavaScript tanto en el lado cliente como en el servidor.

A diferencia de muchos otros lenguajes como php que se apoyan en servidores web existentes como Apache o LightHTTPD, una aplicación web desarrollada en Node.js tiene la particularidad de que el propio servidor web es también parte de la aplicación. Gracias a las facilidades que da el entorno más varios frameworks específicos como Express esta tarea se facilita enormemente.

### Dependencias

Una particularidad de los proyectos Node.js es que siempre se componen de muchos paquetes externos, y cada aplicación depende de muchas librerías. Las dependencias se especifican en un fichero especial llamado package.json, donde además de la configuración del proyecto hay un listado con las librerías que necesita cada proyecto.

Generalmente los proyectos Node.js se alojan en repositorios sin esas dependencias y se distribuyen sin las librerías; éstas se instalan en el momento que la aplicación se despliega por primera vez mediante el comando

```
npm install
```

En nuestra container Docker para aplicaciones, por tanto, tendremos que preocuparnos de ejecutar esa orden antes de poner en marcha el servicio.

### Dockerfile

Este es el aspecto del fichero de configuración de la imagen. Cada línea está comentada para conocer su propósito:

```
# First we chose node, and its boron version
FROM node:boron
```

```
# Create app directory
RUN mkdir -p /srv/www/src/app
WORKDIR /srv/www/src/app

# Install app dependencies
COPY package.json /srv/www/src/app/
RUN npm install

# Bundle app source
COPY . /srv/www/src/app

EXPOSE 3000
CMD [ "npm", "start" ]
```

### Build y Ejecución

La ejecución se hace a través de `npm start` que en realidad, si observamos el fichero `package.json` lo que hace es ejecutar `node bin/www.js`

```
docker build -t pello/node-express .
docker run -p 3000:3000 -d pello/node-express
```

En el *build* se puede observar que efectivamente se descargan las dependencias de la aplicación.

Al final, ejecutamos el *container* mapeando el puerto 3000 y en el navegador veríamos algo así:



## 5.6 Nginx

Nginx es un proxy web inverso, es decir, en lugar de ser utilizado por los usuarios para navegar más rápido cacheando contenidos ya descargados lo que hace es utilizarse para servir contenido de una web de forma más rápida y eficiente. Se utiliza principalmente para servir contenidos estáticos de las webs como el HTML, JavaScript, CSS, imágenes, etc.

En realidad Nginx es un acelerador para protocolos TCP, pero su uso más extendido es el de proxy para aplicaciones web.

### Uso directo de Nginx

Podemos utilizar nginx de forma directa creando un container y ejecutándolo sobre un directorio ya existente:

```
docker run --name mynginx -p 8888:80 -p 8443:443 -d -v
/c/xampp/htdocs:/usr/share/nginx/html:ro nginx
```

Observa las siguientes opciones:

- Mapeamos un directorio local a uno del container:  
-v /c/xampp/htdocs:/usr/share/nginx/html:ro
- Mapeo de multiples puertos -p 8888:80 -p 8443:443

### Dockerfile

Vamos a publicar una web utilizando Nginx. En este caso en el fichero indicaremos tanto el contenido de la web como de un fichero de configuración propio:

```
FROM nginx
COPY ./public-html/ /usr/share/nginx/html
COPY default.conf /etc/nginx/conf.d/default.conf
```

El fichero de configuración personalizado es el fichero principal de configuración de nginx. Lo que hacemos es modificar el puerto por defecto, algo que se hace en sus primeras líneas:

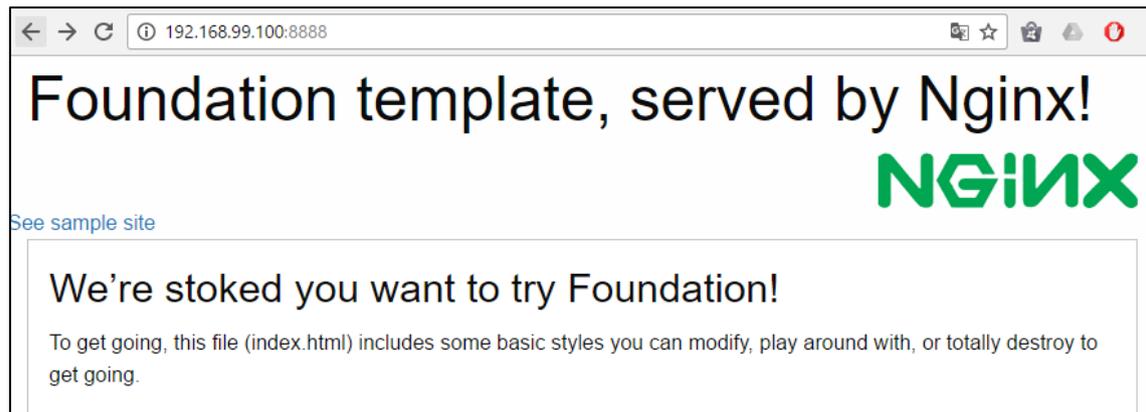
### default.conf

```
server {
    listen 80;
    server_name localhost;
    (Se omite el resto)
```

### Build y ejecución

```
docker build -t pello/nginx .
docker run -d --name my-nginx -p 8888:8080 -p pello/nginx
```

Esto es lo que veremos en la página principal. En este caso sirviendo un diseño basado en el framework Foundation, una alternativa a Bootstrap:



## 5.7 MySQL

Crear un programa en formato de *container* Docker no tiene por qué ser siempre para tener una aplicación de usar y tirar. Un container Docker sirve perfectamente para guardar datos de forma persistente y por tanto los SGBD son ideales para ser integrados como containers.

De hecho, crear aplicaciones utilizando varios containers a la vez pero haciendo que estos sean independientes e intercambiables. De esa manera, por ejemplo, se puede probar una aplicación web sobre un container de BD de desarrollo y luego asociarle un container de producción. La gestión de proyectos de varios containers se hace mediante **Docker composer**, como se muestra más adelante

Vamos a ver cómo ejecutar un container de MySQL estableciendo opciones específicas.

### Ejecución directa

Podemos ejecutar un container MySQL indicando un password para el superusuario mediante una variable de entorno que pasamos en el momento de lanzarlo e indicando el puerto de mapeo:

```
docker run --name my-mysql -e MYSQL_ROOT_PASSWORD=secret -p
3306:3306 -d mysql
```

OJO, si queremos acceder a este MySQL, tendremos que hacerlo de la siguiente manera, creando otro container docker como cliente y pasando los datos de acceso:

```
docker run -it --link my-mysql --rm mysql sh -c 'exec mysql -
h192.168.99.100 -uroot -p'
```

También podemos iniciar una sesión de bash para entrar en el propio container y desde ahí entrar en mysql de forma local:

```
docker exec -it my-mysql bash
```

Aquí vemos como se lanza el Shell y como se accede al MySQL.

```
$ docker exec -it my-mysql bash
root@e9480964dbed:/# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.17 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

### Variables de entorno

Algunas variables de entorno interesantes que podemos utilizar son las siguientes:

- `MYSQL_ROOT_PASSWORD` establece el password de root
- `MYSQL_USER`, `MYSQL_PASSWORD` establecen un usuario/contraseña
- `MYSQL_DATABASE` crea una BD inicial, asignando el usuario creado con `MYSQL_USER` acceso total.
- `MYSQL_ALLOW_EMPTY_PASSWORD` con la opción `yes`, no necesitaremos usar contraseña.

### Ejecución directa con configuración y datos externos

Los datos de MySQL pueden almacenarse dentro del propio container, pero también podemos hacer que MySQL utilice un directorio de datos externo. Mediante esta ejecución, en la que le pasamos los volúmenes de directorio de datos y configuración conseguiremos ese resultado:

```
docker run -d -n my-mysql \
-e"MYSQL_ROOT_PASSWORD=secret" \
--publish 3306:3306 \
--volume=/home/docker/my-mysql/conf.d:/etc/mysql/conf.d \
--volume=/home/docker/mysql-datadir:/var/lib/mysql \
Mysql
```

Los directorios `/home/docker` de origen debe existir en nuestro sistema, obviamente.

### Creando una BD inicial

Podemos hacer que la instancia de MySQL contenga una BD concreta cargada con datos si al hacer el build metemos el volcado de la BD a una carpeta `/docker-entrypoint-initdb.d`. Dentro de esa carpeta, todos aquellos ficheros `.sh`, `.sql` o `.sql` comprimidos se ejecutarán y llenarán la BD.

Por lo tanto, en nuestro directorio donde haremos el *build* de esta imagen, tendremos un fichero con el volcado. En el Dockerfile indicaremos cuál y cómo copiarlo a `/docker-entrypoint-initdb.d`

### Dockerfile

Mediante un Dockerfile podremos crear un container personalizado, con datos iniciales y sin necesidad de pasar tantos parámetros.

En el fichero podemos indicar tanto la configuración de MySQL como pasar un volcado de la BD inicial. Puede pasarse en formato comprimido siempre que la extensión sea sql.gz.

```
FROM mysql
MAINTAINER Pello Altadill
# We copy our db config and dump
COPY mysql/ /etc/
ADD dump/series.sql.gz /docker-entrypoint-initdb.d

ENV MYSQL_DATABASE=series \
    MYSQL_USER=seriesuser \
    MYSQL_PASSWORD=secret \
    MYSQL_ROOT_PASSWORD=secret

EXPOSE 3306:3306
CMD ["mysqld"]
```

### Build y ejecución

Para ejecutar a partir del Dockerfile anterior, simplemente podemos hacer:

```
docker build -t pello/mysql .
docker run -d --name my-pello-mysql pello/mysql
```

Y ahora podemos entrar en el Shell y comprobar que efectivamente la BD se ha creado con su usuario/contraseña y los permisos listos:

```
$ docker exec -it my-pello-mysql bash
root@b840b4943774:/# mysql -u seriesuser series -p
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.17 MySQL Community Server (GPL)
```

### ¡OJO!

Ese container puede iniciarse/pararse siempre que haga falta, pero obviamente, si se elimina se perderán los datos. Si esto no es lo que interesa puede pensarse en otra estrategia para gestionar los datos, o bien en un volumen externo, o bien en otro container.

## 5.8 MongoDB

MongoDB es un SGBD NoSQL, es decir, es una base de datos orientada a documentos que no tiene nada que ver con el clásico modelo relacional. Las aplicaciones más frecuentes de MongoDB suelen estar vinculadas al Big Data o a proyectos que precisan ser rápidamente escalables. Son aplicaciones en cuyos datos se renuncia a la integridad pero que a la vez son muy flexibles y permiten un alto rendimiento en consultas.

Al margen de las diferencias con las BBDD SQL tradicionales, sigue siendo un almacén de datos persistente y por tanto hay que tener algunas consideraciones similares.

### Ejecución directa

Podemos ejecutar un container MySQL de forma directa, sin contraseñas ni nada:

```
docker run --name my-mongo -d mongo
```

Podemos pasarle algunas opciones de inicio interesantes:

- [nombre]
- --auth nos obligará a usar autenticación
- --smallfiles en el inicio reservará unos ficheros de datos menores
- --syslog mandará los logs al demonio syslog

Y a través del Shell del container ya podemos acceder:

```
docker exec -it my-mongo bash
```

### Autenticación

Si queremos crear un container con autenticación, debemos añadir un usuario con su contraseña y permisos en una BD especial llamada admin.

Lo primero sería iniciar un container de mongo con la opción --auth:

```
docker run --name my-mongo -d mongo --auth
```

A continuación entramos en el Shell dentro de esa BD:

```
docker exec -it my-mongo bash
```

Y desde el Shell entramos en mongo a la BD admin y creamos un usuario

```
db.createUser({ user: 'myuser', pwd: 'secret', roles: [ { role: "userAdminAnyDatabase", db: "admin" } ] });
```

### Dockerfile

En este Dockerfile vamos a especificar que el directorio donde Mongo guarda los datos lo montaremos como un volumen. Eso nos permitirá adjuntarlo como un volumen reutilizable docker.

```
FROM mongo
MAINTAINER Pello Altadill

# With this, we plan to use data/db as external storage
VOLUME ["/data/db"]
WORKDIR /data

EXPOSE 27017:27017
```

### Build y ejecución

El build no tendrá ningún misterio:

```
docker build -t pello/mongo .
```

Para ejecutar a partir del Dockerfile anterior y poder utilizar un volumen Docker, primero crearemos un volumen en nuestro sistema:

```
docker volumen create --name mongodata
```

Y ahora ejecutamos pasando ese volumen.

```
docker run -d --name pello-mongo -v mongodata:/data pello/mongo
```

De esta manera, los **datos manejados serán totalmente persistentes** aunque destruyamos el container o la imagen.

## 6 Recetario de Docker compose

Para crear una aplicación que implique el uso de múltiples containers, debemos utilizar la herramienta docker compose. Esta herramienta es capaz de crear múltiples containers a partir de las especificaciones indicadas en un fichero llamado docker-compose.yml, donde se definen cada uno de los containers y lo que es más importante, la relación entre ellos.

### 6.1 Aplicación LAMP I

En este ejemplo vamos a crear una aplicación que consiste en dos containers:

1. Apache+php: usaremos un container especial que ya tiene instalado el soporte de php para mysql
2. Mysql: el container oficial de Mysql

#### docker-compose.yml

Este sería el contenido del fichero donde se ve claramente cada container y su configuración:

```
mysqlcontainer:
  image: mysql
```

```
container_name: my_mysql_container

ports:
  - 3306:3306

environment:
  MYSQL_ROOT_PASSWORD: root
  MYSQL_DATABASE: sampledb
  MYSQL_USER: sampleuser
  MYSQL_PASSWORD: secret

apachephpcontainer:
  image: nimmis/apache-php5
  container_name: my_php_apache_container

ports:
  - "8888:80"
  - "8443:443"

volumes:
  - ./public:/var/www/html

links:
  - mysqlcontainer
```

Es especialmente relevante la configuración de “links”. Esa línea es la que establece la conexión entre ambos containers.

### Build y ejecución

Para ejecutarlo, primero nos debemos situar en el mismo directorio donde este el fichero docker-compose.yml

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

Podemos verificar en cualquier momento el estado de los containers con

```
docker-compose ps
```

```
$ docker-compose ps
      Name                    Command                                State      Ports
-----
my_mysql_container          docker-entrypoint.sh mysqld           Up         0.0.0.0:3306->3306/tcp
my_php_apache_container    /my_init                               Up         0.0.0.0:8443->443/tcp, 0.0.0.0:8888->80/tcp
```

Dentro del container my\_php\_apache\_container tenemos un simple php como prueba de concepto. Observa el nombre de host para la conexión:

```
<?php
echo "<h1>PHP with Docker Works!</h1>";
```

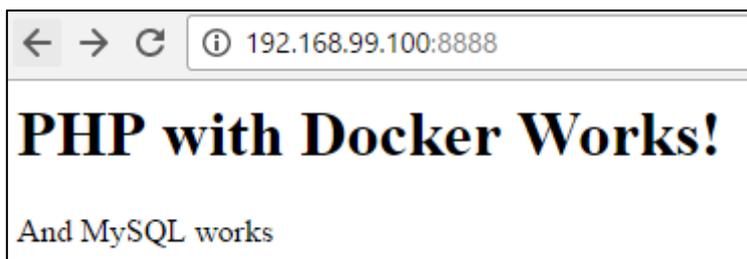
```

$conn = mysqli_connect("mysqlcontainer","sampleuser","secret","sampledb") or
die("Connection failed: ");

echo "And MySQL works";
    
```

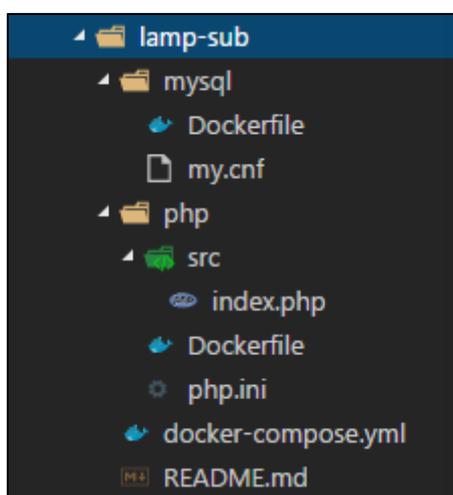
Como se puede ver, el nombre de host de MySQL, es el mismo que hemos establecido en el fichero docker-compose.yml. Docker crea una red entre los dos containers y los entre ellos son accesibles a través de esos nombres.

En el navegador se puede verificar que efectivamente, ya están conectados.



## 6.2 Aplicación LAMP II

Esta es una variante de la composición anterior. En este caso tenemos un fichero docker compose pero por cada imagen utilizada (mysql y php) se utiliza un Dockerfile específico. Cada imagen por tanto se organiza en carpetas separadas y la configuración se hace entre el fichero docker-compose.yml y el Dockerfile de cada imagen.



Es una forma de repartir la configuración que puede ser interesante en proyectos más grandes.

### docker-compose.yml

Este es el nuevo aspecto del docker-compose, donde a través de la directiva build dejamos en manos los detalles de cada imagen a cada Dockerfile correspondiente

```

version: '2'

services:
  mysqlcontainer:
    
```

```
container_name: my_mysql_container
build: ./mysql
phpcontainer:
  container_name: my_php_container
  build: ./php
  ports:
    - "8888:80"
  depends_on:
    - mysqlcontainer
```

### Dockerfiles

Este sería el dockerfile de la imagen de php. Como se puede ver utiliza la base oficial de php pero luego le añadimos la extensión de mysqli.

```
FROM php:7.0-apache
COPY php.ini /usr/local/etc/php/
COPY src/ /var/www/html/
RUN apt-get update \
  && apt-get install -y libmcrypt-dev \
  && docker-php-ext-install pdo_mysql mysqli
RUN chown -R www-data:www-data /var/www/html
RUN chmod -R 755 /var/www/html
EXPOSE 8888:80
```

En cuanto al Dockerfile de MySQL, no tiene gran diferencia respecto al visto anteriormente

### 6.3 Aplicación MEAN

Una aplicación MEAN es una aplicación Web fullstack que se basa en cuatro ingredientes:

1. MongoDB
2. Framework web MVC Express
3. Angular
4. Node.js

En este ejemplo vamos a crear este tipo de aplicación utilizando tres imágenes:

1. Nginx: para los contenidos estáticos y como balanceador de carga
2. Node.js: como servidor web de backend
3. MongoDB: para la persistencia de datos

Como se ve en la aplicación, la configuración está lista para incluir más instancias de Node.js de tal manera que Nginx puede usarlos de forma balanceada. Además separamos la configuración específica en cada fichero Dockerfile

## docker-compose.yml

### 6.3.1 Nginx

La configuración para conseguir el balanceo de carga empezaría así

#### Nginx.conf

```
# Change this if you add more node servers
worker_processes 2;

events { worker_connections 1024; }

http {
    upstream my-node-app {
        least_conn;
        server nodecontainer1:3000 weight=10 max_fails=3 fail_timeout=30s;
        #server nodecontainer2:8080 weight=10 max_fails=3 fail_timeout=30s;
        #server nodecontainer3:8080 weight=10 max_fails=3 fail_timeout=30s;
    }
    ...
}
```

#### Dockerfile

```
FROM nginx
# Copy custom configuration file
COPY nginx.conf /etc/nginx/nginx.conf
# Expose port
EXPOSE 8888
```

### 6.3.2 Node.js

El proyecto ya tiene integrado el framework Express y las librerías de acceso a MongoDB. Dentro tiene una simple consulta a Mongo y muestra el resultado.

#### Dockerfile

```
FROM node:boron
# Create app directory
RUN mkdir -p /srv/www/src/app
WORKDIR /srv/www/src/app
# Install app dependencies
COPY src/package.json /srv/www/src/app/
RUN npm install
# Bundle app source
```

```
COPY src/ /srv/www/src/app
EXPOSE 3000
CMD [ "npm", "start" ]
```

### 6.3.3 MongoDB

En este caso, tratamos de meter datos iniciales a través de la orden mongorestore

#### Dockerfile

```
# This is just a comment.
FROM mongo
MAINTAINER Pello Altadill

# Copy dumped db and restore
WORKDIR /home
COPY dump /home/dump
CMD mongod --smallfiles --fork --syslog; mongorestore

EXPOSE 27017:27017
```

## 7 DockerHub y Git

Es habitual que se tengan proyectos alojados en repositorios como GitHub para gestionar proyectos de software. Ahora, gracias a Docker, podemos convertir nuestros proyectos en imágenes Docker para facilitar su implantación. Por lo tanto, nuestras aplicaciones pueden alojarse en Github en forma de código fuente y en DockerHub en forma de imagen.

### 7.1.1 Integrando Github con Dockerhub

Lo normal es que en el día a día se trabaje con el repositorio Git, y este se actualizará más a menudo. Gracias a la integración con Dockerhub, podemos hacer que cada vez que se haga un commit o se guarden cambios en Github, esto también actualice la imagen en Dockerhub.

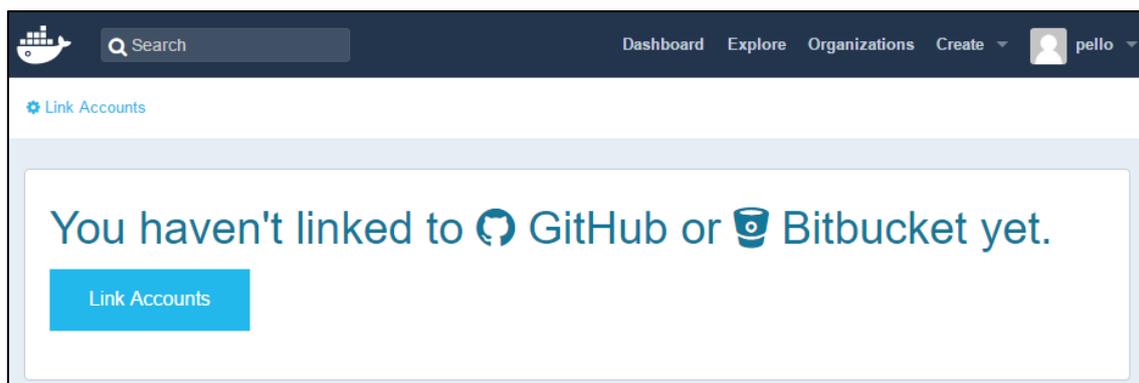
Para esto debemos activar la integración dentro de DockerHub y además incluir un fichero específico dentro del proyecto. Gracias a esto podremos pasar nuestro repositorio GitHub a un repositorio DockerHub de forma automática.

### 7.1.2 Asociando DockerHub a un proyecto Github

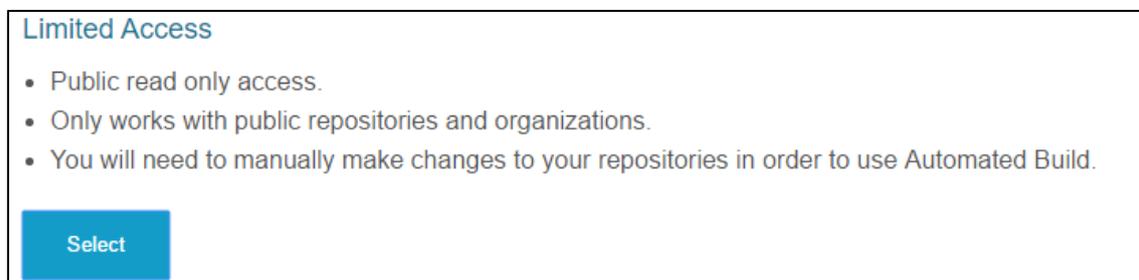
Debemos darnos de alta en DockerHub. A continuación, ya podemos vincular una cuenta de GitHub o Bitbucket. En el menú, vamos a Create > Create Automated Build:



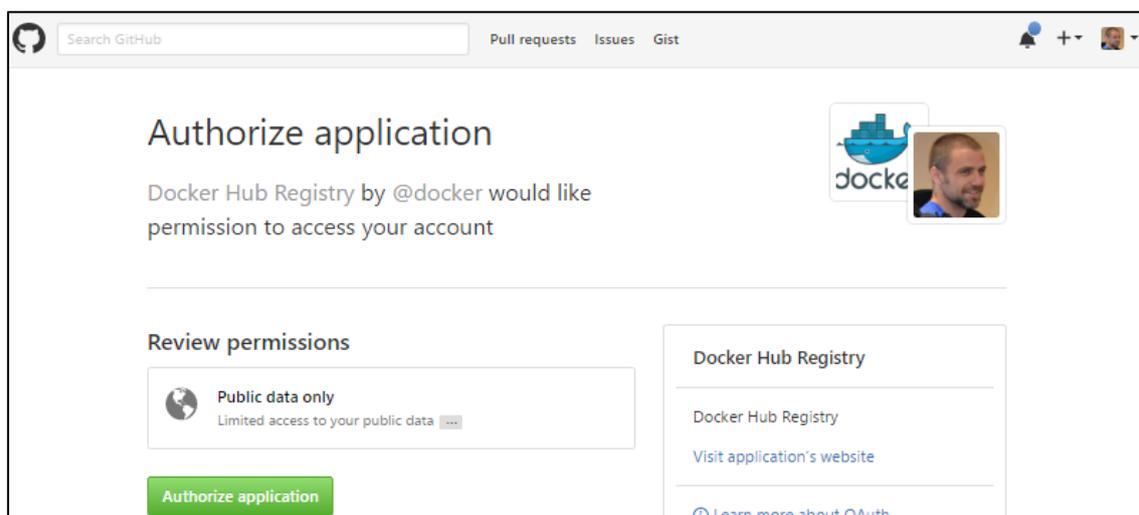
En la siguiente página seleccionaremos GitHub:



Seleccionamos GitHub y en concreto el acceso limitado, o de solo lectura. El otro tipo de acceso es más apropiado para repositorios privados:



Esto nos llevará a la página de GitHub, tendremos que validarnos y aceptar el vínculo:

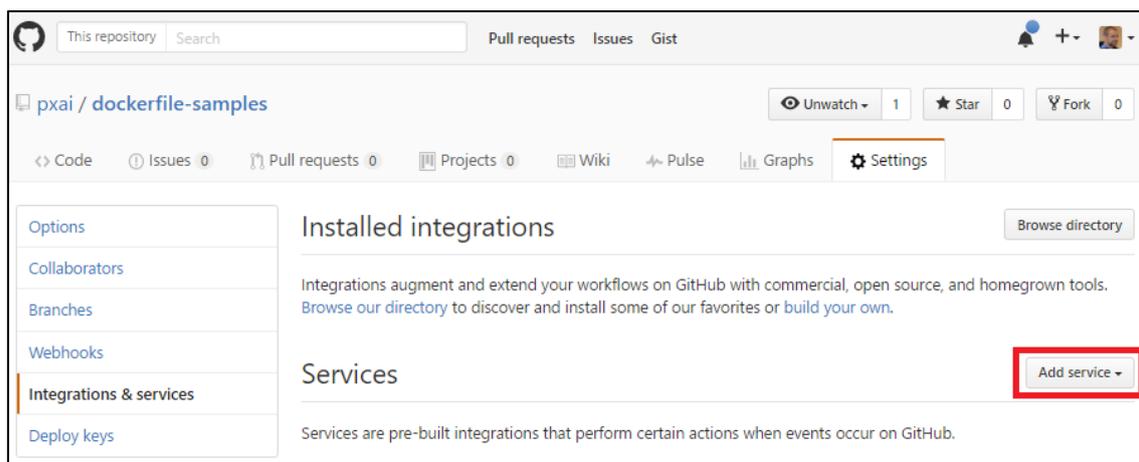


Volveremos de vuelta a DockerHub y podremos ajustar la configuración de notificaciones, etc.

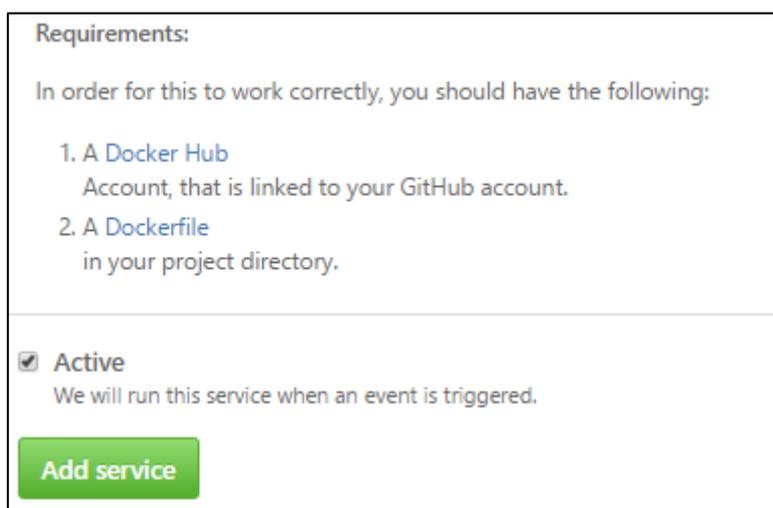
### 7.1.3 Añadir integración de servicio en GitHub

Las integraciones con servicios en GitHub nos permiten lanzar tareas cada vez que se hace un *commit* sobre un repositorio. Esto resulta tremendamente útil para que nuestro proyecto inicie un proceso de integración continua o simplemente automatice tareas como la creación de imágenes Docker.

Los sistemas de integración se añaden dentro de los settings de un repositorio:

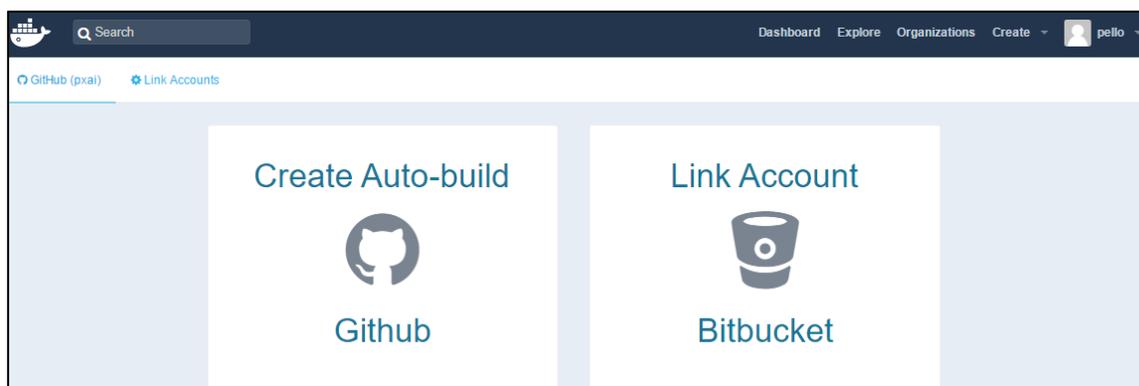


Desplegamos Add Service y seleccionamos Docker. Lo que te indica GitHub es que básicamente necesitarás que el proyecto tenga un fichero Dockerfile y a su vez un repositorio de destino en DockerHub.

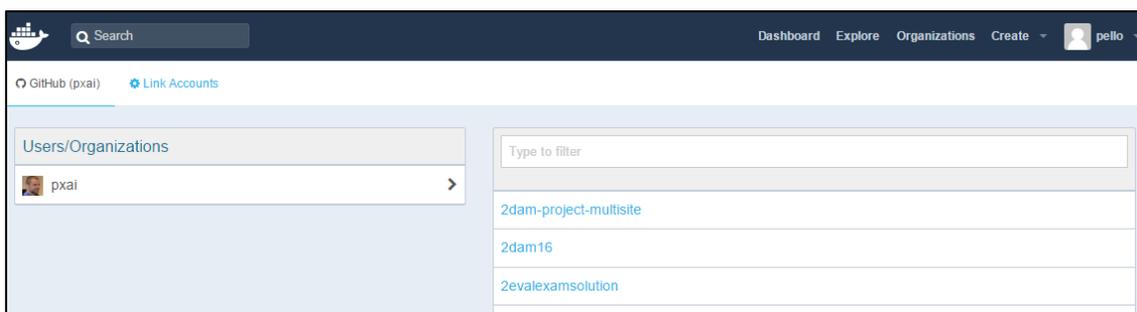


#### 7.1.4 Creando el build automático

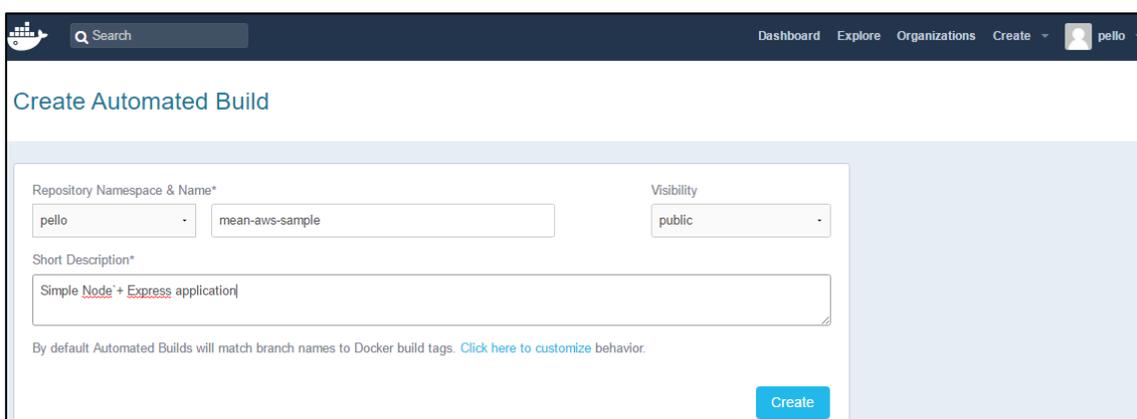
Una vez añadido el servicio y el vínculo vamos a DockerHub y podemos seleccionar para qué proyecto de GitHub haremos un build automático, desde Menú > Create > Create Auto-Build



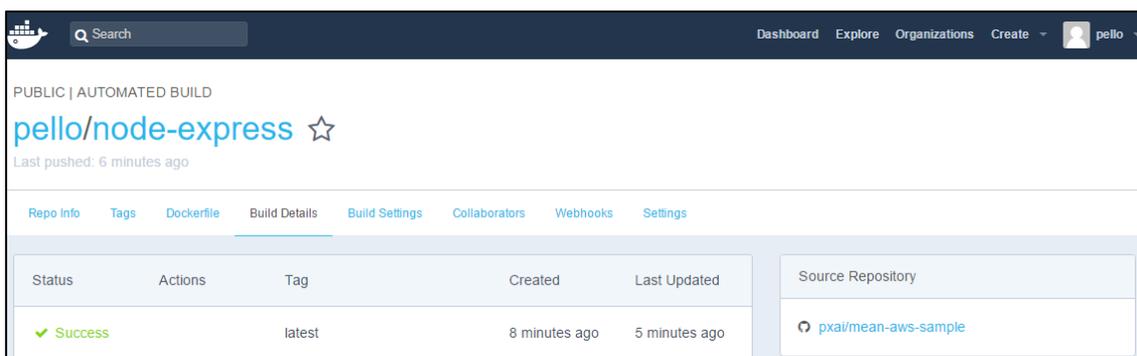
Gracias al vínculo veremos la lista de proyectos de GitHub y seleccionaremos el que nos interese



Indicamos alguna opción y listo:



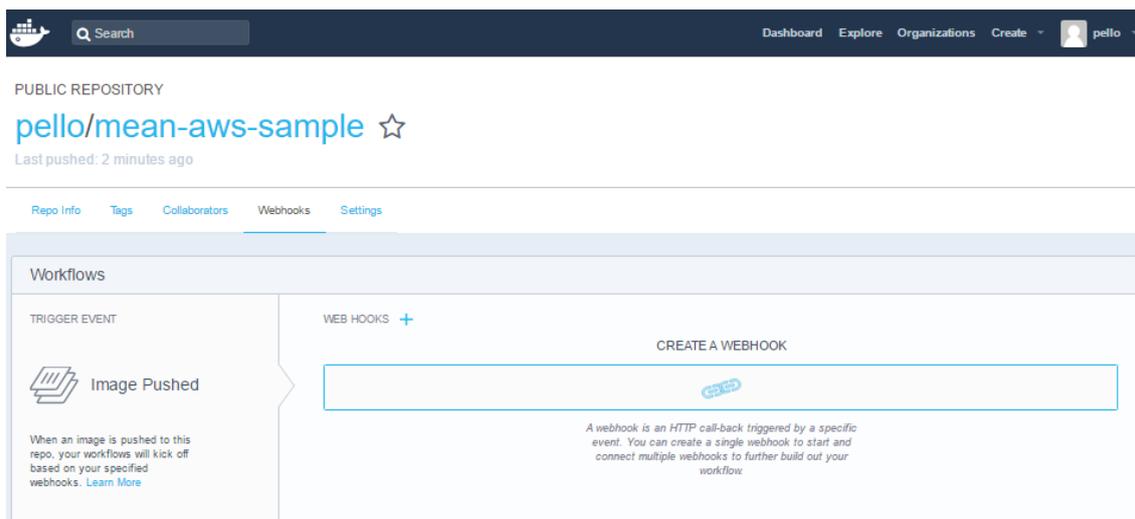
Esto generará un repositorio DockerHub que se actualizará automáticamente en cada commit a GitHub. Por lo tanto, en cada cambio que se haga en la aplicación ¡obtendremos una imagen Docker actualizada y lista para ser utilizada!



### 7.1.5 Integración desde DockerHub a otros servicios

También podemos configurar directamente un proyecto que tengamos en DockerHub para que cuando hagamos un push se lance una petición a otro servicio externo.

Dentro de un repositorio de DockerHub podemos integrar otros servicios creando WebHooks.



PUBLIC REPOSITORY

**pello/mean-aws-sample** ☆

Last pushed: 2 minutes ago

Repo Info Tags Collaborators **Webhooks** Settings

**Workflows**

TRIGGER EVENT

 Image Pushed

When an image is pushed to this repo, your workflows will kick off based on your specified webhooks. [Learn More](#)

WEB HOOKS +

CREATE A WEBHOOK

A webhook is an HTTP call-back triggered by a specific event. You can create a single webhook to start and connect multiple webhooks to further build out your workflow.

Tendremos que indicar un nombre y una URL concreta hacia otro servicio

## 8 Referencias

### 8.1 URLs

- <https://www.docker.com/> Web oficial de docker
- <https://docs.docker.com/compose/compose-file/> Referencia de compose-file
- [https://docs.docker.com/engine/userguide/eng-image/dockerfile\\_best-practices/](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) Dockerfile, best practices
- <https://hub.docker.com> Docker Hub, repositorio de imágenes
- <https://github.com/veggie Monk/awesome-docker> Proyecto con guía de recursos Docker.
- <https://github.com/pxai/dockerfile-samples> Varios proyectos de ejemplo Docker

### 8.2 Libros

- Using Docker, OReilly
- Docker Up & Running, Oreilly
- Docker Cookbook, OReilly
- Docker in practice, Manning
- The Docker Book, <https://www.dockerbook.com/>

### 8.3 Cursos

- <https://www.katacoda.com/courses/docker>
- <https://katacoda.com/courses/docker-orchestration>
- <https://cloudacademy.com/cloud-computing/introduction-to-docker-course/>
- <https://training.docker.com/category/self-paced-online>