

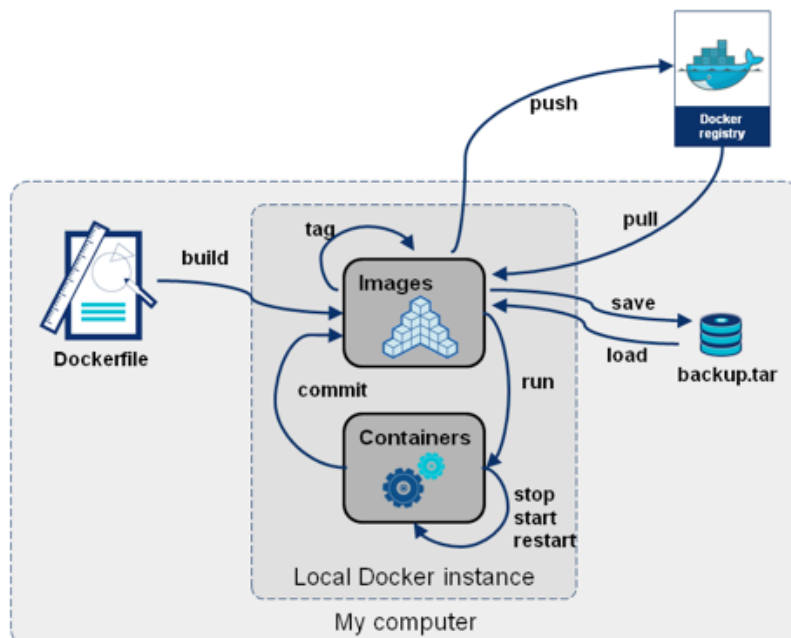
# Creación de imágenes en docker: Dockerfile

---

- Creación de imágenes en docker: Dockerfile
- Creación de una nueva imagen a partir de un contenedor
- Creación de imágenes con fichero Dockerfile
  - El fichero Dockerfile
    - Instrucciones **RUN**, **CMD** y **ENTRYPOINT**
    - Instrucciones **COPY** y **ADD**
  - Construyendo imágenes con docker build
  - Ejemplo de Dockerfile
  - Buenas prácticas al crear Dockerfile
- Distribución de imágenes
  - Distribución a partir de un fichero
  - Distribución usando Docker Hub
- Ejemplo 1: Construcción de imágenes con una página estática
  - Versión 1: Desde una imagen base
  - Versión 2: Desde una imagen con apache2
  - Versión 3: Desde una imagen con nginx
- Ejemplo 2: Construcción de imágenes con una una aplicación PHP
  - Versión 1: Desde una imagen base
  - Versión 2: Desde una imagen con PHP instalado
- Ejemplo 3: Construcción de imágenes con una una aplicación Python
- Imágenes multistage
  - Creando el proyecto NestJS y construcción de una imagen Docker de forma básica
  - Haciendo el Dockerfile con Multi-Stage
- Ejemplo 4: Construcción de imágenes con una una aplicación Angular
- Ejemplo 5: Construcción de imágenes con una una aplicación Spring Boot
  - Versión 1: A partir del fichero JAR ya construido
  - Versión 2: A partir del fichero JAR ya construido pero deconstruido después
  - Versión 3: A través de una imagen Multi-Stage
- Ejercicios
  - Ejercicio 1
  - Ejercicio 2
  - Ejercicio 3
  - Ejercicio 4
  - Ejercicio 5
  - Ejercicio 6
- Bibliografía

Hasta ahora hemos creado contenedores a partir de las imágenes que encontramos en Docker Hub. Estas imágenes las han creado otras personas.

Para crear un contenedor que sirva nuestra aplicación, tendremos que crear una imagen personalizada, es lo que llamamos "dockerizar" una aplicación.



## Creación de una nueva imagen a partir de un contenedor

La primera forma para personalizar las imágenes es partiendo de un contenedor que hayamos modificado.

1. Arranca un contenedor a partir de una imagen base.

```
$ docker run -it --name contenedor debian bash
```

2. Realizar modificaciones en el contenedor (instalaciones, modificación de archivos,...).

```
root@2df2bf1488c5:/# apt update && apt install apache2 -y
root@75f87f84a091:/# echo "<h1>Curso Docker</h1>" >
/var/www/html/index.html
root@75f87f84a091:/# exit
```

3. Crear una nueva imagen partiendo de ese contenedor usando **docker commit**. Con esta instrucción se creará una nueva imagen con las capas de la imagen base más la capa propia del contenedor. Si no indico etiqueta en el nombre, se pondrá la etiqueta **latest**.

```
$ docker commit contenedor lmlopez/myapache2:v1
sha256:017a4489735f91f68366f505e4976c111129699785e1ef609aefb51615f98fc
4

$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
```

```
lmlopez/myapache2      v1      017a4489735f      44 seconds
ago      243MB
...
```

4. Podríamos crear un nuevo contenedor a partir de esta nueva imagen, pero al crear una imagen con este método **no podemos configurar el proceso que se va a ejecutar por defecto al crear el contenedor** (el proceso por defecto que se ejecuta sería el de la imagen base). Por lo tanto en la creación del nuevo contenedor tendríamos que indicar el proceso que queremos ejecutar. En este caso para ejecutar el servidor web apache2 tendremos que ejecutar el comando `apache2ctl -D FOREGROUND`:

```
$ docker run -d -p 8080:80 \
    --name servidor_web \
    lmlopez/myapache2:v1 \
    bash -c "apache2ctl -D FOREGROUND"
```

## Creación de imágenes con fichero Dockerfile

El método anterior tiene algunos inconvenientes:

- **No se puede reproducir la imagen.** Si la perdemos tenemos que recordar toda la secuencia de órdenes que habíamos ejecutado desde que arrancamos el contenedor hasta que teníamos una versión definitiva e hicimos `docker commit`.
- **No podemos configurar el proceso que se ejecutará en el contenedor creado desde la imagen.** Los contenedores creados a partir de la nueva imagen ejecutarán por defecto el proceso que estaba configurado en la imagen base.
- **No podemos cambiar la imagen de base.** Si ha habido alguna actualización, problemas de seguridad, etc. con la imagen de base tenemos que descargar la nueva versión, volver a crear un nuevo contenedor basado en ella y ejecutar de nuevo toda la secuencia de órdenes.

Por todas estas razones, el método preferido para la creación de imágenes es el uso de ficheros `Dockerfile` y el comando `docker build`. Con este método vamos a tener las siguientes ventajas:

- **Podremos reproducir la imagen fácilmente** ya que en el fichero `Dockerfile` tenemos todas y cada una de las órdenes necesarias para la construcción de la imagen. Si además ese `Dockerfile` está guardado en un sistema de control de versiones como git podremos, no sólo reproducir la imagen si no asociar los cambios en el `Dockerfile` a los cambios en las versiones de las imágenes creadas.
- **Podremos configurar el proceso que se ejecutará por defecto en los contenedores creados a partir de la nueva imagen.**
- Si queremos cambiar la imagen de base esto es extremadamente sencillo con un `Dockerfile`, únicamente tendremos que modificar la primera línea de ese fichero tal y como explicaremos posteriormente.

## El fichero Dockerfile

Un fichero **Dockerfile** es un conjunto de instrucciones que serán ejecutadas de forma secuencial para construir una nueva imagen docker. Cada una de estas instrucciones crea una nueva capa en la imagen que estamos creando.

Hay varias instrucción que podemos usar en la construcción de un **Dockerfile**, pero la estructura fundamental del fichero es:

- Indicamos imagen base: FROM
- Metadatos: LABEL
- Instrucciones de construcción: RUN, COPY, ADD, WORKDIR
- Configuración: Variable de entornos, usuarios, puertos: USER, EXPOSE, ENV
- Instrucciones de arranque: CMD, ENTRYPOINT

Veamos las principales instrucciones que podemos usar:

- **FROM:** Sirve para especificar la imagen sobre la que voy a construir la mía. Ejemplo: **FROM php:7.4-apache**.
- **LABEL:** Sirve para añadir metadatos a la imagen mediante clave=valor. Ejemplo: **LABEL company=salestriana**.
- **COPY:** Para copiar ficheros desde mi equipo a la imagen. Esos ficheros deben estar en el mismo contexto (carpeta o repositorio). Su sintaxis es **COPY [--chown=<usuario>:<grupo>] src dest**. Por ejemplo: **COPY --chown=www-data:www-data myapp /var/www/html**.
- **ADD:** Es similar a COPY pero tiene funcionalidades adicionales como especificar URLs y tratar archivos comprimidos.
- **RUN:** Ejecuta una orden creando una nueva capa. Su sintaxis es **RUN orden / RUN ["orden","param1","param2"]**. Ejemplo: **RUN apt update && apt install -y git**. En este caso es muy importante que pongamos la opción **-y** porque en el proceso de construcción no puede haber interacción con el usuario.
- **WORKDIR:** Establece el directorio de trabajo dentro de la imagen que estoy creando para posteriormente usar las órdenes RUN,COPY,ADD,CMD o ENTRYPOINT. Ejemplo: **WORKDIR /usr/local/apache/htdocs**.
- **EXPOSE:** Nos da información acerca de qué puertos tendrá abiertos el contenedor cuando se cree uno en base a la imagen que estamos creando. Es meramente informativo. Ejemplo: **EXPOSE 80**.
- **USER:** Para especificar (por nombre o UID/GID) el usuario de trabajo para todas las órdenes RUN,CMD Y ENTRYPOINT posteriores. Ejemplos: **USER jenkins / USER 1001:10001**.
- **ARG:** Para definir variables para las cuales los usuarios pueden especificar valores a la hora de hacer el proceso de build mediante el flag **--build-arg**. Su sintaxis es **ARG nombre\_variable** o **ARG nombre\_variable=valor\_por\_defecto**. Posteriormente esa variable se puede usar en el resto de la órdenes de la siguiente manera **\$nombre\_variable**. Ejemplo: **ARG usuario=www-data**. No se puede usar con ENTRYPOINT Y CMD.
- **ENV:** Para establecer variables de entorno dentro del contenedor. Puede ser usado posteriormente en las órdenes RUN añadiendo \$ delante de el nombre de la variable de entorno. Ejemplo: **ENV WEB\_DOCUMENT\_ROOT=/var/www/html**. No se puede usar con ENTRYPOINT Y CMD.
- **ENTRYPOINT:** Para establecer el ejecutable que se lanza siempre cuando se crea el contenedor con **docker run**, salvo que se especifique expresamente algo diferente con el flag **--entrypoint**. Su sintaxis es la siguiente: **ENTRYPOINT <command> / ENTRYPOINT ["executable","param1","param2"]**. Ejemplo: **ENTRYPOINT ["/usr/sbin/apache2ctl","-D","FOREGROUND"]**.

- **CMD**: Para establecer el ejecutable por defecto (salvo que se sobrescriba desde la orden **docker run**) o para especificar parámetros para un **ENTRYPOINT**. Si tengo varios sólo se ejecuta el último. Su sintaxis es **CMD param1 param2 / CMD ["param1","param2"] / CMD ["command","param1"]**. Ejemplo: **CMD ["-c" "/etc/nginx.conf"] / ENTRYPOINT ["nginx"]**.

## Instrucciones **RUN**, **CMD** y **ENTRYPOINT**

La sintaxis de Dockerfile a menudo nos permite conseguir el mismo resultado de muchas formas diferentes. Este es el caso con **RUN**, **CMD** y **ENTRYPOINT**, todas estas instrucciones permiten ejecutar comandos. Aunque cada uno de ellos sirve para una situación diferente.

- **RUN** — Se usa para ejecutar comandos relacionados con la instalación de paquetes.
- **CMD** — Indica el comando y argumentos que va a ejecutar el entypoint.
- **ENTRYPOINT** — Se encarga de ejecutar un comando cuando el contenedor arranca. Por defecto el entypoint es **/bin/sh -c**.

Por Ejemplo:

```
FROM ubuntu
ENTRYPOINT ["sleep"]
CMD ["10"]
```

En este caso, cambiamos el **ENTRYPOINT** por defecto para que se ejecute el comando **sleep** y usamos **CMD** para indicar el argumento **10**.

**RUN** no solo ejecuta una instrucción, sino que además crea una imagen después de haberse ejecutado. Docker internamente cachea todas las imágenes con las que trata, una vez crea una imagen y la tiene almacenada, nunca más la va a crear. Cuando creamos un contenedor, y se generan todas sus capas, es probable que la primera ejecución tarde algo de tiempo. La segunda vez, Docker ya tendrá todas las imágenes que necesita para crear el contenedor en un instante.

Si usamos **RUN**, aprovechamos este mecanismo para guardar el resultado de su ejecución. Por lo que cuando creemos un contenedor por segunda vez ya no se tendrá que calcular más. Esto significa que **RUN** no solo se puede usar para instalar paquetes, pero es especialmente útil para eso.

```
RUN pip install pip
```

Ahora lo importante, **¿cuál es la diferencia entre **ENTRYPOINT** y **CMD**?**

Cuando iniciamos un contenedor, podemos indicar una lista de argumentos que le queremos proporcionar:

```
docker run tu_imagen arg1 arg2
```

Los argumentos que usemos sobrescribirán el valor de **CMD** en caso de haberlo y serán ejecutados junto con el comando del **ENTRYPOINT**.

Lo que se ejecute en el contenedor será el resultado de los valores que tenga **ENTRYPOINT** + los valores que tenga **CMD**, que pueden ser sustituidos con los argumentos con los que ejecutemos el contenedor.

Como el **ENTRYPOINT** por defecto es `/bin/sh -c`, nos permite ejecutar cualquier comando de terminal directamente con **CMD**. En la práctica solemos ver lo siguiente:

```
CMD ["ejecutable", "param1", "param2"]
```

Es equivalente a:

```
CMD ejecutable param1 param2
```

Aunque si cambiamos el **ENTRYPOINT**, el modo de empleo de **CMD** pasa a ser:

```
CMD ["param1", "param2"]
```

**ENTRYPOINT** permite indicarle argumentos, **aunque estos no van a ser sobrescribibles vía terminal**:

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

## Instrucciones **COPY** y **ADD**

La instrucción **COPY**, indica a Docker que coja los archivos de nuestro equipo y los añada a la imagen con la que estamos trabajando. **COPY** crea un directorio en caso de que no exista.

Por ejemplo:

```
COPY . ./app  
  
COPY <fuente> <destino>
```

Indica que queremos coger todos los archivos del directorio actual (.) y trasladarlos a al directorio `./app` de la imagen.

**ADD** hace exactamente lo mismo que **COPY**, pero además añade dos funcionalidades. La primera es que permite indicar contenidos vía URL, y la segunda es la extracción de archivos comprimidos.

```
ADD http://example.com/directorio /usr/local/remote/  
ADD recursos/jdk-7u79-linux-x64.tar.gz /usr/local/tar/
```

Para una descripción completa sobre el fichero **Dockerfile**, puedes acceder a la [documentación oficial](#).

## Construyendo imágenes con docker build

El comando **docker build** construye la nueva imagen leyendo las instrucciones del fichero **Dockerfile** y la información de un **entorno**, que para nosotros va a ser un directorio (aunque también podemos guardar información, por ejemplo, en un repositorio git).

La creación de la imagen es ejecutada por el *docker engine*, que recibe toda la información del entorno, por lo tanto es recomendable guardar el **Dockerfile** en un directorio vacío y añadir los ficheros necesarios para la creación de la imagen. El comando **docker build** ejecuta las instrucciones de un **Dockerfile** línea por línea y va mostrando los resultados en pantalla.

Tenemos que tener en cuenta que cada instrucción ejecutada crea una imagen intermedia, una vez finalizada la construcción de la imagen nos devuelve su id. Algunas imágenes intermedias se guardan en **caché**, otras se borran. Por lo tanto, si por ejemplo, en un comando ejecutamos **cd /scripts/** y en otra línea le mandamos a ejecutar un script (**./install.sh**) no va a funcionar, ya que ha lanzado otra imagen intermedia. Teniendo esto en cuenta, la manera correcta de hacerlo sería:

```
cd /scripts/;./install.sh
```

Para terminar indicar que la creación de imágenes intermedias generadas por la ejecución de cada instrucción del **Dockerfile**, es un mecanismo de caché, es decir, si en algún momento falla la creación de la imagen, al corregir el **Dockerfile** y volver a construir la imagen, los pasos que habían funcionado anteriormente no se repiten ya que tenemos a nuestra disposición las imágenes intermedias, y el proceso continúa por la instrucción que causó el fallo.

## Ejemplo de Dockerfile

Vamos a crear un directorio (**nuestro entorno**) donde vamos a crear un **Dockerfile** y un fichero **index.html**:

```
cd build  
~/build$ ls  
Dockerfile  index.html
```

El contenido de **Dockerfile** es:

```
FROM debian:buster-slim  
MAINTAINER Luis Miguel López "luismi.lope@triana.salesianos.edu"  
RUN apt-get update && apt-get install -y apache2
```

```
COPY index.html /var/www/html/  
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Para crear la imagen uso el comando `docker build`, indicando el nombre de la nueva imagen (opción `-t`) y indicando el directorio contexto.

```
$ docker build -t lmlopez/myapache2:v2 .  
...
```

Nota: Pongo como directorio el `.` porque estoy ejecutando esta instrucción dentro del directorio donde está el `Dockerfile`.

Una vez terminado, podríamos ver que hemos generado una nueva imagen:

```
$ docker images  
REPOSITORY          TAG          IMAGE ID          CREATED  
SIZE  
lmlopez/myapache2   v2           3bd28de7ae88     43 seconds  
ago               195MB  
...
```

Si usamos el parámetro `--no-cache` en `docker build` haríamos la construcción de una imagen sin usar las capas cacheadas por haber realizado anteriormente imágenes con capas similares.

En este caso al crear el contenedor a partir de esta imagen no hay que indicar el proceso que se va a ejecutar, porque ya se ha indicado en el fichero `Dockerfile`:

```
$ docker run -d -p 8080:80 --name servidor_web lmlopez/myapache2:v2
```

## Buenas prácticas al crear Dockerfile

- **Los contenedores deber ser "efímeros"**: Cuando decimos "efímeros" queremos decir que la creación, parada, despliegue de los contenedores creados a partir de la imagen que vamos a generar con nuestro `Dockerfile` debe tener una mínima configuración.
- **Uso de ficheros `.dockerignore`**: Como hemos indicado anteriormente, todos los ficheros del contexto se envían al *docker engine*, es recomendable usar un directorio vacío donde vamos creando los ficheros que vamos a enviar. Además, para aumentar el rendimiento, y no enviar al daemon ficheros innecesarios podemos hacer uso de un fichero `.dockerignore`, para excluir ficheros y directorios.
- **No instalar paquetes innecesarios**: Para reducir la complejidad, dependencias, tiempo de creación y tamaño de la imagen resultante, se debe evitar instalar paquetes extras o innecesarios. Si algún



paquete no es necesario durante la creación de la imagen, lo mejor es desinstalarlo durante el proceso.

- **Minimizar el número de capas:** Debemos encontrar el balance entre la legibilidad del Dockerfile y minimizar el número de capa que utiliza.
- **Indicar las instrucciones a ejecutar en múltiples líneas:** Cada vez que sea posible y para hacer más fácil futuros cambios, hay que organizar los argumentos de las instrucciones que contengan múltiples líneas, esto evitará la duplicación de paquetes y hará que el archivo sea más fácil de leer. Por ejemplo:

```
RUN apt-get update && apt-get install -y \  
git \  
wget \  
apache2 \  
php5
```

## Distribución de imágenes

---

Una vez que hemos creado nuestra imagen personalizada, es la hora de distribuirla para desplegarla en el entorno de producción. Para ello vamos a tener varias posibilidades:

1. Utilizar la secuencia de órdenes `docker commit` / `docker save` / `docker load`. En este caso la distribución se producirá a partir de un fichero.
2. Utilizar la pareja de órdenes `docker commit` / `docker push`. En este caso la distribución se producirá a través de DockerHub.
3. Utilizar la pareja de órdenes `docker export` / `docker import`. En este caso la distribución de producirá a través de un fichero.

En este curso nos vamos a ocupar únicamente de las dos primeras ya que la tercera se limita a copiar el sistema de ficheros sin tener en cuenta la información de las imágenes de las que deriva el contenedor (capas, imagen de origen, autor etc..) y además si tenemos volúmenes o bind mounts montados los obviará.

### Distribución a partir de un fichero

1. Guardar esa imagen en un archivo .tar usando el comando `docker save`:

```
$ docker save lmlopez/myapache2:v1 > myapache2.tar
```

2. Distribuir el fichero `.tar`.
3. Si me llega un fichero .tar puedo añadir la imagen a mi repositorio local:

```
$ docker load -i myapache2.tar
6a30654d94bc: Loading layer
[=====>] 132.4MB/132.4MB
Loaded image: lmlopez/myapache2:v1
```

## Distribución usando Docker Hub

1. Autenticarme en Docker Hub usando el comando `docker login`.

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub...
Username: lmlopez
Password:
...
Login Succeeded
```

2. Distribuir ese fichero subiendo la nueva imagen a DockerHub mediante `docker push`. Nota: El nombre de la imagen tiene que tener como primera parte el nombre del usuario de DockerHub que estamos usando.

```
$ docker push lmlopez/myapache2:v2
The push refers to repository [docker.io/lmlopez/myapache2:v2]
6a30654d94bc: Pushed
4762552ad7d8: Mounted from library/debian
latest: digest:
sha256:25b34b8342ac8b73058aa07ec935dcf5d33db7544da9a216050e1d2077a
size: 741
```

3. Ya cualquier persona puede bajar la imagen usando `docker pull`.

## Ejemplo 1: Construcción de imágenes con una página estática

En este ejemplo vamos a crear una imagen con una página estática. Vamos a crear tres versiones de la imagen, y puedes encontrar los ficheros en este [directorio](#) del repositorio en el que se basan estos apuntes.

### Versión 1: Desde una imagen base

Tenemos un directorio, que en Docker se denomina contexto, donde tenemos el fichero `Dockerfile` y un directorio, llamado `public_html` con nuestra página web:

```
$ ls
Dockerfile  public_html
```

En este caso vamos a usar una imagen base de un sistema operativo sin ningún servicio. El fichero **Dockerfile** será el siguiente:

```
FROM debian
RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf /var/lib/apt/lists/*
ADD public_html /var/www/html/
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Al usar una imagen base **debian** tenemos que instalar los paquetes necesarios para tener el servidor web, en este caso **apache2**. A continuación añadiremos el contenido del directorio **public\_html** al directorio **/var/www/html/** del contenedor y finalmente indicamos el comando que se deberá ejecutar al crear un contenedor a partir de esta imagen: iniciamos el servidor web en segundo plano.

Para crear la imagen ejecutamos:

```
$ docker build -t lmlopez/ejemplo1:v1 .
```

Comprobamos que la imagen se ha creado:

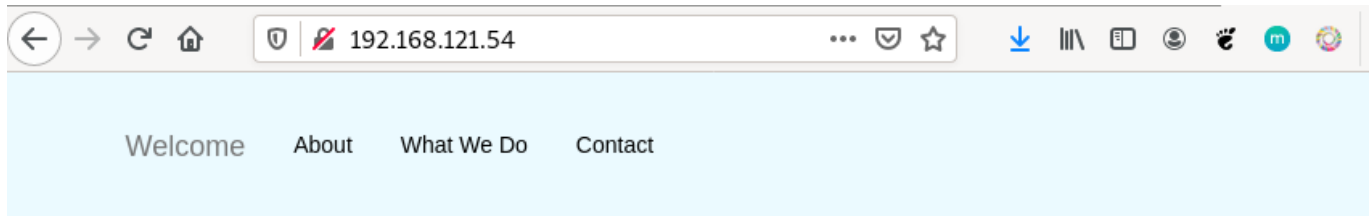
```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/ejemplo1	v1	8c3275799063	1 minute
ago	226MB		

Y podemos crear un contenedor:

```
$ docker run -d -p 80:80 --name ejemplo1 lmlopez/ejemplo1:v1
```

Y acceder con el navegador a nuestra página:



## Curso: Introducción Docker

Responsive one-page scrolling template with sticky menu on the top of the page. The active page gets highlighted as the user scrolls on the page or selects an option in the menu.

A back to top arrow fades in as the user starts scrolling down the page which takes back to the welcome screen with one click.

You can edit and publish this template but please leave a reference to [HTML5 Templates](#). Thank You!

[Click To Scroll Down!](#)

### Versión 2: Desde una imagen con apache2

En este caso el fichero **Dockerfile** sería el siguiente:

```
FROM httpd:2.4
ADD public_html /usr/local/apache2/htdocs/
EXPOSE 80
```

En este caso no necesitamos instalar nada, ya que la imagen tiene instalado el servidor web. En este caso y siguiendo la documentación de la imagen el *\*DocumentRoot* es `/usr/local/apache2/htdocs/`. No es necesario indicar el **CMD** ya que por defecto el contenedor creado a partir de esta imagen ejecutará el mismo proceso que la imagen base, es decir, la ejecución del servidor web.

De forma similar, crearíamos una imagen y un contenedor:

```
$ docker build -t lmlopez/ejemplo1:v2 .
$ docker run -d -p 80:80 --name ejemplo1 josedom24/ejemplo1:v2
```

### Versión 3: Desde una imagen con nginx

En este caso el fichero **Dockerfile** sería:

```
FROM nginx
ADD public_html /usr/share/nginx/html
EXPOSE 80
```

De forma similar, crearíamos una imagen y un contenedor:

```
$ docker build -t lmlopez/ejemplo1:v3 .  
$ docker run -d -p 80:80 --name ejemplo1 josedom24/ejemplo1:v3
```

## Ejemplo 2: Construcción de imágenes con una aplicación PHP

---

En este ejemplo vamos a crear una imagen con una página desarrollada con PHP. Vamos a crear dos versiones de la imagen, y puedes encontrar los ficheros en este [directorio](#) del repositorio en el que se basan estos apuntes.

### Versión 1: Desde una imagen base

En el contexto vamos a tener el fichero **Dockerfile** y un directorio, llamado **app** con nuestra aplicación.

En este caso vamos a usar una imagen base de un sistema operativo sin ningún servicio. El fichero **Dockerfile** será el siguiente:

```
FROM debian  
RUN apt-get update && apt-get install -y apache2 libapache2-mod-php7.3  
php7.3 && apt-get clean && rm -rf /var/lib/apt/lists/*  
ADD app /var/www/html/  
RUN rm /var/www/html/index.html  
EXPOSE 80  
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Al usar una imagen base **debian** tenemos que instalar los paquetes necesarios para tener el servidor web, php y las librerías necesarias. A continuación eliminamos la lista de ubicaciones desde donde se instala software, para evitar que se instalen elementos posteriormente. Después añadiremos el contenido del directorio **app** al directorio **/var/www/html/** del contenedor. Hemos borrado el fichero **/var/www/html/index.html** para que no sea el que se muestre por defecto y finalmente indicamos el comando que se deberá ejecutar al crear un contenedor a partir de esta imagen: iniciamos el servidor web en segundo plano.

Para crear la imagen ejecutamos:

```
$ docker build -t lmlopez/ejemplo2:v1 .
```

Comprobamos que la imagen se ha creado:

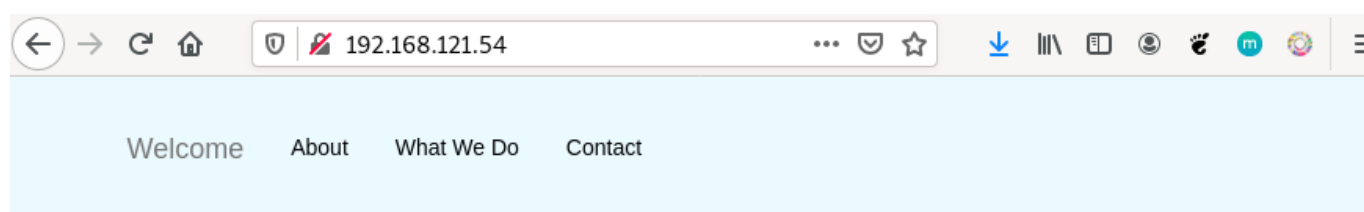
```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/ejemplo2	v1	8c3275799063	1 minute ago
226MB			

Y podemos crear un contenedor:

```
$ docker run -d -p 80:80 --name ejemplo2 lmlopez/ejemplo2:v1
```

Y acceder con el navegador a nuestra página:



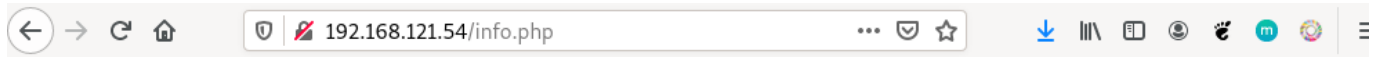
# Curso: Introducción Docker

## Aplicación escrita en PHP

Fecha: 25/03/2021

[Click To Scroll Down!](#)

La aplicación tiene un fichero `info.php` que me da información sobre PHP, en este caso observamos que estamos usando la versión 7.3:



PHP Version 7.3.27-1~deb10u1	
<b>System</b>	Linux a85373d84c85 4.19.0-9-amd64 #1 SMP Debian 4.19.118-2 (2020-04-29) x86_64
<b>Build Date</b>	Feb 13 2021 16:31:40
<b>Server API</b>	Apache 2.0 Handler
<b>Virtual Directory Support</b>	disabled
<b>Configuration File (php.ini) Path</b>	/etc/php/7.3/apache2
<b>Loaded Configuration File</b>	/etc/php/7.3/apache2/php.ini
<b>Scan this dir for additional .ini files</b>	/etc/php/7.3/apache2/conf.d
<b>Additional .ini files parsed</b>	/etc/php/7.3/apache2/conf.d/10-opcache.ini, /etc/php/7.3/apache2/conf.d/10-pdo.ini, /etc/php/7.3/apache2/conf.d/20-calendar.ini, /etc/php/7.3/apache2/conf.d/20-ctype.ini, /etc/php/7.3/apache2/conf.d/20-exif.ini, /etc/php/7.3/apache2/conf.d/20-fileinfo.ini, /etc/php/7.3/apache2/conf.d/20-ftp.ini, /etc/php/7.3/apache2/conf.d/20-gettext.ini, /etc/php/7.3/apache2/conf.d/20-iconv.ini, /etc/php/7.3/apache2/conf.d/20-json.ini, /etc/php/7.3/apache2/conf.d/20-phar.ini, /etc/php/7.3/apache2/conf.d/20-posix.ini, /etc/php/7.3/apache2/conf.d/20-readline.ini, /etc/php/7.3/apache2/conf.d/20-shmop.ini, /etc/php/7.3/apache2/conf.d/20-sockets.ini, /etc/php/7.3/apache2/conf.d/20-sysvmsg.ini, /etc/php/7.3/apache2/conf.d/20-sysvsem.ini, /etc/php/7.3/apache2/conf.d/20-sysvshm.ini, /etc/php/7.3/apache2/conf.d/20-tokenizer.ini
<b>PHP API</b>	20180731
<b>PHP Extension</b>	20180731
<b>Zend Extension</b>	320180731
<b>Zend Extension Build</b>	API320180731,NTS
<b>PHP Extension Build</b>	API20180731,NTS
<b>Debug Build</b>	no
<b>Thread Safety</b>	disabled
<b>Zend Signal Handling</b>	enabled
<b>Zend Memory Manager</b>	enabled

## Versión 2: Desde una imagen con PHP instalado

En este caso el fichero **Dockerfile** sería el siguiente:

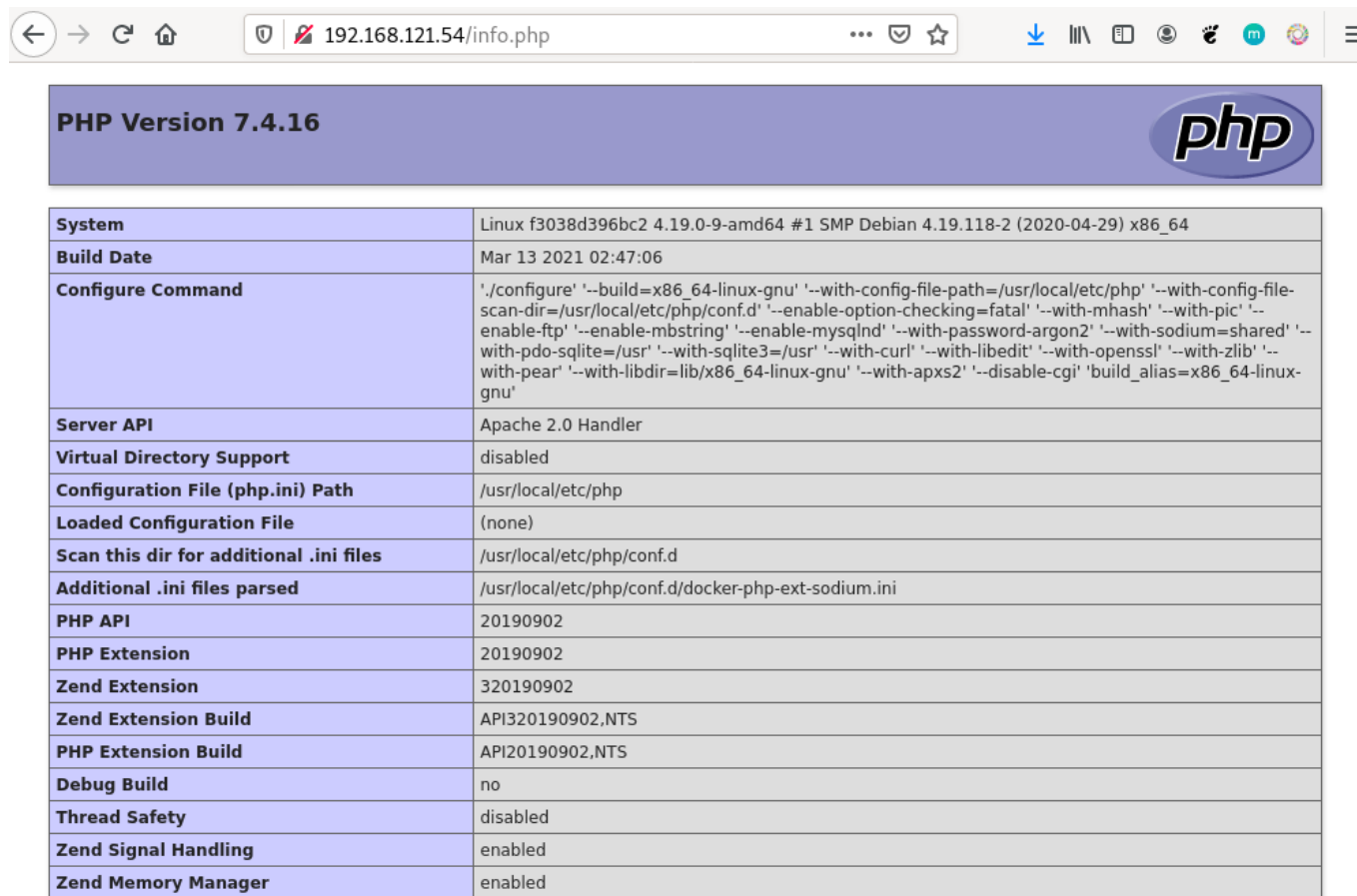
```
FROM php:7.4-apache
ADD app /var/www/html/
EXPOSE 80
```

En este caso no necesitamos instalar nada, ya que la imagen tiene instalado el servidor web y PHP. No es necesario indicar el **CMD** ya que por defecto el contenedor creado a partir de esta imagen ejecutará el mismo proceso que la imagen base, es decir, la ejecución del servidor web.

De forma similar, crearíamos una imagen y un contenedor:

```
$ docker build -t lmlopez/ejemplo2:v2 .
$ docker run -d -p 80:80 --name ejemplo2 lmlopez/ejemplo2:v2
```

Podemos acceder al fichero **info.php** para comprobar la versión de php que estamos utilizando con esta imagen:



PHP Version 7.4.16	
System	Linux f3038d396bc2 4.19.0-9-amd64 #1 SMP Debian 4.19.118-2 (2020-04-29) x86_64
Build Date	Mar 13 2021 02:47:06
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' '--with-pic' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with-password-argon2' '--with-sodium=shared' '--with-pdo-sqlite=/usr' '--with-sqlite3=/usr' '--with-curl' '--with-libedit' '--with-openssl' '--with-zlib' '--with-pear' '--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
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	/usr/local/etc/php/conf.d/docker-php-ext-sodium.ini
PHP API	20190902
PHP Extension	20190902
Zend Extension	320190902
Zend Extension Build	API320190902,NTS
PHP Extension Build	API20190902,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled

## Ejemplo 3: Construcción de imágenes con una aplicación Python

En este ejemplo vamos a construir una imagen para servir una aplicación escrita en Python utilizando el framework flask. La aplicación será servida en el puerto 3000/tcp. Puedes encontrar los ficheros en este [directorio](#) del repositorio en el que están basados estos apuntes.

En el contexto vamos a tener el fichero **Dockerfile** y un directorio, llamado **app** con nuestra aplicación.

En este caso vamos a usar una imagen base de un sistema operativo sin ningún servicio. El fichero **Dockerfile** será el siguiente:

```
FROM debian
RUN apt-get update && apt-get install -y python3-pip && apt-get clean &&
rm -rf /var/lib/apt/lists/*
COPY app /usr/share/app
WORKDIR /usr/share/app
RUN pip3 install --no-cache-dir -r requirements.txt
EXPOSE 3000
CMD [ "python3", "app.py"]
```

Algunas consideraciones:

- Sólo tenemos que instalar pip, que utilizaremos posteriormente para instalar los paquetes Python.



- Copiamos nuestra aplicación en cualquier directorio.
- Con **WORKDIR** nos posicionamos en el directorio indicado. Todas las instrucciones posteriores se realizarán sobre ese directorio.
- Instalamos los paquetes python con pip, que están listados en el fichero **requirements.txt**.
- El proceso que se va a ejecutar por defecto al iniciar el contenedor será **python3 app.py** que arranca un servidor web en el puerto 3000/tcp ofreciendo la aplicación.

Para crear la imagen ejecutamos:

```
$ docker build -t lmlopez/ejemplo3:v1 .
```

Comprobamos que la imagen se ha creado:

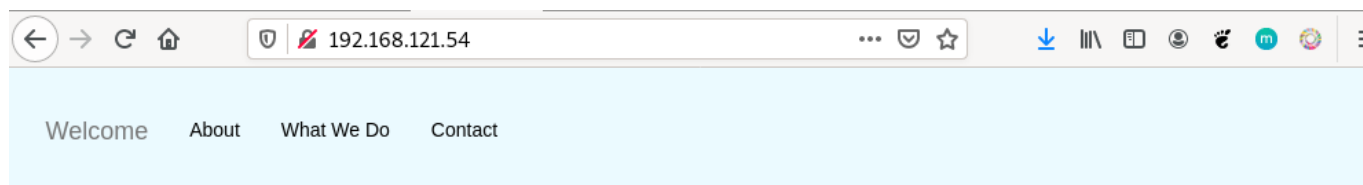
```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/ejemplo3	v1	8c3275799063	1 minute
ago	226MB		

Y podemos crear un contenedor:

```
$ docker run -d -p 80:3000 --name ejemplo2 lmlopez/ejemplo3:v1
```

Y acceder con el navegador a nuestra página:



## Curso: Introducción Docker

### Aplicación escrita en Python

Fecha: 25/3/2021

hostname: 35de9b80e1ea

[Click To Scroll Down!](#)

## Imágenes multistage

Docker Multi-Stage fue introducido en la versión 17.05 y nos permite crear múltiples imágenes de Docker en el mismo Dockerfile, usando múltiples sentencias **FROM** en el mismo fichero.

Vamos a crear un pequeño proyecto con NestJS y a partir del proyecto crearemos el Dockerfile con las dependencias dev y sin ellas, para notar las diferencias entre ambas construcciones de imágenes.

Tenemos un proyecto de ejemplo NestJS en el siguiente repositorio

<https://github.com/ajdelgados/docker-multi-stage>

## Creando el proyecto NestJS y construcción de una imagen Docker de forma básica

NestJS es un framework para crear aplicaciones del lado del servidor con Node.js con un conjunto de tecnologías como Express y TypeScript. Al iniciar debemos instalar de forma global el cli de nest y crear el proyecto

```
npm i -g @nestjs/cli  
nest new docker-multi-state
```

Ya tenemos el proyecto iniciado, vamos al directorio del proyecto y creamos el Dockerfile de forma básica, dejando las dependencias dev

```
FROM node:12.16-alpine  
  
WORKDIR /app/  
COPY . /app/  
RUN npm install && npm run build  
  
ENTRYPOINT [ "npm" ]  
CMD [ "run", "start:prod" ]  
EXPOSE 3000
```

Procedemos a hacer la construcción de la imagen Docker y verificamos su peso

```
docker build -t docker-without-multi-stage .  
docker images
```

Ya podemos ver el peso de la imagen, ahora vamos a cambiar el Dockerfile para aplicar multi-stage.

## Haciendo el Dockerfile con Multi-Stage

Modificando el archivo Dockerfile utilizamos Multi-Stage para eliminar de la imagen final las dependencias dev

```
FROM node:12.16-alpine as builder
LABEL stage=builder
WORKDIR /app/
COPY . /app/
RUN npm install && npm run build

FROM node:12.16-alpine
WORKDIR /app/
COPY --from=builder /app/package.json /app/package-lock.json ./
COPY --from=builder /app/dist ./dist/
RUN npm ci --only=production

ENTRYPOINT [ "npm" ]
CMD [ "run", "start:prod" ]
EXPOSE 3000
```

Implementando el Dockerfile anterior podemos ver que creamos la primera etapa con el nombre **builder** y le colocamos la etiqueta stage con el valor **builder**, eso nos servirá luego para poder filtrar y eliminar las imágenes intermedias que quedan luego de la creación de la imagen final.

En la siguiente etapa, copiamos los archivos necesarios y la versión transpilada para producción, notamos que la sentencia **COPY** tiene el parámetro **--from** con la variable **builder**, eso es para indicar que de la etapa **builder** necesitamos los archivos o directorios.

Volvemos a hacer el build y verificamos el peso de la imagen Docker

```
docker build -t docker-with-multi-stage .
docker images
```

Se puede verificar la diferencia de más de 150MB entre las imágenes finales en su peso y se debe a solo omitir las dependencias dev a la hora de construir la imagen.

También vemos la imagen de la etapa builder, necesitamos eliminar esa imagen, ocupa espacio en nuestro local.

```
docker image prune -f --filter label=stage=builder
```

Ya tenemos las imágenes intermedias eliminadas y una imagen para compartir limpia en el container register y colocar en producción.

## Ejemplo 4: Construcción de imágenes con una aplicación Angular

---

Vamos a utilizar como base el siguiente repositorio: <https://github.com/Accenture/openathon-2019-docker-angular-app>

En este caso vamos a crear un Dockerfile multistage para, en primer lugar, generar el código de producción de nuestra aplicación de Angular, y en segundo lugar, para copiar este código en una nueva imagen más básica que solamente ejecute la aplicación.

En el contexto vamos a tener el fichero **Dockerfile** y otros directorios, entre ellos uno llamado **src** con el código fuente.

Partimos de la imagen de **node**, que tiene todo lo necesario (de partida) para realizar *preparación* del código fuente para producción.

Sobre el repositorio de código hacemos una pequeña modificación, ya que en su momento estuvo bien usar la imagen **:latest** de node, pero al pasar el tiempo, algunas dependencias dan fallos. Usaremos la imagen **node:12**

```
# Utilizamos la imagen de node como base ya que la necesitamos para
"compilar" los fuentes del proyecto Angular. Denominaremos a esta imagen
"build"
FROM node:12 as build

# Recogemos el argumento de entrada si existe, si no usaremos el valor por
defecto (localhost)
ARG ARG_API_URL=localhost

# Asignamos a la variable de entorno API_URL el valor del argumento de
entrada definido en el paso anterior.
# Esta variable de entorno la utiliza el compilador de Angular para
establecerla en el momento de la "compilación"
ENV API_URL=$ARG_API_URL

# Copiamos el fichero package.json a una nueva carpeta de trabajo
COPY ./package.json /usr/angular-workdir/
WORKDIR /usr/angular-workdir

# Lanzamos el comando npm install para que node se descargue todas las
dependencias
# definidas en nuestro fichero package.json
RUN npm install

# Copiamos todo el código del proyecto a la carpeta de trabajo
COPY ./ /usr/angular-workdir

# Ahora que tenemos todas las dependencias y todo el código podemos
generar
# nuestro entregable ejecutando el siguiente comando.
RUN npm run buildProd
```

Nótese que a esta primera imagen se le ha dado un *alias* en el **FROM**, de forma que lo podremos utilizar en la instrucción **COPY** más adelante.

En una segunda fase, queremos aprovechar ese código para desplegarlo en una imagen basada en nginx para poder ejecutarlo.

```
# Llega el momento de preparar el servidor web, para ello usaremos la
imagen base
# de Nginx
FROM nginx

# Copiamos el fichero nginx.conf a la ruta adecuada en la imagen nginx
COPY ./nginx.conf /etc/nginx/nginx.conf

# Borramos todos los ficheros que pudieran existir en la ruta donde
desplegaremos
# el desplegable que hemos generado antes
RUN rm -rf /usr/share/nginx/html/*

# Finalmente copiamos nuestro entregable desde la imagen de node a la ruta
de despliegue
# en la imagen de Nginx
COPY --from=build /usr/angular-workdir/dist/HelloWorld
/usr/share/nginx/html
```

Podemos ver como la instrucción **COPY** va a tomar los ficheros del *stage* anterior para copiarlos en la imagen definitiva.

En este caso no necesitamos **ENTRYPOINT** o **CMD** ya que el que tiene por defecto la imagen de nginx nos sirve, ya que las aplicaciones de Angular no son más que aplicaciones html, js y css.

Para crear la imagen ejecutamos:

```
$ docker build -t lmlopez/angular:1.0 .
```

Comprobamos que la imagen se ha creado:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/angular	1.0	1c508b70198c	1 minute
ago	193MB		

Y podemos crear un contenedor:

```
$ docker run -d -p 8081:80 --name angular lmlopez/angular:1.0
```

Ahora, como ejercicio, deberías crear un segundo Dockerfile, pero que realizara la construcción de la imagen sin el uso de multistage. Compara el tamaño de la imagen generada con el de la imagen multistage y verás como esta última es mucho más pequeña.

## Ejemplo 5: Construcción de imágenes con una aplicación Spring Boot

En este ejemplo, que tendrá varias versiones, vamos a construir la imagen para una aplicación de Spring Boot. Podemos tomar como referencia esta carpeta del repositorio:

<https://github.com/lmlopezmagana/spring-boot-docker/tree/spring-boot-3/EjemploDocker01>.

### Versión 1: A partir del fichero JAR ya construido

Esta primera versión vamos a construir el JAR de forma externa a Docker, y posteriormente vamos a pasarlo a la imagen. Para construir el JAR podemos invocar el siguiente comando dentro de la carpeta en la que se encuentre el fichero `pom.xml`:

```
$ mvn clean install
```

Esto generará la siguiente ruta:

```
$ ls -lart target

total 37152
drwxr-xr-x@ 11 lmlopez  staff      352 Nov 16 13:04 ..
drwxr-xr-x@  3 lmlopez  staff       96 Nov 16 13:04 generated-sources
drwxr-xr-x@  3 lmlopez  staff       96 Nov 16 13:04 maven-status
drwxr-xr-x@  4 lmlopez  staff      128 Nov 16 13:04 classes
drwxr-xr-x@  3 lmlopez  staff       96 Nov 16 13:04 generated-test-
sources
drwxr-xr-x@  3 lmlopez  staff       96 Nov 16 13:04 test-classes
drwxr-xr-x@  4 lmlopez  staff      128 Nov 16 13:04 surefire-reports
drwxr-xr-x@  3 lmlopez  staff       96 Nov 16 13:04 maven-archiver
-rw-r--r--@  1 lmlopez  staff    3141 Nov 16 13:04 EjemploDocker01-
0.0.1-SNAPSHOT.jar.original
drwxr-xr-x@ 11 lmlopez  staff      352 Nov 16 13:04 .
-rw-r--r--@  1 lmlopez  staff 19014241 Nov 16 13:04 EjemploDocker01-
0.0.1-SNAPSHOT.jar
```

Es decir, que hemos generado el fichero `.jar` de la aplicación. Podemos comprobar que funciona si ejecutamos el comando

```
$ java -jar EjemploDocker01-0.0.1-SNAPSHOT.jar
```

Ahora veamos el Dockerfile

```
FROM amazoncorretto:17-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Vemos que en la construcción vamos a poder proporcionar como argumento la ruta del fichero JAR; si no se hace, por defecto se copian todos los ficheros jar de la carpeta target.

A partir de ahí, tan solo tenemos que ejecutar el JAR como lo haríamos sin Docker.

```
$ docker build -t lmlopez/springboot-ej01:1.0 .
```

Comprobamos que la imagen se ha creado:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/springboot-ej01	1.0	ed609bce6be3	1
minute ago	414MB		

Y podemos crear un contenedor:

```
$ docker run -d -p 8083:8080 --name springbootej01 lmlopez/springboot-ej01:1.0
```

Prueba con otras imágenes de base que incluyan Java 17. Hasta hace no mucho estaba disponible la imagen de openjdk, pero ya está deprecada. Puedes empezar probando con eclipse temurin, y comprobar que imagen ocupa menos.

## Versión 2: A partir del fichero JAR ya construido pero deconstruido después

Un *fat jar* de Spring Boot tiene algunas capas debido a la forma en que se *empaqueta* este fichero. Si lo descomprimos, podemos dividirlo entre dependencias internas y externas. A partir de esta estructura de ficheros, podemos copiar en capas individuales en nuestra imagen. De todas ellas, es posible que la única que cambie es la del código, ya que las dependencias en un proyecto suelen ser estables. Así, la creación de imágenes para nuevas versiones, y la carga de contenedores será más rápida.

Primero, una vez obtenido el JAR, ejecutamos los siguientes comandos:

```
$ mkdir target/dependency
$ (cd target/dependency; jar -xf ../*.jar)
```

Ahora veamos el Dockerfile

```
FROM amazoncorretto:17-alpine
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "com.salesianostriana.dam.EjemploDocker02Application"]
```

- En primer lugar, creamos un usuario y un grupo, para que la aplicación no se ejecute como root. Así será menos vulnerable.
- Establecemos como usuario/grupo los recién creados.
- Añadimos un argumento que se puede pasar en la construcción con el valor por defecto como la ruta que hemos creado arriba.
- Copiamos las 3 carpetas que se han generado al descomprimir el *fat jar*. Cada una de ellas irá a una capa diferente de la imagen. La última contiene nuestro código fuente compilado.
- Ejecutamos la aplicación; en lugar de como un jar, le indicamos el *classpath* con la ruta de todas las librerías y como punto de entrada la clase anotada con `@SpringBootApplication`.

Ahora podemos construir la imagen.

```
$ docker build -t lmlopez/springboot-ej02:1.0 .
```

Comprobamos que la imagen se ha creado:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
lmlopez/springboot-ej02	1.0	a204be2f5a2e	1
minute ago	306MB		

Y podemos crear un contenedor:

```
$ docker run -d -p 8080:8080 --name springbootej02 lmlopez/springboot-ej02:1.0
```



## Versión 3: A través de una imagen Multi-Stage

Hasta ahora hemos tenido que crear el jar desde fuera de Docker. Esto también lo podemos hacer dentro de docker usando un Dockerfile multistage, y copiando el resultado de un *stage* en otro.

El fichero Dockerfile debería ser similar a este:

```
FROM amazoncorretto:17-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM amazoncorretto:17-alpine
VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "com.salesianostriana.dam.EjemploDocker03Application"]
```

Como podemos ver, combinamos el Dockerfile de la version 2 con una *stage* anterior donde se descargan las dependencias y se compila el código.

Ahora, la copia de las capas de la aplicación ya no se hace directamente desde nuestro host, sino que se realiza desde el *stage* anterior donde se ha compilado.

Como es lógico, la compilación es más lenta, ya que no se podrá aprovechar la caché de la *stage* de compilación. Sin embargo, la compilación se hará independiente del host donde se ejecute. Esto nos permite también agregar algunos elementos propios de nuestro equipo de desarrollo a la compilación, ya que podemos crear una imagen base que tenga elementos de seguridad, paquetes propios, etc...

Ahora podemos construir la imagen.

```
$ docker build -t lmlopez/springboot-ej03:1.0 .
```

Comprobamos que la imagen se ha creado:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
------------	-----	----------	---------

SIZE				
lmlopez/springboot-ej03	1.0	d12a378f6e73	1	
minute ago	306MB			

Y podemos crear un contenedor:

```
$ docker run -d -p 8080:8080 --name springbootej03 lmlopez/springboot-ej03:1.0
```

Puedes investigar ahora con una aplicación más compleja de ejemplo, como Pet Clinic  
<https://github.com/docker-samples/spring-petclinic-docker>

## Ejercicios

---

### Ejercicio 1

Crea una imagen que sirva la web estática que está en el directorio site del siguiente repositorio:

<https://github.com/josejuansanchez/lab-cep-awesome-docker>

Utiliza la imagen base que prefieras (httpd, nginx, ubuntu...) (puedes probar con más de una).

### Ejercicio 2

Crea una imagen que ejecute la siguiente aplicación web desarrollada con Python y Flask.

<https://github.com/josejuansanchez/lab-cep-flask-app>

Una vez que hayas creado la imagen publícala en Docker Hub.

### Ejercicio 3

Crea una imagen que ejecute la siguiente aplicación web desarrollada en Node.js.

<https://github.com/docker-training/node-bulletin-board>

### Ejercicio 4

Crea una imagen que utilice como base la imagen de nginx e instala/configura los paquetes necesarios para que pueda servir páginas PHP.

Incluya un archivo index.php dentro de la imagen con el siguiente contenido:

```
<?php
phpinfo();
?>
```

En la siguiente referencia puede encontrar los pasos necesarios para instalar y configurar PHP-FPM (PHP FastCGI Process Manager) en el servidor web Nginx.

<https://josejuansanchez.org/iaw/practica-06-teoria/index.html>

## Ejercicio 5

A partir de algunos de los ejercicios que has hecho en PMDM o DI sobre Angular, crea una imagen con dicha aplicación. Optimiza el Dockerfile para que ocupe lo mínimo posible.

## Ejercicio 6

A partir de algunos de los ejercicios que has hecho en AD/PSP sobre Spring Boot, crea una imagen con dicha aplicación, aplicando las diferentes versiones a tu Dockerfile. Optimiza el Dockerfile para que ocupe lo mínimo posible.

## Bibliografía

---

1. [https://iesgn.github.io/curso\\_docker\\_2021/sesion6/](https://iesgn.github.io/curso_docker_2021/sesion6/) (Visitado en Nov-2023)
2. [Curso de Introducción a Docker](#) (Visitado en Nov-2023)
3. <https://github.com/ajdelgados/docker-multi-stage#:~:text=https%3A//ajdelgados.com/2020/04/21/docker%2Dmulti%2Dstage%2Dpara%2Dcrear%2Dimagen%2Dlimpia/> (Visitado en Nov-2023)
4. <https://spring.io/guides/topicals/spring-boot-docker/> (Visitado en Nov-2023)