6. Construyendo mis propias imágenes

Hasta este capítulo hemos estado usando contenedores basados en imágenes de terceros que nos descargábamos desde DockerHub. En este capítulo vamos a realizar la **personalización de las imágenes** para que se adecúen a nuestras necesidades.

Esta personalización para conseguir nuestras propias imágenes la vamos a conseguir **de dos maneras:**

- **Partiendo de un contenedor que tenemos en ejecución** y sobre el que hemos realizado modificaciones.
- **De manera declarativa** a través del **fichero Dockerfile** y un proceso de construcción que veremos que puede ser manual o automático.

¿QUÉ ES PERSONALIZAR UNA IMAGEN?

Pero exactamente, ¿en qué consiste exactamente esto de la personalización de las imágenes?.

Si recordamos un poco los capítulos anteriores tras descargar las imágenes de DockerHub los contenedores que habíamos usado venían literalmente "pelados" incluyendo lo mínimo de lo mínimo para poder trabajar. Ni siquiera tenían cargados los repositorios y se queríamos instalar algo nuevo teníamos que ejecutar ***apt update*** *(o similar).*

Nosotros hemos **instalado aplicaciones**, hemos **incluido el código de nuestras aplicaciones** y si hubiera sido necesario habríamos **configurado los sistemas o servicios** para adecuarlos a nuestras necesidades. Esto es precisamente personalizar los contenedores.

¿Pero qué objetivo conseguimos con esta personalización?. Lo que buscamos es distribuir nuestras imágenes para puedan ser usadas sin problemas en cualquier sistema en el que docker esté instalado.

¿Y cómo encaja esto con nuestra práctica docente?. Pues se relaciona directamente con algunos de los objetivos que nos habíamos planteado al inicio del curso:

- Al hacer nuestras propias imágenes **podemos distribuirlas a los alumnos con TOTAL seguridad** de que no van a tener ningún problema al ahora de acceder y ejecutar esas imágenes.
- Si los **alumnos tienen que entregar sus proyectos** y lo hacen en un contenedor **no** vamos a tener que **montar cada vez un entorno diferente** con todas sus dependencias para poder poner en marcha dichos proyectos.
- Podremos **compartir nuestras imágenes** con la comunidad de docentes.

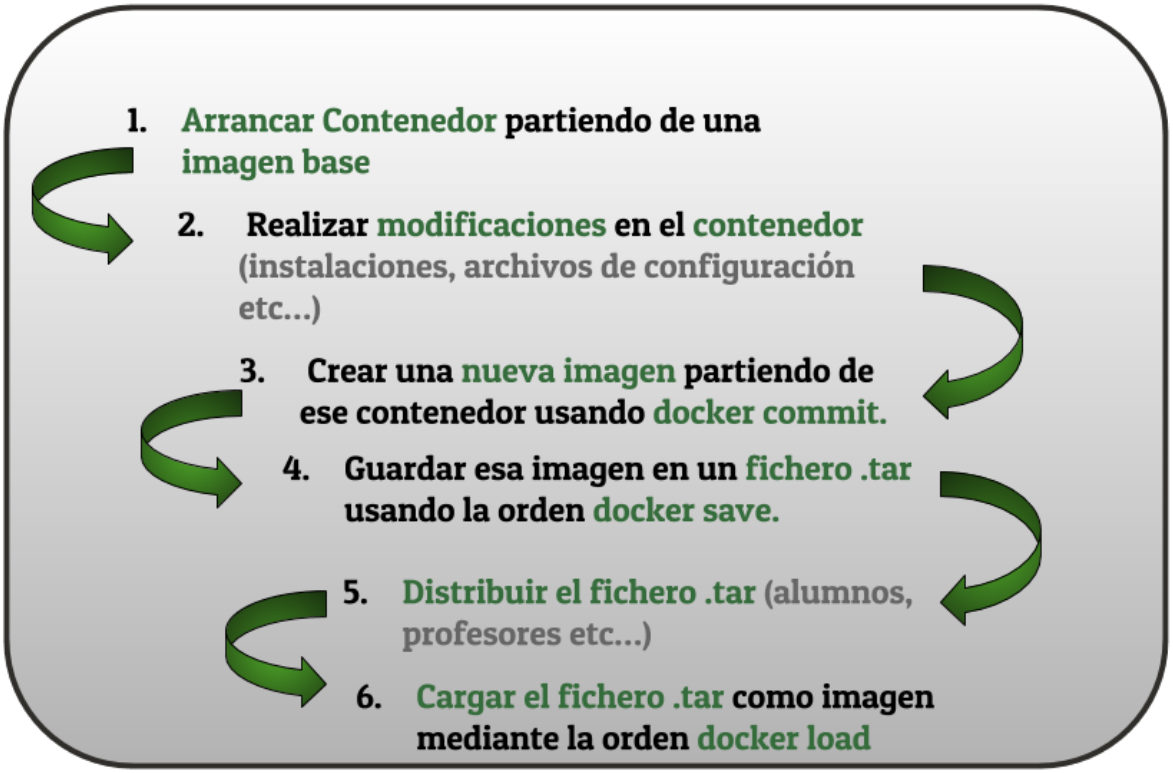
6.1 Desde un contenedor en ejecución

La primera forma para personalizar las imágenes y distribuirlas es partiendo de un contenedor en ejecución. 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.**

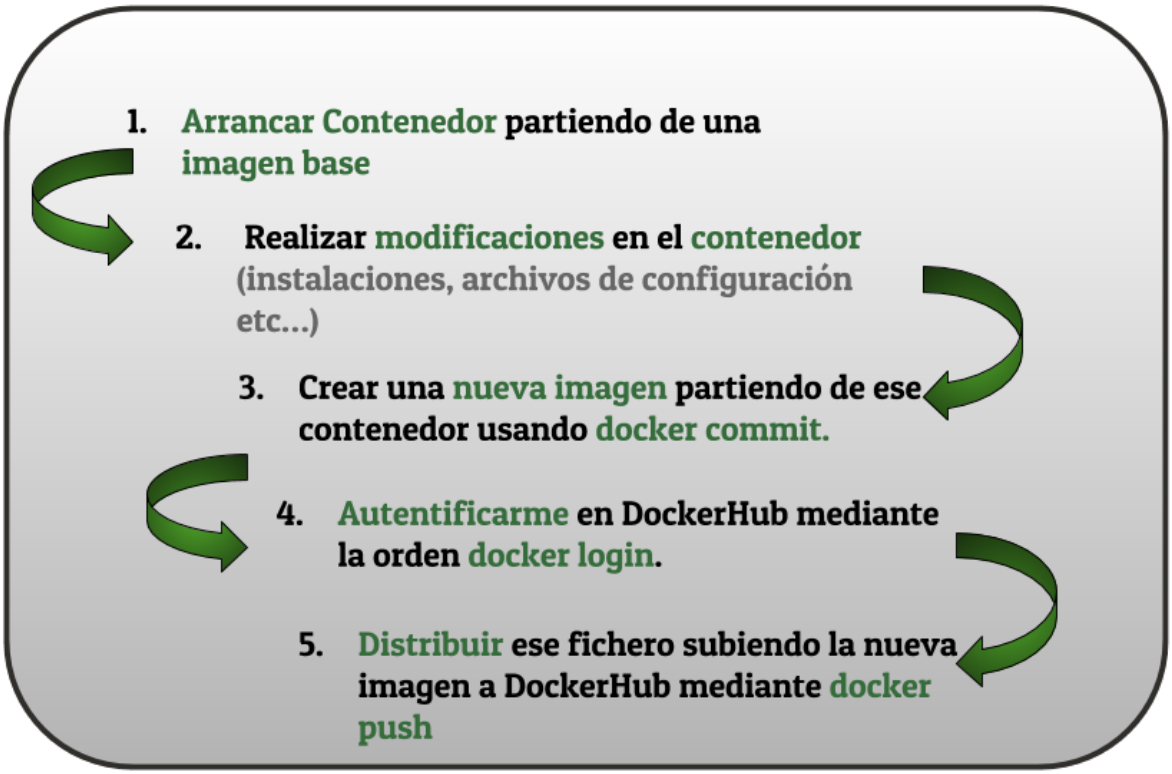
En este curso **nos vamos a ocupar únicamente de las dos primeras** ya que la tercera se limita a copiar el sistema de ficheros sin tener en cuenta la informácion de las imágenes de las que deriva el contenedor (capas, imagen de origen, autor etc..) y además si tenemos volúmenes o bind mounts montados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

En el primer caso el flujo es ejecución es el siguiente:



Juan Diego Pérez Jiménez. *Creación y distribución de nuevas imágenes mediante ficheros* (Dominio público)

Si optamos por la segunda opción el flujo de trabajo es el siguiente:



Juan Diego Pérez Jiménez. *Creación y distribución a través de DockerHub* (Dominio público)

IMPORTANTE: AL HACER COMMIT DEBEMOS AÑADIR EL NOMBRE DE NUESTRO USUARIO DE DOCKERHUB SI QUEREMOS SUBIRLO.

Cada uno de estos comandos tiene una serie de opciones algunas de las cuales las ilustraremos con una serie de ejemplos:

```
# Creación de una nueva imagen a partir del contenedor con nombre ejemplo (tag=latest)

> docker commit ejemplo usuarioDockerHub/ubuntu20netutils

# Igual que la anterior pero añadiendo versión (tag)

> docker commit ejemplo usuarioDockerHub/ubuntu20netutils:1.0

# Igual que la anterior pero pausando el contenedor durante el commit (--pause/-p) y añadiendo un mensaje describiendo el commit (--message/-m)

> docker commit -m "Versión con Nmap" -p ejemplo usuarioDockerHub/ubuntu20netutils:1.1

# Igual que la anterior pero añadiendo la información del autor (--author/-a)

> docker commit -a "Juan Diego Pérez" -m "Versión con Nmap" -p ejemplo
```

```
usuarioDockerHub/ubuntu20netutils:1.1

# Guardar la imagen ubuntu20netutils:1.1 al ficher u20v1.1.tar

> docker save usuarioDockerHub/ubuntu20netutils:1.1 > u20v1.1.tar

# Lo mismo que en el apartado anterior sin la redirección y especificando el fichero (--output / -o)

> docker save --output u20v1.1.tar usuarioDockerHub/ubuntu20netutils:1.1

# Carga la imagen con nombre imagen.tar (--input / -i)

> docker load --input imagen.tar

# Autentificación en DockerHub

> docker login

# Subir una imagen ubuntu20netutils:1.1 a DockerHub

> docker push usuarioDockerHub/ubuntu20netutils:1.1

# Subir una imagen ubuntu20netutils:1.1 a DockerHub suprimiendo la salida que se muestra sobre la información del proceso de subida (--quiet /  -q)

> docker push -q usuarioDockerHub/ubuntu20netutils:1.1

# Subir a DockerHub todas las versiones (tags) de la imagen ubuntu20netutils (--all-tags / -a)

> docker push -a usuarioDockerHub/ubuntu20netutils
```

Lo entenderemos mejor con el siguiente vídeo.

https://youtu.be/eWkqN9U5yJU?list=PL-8CyWabyNa85xowmOeBMCspbrn6qNWgl

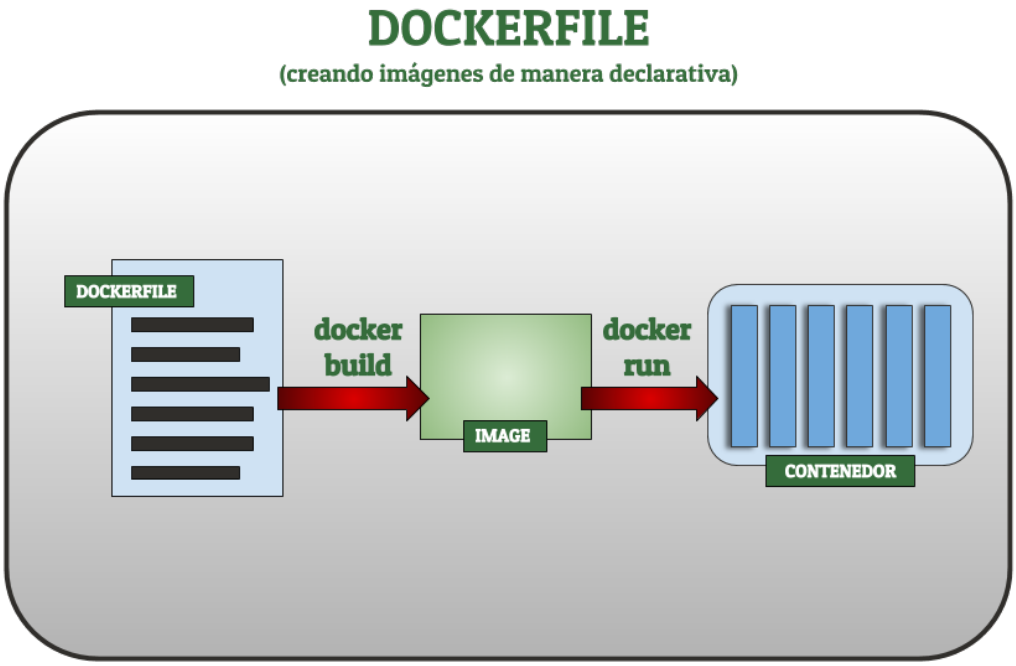
6.2 El fichero Dockerfile

En el apartado anterior hemos visto cómo crear y distribuir mis **nuevas imágenes partiendo de un contenedor.** Esta forma suele ser la preferida cuando empezamos porque es fácil si tenemos ciertos conocimientos de sistemas y porque no hace falta muchos conocimiento sobre docker y su entorno. Sin embargo este tipo de flujo de trabajo aunque fácil, tiene unos inconvenientes importantes:

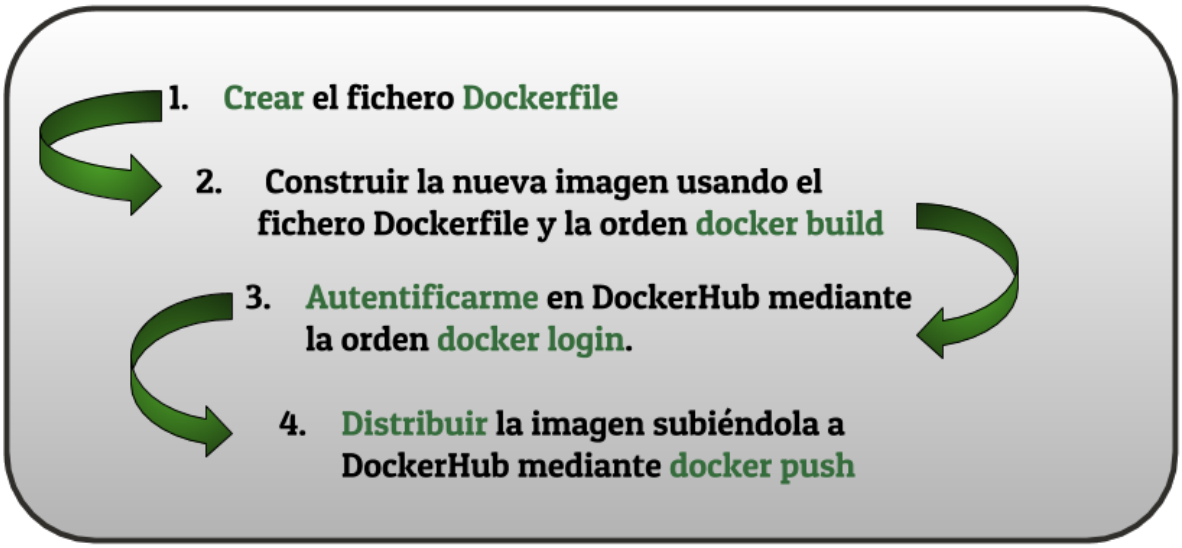
- **No se puede reproducir la imagen.** Si la perdemos tenemos que recordar toda la secuencia de órdenes que habíamos ejecutado desde que arrancamos el contenedor hasta que teníamos una versión definitiva e hicimos docker commit.

- **No podemos 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.

Frente a estos inconvenientes el enfoque preferido es utilizar **ficheros Dockerfile,** que son una **forma declarativa de construir nuevas imágenes.** Este proceso de construcción queda descrito en las siguientes imágenes:



Juan Diego Pérez Jiménez. *Creación de imágenes con Dockerfile* (Dominio público)



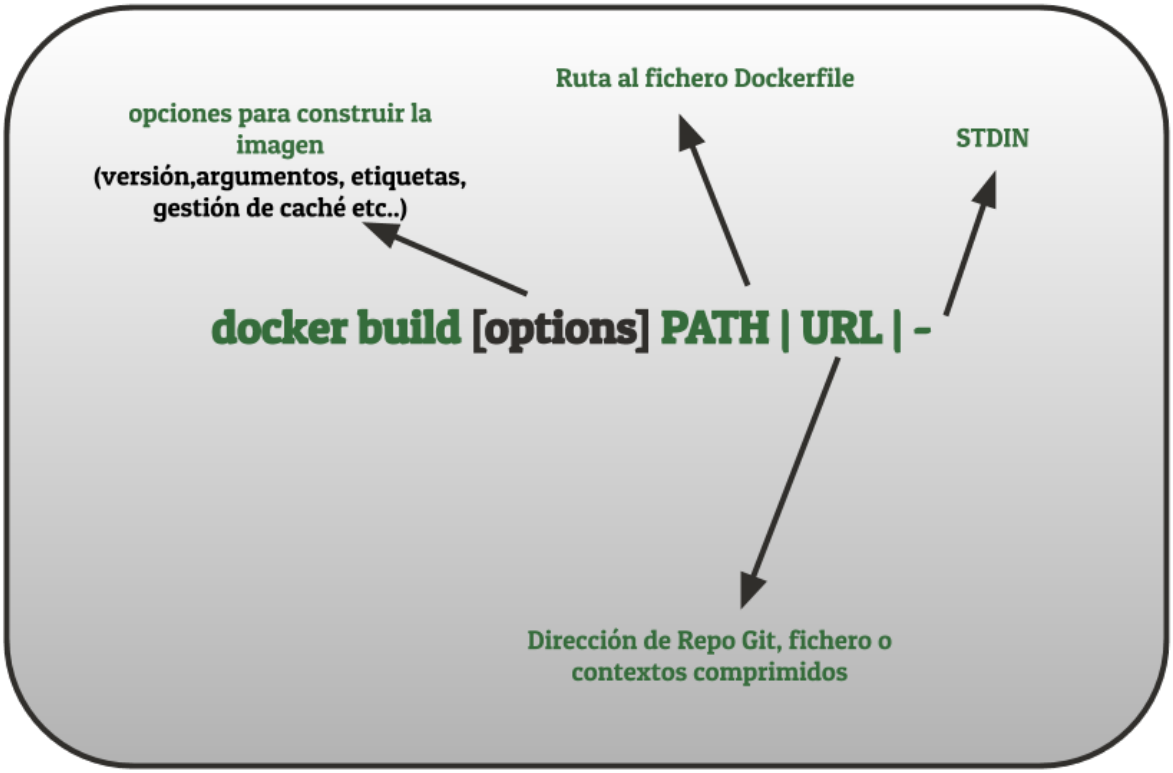
Juan Diego Pérez Jiménez. *Proceso de creación y distribución usando docker build* (Dominio público)

Si trabajamos así y aunque el proceso de construcción del Dockerfile es costoso al principio, vamos a evitar los dos problemas citados anteriormente:

- **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 Dockfile 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.
- 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.

USO DE DOCKER BUILD

Por lo tanto para construir las imágenes necesitamos un **fichero Dockerfile dentro de un contexto**, ya sea en mi equipo o un repositorio exterior, y la **orden docker build** cuya estructura general es la siguiente:



Esta orden tiene muchas opciones pero como estamos acostumbrados a lo largo del curso vamos a ver algunas de las más importantes mediante ejemplos.:

```
# Construcción de una imagen sin nombre ni versión estando el Dockerfile en el mismo directorio donde se ejecuta docker build

> docker build .

# Construcción de una imagen especificando nombre y versión estando el Dockerfile en el mismo directorio donde se ejecuta docker build (--tag/-t)
```

```
> docker build -t  usuario/nombre_imagen:1.0 .

# Construcción de una imagen especificando un repositorio en GitHub donde se encuentra el Dockerfile. Ese repositorio es el contexto de construcción

> docker build -t usuarioDockerHub/nombre_imagen:1.1 https://github.com/...../nombre_repo.git#nombre_rama_git

# Construcción de una imagen usando una variable de entorno estando el Dockerfile en el mismo directorio donde se ejecuta docker build (--build-arg)

> docker build --build-arg user=usuario -t  usuarioDockerHub/nombre_imagen:1.0 .

# Construcción de una imagen sin usar las capas cacheadas por haber realizado anteriormente imágenes con capas similares y estando el Dockerfile en el mismo directorio donde se ejecuta docker build (--no-cache)

> docker build --no-cache -t  usuarioDockerHub/nombre_imagen:1.0 .

# Construcción de una imagen especificando nombre,versión y especificando la ruta al fichero Dockerfile mediante el flag --file/-f

> docker build -t  usuario/nombre_imagen:1.0 -f /home/usuario/DockerProject/Dockerfile
```

NOTA: Si quiero que la imagen construida sea distribuida mediante DockerHub debo poner como prefijo de la imagen mi nombre de usuario de DockerHub.

6.2.1 Resumen de comandos Dockerfile

En el apartado anterior hemos descrito como construir nuevos contenedores mediante la **docker build** y los **ficheros Dockerfile** pero, ¿qué son y qué contienen esos ficheros declarativos que me sirven para construir mis imágenes?.

Si queremos hacer una definición más o menos formal de estos ficheros podríamos decir que:

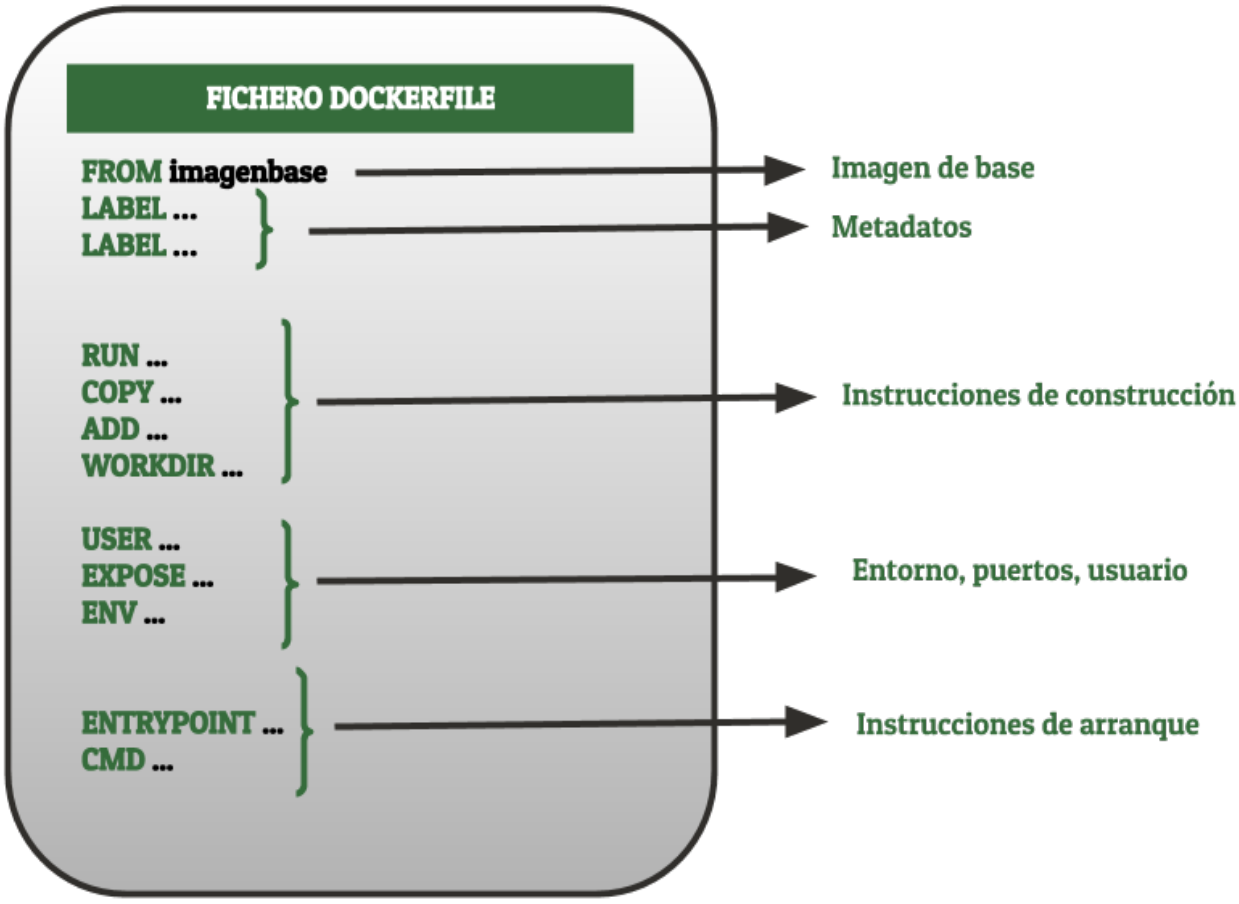
"Un fichero Dockerfile es un conjunto de instrucciones que serán ejecutadas de forma secuencial para construir una nueva imagen docker".

Cada una de estas instrucciones crea una nueva capa en la imagen que estamos creando. Si no ha habido cambios estas instrucciones son cacheadas y la capa previamente creada usada. Esto me permite reusar capas entre imágenes que están construidas de forma similar y basándose en las mismas imágenes de base. Esta **estructura en forma de capas de las imágenes** la podemos ver cuando ejecutamos la orden docker pull. Podemos apreciarlo en la siguiente imagen.

Juan Diego Pérez Jiménez. *Capas de una imagen docker* (Dominio público)

Cada una de las líneas que nos aparecen en el proceso de descarga es una de las capas que conforman la imagen. En este caso tres capas, cada una con su identificador.

¿ Y las instrucciones que puede contener el fichero Dockerfile?. Son varias que explicaremos posteriormente, pero de manera general podemos decir que dichos ficheros tienen la siguiente estructura.



Estructura de los ficheros Dockerfile. *Estructura de los ficheros Dockerfile* (Dominio público)

ORDENES FICHEROS DOCKERFILE

Las órdenes más comunes son:

- **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=iesalixar***
- **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 EN 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 EN 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 síntaxis es la siguiente: ***ENTRYPOINT <command> / ENTRYPOINT ["executable","param1","param2"].*** Ejemplo: **ENTRYPOINT ["service","apache2","start"]**
- **CMD:** Para establecer el ejecutable por defecto (salvo que se sobreescriba desde la order 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”].***

FICHERO .dockerignore

Antes de que se ejecuten las órdenes ADD y COPY de los Dockerfile el proceso de construcción de docker build mira si en el contexto de la construcción existe un fichero **.dockerignore.** El funcionamiento de este tipo de ficheros es **análogo al funcionamiento de los ficheros .gitignore** que excluyen una serie de ficheros del control de versiones.

EN ESTE CASO LOS ARCHIVOS QUE SE RECOJAN EN EL FICHERO .DOCKERIGNORE NO PASARÁN A LA IMAGEN EN EL PROCESO DE CONSTRUCCIÓN DE LA IMAGEN.

Algunos ejemplos de ficheros .dockerignore.

Un ejemplo genérico:

```
# Esa carpeta app tiene el contenido de un repositorio

#Excluyo la carpeta .git

app/.git

#Excluyo el fichero .gitignore

app/.gitignore

#Excluyo todos los archivos dentro de la carpeta log pero dejo la carpeta

app/log/*

#Excluyo todos los archivos dentro de la carpeta tmp pero dejo la carpeta.

app/tmp/*

#Excluyo el archivo README.md

app/README.md
```

Un ejemplo para una aplicación node:

```
# Esa carpeta nodeapp tiene el contenido de un repositorio

#Excluyo la carpeta .git

nodeapp/.git

#Excluyo el fichero .gitignore

nodeapp/.gitignore
```

```
#Excluyo la carpeta node_modules. Eso me obliga a hacer npm install al arrancar el contenedor

nodeapp/node_modules

#Excluyo el fichero creado por el editor de código

.vscode
```

Para más información sobre los comodines y reglas que puedo tener en un fichero .dockerignore os dejo este enlace: https://docs.docker.com/engine/reference/builder/#dockerignore-file

REFERENCIA COMPLETA FICHEROS DOCKERFILE

Podéis encontrar una referencia completa en este enlace
https://docs.docker.com/engine/reference/builder/

6.2.2 Ejemplos de ficheros Dockerfile

DOCKERFILE Proyecto Tomcat

Dockerfile que hace modificaciones en un servidor de Aplicaciones Tomcat

```
FROM tomcat:9.0.39-jdk11

# Installing basic tools
RUN apt update && apt install -y nano && apt install -y vim

# Enable manager app, host manager app and docs apss
RUN mv /usr/local/tomcat/webapps.dist/* /usr/local/tomcat/webapps
RUN rm -rf /usr/local/tomcat/webapps.dist

# Copying my tomcat-users.xml to the container
# TO-BE-DONE user password as a docker build arg
COPY mytomcat-users.xml /usr/local/tomcat/conf/tomcat-users.xml

# Context Modifying from all default apps
COPY mycontext.xml /usr/local/tomcat/webapps/host-manager/META-INF/context.xml
COPY mycontext.xml /usr/local/tomcat/webapps/manager/META-INF/context.xml

# COPY THE APP


COPY webapp.war /usr/local/tomcat/webapps/webapp.war
```

DOCKERFILE Código de un repositorio (WORDPRESS)

Ocultar

Dockerfile que descarga el código de WP actualizado y lo pone en funcionamiento en un servidor Web Apache con PHP.

```
FROM php:7.4-apache

RUN apt update && apt install -y git

WORKDIR /var/www/html

RUN git clone https://github.com/WordPress/WordPress.git .

RUN chown -R www-data:www-data /var/www/html


RUN docker-php-ext-install mysqli
```

DOCKERFILE para proyecto Django

Dockerfile para un proyecto Python - Django.

```
FROM python:3.8.7

ARG DEBUG_PORT=5687
ARG APP_PORT=8000

COPY djangoapp /app
COPY ./startapp.sh /app/

WORKDIR /app
RUN chmod +x startapp.sh
RUN ls /app
RUN pip install -r requirements.txt
RUN sed -i "s/\$1/$APP_PORT/" startapp.sh
RUN sed -i "s/\$2/$DEBUG_PORT/" startapp.sh
RUN more startapp.sh


ENTRYPOINT ["/app/startapp.sh"]
```

IMPORTANTE: SI EL DOCKERFILE NO ESPECIFICA UN CMD O UN ENTRYPOINT LAS IMÁGENES GENERADAS HEREDAN ESAS ÓRDENES DE LA IMAGEN BASE. ESTO DE ESPECIAL RELEVANCIA CUANDO USAMOS CONTENEDORES CON SERVICIOS.

Evidentemente estos ejemplos son de ficheros Dockerfiles sencillos. Desde el punto de vista del desarrollo lo que queremos es **PONER NUESTRO CÓDIGO** en un sistema que ya esté montado y que sea lo más similar posible a producción o que, en nuestro caso como profesores:

- Genere una **imagen personalizada** para que **tanto alumnos como profesor** tengan el **mismo entorno y/o el mismo código.**
- Genere una **imagen con el código de algún proyecto de los alumnos** y que de esa manera **el profesor no tenga ningún problema en hacer funcionar** dicho proyecto.

Si queremos ver ficheros Dockerfile más complejos una simple búsqueda en GitHub de este término no devolverá miles de resultados. Os recomiento los creados por la empresa andaluza Bitnami cuyo perfil en GitHub es este https://github.com/bitnami.