Introducción a Docker  
UD 04. Gestión de imágenes en Docker

short line



Autor: Sergi García Barea

Actualizado Febrero 2025

Licencia

**Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA)**: No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

[**1. Introducción 3**](#_dbh0n1vac4c8)

[**2. Listando imágenes locales y para su descarga 3**](#_vyhbfp4t666x)

[2.1 Listando imágenes locales 3](#_iol64q9i0w94)

[2.2 Listando imágenes para su descarga 3](#_603e4rcqxe8h)

[**3. Descargando y eliminando imágenes (y contenedores) locales 4**](#_6lyvzg2kpq1f)

[3.1 Descargando imágenes con “docker pull” 4](#_ht3tmouzfilx)

[3.2 Observar el historial de una imagen descargada 4](#_dh3d0i4sede3)

[3.3 Eliminando imágenes con “docker rmi” 4](#_507o0k1jlnpi)

[3.4 Eliminando contenedores con “docker rm” 5](#_ltn8bzci05fz)

[3.5 Eliminando todas las imágenes y contenedores con “docker system prune -a” 6](#_r40l00y22gob)

[**4. Creando nuestras propias imágenes a partir de un contenedor existente 6**](#_28vw6on9kext)

[**5. Exportando/importando imágenes locales a/desde ficheros 7**](#_cvqqtqbnce3e)

[**6. Subiendo nuestras propias imágenes a un repositorio (Docker Hub) 8**](#_e9sdebhk9jbe)

[6.1 Paso 1: creando repositorio para almacenar la imagen en Docker Hub 8](#_a88rehcko14m)

[6.2 Paso 2: almacenando imagen local en repositorio Docker Hub 9](#_ta5sbrcu138k)

[**7. Generar automáticamente nuestras propias imágenes mediante Dockerfile 9**](#_3l3osjhykuaf)

[7.1 Editor Visual Studio Code y plugins asociados a Docker 9](#_e9q761413tdo)

[7.2 Creando nuestro primer Dockerfile 10](#_b8fv9ikvt4n4)

[7.3 Otros comandos importantes de Dockerfile 11](#_ih1cpx4sfzoc)

[7.3.1 Comando EXPOSE 12](#_y26fzwq94x5r)

[7.3.2 Comando ADD/COPY 12](#_6meccoq5febc)

[7.3.3 Comando ENTRYPOINT 12](#_rprs4dvutxhg)

[7.3.4 Comando USER 13](#_whx5yvvioyl1)

[7.3.5 Comando WORKDIR 13](#_unir0oxaes4t)

[7.3.6 Comando ENV 13](#_88041cwfcgu6)

[7.3.7 Otros comandos útiles: ARG, VOLUME, LABEL, HEALTHCHECK 13](#_uekjdcntrv2u)

[**8. Trucos para hacer nuestras imágenes más ligeras 13**](#_xrbtq9hm5hsf)

[**9. Bibliografía 14**](#_g1qlmy5ta6mu)

UD04. Gestión de imágenes en Docker

# Introducción

Hasta ahora, hemos visto cómo descargar y trabajar con imágenes de terceros en Docker. En esta unidad explicaremos cómo gestionar las imágenes de contenedores Docker (listado, eliminación, historia, etc.) así como su creación tanto de forma manual como utilizando el comando “***docker build***” con los llamados “***Dockerfiles***”.

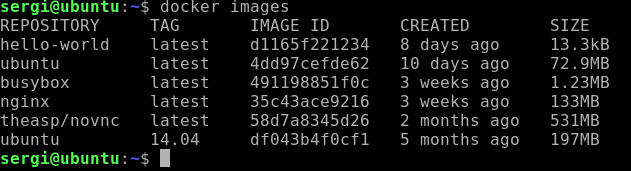
# Listando imágenes locales y para su descarga

## Listando imágenes locales

Podemos obtener información de qué imágenes tenemos almacenadas localmente usando

| docker images |
| --- |

Obteniendo un resultado similar al siguiente, donde vemos información acerca de las imágenes



Podemos utilizar filtros sencillos usando la nomenclatura ***“docker images [REPOSITORIO[:TAG]]”***.

| docker images ubuntu:14.04 |
| --- |

Nos mostrará la imagen del repositorio “ubuntu” en su versión “14.04”.



Si queremos utilizar algún filtro avanzado, podemos usar la opción “-f”. Aquí un ejemplo, filtrando las imágenes que empiecen por “u” y acabe su etiqueta en “04”.

| docker images -f=reference="u\*:\*04" |
| --- |



❕ **Atención:**  no confundir este comando con ***“docker image”*** (sin la s final).

Más información en <https://docs.docker.com/engine/reference/commandline/images/>

## Listando imágenes para su descarga

Podemos obtener información de imágenes que podemos descargar en el registro (por defecto, Docker Hub) utilizando el comando “docker search”. Por ejemplo, con el siguiente comando:

| docker search ubuntu |
| --- |

Nos aparecen aquellas imágenes disponibles en el registro (Docker Hub) con esa palabra.

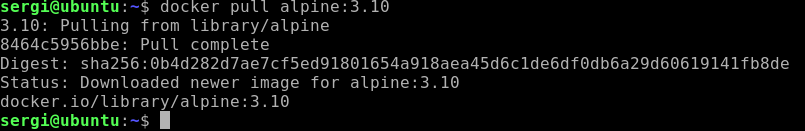
# Descargando y eliminando imágenes (y contenedores) locales

## Descargando imágenes con “docker pull”

Podemos almacenar imágenes localmente desde el registro sin necesidad de crear un contenedor mediante el comando ***“docker pull”***, claramente inspirado en sistemas de control de versiones como ***“git”***. Para conocer sus nombres y versiones, podemos usar el comando “docker search” explicado anteriormente o visitar <https://hub.docker.com/>.

| docker pull alpine:3.10 |
| --- |

Este comando nos descarga la imagen ***“alpine”*** con el tag ***“3.10”***, como vemos aquí:

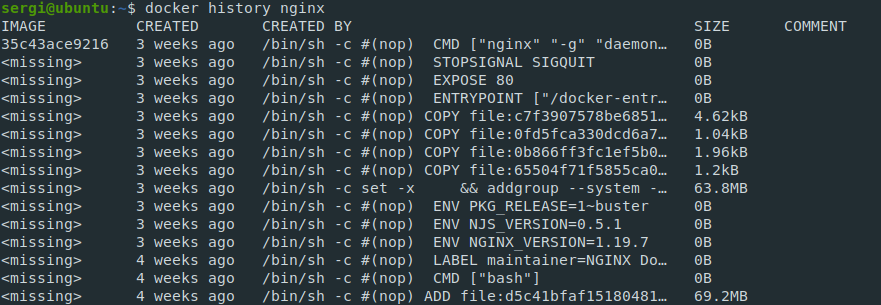


## Observar el historial de una imagen descargada

Podéis observar el historial de una imagen descargada, es decir, en qué versiones se basa, usando el comando ***“docker history”.*** Por ejemplo con:

| docker history nginx |
| --- |

Obtenemos lo siguiente:

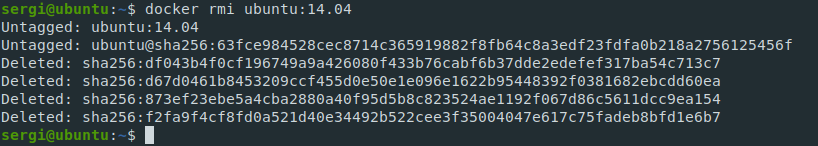


## Eliminando imágenes con “docker rmi”

Con el comando ***“docker rmi”*** podemos eliminar imágenes almacenadas localmente.

| docker rmi ubuntu:14.04 |
| --- |

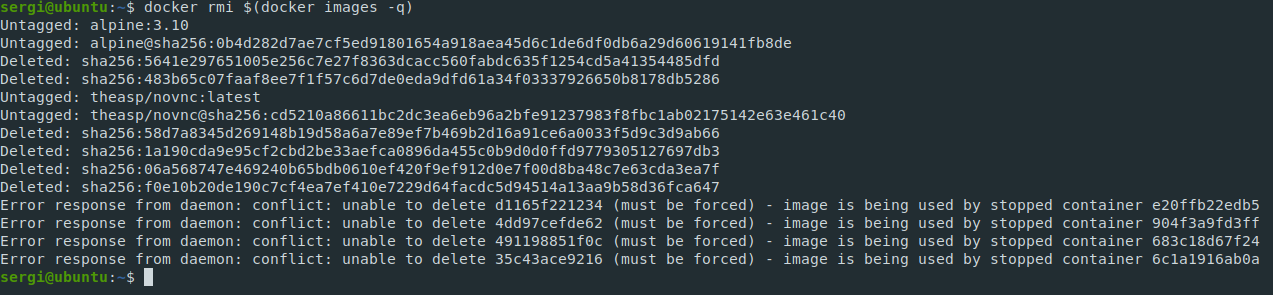
Elimina la imagen ubuntu con la etiqueta 14.04



Una forma de eliminar **todas** las imágenes locales, que no estén siendo usadas por un contenedor, combinando ***“docker images -q”*** para obtener la lista y ***“docker rmi”*** es la siguiente:

| docker rmi $(docker images -q) |
| --- |

Aquí se observa el borrado, excepto de aquellas usadas por un contenedor:



## Eliminando contenedores con “docker rm”

Aprovechando que tratamos el borrado de imágenes, comentamos cómo borrar contenedores parados (si un contenedor está en marcha, debe ser parado antes del borrado).

Con la siguiente orden se puede borrar un contenedor por identificador o nombre

| docker rm IDENTIFICADOR/NOMBRE |
| --- |

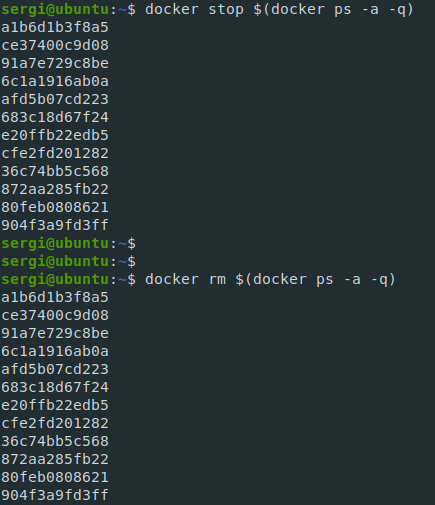
Asimismo, una forma de borrar todos los contenedores (que estén parados), de forma similar a como vimos en el anterior punto, es la siguiente:

**Paso 1 (opcional):** paramos todos los contenedores:

| docker stop $(docker ps -a -q) |
| --- |

**Paso 2: borramos todos los contenedores:**

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



## Eliminando todas las imágenes y contenedores con “docker system prune -a”

Una forma de realizar las operaciones anteriores de golpe, es usando “*docker system prune -a*”, que elimina toda imagen y contenedor parado.

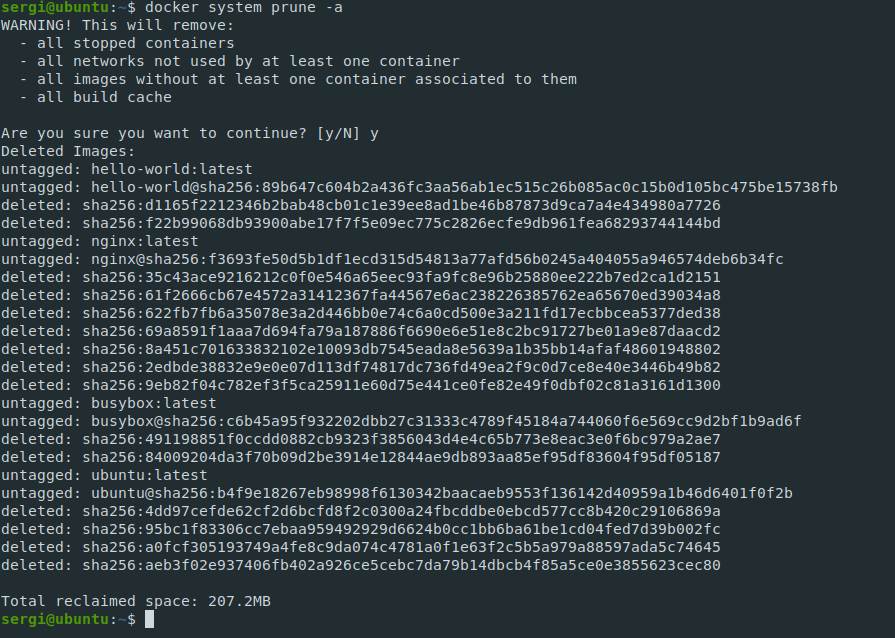
**Paso 1 (opcional):** paramos todos los contenedores:

| docker stop $(docker ps -a -q) |
| --- |

**Paso 2: borramos todos los contenedores:**

| docker system prune -a |
| --- |

Obteniendo algo similar a esto:



# Creando nuestras propias imágenes a partir de un contenedor existente

El sistema de imágenes de Docker funciona como un control de versiones por capas, de forma similar a la herramienta ***“git”*** para control de versiones. Podemos entender que un contenedor es como una “capa temporal” de una imagen, por lo cual, podemos hacer un “***commit***” y convertir esa “capa temporal” en una imagen. La sintaxis más habitual es la siguiente

| docker commit -a "autor" -m "comentario" ID/NOMBRE-CONTENEDOR usuario/imagen:[version] |
| --- |

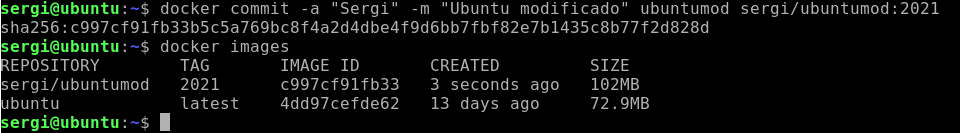
Por ejemplo, si tenemos un contenedor con nombre “***ubuntumod***” que simplemente es un contenedor basado en la imagen “***ubuntu***” en el que se ha instalado un programa y hacemos:

| docker commit -a "Sergi" -m "Ubuntu modificado" IDCONTENEDOR sergi/ubuntumod:2021 |
| --- |

y tras ello, comprobamos las imágenes con

| docker images |
| --- |

observamos lo siguiente:



Hemos obtenido lo siguiente: una nueva imagen, con nombre “***sergi/ubuntumod***” con tag “***2021***”, donde “sergi” actúa como nombre de usuario para usarlo en un repositorio remoto (recordamos nuevamente, que por defecto es “Docker Hub”).

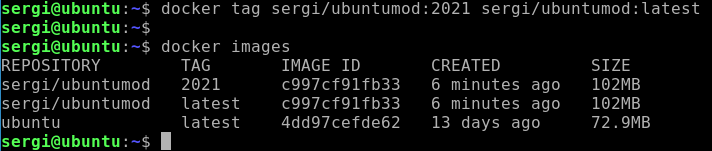
Ahora ya podríamos crear nuevos contenedores con esa imagen, usando por ejemplo:

| docker run -it sergi/ubuntumod:2021 |
| --- |

Si quisiéramos añadir una nueva etiqueta a la imagen, como “latest”, podemos usar el comando “***docker tag***”, teniendo en cuenta que una misma imagen puede tener varias etiquetas:

| docker tag sergi/ubuntumod:2021 sergi/ubuntumod:latest |
| --- |

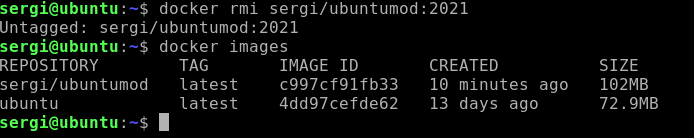
Obtendremos algo similar a:



Para eliminar una etiqueta, simplemente deberemos borrar la imagen con “***docker rmi***”. La imagen se mantendrá mientras al menos tenga una etiqueta. Por ejemplo con:

| docker rmi sergi/ubuntumod:2021 |
| --- |

quedaría así:



Más información de los comandos en:

* Docker commit <https://docs.docker.com/engine/reference/commandline/commit/>
* Docker tag <https://docs.docker.com/engine/reference/commandline/tag/>

# Exportando/importando imágenes locales a/desde ficheros

Una vez tengamos una imagen local en nuestro sistema, podemos hacer una copia de la misma, ya sea como copia de seguridad o como forma de transportarla a otros sistemas mediante el comando “***docker save”***. Por ejemplo, se puede hacer de estas dos formas:

| docker save -o copiaSeguridad.tar sergi/ubuntumod |
| --- |

o de forma alternativa

| docker save sergi/ubuntumod > copiaSeguridad.tar |
| --- |

Si queremos importar el fichero para crear una imagen en nuestra máquina, podemos usar “***docker load***”. Por ejemplo, se puede hacer de estas dos formas:

| docker load -i copiaSeguridad.tar |
| --- |

o de forma alternativa

| docker load < copiaSeguridad.tar |
| --- |

Más información sobre los comandos:

* Docker save: <https://docs.docker.com/engine/reference/commandline/save/>
* Docker load: <https://docs.docker.com/engine/reference/commandline/load/>

# Subiendo nuestras propias imágenes a un repositorio (Docker Hub)

Podemos subir una imagen a un repositorio (por defecto Docker Hub). Para ello, debemos realizar los siguientes pasos:

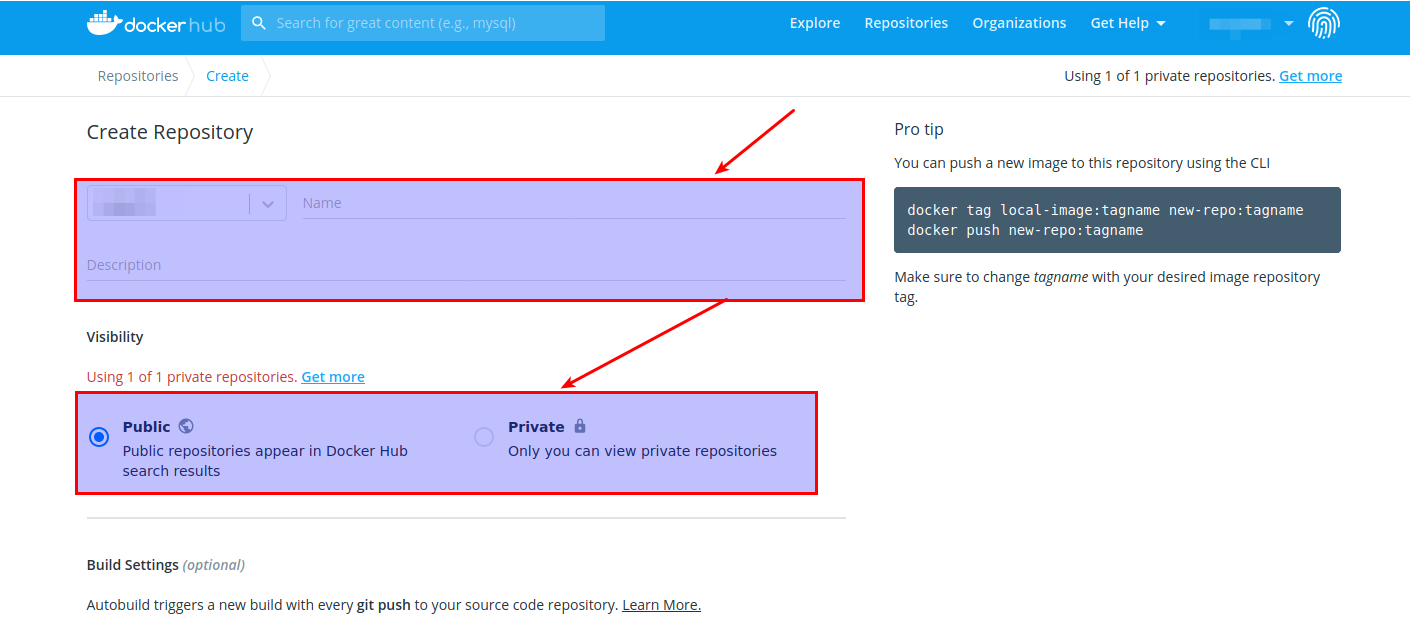
## Paso 1: creando repositorio para almacenar la imagen en Docker Hub

En primer lugar, debéis crearos una cuenta en <https://hub.docker.com> e iniciar sesión. Una vez iniciada sesión, debéis acceder a “***Repositories***” y ahí a “***Create repository***” de forma similar a como se ve en la imagen siguiente:



Tras ello, podréis quedar un repositorio con vuestra cuenta y elegir si dicho repositorio es público (cualquiera puede acceder) o privado (solo puede acceder dueño o autorizados).

La pantalla de creación del repositorio tiene un aspecto similar a este:



Una vez creado, si tu usuario es “***sergi***” y la imagen se llama “***prueba”***, podremos referenciarla en distintos contextos como “***sergi/prueba***”

## Paso 2: almacenando imagen local en repositorio Docker Hub

En primer lugar, deberemos iniciar sesión mediante consola al repositorio mediante el comando

| docker login |
| --- |

Una vez iniciada sesión, debemos hacer un “commit” local de la imagen, siguiendo la estructura vista en puntos anteriores. Un ejemplo podría ser:

| docker commit -a "Sergi" -m "Ubuntu modificado" IDCONTENEDOR sergi/prueba |
| --- |

Hecho este “commit” local, debemos subirlo usando “docker push”

| docker push sergi/prueba |
| --- |

Una vez hecho eso, si la imagen es pública (o privada con permisos), cualquiera podrá descargarla y crear contenedores usando ***“docker pull”*** o ***“docker run”***.

Más información de los comandos:

* Docker login <https://docs.docker.com/engine/reference/commandline/login/>
* Docker push <https://docs.docker.com/engine/reference/commandline/push/>

# Generar automáticamente nuestras propias imágenes mediante Dockerfile

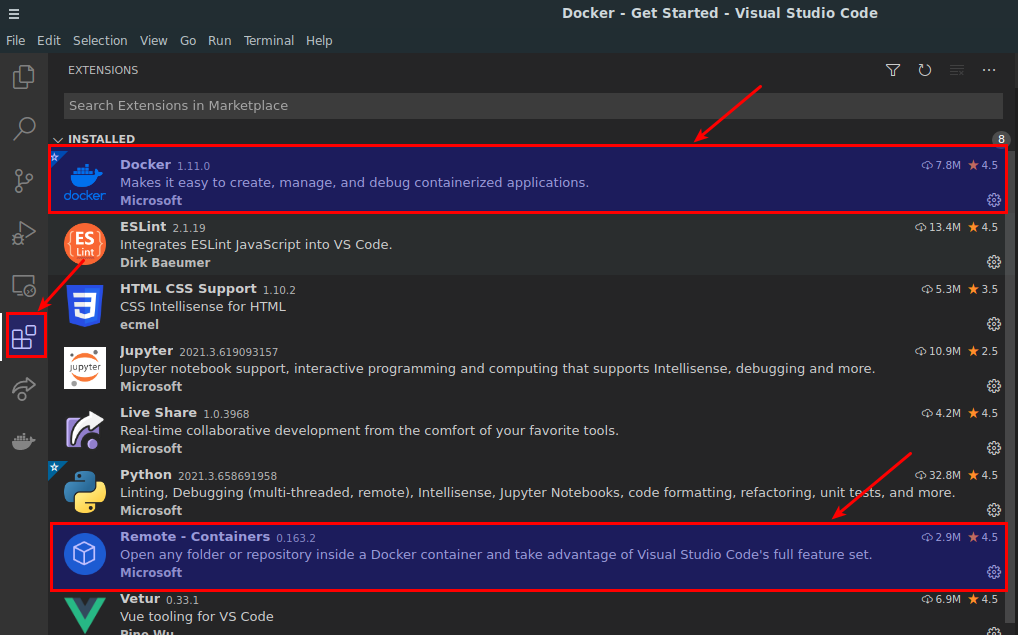
Docker nos permite generar de forma automática nuestras propias imágenes usando “***docker build***” y los llamados “***Dockerfile”***.

## Editor Visual Studio Code y plugins asociados a Docker

Los ficheros “***Dockerfile***” pueden crearse con cualquier editor de texto, pero desde aquí recomendamos el editor multiplataforma “***Visual Studio Code***” <https://code.visualstudio.com/>

Para saber más sobre cómo usar este editor podéis usar <https://code.visualstudio.com/learn>

Al instalarlo, si detecta Docker instalado en el sistema, el propio editor nos sugerirá una serie de plugins. Merece la pena instalarlos. Si no, siempre podéis buscar en plugins manualmente. Yo personalmente, os recomiendo estos dos que podéis ver en la imagen:



## Creando nuestro primer Dockerfile

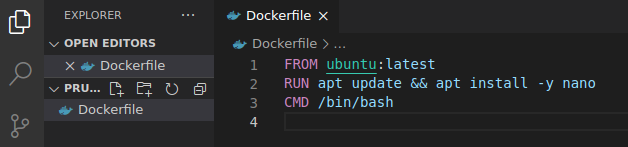
Empezaremos creando un sencillo ***“Dockerfile”*** donde crearemos una imagen de Ubuntu con el editor de texto “nano” instalado. Para ello indicaremos:

* De qué imagen base partiremos.
* Qué comandos lanzaremos sobre la imagen base, para crear la nueva imagen
* Qué comando se asociará por defecto al lanzar un contenedor con la nueva imagen

Creamos el fichero “***Dockerfile***” (Visual Studio Code le pondrá un icono de la ballena) y añadimos:

| **FROM** ubuntu:latest **RUN** apt update && apt install -y nano *# Aquí un comentario* **CMD** /bin/bash |
| --- |

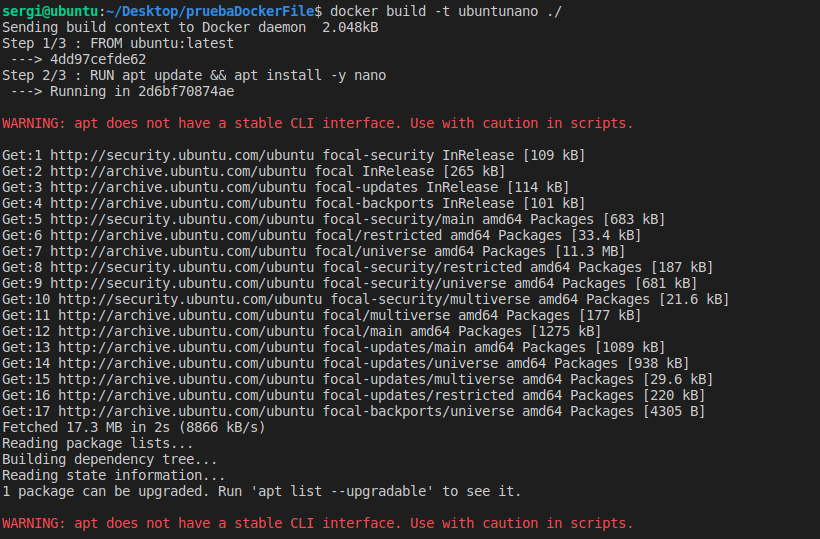
En el editor quedará de una forma similar a:



Si ahora usamos el comando “*docker build*” de la siguiente forma:

| docker build -t ubuntunano ./ |
| --- |

Obtendremos algo similar a



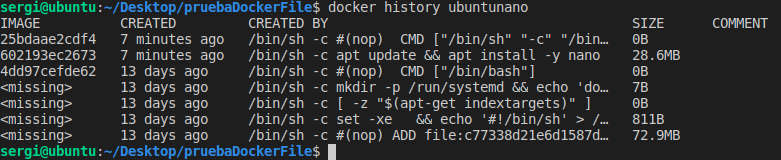
Lo que hemos hecho es “ejecutar” lo que marca él “***Dockerfile***”. El resultado se ha guardado en una nueva imagen local cuyo nombre hemos especificado con la opción ***“-t”***. El lugar donde se encontraba el “***Dockerfile***” se ha indicado mediante “***./***” (directorio actual).

❕ **Atención:** el fichero debe llamarse exactamente “***Dockerfile***”, respetando mayúsculas y minúsculas.

Si queréis especificar un nombre de fichero distinto a buscar en el directorio, puede usarse la opción “-f”, como en este ejemplo:

| docker build -t ubuntunano -f Dockerfile2 ./ |
| --- |

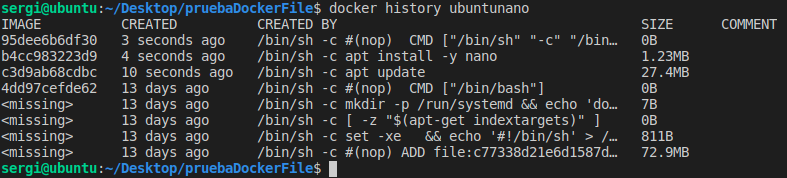
Podemos observar la historia de la imagen que hemos creado con “***docker history***”



Donde observamos, que nuestra imagen ha crecido en “28.6MB” al hacer “***apt update && apt install -y nano***”. Aunque en el anterior “Dockerfile” hemos usado un solo RUN, podríamos haber utilizado varios RUN en lugar de uno, escrito de una forma secuencial, como vemos en:

| **FROM** ubuntu:latest **RUN** apt update  **RUN** apt install -y nano **CMD** /bin/bash |
| --- |

Aquí, simplemente, habría más capas intermedias, como se observa en “docker history” si generamos la imagen con la secuencia anterior



Los comandos vistos (FROM, RUN y CMD) admiten distintos formatos. Para saber más podemos visitar su ayuda: <https://docs.docker.com/engine/reference/builder/>

## Otros comandos importantes de Dockerfile

Al crear un “Dockerfile” hay multitud de comandos. A continuación explicamos los comandos más importantes a utilizar:

### Comando EXPOSE

En primer lugar, **repasamos la diferencia entre exponer y publicar puertos en Docker**:

* Si no se expone ni publica un puerto, este solo es accesible desde el interior del contenedor.
* Exponer un puerto, indica que ese puerto es accesible tanto dentro del propio contenedor como por otros contenedores, pero no desde fuera (incluido el anfitrión).
* Publicar un puerto, indica que el puerto es accesible desde fuera del contenedor, por lo cual debe asociarse a un puerto del anfitrión.

La opción EXPOSE nos permite indicar los puertos por defecto expuestos que tendrá un contenedor basado en esta imagen. Es similar a la opción “***--expose***” de “***docker run***” (y de paso, recordamos que “docker run” con “***-p***” los publica). Por ejemplo, para exponer los puertos 80, 443 y 8080 indicaremos:

| **EXPOSE** 80 443 8080 |
| --- |

### Comando ADD/COPY

**ADD** y **COPY** son comandos para copiar ficheros de la máquina anfitriona al nuevo contenedor. Se recomienda usar **COPY**, excepto que queramos descomprimir un “zip”, que **ADD** permite su descompresión. Más información sobre la diferencia entre **ADD** y **COPY:**

<https://nickjanetakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerile>

Ejemplo de uso de **ADD**:

| **ADD** ./mifichero.zip /var/www/html |
| --- |

Descomprimirá el contenido de “***mifichero.zip***” en el directorio destino de la nueva imagen.

Ejemplo de uso de **COPY**:

| **COPY** ./mifichero.zip /var/www/html |
| --- |

O incluso accediendo desde la web.

| **COPY** http://miservidor.commifichero.zip /var/www/html |
| --- |

En este caso, copiará el fichero “***mifichero.zip***” en el directorio destino de la nueva imágen.

### Comando ENTRYPOINT

Por defecto, los contenedores Docker están configurados para que ejecuten los comandos que se lancen mediante **“*/bin/sh -c*”**. Dicho de otra forma, los comandos que lanzábamos, eran parámetros para **“*/bin/sh -c*”**. Podemos cambiar qué comando se usa para esto con **ENTRYPOINT**. Por ejemplo:

| **ENTRYPOINT** ["cat"] **CMD** ["/etc/passwd"] |
| --- |

Indicaremos que los comandos sean lanzados con “***cat***”. Al lanzar el comando “***/etc/passwd***”, realmente lo que haremos es que se lanzará “***cat /etc/passwd***”.

### Comando USER

Por defecto, todos los comandos lanzados en la creación de la imagen se ejecutan con el usuario root (usuario con UID=0). Para poder cambiar esto, podemos usar el comando **USER**, indicando el nombre de usuario o UID con el que queremos que se ejecute el comando. Por ejemplo:

| **USER** sergi **CMD** id |
| --- |

Mostrará con el comando “id” el uid y otra información del usuario “sergi”.

### Comando WORKDIR

Cada vez que expresamos el comando **WORKDIR**, estamos cambiando el directorio de la imagen donde ejecutamos los comandos. ***Si este directorio no existe, se crea.*** Por ejemplo:

| **WORKDIR** /root **CMD** mkdir tmp **WORKDIR** /var/www/html **CMD** mkdir tmp |
| --- |

Creará la carpeta “***tmp***” tanto en “***/root***” como en “***/var/www/html***”. Si los directorios “***/root***” o “***/var/www/html***” no hubieran existido, los hubiera creado.

### Comando ENV

El comando ENV nos permite definir variables de entorno por defecto en la imagen.

| **ENV v1=”valor1” v2=”valor2”** |
| --- |

Esto definirá las variables de entorno “v1” y “v2” con los valores “valor1” y “valor2”.

### Otros comandos útiles: ARG, VOLUME, LABEL, HEALTHCHECK

Aquí comentamos comandos útiles:

* **ARG**: permite enviar parámetros al propio “***Dockerfile***” con la opción “***--build-arg***” del comando “***docker build***”.
* **VOLUME**: permite establecer volúmenes por defecto en la imágen. Hablaremos de los volúmenes más adelante en el curso.
* **LABEL**: permite establecer metadatos dentro de la imagen mediante etiquetas. Uno de los casos más típicos, sustituyendo al comando MAINTAINER, que esta “deprecated” es:
  + **LABEL** maintainer="sergi.profesor@gmail.com"
* **HEALTHCHECK**: permite definir cómo se comprobará si ese contenedor está funcionando correctamente o no. Útil para sistemas orquestadores como “***Docker swarn”***, aunque otros como “***Kubernetes***” incorporan su propio sistema

# Trucos para hacer nuestras imágenes más ligeras

Al crear imágenes, es habitual aumentar el tamaño de las imágenes base. Algunos consejos para dentro de lo posible, aumentar el tamaño lo menos posible:

* Usar imágenes base ligeras, tipo “Alpine”.
* No instalar programas innecesarios, incluso evitando herramientas tipo Vim, Nano, etc.
* Minimizar capas en “Dockerfile”
  + Mejor usar “***RUN apt-get install -y <packageA> <packageB>”*** que usar ***“RUN apt-get install -y <packageA>***” y tras ello “***RUN apt-get install -y <packageB>***”
* Utiliza comandos de limpieza tras instalaciones con “***apt***”, tales como:
  + “***rm -rf /var/lib/apt/lists/\****” tras crear una imagen para borrar las listas generadas al realizar “apt update”.
  + “***apt-get purge --auto-remove && apt-get clean***” para eliminar temporales de apt.
* Al usar “**apt install**” usar la opción “***no-install-recommend***”, para que no instale paquetes recomendados asociados al paquete instalado.
* Analiza tus “Dockerfile” con <https://www.fromlatest.io/#/> y sigue sus consejos.

Más información en:

* <https://hackernoon.com/tips-to-reduce-docker-image-sizes-876095da3b34>
* <https://medium.com/sciforce/strategies-of-docker-images-optimization-2ca9cc5719b6>

# Bibliografía

[1] Docker Docs <https://docs.docker.com/>

[2] Visual Studio Code Learn <https://code.visualstudio.com/learn>

[3] FROM:latest <https://www.fromlatest.io/#/>