

Docker: Introducción y guía rápida

J. Miguel-Alonso, 2022

Ejemplo 1

Ponemos un contenedor en ejecución basado en la imagen "alpine". Le asignamos un nombre, "alpine_orig_cont". Modificamos un fichero en el contenedor. Creamos una imagen "alpine_new" a partir del contenedor "alpine_orig_cont". Vemos la historia de la imagen. Paramos y eliminamos el contenedor "alpine_orig_cont". Creamos un nuevo contenedor a partir de la imagen "alpine_new". Paramos y eliminamos el contenedor.

```
$ docker run -it --name=alpine_orig_cont alpine /bin/sh
# hacer alguna modificación (crear un fichero) y
# salir con Ctrl-PQ, que deja el contenedor en ejecución
/ # touch hello
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
3de4f9e5bb43   alpine    "/bin/sh"               5 minutes ago   Up 5 minutes           alpine_orig_cont
$ docker commit alpine_orig_cont alpine_new
sha256:2cf018e1ed517bf2206e57d5e3c05957d970eb1c0b56f9e152dd3e11da8d4f55
$ docker image ls
REPOSITORY      TAG         IMAGE ID          CREATED          SIZE
alpine new      latest     2cf018e1ed51     4 minutes ago   5.54MB
...
$ docker history alpine_new
IMAGE          CREATED          CREATED BY          SIZE      COMMENT
2cf018e1ed51   About a minute ago   /bin/sh             12B
9c6f07244728   2 months ago       /bin/sh -c #(nop)  CMD ["/bin/sh"]     0B
<missing>      2 months ago       /bin/sh -c #(nop)  ADD file:2a949686d9886ac7c... 5.54MB
$ docker stop alpine_orig_cont
alpine_orig_cont
$ docker rm alpine_orig_cont
alpine_orig_cont
$ docker run -it alpine_new /bin/sh
/ # ls
bin      etc      home    media   opt      root    sbin     sys      usr
dev      hello   lib     mnt     proc     run     srv      tmp      var
# salir con Ctrl-D, que termina el shell y también el contenedor
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
28dfc79d7184   alpine_new      "/bin/sh"               About a minute ago   Exited (0)
42 seconds ago   blissful_noyce
$ docker rm blissful_noyce
blissful_noyce
```

Ejemplo 2

Creamos un contenedor con "docker run" y vemos su estado con "docker ps". Listamos las imágenes disponibles localmente con "docker image ls".

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:18a657d0cc1c7d0678a3f9bea8b7eb4918bba25968d3e1b0adebfa71caddbc346
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
1018ccdba944   hello-world  "/hello"               About a minute ago   Exited (0) About a minute ago   vibrant_lumiere
$ docker image ls
REPOSITORY      TAG         IMAGE ID          CREATED          SIZE
alpine new      latest     2cf018e1ed51     19 minutes ago   5.54MB
hello-world     latest     feb5d9fea6a5     12 months ago   13.3kB
```

Nótese que el contenedor "vibrant_lumiere" basado en la imagen "hello-world:latest" ha sido creado y ejecutado,

pero su ejecución ha terminado inmediatamente. Por eso no lo vemos con "docker ps". El contenedor, sin embargo, queda en estado "Exited (0)" (ha terminado). Para verlo tenemos que usar la opción "-a" del comando para listar los contenedores: "docker ps -a".

Ejemplo 3

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS          PORTS          NAMES
3afe9180839b   hello-world    "/hello"                4 minutes ago  Exited (0) 4 minutes ago          clever_haslett
$ docker start -i 3afe9180839b
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
$ docker rm clever_haslett
clever_haslett
```

Se lanza un contenedor a partir de la imagen "hello-world", que ya está disponible localmente -- por eso no se descarga desde Docker Hub. Docker asigna al contenedor el nombre "clever_haslett" y el ID "3afe9180839b". Al iniciarse, el contenedor ejecuta el comando por omisión, en este caso "/hello", cuyo resultado se vuelca en pantalla. El comando termina tras escribir un mensaje, por lo que el contenedor queda en estado "Exited (0)", lo que se puede ver con "docker ps -a". El "(0)" tras "Exited" es un código de terminación, que normalmente significa "terminación sin errores" -- los valores distintos de cero pueden indicar diferentes errores.

Podemos volver a ejecutarlo con "docker start -i". Lógicamente, enseguida vuelve a quedar en estado "Exited". Lo eliminamos con "docker rm" usando su nombre, aunque podríamos haber usado su ID.

Ejemplo 4

Lanzamos un contenedor basado en la imagen "hello-world" usando la opción "--rm" de "docker run". El contenedor termina tras escribir su mensaje, y queda automáticamente borrado.

```
$ docker run --rm hello-world

Hello from Docker!
...
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS          PORTS          NAMES
```

Lanzamos un contenedor con una distribución Linux "alpine" y nombre "a1". El comportamiento de este contenedor es ejecutar cierto comando ("/bin/sh", se puede ver con "docker ps -a") que, al no estar asociado a un terminal, acaba inmediatamente. El contenedor queda en estado "Exited". Lo borramos manualmente.

```
$ docker run --name=a1 alpine
$ docker rm a1
```

Lanzamos un contenedor "a2" basado en alpine, pero esta vez asociamos al mismo un terminal interactivo. Pasamos a estar en un shell dentro del contenedor, y podemos ejecutar comandos. Al pulsar Ctrl-D el shell del contenedor termina y volvemos al host. El contenedor quedaría en estado "Exited", pero al haberlo lanzado con "--rm", se elimina inmediatamente.

```
$ docker run --rm --name a2 -it alpine
/ # ls
bin    etc    lib    mnt    proc  run    srv    tmp    var
dev    home  media  opt    root  sbin   sys    usr
/ # Ctrl-D
$
```

Lanzamos un contenedor a3 basado en alpine y lo dejamos como demonio, ejecutando un shell interactivo (que no termina, pero al que no estamos conectados). Podemos acceder a ese shell con "docker attach". Si salimos del shell de a3 con Ctrl-D (o escribiendo "exit"), termina el shell y por lo tanto el contenedor. Si antes de terminar el shell

escribimos la secuencia de escape Ctrl-PQ abandonamos el contenedor, que queda como demonio, y regresamos al shell del host.

```
$ docker run --rm --name a3 -itd alpine
d4965f0ec6157ed830486d3d6edac26a9c962ac1c5ccb43d8614f9e414c0ae24
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
d4965f0ec615   alpine    "/bin/sh" 6 seconds ago Up 5 seconds a3

$ docker attach a3
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home  media opt    root   sbin   sys    usr
/ # read escape sequence Ctrl-PQ
```

En este ejemplo "docker exec" se comporta de forma similar a "docker attach", pero no es lo mismo. En este caso ejecutamos un nuevo shell en el contenedor activo a3. Al terminar con "exit" termina ese nuevo shell, pero no el que se estaba ejecutando desde el lanzamiento. El último comando lanza un trabajo no interactivo, "ls -l" que se ejecuta y termina. El contenedor a3 sigue activo, en estado "Up".

```
$ docker exec -it a3 /bin/sh
/ # exit
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
d4965f0ec615   alpine    "/bin/sh" 9 minutes ago Up 9 minutes a3
$ docker exec a3 ls -l
total 56
drwxr-xr-x  2 root    root      4096 Aug  9 08:47 bin
drwxr-xr-x  5 root    root      360 Oct 14 09:40 dev
...
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
d4965f0ec615   alpine    "/bin/sh" 9 minutes ago Up 9 minutes a3
```

Ejemplo 5

Creamos un contenedor a4 y comprobamos que está en estado "Created" (pero no "Up"). Lo ponemos en marcha, cambiando a estado "Up". Asociamos el terminal del host a a4. Salimos de a4 con Ctrl-PQ. Lo ponemos en pausa. Lo hacemos continuar. Lo matamos y comprobamos que ha terminado con un código de error. Lo borramos.

```
$ docker create --name=a4 -it alpine /bin/sh
8e1a978ed743f300db780818872ba7a866c9eabf57afe684cfc5b6b812c48594
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
8e1a978ed743   alpine    "/bin/sh" 12 seconds ago Created              a4
$ docker start a4
a4
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
8e1a978ed743   alpine    "/bin/sh" 34 seconds ago Up 5 seconds a4
$ docker attach a4
/ # ... type some commands ...
/ # read escape sequence Ctrl-PQ
$ docker pause a4
a4
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS             PORTS   NAMES
8e1a978ed743   alpine    "/bin/sh" 12 seconds ago Up 5 seconds (Paused) a4
$ docker unpause a4
a4
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
8e1a978ed743   alpine    "/bin/sh" About a minute ago Up About a minute a4
$ docker kill a4
a4
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
```

```
8e1a978ed743  alpine  "/bin/sh"  2 minutes ago  Exited (137) 8 seconds ago  a4
$ docker rm a4
a4
```

Ejemplo 6

Ejecutamos el contenedor e5 basado en alpine, dejándolo como demonio. Lo inspeccionamos y vemos alguna de su información (estado, asociación de volúmenes, dirección IP...). Vemos que, con respecto a la imagen original, e5 ha cambiado algo: se ha añadido el directorio "/hostdir". Por último vemos algunas estadísticas de uso de CPU y memoria.

```
$ docker run --name=a5 -itd -v $(pwd):/hostdir alpine
59b1c56ce4ddb16029a816e1a33d5dc5937d21d33a6e991acf15ae2c13cfb1dd
$ docker inspect a5
[
  {
    "Id": "59b1c56ce4ddb16029a816e1a33d5dc5937d21d33a6e991acf15ae2c13cfb1dd",
    "Created": "2022-10-14T10:51:02.875625448Z",
    "Path": "/bin/sh",
    "Args": [],
    "State": {
      "Status": "running",
      ...
    "HostConfig": {
      "Binds": [
        "/home/ubuntu:/hostdir"
      ],
      ...
      "PortBindings": {},
      ...
      "IPAddress": "172.17.0.2",
      ...
    }
  }
]
$ docker diff a5
A /hostdir
$ docker stats --no-stream
CONTAINER ID   NAME    CPU %     MEM USAGE / LIMIT   MEM %     NET I/O       BLOCK I/O      PIDS
59b1c56ce4dd   a5      0.00%    380KiB / 3.832GiB   0.01%    3.27kB / 0B   0B / 0B       1
```

Ejemplo 7

Creamos un contenedor e6 basado en alpine. Lo renombramos como e8. Comprobamos que está en ejecución, estado "Up". Asociamos nuestro terminal al contenedor en ejecución. Vemos los directorios que tiene, y creamos uno nuevo, "usuario". En dicho directorio creamos dos ficheros de texto, "f1.txt" y "f2.txt". Volvemos al host con Ctrl-PQ, dejando el contenedor en ejecución. Creamos un fichero de texto "f3.txt" en el host. Lo copiamos al directorio "usuario" de e8. Luego creamos un directorio "us_cont" en el host y copiamos en el mismo todo el contenido del directorio "usuario" de e8. Vemos que en el directorio copiado están tanto los ficheros que se crearon en el contenedor como el "f3.txt" que se copió desde el host.

Tras ello, volcamos el sistema de ficheros completo de e8 en un archivo "cont_e8.tar".

```
$ docker run -itd --name=e6 alpine
67d5c0a7fa8bf47e7601cad8ad831de8e41b70eedd14b66bfc3740faf724fd95
$ docker rename e6 e8
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
67d5c0a7fa8b   alpine    "/bin/sh"  50 seconds ago  Up 49 seconds           e8
$ docker attach e8
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home  media  opt    root   sbin   sys    usr
/ # mkdir usuario
/ # echo "blablabla" > /usuario/f1.txt
/ # echo "blebleble" > /usuario/f2.txt
/ # read escape sequence
$ echo "ggggggg" > f3.txt
```

```
$ docker cp ./f3.txt e8:/usuario
$ mkdir us_cont
$ docker cp e8:/usuario ./us_cont
$ ls us_cont/
usuario
$ ls us_cont/usuario/
f1.txt f2.txt f3.txt
$ docker export e8 > cont_e8.tar
$ ls
cont_e8.tar ...
```

Ejemplo 8

Creamos la red bridge "interna". Creamos un contenedor "alp" basado en alpine. Creamos otro contenedor "nginx" basado en nginx, mapeando su puerto interno 80 con el puerto 8080 del host. Comprobamos que los contenedores están creados y los mapeos son correctos. Verificamos que, efectivamente, nginx responde a accesos web en el puerto 8080 del host (tanto de localhost como de su dirección IP "pública").

Nos conectamos a la consola del contenedor "alp" y vemos que puede acceder por nombre al contenedor "nginx". Tras ello paramos los contenedores, los borramos y eliminamos la red "interna".

```
$ docker network create interna
9092eca5e86e1a2a07cac8feadecla414060c904d142361328138ab8044fea9f
$ docker run -itd --name alp --network interna alpine
89334efb1c3a6df3949539fc5fe455bda8b7759c6749f1984c358195bd6a6574
$ docker run -itd --name nginx --network interna -p 8080:80 nginx
c9f1ed72a6f7cfeee0fd58460eb04a12b0e78f3e23402a1533cf094a0e3a67fa
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS
NAMES
c9f1ed72a6f7   nginx     "/docker-entrypoint...." About a minute ago Up About a minute 0.0.0.0:8080->80/tcp, :::8080->80/tcp   nginx
89334efb1c3a   alpine    "/bin/sh"                About a minute ago Up About a minute
alp
$ wget http://127.0.0.1:8080
--2022-10-17 05:29:57-- http://127.0.0.1:8080/
Connecting to 127.0.0.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 615 [text/html]
Saving to: 'index.html.1'
index.html.1      100%[=====>]          615  --.-KB/s    in 0s
2022-10-17 05:29:57 (69.2 MB/s) - 'index.html.1' saved [615/615]
$ docker attach alp
/ # ping -c 3 nginx
PING nginx (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.119 ms
64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.328 ms
64 bytes from 172.19.0.3: seq=2 ttl=64 time=0.331 ms
--- nginx ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.119/0.259/0.331 ms
/ # read escape sequence
$ docker stop alp nginx
$ docker rm $(docker ps --filter status=exited -q)
$ docker network rm interna
```

Por cada contenedor lanzado en una red "bridge", en el host se crea una interfaz de red virtual asociada a dicho contenedor.

Ejemplo 9

Partiendo de un sistema sin contenedores en ejecución vemos tres interfaces: loopback, la que da acceso a Internet al host (en este caso, "enp0s3" y la virtual que ofrece comunicación interna con los contenedores ("docker0"). Tras crear un contenedor, aparece una interfaz virtual nueva (veth6d0f404@if22) asociada al contenedor. Esa última interfaz no se usa directamente -- solo a través de su contenedor.

```
$ ip -br address # Sin contenedores
lo                UNKNOWN      127.0.0.1/8  ::1/128
enp0s3            UP              10.0.2.15/24 fe80::840d:eb3e:c158:af1c/64
docker0           DOWN          172.17.0.1/16 fe80::42:9bff:fe67:8e39/64
$ docker run -itd alpine
30464be3a264d5de51ebfebcc47e7b8095a34a5fbd128c0f23ed8d337bb4aea4
$ ip -br address
lo                UNKNOWN      127.0.0.1/8  ::1/128
enp0s3            UP              10.0.2.15/24 fe80::840d:eb3e:c158:af1c/64
docker0           UP              172.17.0.1/16 fe80::42:9bff:fe67:8e39/64
veth6d0f404@if22  UP              fe80::8c69:a6ff:fe78:6663/64
```

Ejemplo 10

Ejecutamos "docker run --rm -d --network host --name my_nginx nginx", lo que lanza un contenedor que ejecuta el servidor nginx. Dicho servidor se asociará al puerto 80 del host – si está disponible; en caso contrario obtendríamos un error porque esa asociación no es posible ya que hay algún otro proceso ocupando el puerto.

```
$ docker run --rm -d --network host --name my_nginx nginx
b67cd8b206858b78fd6df03e637fb385f8ab6c58b79452660e6db2cc073d348b
$ ip -br address
lo                UNKNOWN      127.0.0.1/8  ::1/128
enp0s3            UP              10.0.2.15/24 fe80::840d:eb3e:c158:af1c/64
docker0           DOWN          172.17.0.1/16 fe80::42:9bff:fe67:8e39/64
$ sudo netstat -tulpn | grep :80
tcp        0      0 0.0.0.0:80          0.0.0.0:*          LISTEN      7534/nginx: master
tcp6       0      0 :::80              :::*                LISTEN      7534/nginx: master
$ docker stop my_nginx # no hace falta borrarlo por la opción --rm
my_nginx
```

Ejemplo 11

Creamos un contenedor a9 (alpine) y asociamos el directorio "\$HOME/us_cont" del host al directorio "/cont" de a9. Nos conectamos al contenedor y vemos que existe el directorio "/cont", y tiene contenidos. Salimos al host. Vemos que no hay volúmenes creados. Creamos uno llamado "shared". Lanzamos un contenedor e10 (alpine), asociando el volumen "shared" a su directorio "/cont". Nos conectamos a e10 y vemos que el directorio "/cont" existe, pero está vacío. Volvemos al host y usamos "docker cp" para copiar un sencillo fichero de texto en el directorio "/cont" de e10. Comprobamos que está ahí. Desde el host, intentamos eliminar el volumen y no podemos porque está en uso.

```
$ docker run -itd --name a9 -v $HOME/us_cont:/cont alpine
025fd3653fdb54cc6f0773325c1c7d995a2f6b9bb86a5083cd7092aa3ab67661
$ docker attach a9
/ # ls cont
usuario
/ # ls cont/usuario
f1.txt f2.txt f3.txt
/ # read escape sequence
$ docker volume ls
DRIVER      VOLUME NAME
$ docker volume create shared
shared
$ docker volume ls
DRIVER      VOLUME NAME
local       shared
$ docker run -itd --name a10 -v shared:/cont alpine
62fa1487e4debf5a63640a9b6cb921539264265f1dc9a268076e5fc168b6417a
$ docker attach a10
/ # ls
bin    dev    home   media  opt     root   sbin   sys    usr
cont  etc    lib    mnt     proc    run    srv    tmp    var
/ # ls cont
/ # read escape sequence
$ echo "aaaaa" > f10.txt
$ docker cp ./f10.txt a10:/cont
$ docker attach a10
```

```
/ # cat /cont/f10.txt
aaaaa
/ # read escape sequence
$ docker volume rm shared
Error response from daemon: remove shared: volume is in use -
[62fa1487e4debf5a63640a9b6cb921539264265f1dc9a268076e5fc168b6417a]
```

Ejemplo 12

La primera ejecución de "docker run hello-world" descargó de Docker Hub la imagen "hello-world:latest", que quedó almacenada localmente. La eliminamos con "docker rmi" (o "docker image rm"). Una nueva ejecución de un contenedor basado en "hello-world" ("docker run") hace que se descargue otra vez dicha imagen. La inspeccionamos con "docker image inspect". Usamos "docker image ls" (o "docker images") para obtener un listado de imágenes locales – entre las que está "hello-world".

Tras ello hacemos un backup de la imagen "hello-world" en el archivo "my-hello.tar" con "docker save". Eliminamos la imagen. La recuperamos con "docker load". Nótese que la imagen recuperada sigue manteniendo las etiquetas "hello-world:latest", no toma el nombre del fichero de entrada.

```
$ docker rmi hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:e18f0a777aefabe047a671ab3ec3eed05414477c951ab1a6f352a06974245fe7
$ docker run --name hello --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Digest: sha256:e18f0a777aefabe047a671ab3ec3eed05414477c951ab1a6f352a06974245fe7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
...

$ docker image inspect hello-world
[
  {
    "Id": "sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412",
    "RepoTags": [
      "hello-world:latest"
    ]
  }
]
...

$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-hello             latest              8dba4ea69dd1       19 minutes ago     13.3kB
nginx                latest              76c69feac34e       2 weeks ago        142MB
...

$ docker save -o my-hello.tar hello-world
$ docker image rm hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:e18f0a777aefabe047a671ab3ec3eed05414477c951ab1a6f352a06974245fe7
$ docker load -i my-hello.tar
Loaded image: hello-world:latest
```

Ejemplo 13

Ejecutamos un contenedor "calc" basado en la imagen "alpine". Su función es escribir una suma y terminar. Creamos con "docker commit" una imagen "calc_img" a partir de ese contenedor. Vemos que funciona: podemos crear un contenedor a partir de la nueva imagen. Lo hacemos. Comprobamos que localmente tenemos la imagen original ("alpine") y la nueva ("calc_img"). Además, tenemos en ejecución dos contenedores, uno llamado "calc" basado en "alpine", y otro llamado "bold_sammet" basado en "calc_img"). Intentamos eliminar la imagen "calc_img" sin éxito, porque está en uso por "bold_sammet".

Volcamos el contenido completo del sistema de ficheros del contenedor "bold_sammet" en un archivo .tar con "docker export". Importamos con "docker import" dicho archivo y el resultado es una nueva imagen almacenada localmente, pero sin nombre ni etiqueta. Utilizamos "docker tag" para asignárselos.

```
$ docker run --name calc alpine echo "$((40+2))"
42
$ docker commit calc calc_img
sha256:43fa84b4d305674d95deb09d5b0e6ab33036b6320ae0903b57d1025486dd1e90
$ docker run calc_img
42
$ docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
calc_img            latest       36093b3c4088      40 seconds ago    5.54MB
alpine              latest       9c6f07244728      2 months ago      5.54MB
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND          CREATED           STATUS              PORTS          NAMES
a2cccflb3bc9   calc_img       "echo 42"        4 minutes ago     Exited (0) 4 minutes ago
06168ce1903b   alpine         "echo 42"        6 minutes ago     Exited (0) 6 minutes ago
...
$ docker image rm calc_img
Error response from daemon: conflict: unable to remove repository reference "calc_img" (must force) - container a2cccflb3bc9 is using its referenced image 36093b3c4088

$ docker export bold_sammet > calc.tar
$ docker import calc.tar
sha256:ba4d9bf74fa5a25e4520756ab69adffe1227aed8ad28d66ab675ecc9709caad9
$ docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
<none>              <none>       ba4d9bf74fa5      7 seconds ago     5.54MB
calc_img            latest       36093b3c4088      5 minutes ago     5.54MB
alpine              latest       9c6f07244728      3 months ago      5.54MB
...
$ docker tag ba4d9bf74fa5 my_new_calc:v1.0
$ docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
my_new_calc         v1.0         ba4d9bf74fa5      10 minutes ago    5.54MB
calc_img            latest       36093b3c4088      16 minutes ago    5.54MB
alpine              latest       9c6f07244728      3 months ago      5.54MB
...
```

Ya tenemos una nueva imagen "my_new_calc:v1.0". Si intentamos ejecutar un contenedor basado en "my_new_calc" obtendremos un error, porque estará buscando la etiqueta ":latest" y no existe. Tendremos más éxito usando el nombre completo de la imagen, con su etiqueta. Aun así, el contenedor no se puede lanzar correctamente: no tiene asociado un comando a ejecutar en el momento del lanzamiento. Podemos comprobarlo usando "docker inspect" y buscando en el resultado "Cmd:". Veremos que aparece "null". Eso no pasaba con la imagen original, "calc_img", que tiene ""Cmd": ["echo", "42"]". Sin embargo se pueden crear contenedores a partir de la nueva imagen especificando el comando que tienen que ejecutar.

```
$ docker run my_new_calc
Unable to find image 'my_new_calc:latest' locally
docker: Error response from daemon: pull access denied for my_new_calc, repository does not exist or may require 'docker login': denied: requested access to the resource is denied.
See 'docker run --help'.
$ docker run my_new_calc:v1.0
docker: Error response from daemon: No command specified.
See 'docker run --help'.
$ docker run my_new_calc:v1.0 ls
bin
dev
etc
...
```

Como se ve, las imágenes creadas a partir de un .tar obtenido vía "docker export" están bastante incompletas. Solo es una instantánea del sistema de ficheros. Una imagen "normal" tiene además etiquetas, comandos por omisión, variables de entorno y otras propiedades.

Ejemplo 14

Nos identificamos ante Docker Hub como usuario "dhid". Tenemos una imagen local, "calc_img:latest" generada en un ejemplo anterior. Queremos que vaya al repositorio privado "dhid/ipmd" de Docker Hub ("docker.io/dhid/ipmd"). Para ello tenemos que cambiar el nombre "repositorio:etiqueta" a la mencionada imagen. Lo hacemos con "docker tag". Tras ello podemos subir la imagen al registro.

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over
to https://hub.docker.com to create one.
Username: dhid
Password:
WARNING! Your password will be stored unencrypted in /home/osboxes/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
$ docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
my_new_calc         v1.0        ba4d9bf74fa5     10 minutes ago   5.54MB
calc_img            latest      36093b3c4088     16 minutes ago   5.54MB
alpine              latest      9c6f07244728     3 months ago     5.54MB
...
$ docker tag calc_img dhid/ipmd:latest
$ docker push dhid/ipmd
Using default tag: latest
The push refers to repository [docker.io/dhid/ipmd]
994393dc58e7: Mounted from library/alpine
latest: digest: sha256:deb4540708cccba9f1363e4c181faabaf5f9160e94268b3b9461a1ecff0b85f2 size: 528
```

Ejemplo 15

Nota: en <https://github.com/acpmialj/introdocker> están los ficheros necesarios para este ejemplo.

El objetivo de este ejemplo es crear una imagen que ejecute un programa basado en Python y Flask. Flask permite escribir de forma sencilla programas Python que se comportan como servidores basados en el protocolo HTTP. Nuestro programa usará el servicio Redis (una base de datos en memoria) para almacenar parejas clave-valor.

Para la construcción necesitaremos una imagen de base que ya tenga instalado python, instalar en ella las librerías Python para Flask y Redis, y añadir el programa que nos interesa.

Asumimos que el Dockerfile, y todos los ficheros adicionales necesarios, están en un directorio "\$HOME/flask", que es nuestro directorio-proyecto actual. Empezamos por la aplicación Python/Flask que se ejecutará en el contenedor, llamada "app.py":

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}</h3>" \
          "<b>Hostname:</b> {hostname}<br/>" \
          "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)
```

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Para que esta aplicación funcione, es necesario tener una serie de librerías instaladas. Indicamos las dependencias en un fichero de texto "requirements.txt":

```
Flask
Redis
```

Ya tenemos los elementos necesarios. Tenemos que escribir el "Dockerfile" para la construcción de la imagen:

```
# Partimos de una base oficial de Python 2.7
FROM python:2.7-slim

# Cuando se inicie el contenedor, este será su directorio de trabajo
WORKDIR /app

# Copiamos todos los archivos de nuestro proyecto al directorio /app del contenedor:
# Se copian los ficheros locales del host app.py y requirements.txt
COPY . /app

# Ejecutamos pip para instalar las dependencias en el contenedor
# Dichas dependencias están en "requirements.txt"
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Indicamos que este contenedor se comunica por el puerto 80/tcp
EXPOSE 80

# Declaramos una variable de entorno
ENV NAME World

# Cuando se arranque el contenedor, por omisión ejecutará el comando siguiente
CMD ["python", "app.py"]
```

Para la construcción partimos de una imagen previa ("python:2.7-slim") del registro de Docker Hub, copiamos en ella los ficheros del directorio actual ("."). El fichero "requirements.txt" se usa para instalar los paquetes python necesarios. El fichero "app.py" forma parte de la línea CMD, que indica el programa a ejecutar cuando se lance un contenedor. Se usa además ENV para declarar la variable de entorno NAME, y EXPOSE para declarar que nuestro programa va a usar el puerto 80.

Todo listo. Nos aseguramos de que todo lo necesario está en "\$HOME/flask", y que este es nuestro directorio-proyecto de trabajo. Podemos ejecutar "docker build ." preparar la imagen. Si añadimos la opción "-t" asignaremos un nombre a dicha imagen. Si no se indica etiqueta, se usa ":latest". En este ejemplo la imagen se llamará "myflask". La secuencia completa queda así:

```
$ mkdir flask
$ cd flask
$ nano Dockerfile
$ nano requirements.txt
$ nano app.py
$ ls
app.py Dockerfile requirements.txt
$ docker build -t myflask .
Sending build context to Docker daemon 5.12kB
Step 1/7 : FROM python:2.7-slim
2.7-slim: Pulling from library/python
123275d6e508: Pull complete
...
Collecting Flask
  Downloading Flask-1.1.4-py2.py3-none-any.whl (94 kB)
Collecting Redis
  Downloading redis-3.5.3-py2.py3-none-any.whl (72 kB)
...
Successfully installed Flask-1.1.4 Jinja2-2.11.3 MarkupSafe-1.1.1 Redis-3.5.3 Werkzeug-1.0.1 click-7.1.2 itsdangerous-1.1.0
...
```

```
Successfully built da163f72de52
Successfully tagged myflask:latest
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myflask	latest	da163f72de52	22 seconds ago	159MB

Al lanzar un contenedor basado en la nueva imagen, ejecutará lo que se indicó en la línea CMD del Dockerfile. El resultado es que el terminal queda ligado al contenedor, mostrando las salidas generadas por la aplicación flask "app.py".

```
$ docker run --rm -p 4000:80 myflask
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Comprobamos que la aplicación funciona. Desde un navegador nos conectamos a <http://localhost:4000> y el resultado es:

```
Hello World!
Hostname: dda7a083b450
Visits: cannot connect to Redis, counter disabled
```

El terminal mostrará:

```
172.17.0.1 - - [30/Nov/2022 12:07:31] "GET / HTTP/1.1" 200 -
```

Pulsando Ctrl-C se saldrá de la aplicación y del contenedor, que será borrado inmediatamente. Nótese que el acceso al sitio web ha funcionado (nuestra aplicación Python/Flask ha generado contenido), pero se indica que hay un error de acceso a Redis. Es así porque la aplicación está esperando conectarse a un servicio Redis (en el puerto por omisión, 6379), y ese servicio no está en marcha. Vamos a "arreglarlo" de manera no óptima. Primero destruimos todos los contenedores relacionados con Flask. Partimos de que la imagen myflask se ha creado correctamente.

Los pasos a dar son

1. Creamos una red "bridge" llamada "interna"
2. Ejecutamos una instancia de Redis, con el nombre "redis" conectada a la red "interna"
3. Ejecutamos una instancia de myflask conectada a la red "interna". Cuando se lance el contenedor, ejecutará el programa "app.py" que se conectará al servidor con nombre "redis" (el lanzado en el paso anterior).

```
$ docker network create interna
$ docker run --rm --network interna -d --name redis redis
$ docker run --rm -p 4000:80 --network interna -d myflask
```

Comprobamos que ahora el resultado de acceder a <http://localhost:4000> es distinto:

```
Hello World!
Hostname: 8632602aa4b9
Visits: 7
```

Hacemos limpieza con "docker stop" para parar los contenedores (no hace falta "docker rm" por la opción "--rm" al lanzarlos). Eliminamos la red con "docker network rm interna".