Docker: Introducción y guía rápida

Instalación

Docker es un entorno de manejo de contenedores, que contiene todo lo necesario para

1. Trabajar con imágenes
2. Trabajar con contenedores basados en imágenes

Usaremos Docker por línea de comandos, desde un entorno Ubuntu. Para los ejemplos en este documento, se ha usado como anfitrión un equipo con Ubuntu 2.04, 64 bits. En Mac, se puede conseguir un funcionamiento similar usando MacOS. En Windows, se puede instalar Ubuntu sobre el WSL (Windows Subsystem for Linux). Las instrucciones detalladas para instalar Docker engine están en https://docs.docker.com/engine/install/ubuntu/). Se recomienda usar el método de instalación a partir del repositorio apt de Docker.

Instalación de Docker Engine

Eliminar paquetes viejos (puede que no los haya)

```
for pkg in docker.io docker-doc docker-compose podman-docker containerd runc; do sudo apt-get remove $pkg; done
```

Instalar usando el repositorio apt de Docker. Lo primero es configurar el repositorio

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Y luego se instalan los paquetes de Docker

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Comprobamos que todo funciona:

```
sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:4f53e2564790c8e7856ec08e384732aa38dc43c52f02952483e3f003afbf23db
Status: Downloaded newer image for hello-world:latest
…
```

Pasos adicionales

El uso de los comandos de Docker para trabajar con imágenes en una máquina anfitriona (*host*), y ejecutar contenedores requiere ser un usuario (en el host) con privilegios de administrador, por lo que todos los comandos hay que ejecutarlos con "sudo docker …". Para evitarlo, es suficiente con incluir nuestro nombre de usuario en el grupo "docker". Ejecutamos estos comandos:

```
$ sudo groupadd docker # Crea el grupo, si no existe
$ sudo usermod -aG docker $USER
```

Tras estos pasos podemos ejecutar "newgrp docker" para que el efecto sea inmediato en nuestro terminal activo. Sin embargo, si queremos que la configuración sea totalmente efectiva, es necesario reiniciar la máquina host.

Notas de instalación: para las demostraciones, se ha usado como anfitrión un equipo virtual con Ubuntu 2.04, 64 bits, ejecutándose sobre Virtualbox 6.1. Se ha descargado la imagen de osboxes.org, un disco virtual ".vdi". Se ha creado una MV nueva, indicando cuál es el disco virtual (el descargado). Se han instalado las "guest additions" y, desde el directorio del CD virtual, se ha ejecutado "autorun.sh" (sin "sudo"). Después se han modificado los ajustes para usar un teclado con disposición de letras en español, eliminando la disposición "English (US)", se ha habilitado el portapapeles bidireccional y se ha desactivado el bloqueo automático de pantalla.

Principios básicos y definiciones

Una **imagen** es un fichero con todo lo necesario para ejecutar una aplicación o servicio: sistema operativo (SO), librerías, código de aplicación, configuraciones, etc. Una imagen va asociada a un SO (puesto que se usa el kernel del SO de la máquina anfitriona) y a una arquitectura de procesador (por la misma razón). Las imágenes consumen espacio de almacenamiento en disco, pero no otros recursos.

Un **contenedor** es una imagen que ha sido iniciada, es decir, una aplicación en ejecución. Los contenedores consumen recursos (CPU, memoria), acceden a ficheros y se comunican vía red. Nótese que es posible (y muy sencillo) generar varios contenedores a partir de una misma imagen.

Una imagen se identifica por un nombre cualificado o por el ID (hash) que Docker le asigna ("c85146bafb83"). El nombre puede ser una única palabra ("nginx") si se refiere a una imagen oficial de Docker almacenada en su **registro** oficial (**Docker Hub**[1]). Puede tener varias partes separadas por "/", por ejemplo "bitnami/mysql" si se identifica al repositorio de un determinado autor en Docker Hub. Puede tener una etiqueta tras el separador ":" ("nginx:1.15.5-alpine"). Y puede ser más largo si se refiere a una imagen no almacenada en el registro oficial ("mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine"). La parte anterior a ":" la denominamos "repositorio", y la posterior "etiqueta".

Es común usar etiquetas del estilo *nombre_base:versión* o *nombre_base:modificación*. Por ejemplo, "alpine:3.16.2" se refiere a una imagen de una distribución de Linux de pequeño tamaño, en su versión 3.16.2. Cuando no se especifica etiqueta, Docker asume que se usa la etiqueta por omisión ":latest". Por ejemplo, el nombre "ibmcom/iop-hadoop" se refiere a la última versión de una imagen del repositorio "ibmcom/iop-hadoop" que está en el registro Docker Hub, es decir, equivale a "ibmcom/iop-hadoop:latest".

[1] Docker Hub tiene una webUI en https://hub.docker.com, que resulta práctica para la búsqueda de imágenes. Sin embargo, el registro en sí es "docker.io".

Un contenedor se identifica por un nombre, auto-generado por Docker o especificado al lanzarlo (por ejemplo, "web-server-11"), y también por un hash (ID) creado por Docker (por ejemplo, "80f1bc1e7feb").

Internamente una imagen está formada por una serie de **capas**, que se forman a partir de una imagen de base o capa inicial. Esta imagen tiene todo lo necesario para la ejecución de un contenedor basado en ella, incluyendo un sistema de ficheros. Cada paso en la construcción de la imagen (ver más adelante las instrucciones para crear imágenes a partir de **Dockerfiles**) supone modificaciones sobre la anterior, que se guardan de forma incremental – es decir, solo se guardan las diferencias con la capa previa. Y así sucesivamente hasta llegar a la imagen definitiva. Por lo tanto, una imagen la podemos ver como una pila de capas.

En Docker Hub podemos ver fácilmente las capas asociadas a una imagen. Por línea de comandos,

docker history *imagen*

nos muestra las capas de la imagen indicada.

Si a partir de una imagen se crea un contenedor, a partir de un contenedor se puede crear una imagen para su uso posterior en nuevos contenedores. El contenedor habrá hecho cambios sobre lo que contenía la imagen usada al lanzarlo – lo que significa que habrá añadido otra capa. Con

docker commit *contenedor nombre_nueva_imagen*

creamos una imagen a partir de una instantánea del contenedor indicado, que estará en ejecución o temporalmente parado. Eso guarda una imagen, visible con

docker image ls

La imagen está almacenada localmente. Son necesarios unos pasos extra para guardarla en un repositorio externo. *Dicho todo esto, la forma recomendada de crear una nueva imagen es a partir de una de base y un* ***Dockerfile***.

Ejemplo 1

El propósito principal de este ejemplo es mostrar cómo ejecutar un contenedor a partir de una imagen ("docker run") y como crear una imagen a partir de un contenedor ("docker commit). Estos comandos, y otros que aparecen en el ejemplo, se verán en detalle más adelante.

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
sha256:2cf018e1ed517bf2206e57d5e3c05957d970eb1c0b56fce152dd3e11da8d4f55
$ 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
```

Nótese que la imagen que hemos creado con "docker commit" tiene como nombre completo "alpine_new:latest". Si lo partimos, la parte repositorio es "alpine_new" y la etiqueta es ":latest".

Operaciones sobre contenedores

Operaciones básicas

Recordemos que un contenedor es una imagen en ejecución. La forma más sencilla de crearlo es

docker run *imagen*

Por ejemplo, "docker un hello-world". Si no tenemos localmente una imagen "hello-world", se descarga automáticamente de Docker Hub. Se crea el contenedor, asignándole un nombre y un ID aleatorios. El contenedor ejecutará una serie de acciones predefinidas y puede terminar o quedar en ejecución. El contenedor queda ligado al terminal, que no se libera hasta que termine. Una vez terminado, el contenedor queda detenido, pero no se destruye. La imagen descargada tampoco se destruye, queda almacenada localmente para futuro uso.

Podemos usar

docker ps

para ver los contenedores en ejecución. Con "docker ps -a" vemos también los que están parados porque han terminado su ejecución.

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: sha256:18a657d0cc1c7d0678a3fbea8b7eb4918bba25968d3e1b0adebfa71caddbc346
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
$ docker ps -a
```

```
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".

docker rename *contenedor*

cambia el nombre de un contenedor.

docker stop *contenedor*

detiene un contenedor en ejecución.

docker rm contenedor

Elimina el contenedor indicado, que no puede estar en ejecución: o ha parado solo, o lo hemos parado con "docker stop". Una vez eliminado, ya no aparece con "docker ps -a". Si hay muchos contenedores a borrar, se pueden usar "docker rm" en combinación con "docker ps". Por ejemplo, este comando borra todos los contenedores en estado "exited":

docker rm $(docker ps --filter status=exited -q)

Veamos algunas opciones adicionales de "docker run", el comando básico para lanzar contenedores.

docker run *imagen comando*

Pone en marcha un contenedor a partir de *imagen*, ejecuta el *comando* en el mismo, muestra el resultado y termina (cuando el comando termine). Las imágenes pueden tener asociado un comando inicial, predeterminado, que es el que se ejecuta con "docker run *imagen*". Si se indica un comando, sustituye al predeterminado[2]. Indicar un comando asegura que se ejecuta al iniciar el contenedor.

docker run -it *imagen comando*

Crea un contenedor y ejecuta el comando de manera interactiva: todo lo que escribamos a partir de ese momento será procesado por el contenedor. El terminal queda ligado al contenedor. Para volver al host será necesario terminar el comando, o pulsar **Ctrl-PQ**[3].

docker run -d *imagen comando*

Ejecuta el contenedor (y, en el mismo, el comando indicado) como demonio (se desliga del terminal y, normalmente, no se para). Se puede ver con "docker ps".

[2] Con matices, lo veremos más adelante.
[3] Ctrl-PQ (con la tecla Ctrl presionada, pulsar P y seguido Q) es una secuencia de escape que permite desligar el terminal de un contenedor en ejecución, pasando el control al shell del host.

docker exec *contenedor comando*

Ejecuta un comando en un contenedor *que ya está en funcionamiento*. Se pueden usar las opciones "-d" y "-it". Nótese que si el contenedor estaba haciendo algo, ejecutará adicionalmente el comando indicado.

docker run --name=*nombre imagen comando*

Da nombre al contenedor en el momento de crearlo. Si no se usa esta opción, Docker asigna un nombre aleatorio al contenedor.

docker run --rm *imagen comando*

Borra el contenedor en cuanto termina su ejecución. Así no será necesario ejecutar "docker rm".

docker run -v *hostdir*:*containerdir imagen comando*

Al crear el contenedor mapea el directorio *hostdir* del host en un directorio *containerdir* del contenedor. Es una forma muy útil de compartir una carpeta entre host y contenedor.

docker run -p *hostport*:*containerport imagen comando*

Mapea un puerto del contenedor en un puerto del host: todo el tráfico dirigido al puerto *hostport* del host es redirigido al puerto *containerport* del contenedor. Con la opción "-P" se publican todos los puertos necesarios sobre puertos aleatorios del host. Se puede ver el mapeo efectivo con "docker ps".

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" (veremos este comando más adelante). 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 contendor 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 Crlt-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
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

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
$ docker attach a3
/ # exit
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Operaciones adicionales sobre contenedores

docker create *imagen*

Crea un contenedor, pero no lo arranca (no se ve con "docker ps" pero sí con "docker ps -a"). Admite muchas de las opciones de "docker run". El contenedor queda en estado "Created".

docker start *contenedor*

Arranca un contenedor creado con "docker create", o parado (con "docker stop" o simplemente porque ha terminado su ejecución).

docker stop *contenedor*

Detiene un contenedor en ejecución (pasa a estado "Exited"). Lo hace enviándole la señal SIGTERM.

docker restart *contenedor*

Reinicia el contenedor.

docker pause *contenedor*

Pone un contenedor en pausa (pasa a estado "Up (Paused)").

docker unpause *contenedor*

Reanuda la ejecución de un contenedor pausado.

docker wait *contenedor*

Bloquea el terminal del host hasta que el contenedor indicado se detenga.

docker kill *contenedor*

Envía una señal SIGKILL al contenedor en ejecución. Normalmente el contenedor termina (pasa a estado "Exited" con un código de error (por ejemplo, 137). En condiciones normales, un contenedor que termina devolvería un código 0. Nótese que "docker stop" es similar, pero la señal enviada es SIGTERM.

docker attach contenedor

Asocia el terminal del host al contenedor en ejecución, siempre que no esté parado y esté ejecutando un comando que permita interacción (por ejemplo, "/bin/sh" o "/bin/bash").

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
```


Obtención de información sobre contenedores

docker ps *opciones*

Emite una lista, con varios campos, con información sobre los contenedores en ejecución, estado "Up". La opción "-a" hace que se listen los que están en otros estados. La opción "-q" emite únicamente una lista de IDs. La opción "-f" permite especificar filtros que seleccionan los contenedores que se muestran (por nombre, código de estado, etiqueta…).

docker logs *contenedor*

Permite ver los *logs* de un contenedor (lo volcado por STDOUT y STDERR). Si el contenedor envía los logs a otro sitio (un servidor, unos ficheros) hay que buscar alternativas para acceder a los mismos. Por ejemplo, mapeando directorios.

docker inspect *contenedor*

Muestra toda la información sobre un contenedor, en formato JSON: variables de entorno, comandos que se ejecutan al arrancar, volúmenes, mapeos de puertos, direcciones IP…

docker port *contenedor*

Muestra los mapeos de puertos del contenedor con los del host. Mostrará parejas del estilo "7890/tcp -> 0.0.0.0:4321". La parte izquierda corresponde al contenedor, la derecha al host.

docker stats *contenedor*

Muestra estadísticas de uso de recursos. Por omisión este comando captura el terminal hasta pulsar Ctrl-C, mostrando información en tiempo real. Para evitarlo, usar "docker stats --no-stream".

docker diff *contenedor*

Muestra ficheros cambiados en el sistema de ficheros del contenedor.

Ejemplo 6

Ejecutamos el contenedor a5 basado en alpine, dejándolo como demonio. Se usa un flag "-v" para asociar el directorio actual del equipo anfitrión ("$(pwd)") a un directorio del contenedor ("/hostdir")[4]. Esto no formaba parte de la imagen, por lo que e5 difiere de un contenedor generado a partir de "alpine" sin flags. Inspeccionamos a5 y vemos alguna de su información (estado, asociación de volúmenes, dirección IP…). Vemos que, con respecto a la imagen original, a5 tiene algo distinto: 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
```

Importar/exportar datos

El comando "docker cp" copia ficheros o directorios entre un contenedor y el sistema de ficheros local. Nótese *no se está compartiendo un directorio entre host y contenedor*: se están copiando ficheros. El comando es "docker cp *origen destino*". El contenedor puede estar parado o en ejecución.

docker cp *./fichero_o_directorio_host contenedor:/fichero_o_directorio_cont*

[4] Veremos más adelante más detalles sobre cómo compartir ficheros y directorios entre el anfitrión y los contenedores.

Copia el fichero o directorio indicado del host en el fichero o directorio indicado del contenedor. Se puede dar la vuelta a los argumentos, extrayendo ficheros del contenedor. Por ejemplo:

docker cp *contenedor*:/var/logs /tmp/app_logs

copia el contenido del directorio "/var/logs" del contenedor en el directorio "/tmp/app_logs" del host. Si el último argumento es un guión "-" en vez de un *pathname* en el host, la salida es un flujo .tar por STDOUT.

docker export *contenedor*

Vuelca el sistema de ficheros completo del contenedor vía STDOUT (con la opción "-o" se puede especificar un fichero de salida). El formato es .tar. Este fichero se puede usar después con "docker import". Lo veremos más adelante.

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 Ctrl-PQ
$ 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 ...
$ tar tvf cont_e8.tar
-rwxr-xr-x 0/0               0 2023-09-19 09:00 .dockerenv
drwxr-xr-x 0/0               0 2023-08-07 09:09 bin/
lrwxrwxrwx 0/0               0 2023-08-07 09:09 bin/arch -> /bin/busybox
…
```

No olvidemos hacer un poco de limpieza, eliminando ficheros y directorios temporales ("f3.txt", "us_cont", "cont_e8.tar"…). También deberíamos eliminar el contenedor e8.

Redes

Por omisión, cada contenedor tiene una dirección IP interna (no pública), una máscara y una dirección de *gateway* (pasarela por omisión). Todos esos datos se pueden ver con "docker inspect". Con la ayuda de los servicios de Docker, el contenedor puede salir a Internet usando NAT, compartiendo la conexión del host. Si se crean varios contenedores, todos están en la misma subred, interconectados sin restricciones, y se pueden comunicar entre ellos utilizando sus direcciones IP. El host también tiene una interfaz (normalmente llamada "docker0") conectada a dicha red.

docker network inspect bridge

Muestra información sobre una red de contenedores, en este caso la red por omisión llamada "bridge" (pero puede ser otra). El manual de Docker recomienda usar preferiblemente redes creadas manualmente.

Una red creada explícitamente por el usuario tiene algunas ventajas sobre la red por omisión. Al crear una red se especifica el "driver" a emplear. Las características de la red dependen de dicho driver. Si no se indica ninguno, se usa el driver "bridge".

docker network create *red_usuario*

Crea una red *red_usuario* con el driver "bridge". Opera dentro de un único host. Los contenedores que se asocien a dicha red pueden comunicarse entre ellos usando sus direcciones IP y también (esto es importante) sus nombres.

docker network rm *red*

Elimina la red indicada. Primero hay que desconectar todos los contenedores conectados a ella.

docker network ls

Vuelca un listado de redes.

docker network inspect *red*

Vuelca, en formato JSON, información detallada sobre una red, incluyendo la lista de contenedores asociados a ella.

docker network connect *red contenedor*

Conecta un contenedor a una red. Se puede hacer al lanzarlo: "docker run --network=*red*". La red tiene que estar creada previamente.

docker network disconnect *red contenedor*

Desconecta un contenedor de la red indicada.

Los contenedores pueden salir a Internet vía NAT, pero no son accesibles desde fuera (este es el comportamiento habitual de NAT). Recordemos que *se puede hacer mapeo de puertos de un contenedor a puertos del host*, con

"docker run -p *puerto_host:puerto_contenedor*", exponiendo ("publicando") así al exterior servicios implementados en el contenedor.

Ejemplo 8

Escenario 1. Creamos un contenedor "alp" basado en alpine. Creamos otro contenedor "ngin" 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, "ngin" responde a accesos web en el puerto 8080 del host (tanto de localhost como de su dirección IP "pública"). Los dos contenedores están conectados a la red por omisión de Docker. Sin embargo, un contenedor no puede conectarse con el otro usando su nombre: conectándonos al contenedor "alp", los pings a "ngin" fallan. El acceso por dirección IP sí funcionaría.

```
$ docker run -itd --name alp alpine
0ab23ad6c3c5751bb5f525b84811272f0cac6bd9c0f563ad01ff2163d8e84500
$ docker run -itd --name ngin -p 8080:80 nginx
cc370adc9b5c336ecb2bcafea9daba6d3de9835ecf847217432ffe45c35669dd
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
NAMES
cc370adc9b5c   nginx     "/docker-entrypoint.…"   10 seconds ago   Up 10 seconds   0.0.0.0:8080->80/tcp,
:::8080->80/tcp   ngin
0ab23ad6c3c5   alpine    "/bin/sh"                32 seconds ago   Up 32 seconds
alp
$ docker port ngin
80/tcp -> 0.0.0.0:8080
80/tcp -> [::]:8080
$ curl http://localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
…
$ docker attach alp
/ # ping -c 3 ngin
ping: bad address 'ngin'
/ # read escape sequence
$ docker stop alp ngin
alp
ngin
$ docker rm alp ngin
alp
ngin
```

Escenario 2. Creamos la red bridge "interna". Como antes, creamos un contenedor "alp" basado en alpine. Creamos otro contenedor "ngin" basado en nginx, mapeando su puerto interno 80 con el puerto 8080 del host. Verificamos que "ngin" 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 "ngin". Tras ello paramos los contenedores, los borramos y eliminamos la red "interna".

```
$ docker network create interna
9092eca5e86e1a2a07cac8feadec1a414060c904d142361328138ab8044fea9f
$ docker run -itd --name alp --network interna alpine
89334efb1c3a6df3949539fc5fe455bda8b7759c6749f1984c358195bd6a6574
$ docker run -itd --name ngin --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   ngin
```

```
89334efb1c3a   alpine    "/bin/sh"                About a minute ago   Up About a minute
alp
$ docker port ngin
80/tcp -> 0.0.0.0:8080
80/tcp -> :::8080
$ curl http://localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
…
$ docker attach alp
/ # ping -c 3 ngin
PING ngin (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
--- ngin 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 ngin
$ docker rm $(docker ps --filter status=exited -q)
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
90d4007c733e   bridge    bridge    local
72683c2c01c4   host      host      local
23d1d84786ee   interna   bridge    local
05de59f779ee   none      null      local
$ docker network rm interna
```

Nota: se puede acceder al contenedor ngin usando un navegador como Firefox en la máquina anfitriona. En la máquina de prueba, Firefox no funcionaba, y fue necesario ejecutar estos comandos: "sudo snap remove firefox" y después "sudo apt install snap --reinstall firefox".

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

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
```

Otros drivers de red

Cuando se cread una red tipo "bridge" con "docker network create", Docker instala un conmutador (*switch*) Ethernet virtual asociado a una pasarela NAT – la que permite exponer puertos. Con este driver, los contenedores asociados a la red "bridge" tienen una dirección IP interna, en una subred diferente a la del host. Además, se crea un servicio DNS de traducción nombre <-> dirección IP asociado a dicha red.

El driver "**host**" hace que el contenedor use directamente la dirección IP del host. Es decir, el contenedor se comporta como un proceso más ejecutándose en el host.

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
```

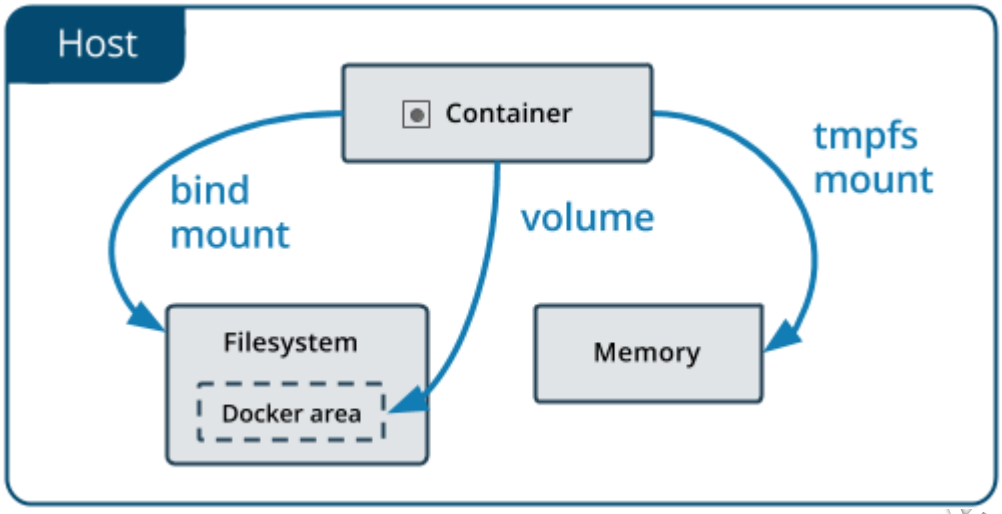
Con "ip address" hemos visto las tres intefaces del host. Tras lanzar el contenedor nginx con la opción "--network host" comprobamos que no se ha añadido ninguna interfaz virtual nueva (lo que sí habría pasado sin la opción indicada). Comprobamos que nginx está asociado al puerto 80 del host. Detenemos (y borramos, por la opción --rm) el contendor.

Además de los drivers "bridge" y "host" tenemos otros. El driver "**overlay**" tiene sentido cuando se usa Docker en modo *swarm*[5]. El driver "**macvlan**" presenta el contenedor como si tuviese su propia dirección MAC, es decir, como si tuviese una interfaz física conectada al mismo conmutador al que está conectado el host. Por lo tanto, también tendrá su propia dirección IP, que puede estar en la misma subred del host – o en otra distinta, si se usa una VLAN diferente. Esta opción es útil cuando un servicio debe usar una dirección específica, diferente a la del host. Por último, se puede desactivar por completo el acceso a la red de un contenedor ejecutándolo con la opción "--network **none**".

Volúmenes

El modelo de almacenamiento de Docker para contenedores es el de la figura. Un ***bind mount*** es un directorio compartido con el host. Un ***volume*** es un volumen gestionado por Docker, al que el host no puede acceder directamente, pero que pueden compartir varios contenedores. Un ***tmpfs mount*** es un directorio desde el punto de vista del contenedor, pero sin reflejo en el sistema de ficheros del host, está únicamente en RAM.

[5] El modo *swarm* de Docker es un sistema de orquestación de contenedores en un grupo de hosts. No lo estudiaremos, ya que su utilidad es similar a la de Kubernetes.



Los *volumes* son para almacenamiento persistente, accesibles por uno o más contenedores. Se crean con "docker volumen create" y su ciclo de vida está separado del de los contenedores que los usan. Se puede usar "docker volume prune" para borrar volúmenes sin uso. Los volúmenes pueden ser anónimos – no se les da un nombre cuando un contenedor lo monta, Docker les asigna un nombre aleatorio.

Ya hemos visto en un ejemplo anterior el uso de *bind mounts*: se gestionan al crear un contenedor, con la opción "-v" de "docker run".

Los *tmpfs mounts* no son persistentes y se pueden usar en un contenedor, por ejemplo, para guardar información sensible (secretos). Se crean con "docker run --tmpfs *dir_contenedor contenedor comando*".

La sintaxis para crear en un contenedor un *bind mount* asociado a una carpeta del host es:

docker run -v *dir_host*:*dir_contenedor contenedor comando*

Los directorios del host y del contenedor deben especificarse con su pathname completo, o bien "$HOME/path/relativo" o "~/path/relativo". Normalmente se especifican directorios, pero también es posible especificar ficheros individuales. Una vez creado un *bind mount*, está visible para el host y para el contenedor. Ambos pueden modificar su contenido, y lo que haga uno será inmediatamente visible por el otro.

Los *volumes* se gestionan de otra forma: hay que crearlos y destruirlos por separado.

docker volume create *volumen*

Crea un volumen persistente que puede ser usado por uno o más contenedores. Es muy importante indicar que los volúmenes no son accesibles directamente desde el host. Todo el acceso a volúmenes tiene que hacerse desde contenedores. Si el host necesita algo almacenado en un volumen, puede copiarlo de un contenedor conectado al mismo, usando "docker cp".

docker run -v *volumen*:*/directorio contenedor comando*

Al ejecutar el contenedor indicado se monta el volumen indicado en el directorio indicado (del contenedor). Si dos contenedores se asocian al mismo volumen, comparten sus datos. Los datos del volumen persisten tras terminar el contenedor que lo usa. Si no se especifica un nombre de volumen, sino solamente un directorio en el contenedor, se crea un volumen anónimo.

docker volume rm *volumen*

Borra un volumen, que no puede estar usado por ningún contenedor.

docker volume prune

Elimina todos los volúmenes locales que no están en uso (ningún contenedor hace referencia a ellos).

docker volume ls

Vuelca un listado de volúmenes.

docker volume inspect *volumen*

Inspecciona un volumen concreto. El volcado es en formato JSON.

docker rm -v *contenedor*

Elimina un contenedor y los volúmenes anónimos asociados al mismo.

docker run --tmpfs *dir_contenedor contenedor comando*

Crea un disco en RAM asociado al director indicado en el contenedor. Dicho disco no es persistente, dura mientras el contenedor está en ejecución.

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 el 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]
```

Imágenes, repositorios, etiquetas, registros

Las imágenes usadas para crear contenedores están guardadas localmente (las que se hayan usado) y en registros como Docker Hub (docker.io). El registro Docker Hub es el más popular, pero no el único. Otras empresas (como Microsoft Azure) tienen sus registros, y se puede crear uno propio.

Podemos ver las imágenes locales con "docker image ls". Entre la información obtenida tenemos una columna REPOSITORY y otra columna TAG. Las dos juntas dan el nombre completo de la imagen, por ejemplo "apache/hadoop:3". En ocasiones es necesario indicar delante el registro: "docker.io/apache/hadoop:3". Si no se indica registro, se asume que se usa Docker Hub. Si no se indica etiqueta, se asume "latest".

Recordemos que cuando lanzamos un contenedor indicamos qué imagen hay que usar. Si la imagen no se encuentra almacenada localmente, se copia del registro/repositorio indicado (teniendo en cuenta que el registro por omisión es Docker Hub) y queda almacenada, con su nombre completo, para usos futuros.

Cuando tenemos una imagen almacenada localmente, el nombre con el que aparece es el completo (repositorio + etiqueta). Esto es importante cuando vamos a subirla a un repositorio, ya que la información relevante se extrae del nombre.

Ciclo de vida de una imagen

docker image ls

Muestra todas las imágenes almacenadas localmente. Se obtiene el mismo resultado con "docker image".

docker build -f *dockerfile*

Crea una imagen, normalmente a partir de un Dockerfile (tomando como base otra imagen). La imagen queda almacenada localmente (si se desea se puede subir a un repositorio con "docker push"). Más adelante veremos cómo son los Dockerfiles. Si no se especifica qué fichero Dockerfile hay que usar, se usa "$PWD/Dockerfile" (un fichero llamado "Dockerfile" en el directorio de trabajo actual). *Aunque hay otras formas de crear imágenes, esta es la más recomendable.*

docker commit *contenedor nombre_nueva_imagen*

Crea una imagen a partir de un contenedor. Si el contenedor está en ejecución, queda temporalmente bloqueado.

docker image rm *imagen*

Elimina una imagen del almacén local. Se consigue lo mismo con "**docker rmi** *imagen*".

docker save -o *volcado.tar imagen*

Vuelca en formato .tar el contenido de la imagen. Si no se usa la opción "-o", el volcado se hace por STDOUT. Incluye todas las capas, etiquetas y versiones. Es una especie de *backup* de una imagen, en un fichero que se puede almacenar, enviar a otra persona, o usar más adelante con "docker load". No es exactamente una imagen (es un .tar), pero es fácil obtener a partir de este fichero la imagen original, con "docker load".

docker load -i *volcado.tar*

Crea una imagen, almacenada localmente, a partir de un archivo .tar. Sin la opción "-i", se puede hacer "docker load < *volcado.tar*", puesto que el comando espera la entrada por STDIN.

docker export *contenedor* > *volcado.tar*

Crea un archivo .tar a partir de un contenedor[6] con una instantánea del sistema de ficheros. Así, dos contenedores creados a partir de la misma imagen pueden generar ficheros .tar distintos cuando se exportan.

docker import < *volcado.tar*

Importa el .tar resultado de una operación "docker export" y a partir del mismo crea una imagen sin etiquetar. Esa imagen puede servir de base para crear nuevos contenedores, que tendrán el contenido del sistema de ficheros que se exportó.

docker tag *imagen_origen imagen_destino*

Asigna un nombre y, si se especifica, una etiqueta a una imagen local. El nombre puede ser completo: *repositorio:etiqueta*.

docker history *imagen*

Muestra la historia de la imagen, con todas sus capas.

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 desde el registro de Docker. 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", sin necesidad de acudir al registro. 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!
```

[6] Nótese que "docker save" crea un .tar a partir de una imagen, mientras que "docker export" lo hace a partir de un contenedor.

```
...

$ 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 (pero sin ser borrado). 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    3b07dc0c9a61   40 seconds ago   5.54MB
alpine                      latest    7e01a0d0a1dc   2 months ago     5.54MB
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND     CREATED             STATUS                       PORTS     NAMES
a2cccf1b3bc9   calc_img   "echo 42"   4 minutes ago       Exited (0) 4 minutes ago             bold_sammet
06168ce1903b   alpine     "echo 42"   6 minutes ago       Exited (0) 6 minutes ago             calc
...
$ docker image rm calc_img
Error response from daemon: conflict: unable to remove repository reference "calc_img" (must force) -
container a2cccf1b3bc9 is using its referenced image 3b07dc0c9a61
```

En estos momentos podemos ver las capas que forman una imagen. Veamos primero las capas de alpine, y luego las de calc_img. Se puede comprobar que el segundo está basado en el primero, con una capa más.

```
$ docker history alpine
IMAGE          CREATED       CREATED BY                                      SIZE      COMMENT
7e01a0d0a1dc   6 weeks ago   /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
```

```
<missing>      6 weeks ago   /bin/sh -c #(nop) ADD file:32ff5e7a78b890996…   7.34MB
 $ docker history calc_img
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
3b07dc0c9a61   40 seconds ago   echo 42                                         0B
7e01a0d0a1dc   6 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      6 weeks ago      /bin/sh -c #(nop) ADD file:32ff5e7a78b890996…   7.34MB
```

Veamos otra forma de crear imágenes. Volcamos el contenido completo del sistema de ficheros del contenedor "bold_sammet" (el nombre que haya asignado Docker) 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 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.

Descargando y subiendo imágenes a registros

Llamamos **registro** a un almacén de imágenes. Por omisión se usa Docker Hub (docker.io), pero puede ser otro.

Un **repositorio** es un conjunto de imágenes que comparten la primera parte del nombre pero tienen diferentes **etiquetas**. Las imágenes correspondientes a un repositorio podemos tenerlas de forma local, o en un registro. Un repositorio en un registro puede ser público o privado – aunque el registro esté "abierto al público" como Docker Hub.

Cuando descargamos una imagen ("docker pull", "docker run", etc.) dicha imagen queda como copia local, manteniendo su nombre completo (repositorio:etiqueta). Si queremos subirla, es fundamental que esté etiquetada. Si no lo está, debemos usar previamente "docker tag".

Recordemos que la parte repositorio de un nombre de imagen puede tener una de estas formas:

- *id* -- para imágenes "oficiales" de Docker Hub (docker.io/library/*id*)
- *usuario/id* -- para imágenes de usuarios de Docker Hub (docker.io/*usuario/id*)
- *registro/usuario/id* -- para imágenes en otros registros diferentes de Docker Hub

Si vamos a trabajar con un repositorio privado, necesitamos identificarnos ante el mismo con "docker login". Si usamos Docker Hub para descargar imágenes públicas no se necesita identificación – aunque sí será necesaria para subir imágenes.

docker login *registro*

Nos conecta al registro indicado. Si no se especifica cuál, se usa el registro oficial de Docker, "Docker Hub" (docker.io).

docker logout

Nos desconecta del registro al que estemos conectados.

docker search *término_a_buscar*

Realiza una búsqueda de imágenes en el registro. Por omisión se usa Docker Hub. Por comodidad, la búsqueda de imágenes se puede hacer también a través de su interfaz web (https://hub.docker.com/).

docker pull *repositorio:etiqueta*

Descarga una imagen, que queda guardada localmente para su uso futuro. La etiqueta es opcional: si no se pone, se asume ":latest".

docker push *repositorio:etiqueta*

Carga una imagen indicada en el repositorio indicado (que puede incluir la dirección del registro). La imagen tiene que estar correctamente etiquetada. Si no se indica registro se usa Docker Hub.

Ejemplo 14

Nos identificamos ante Docker Hub como usuario "dockuser". Tenemos una imagen local, "calc_img:latest" generada en un ejemplo anterior. Queremos que vaya al repositorio privado " dockuser /ipmd" de Docker Hub ("docker.io/

dockuser /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: dockuser
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 dockuser/ipmd:latest
$ docker push dockuser/ipmd
Using default tag: latest
The push refers to repository [docker.io/dockuser/ipmd]
994393dc58e7: Mounted from library/alpine
latest: digest: sha256:deb4540708cccba9f1363e4c181faabaf5f9160e94268b3b9461a1ecff0b85f2 size: 528
```

La versión gratuita de Docker Hub solo nos deja usar un repositorio público, pero se pueden tener múltiples etiquetas. Por tanto, podemos guardar diferentes imágenes pero todas tendrán igual la parte inicial: solo se diferenciarán en lo que va tras ":". Por ejemplo "dockeruser/ipmd:myalpine", "dockeruser/ipmd:mynginx", etc.

Imágenes y capas

Cuando descargamos una imagen, en realidad no se descarga como un fichero único: se descargan una por una las capas que la forman. Con todas ellas se construye la imagen. Hay una razón para explicar este comportamiento: reducir el trasiego de datos relacionado con imágenes, y la complejidad de construirlas. Veamos un ejemplo sencillo:

- Tenemos una imagen de partida Img1, con una única capa C1. Es una imagen que crea contenedores con una distribución Linux alpine básica.
- Creamos una imagen Img2 a partir de Img1, que añade a Img1 una capa con una instalación de Apache Web Server. Sus capas son, por tanto, C1 + C2.
- Creamos una imagen Img3 a partir de Img2, que añade una aplicación web concreta. Sus capas son, por tanto, C1 + C2 + C3.

Si en el futuro voy a construir una nueva imagen Img4 con una aplicación web diferente, no tengo que partir de Img1, puedo hacerla partiendo de Img2. Por lo tanto, solo es necesario modificar la tercera capa (C3b).

Si las cuatro imágenes están guardadas en un repositorio, no tengo

- Cuatro copias de C1: una para Img1, otra para Img2, otra para Img3, otra para Img4
- Tres copias de C2: una para Img2, otra para Img3, otra para Img4
- Una copia de C3 para Img3
- Una copia de C3b para Img4

Esto totalizaría 9 capas, muchas de ellas repetidas. Lo que se almacena en realidad es

- Una copia de C1. Img1, Img2, Img3 e Img4 parten de esta capa.
- Una copia de C2. Img2, Img3 e Img4 usan esta capa (por encima de C1).
- Una copia de C3. Img3 se completa con esta capa.
- Una copia de C3b. Img4 se completa con esta capa.

Todas las capas tienen un identificador hash único. Partiendo de un almacén de imágenes local vacío, un "docker pull Img3" descargaría las capas C1, C2 y C3. Un posterior "docker pull Img4" descargaría únicamente la capa C3b.

Recordemos que podemos ver las capas de una imagen con "docker image inspect".

Opciones adicionales sobre contenedores

Reinicio automático

Al ejecutar "docker run" se pueden especificar opciones adicionales de interés.

docker run -d --restart *opción_reinicio imagen*

Indica qué debe hacer Docker cuando un contenedor, que implementa un servicio, falla. Las opciones de reinicio son "no" (opción por omisión, no hacer nada), "unless-stopped" (lo reinicia siempre que no se haya detenido manualmente), "on-failure" (el contenedor es reiniciado si termina por un error; se puede indicar un máximo de reintentos).

Limitación de recursos

Un contenedor tiene, por omisión, acceso a todos los recursos del host. Sin embargo se pueden controlar esos recursos con opciones de "docker run". Muchas de estas opciones llevan un parámetro seguido de un sufijo que indica unidades: b, k, m, g para indicar múltiplos (binarios) de byte.

--memory=*valor*

Permite especificar la memoria máxima a usar. El valor mínimo es 6m. También se puede usar la versión corta (-m *valor*).

--cpus=*valor*

Indica los recursos de CPU disponibles para el contenedor. Puede ser un valor no entero. Por ejemplo, en un host con 2 CPUs se puede indicar "1.5".

Si hay instalada una o más GPUs NVIDIA (y se ha instalado "apt-get install nvidia-container-runtime") se pueden usar desde un contenedor:

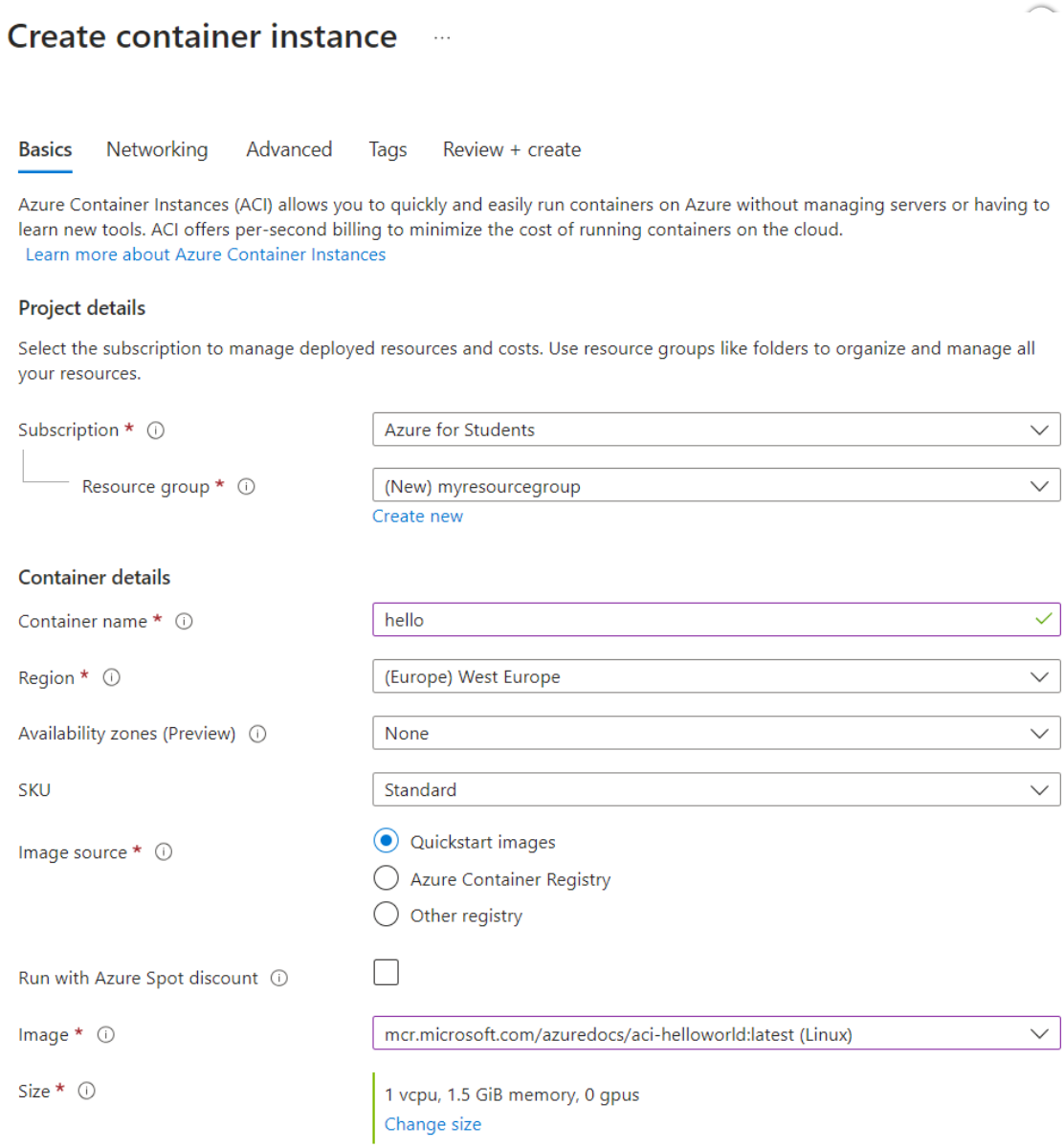
--gpus all

Permite al contendor usar todas las GPUs disponibles. Para más opciones, véase https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/user-guide.html

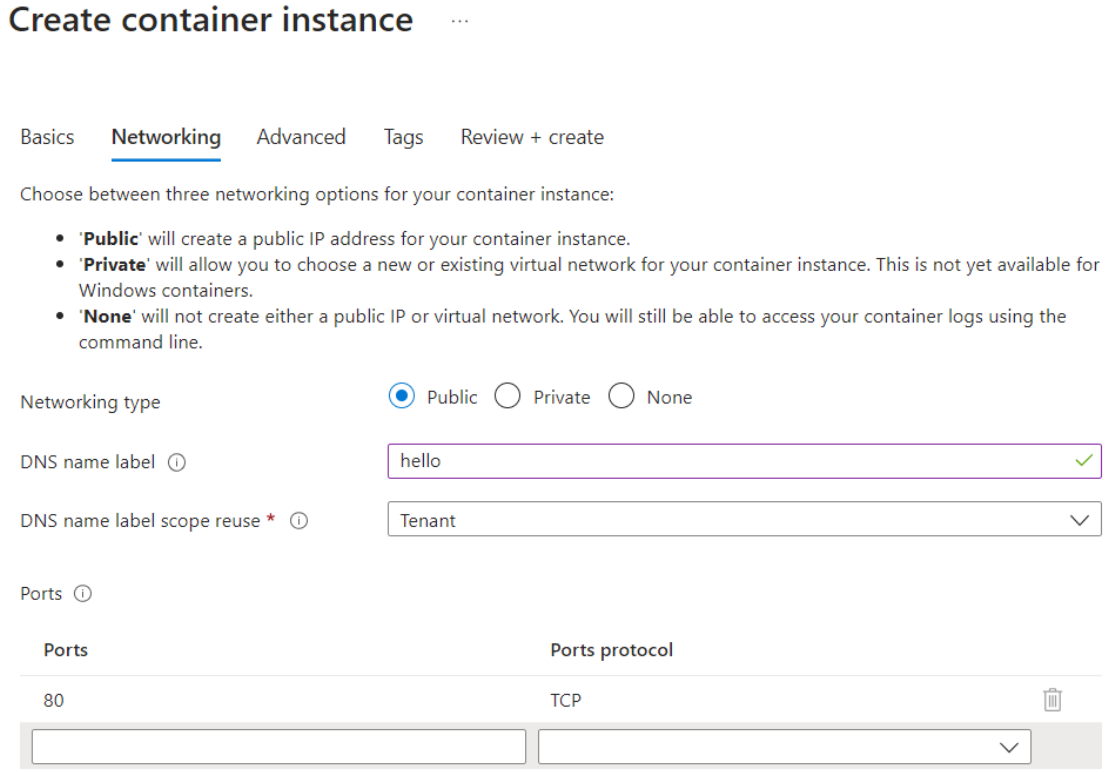
Docker en Azure

Desde el portal (se puede usar CLI): https://learn.microsoft.com/es-es/azure/container-instances/container-instances-quickstart-portal

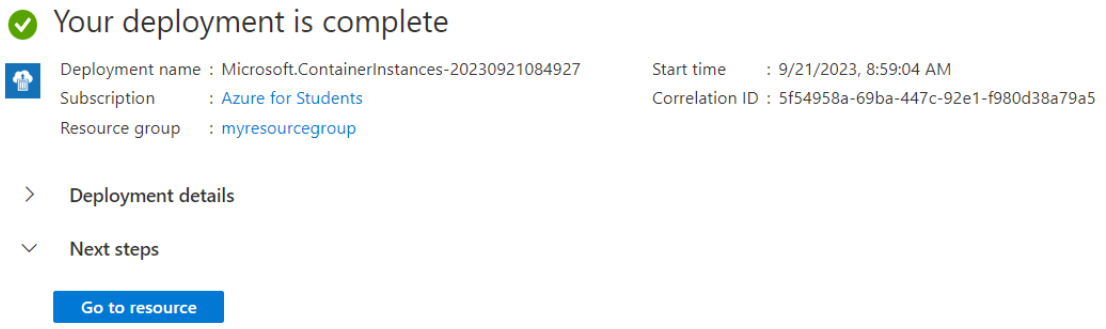
Lo primero es crear un recurso "Container Instances". Al hacerlo hay que facilitar una suscripción, un grupo de recursos (crear nuevo: myresourcegroup), un nombre de contenedor (hello), un origen de la imagen (Quickstart images de Azure), una imagen (aci-helloworld:latest, Linux).



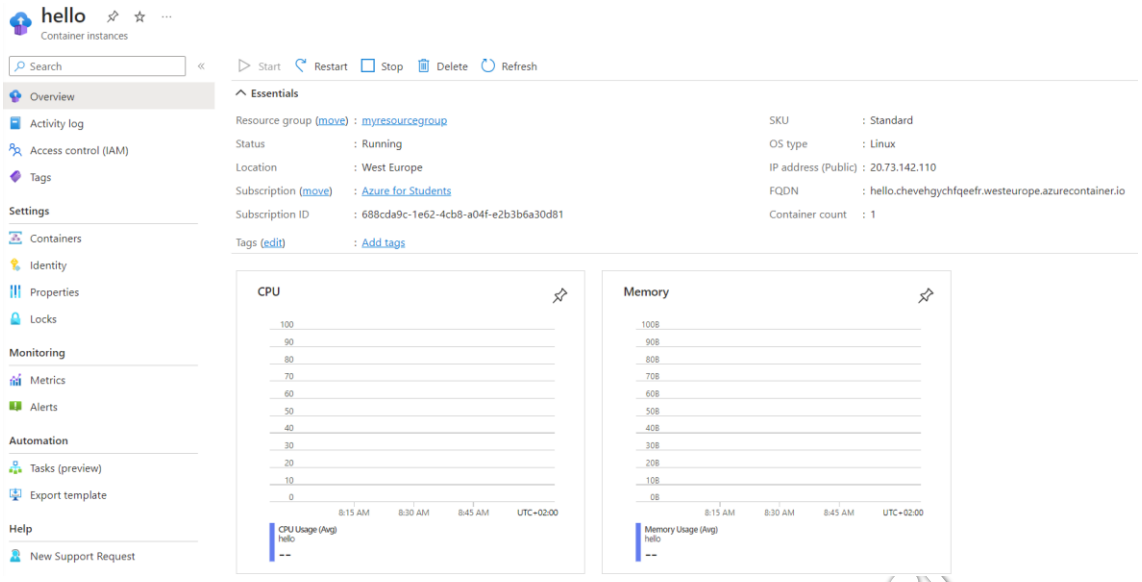
Para acceder al contenedor necesitamos que tenga conexión a Internet. Pulsamos el botón "Next: Networking" y seleccionamos opciones: Networking type (Public), DNS name label (hello). El puerto 80 ya aparece abierto.



Tras esto ya podemos pulsar "Review + create" y lanzar el contenedor. Pasaremos a esta pantalla:



Y si vamos al recurso, pasaremos a esta otra:



Copiemos el valor que aparece indicado como FQDN, en este caso hello.chevehgychfqeefr.westeurope.azurecontainer.io. Y abramos un navegador conectado a ese nombre (http://hello.chevehgychfqeefr.westeurope.azurecontainer.io).

En la página de control del contenedor podemos ver que, tras el acceso, ha aumentado el uso de la CPU, de la memoria, y de los bytes transmitidos.

Terminamos la demostración haciendo limpieza. Para ello lo más fácil es eliminar el grupo de recursos. Eso también eliminará el contenedor.

Generando imágenes con Dockerfiles

La forma más recomendable de construir imágenes para Docker es partir de una imagen de base y enriquecerla, añadiendo capas, usando para ello el comando "docker build" y un *Dockerfile*.

Los Dockerfiles especifican cómo construir una imagen a partir de otra de base, normalmente obtenida de un repositorio en un registro público. Lo habitual es crear un directorio-proyecto, y en el mismo crear un fichero con el nombre "Dockerfile". Cuando estamos en ese directorio, la ejecución de

docker build .

lanza la creación de la imagen a partir del Dockerfile. Alternativamente, se puede especificar explícitamente el Dockerfile a usar en la construcción:

docker build -f */path/to/a/Dockerfile*

La imagen queda almacenada localmente, sin nombre (habrá que usar "docker tag" para dárselo). La opción "-t *nombre*" permite especificar el nombre de la imagen resultante.

Antes de procesar el Dockerfile, "docker build" comprueba su sintaxis. Si es incorrecta, no se hace nada.

El directorio con el proyecto puede contener recursos necesarios para crear la imagen, como por ejemplo ficheros de datos a montar (*bind mounts*) o programas/scripts a cargar (con "docker cp".

El Dockerfile puede verse como una colección de instrucciones que se ejecutan secuencialmente. El formato de cada instrucción es "*instrucción argumentos*". Se ignoran los espacios en blanco y las líneas que empiezan por "#", que son comentarios. Se pueden poner sentencias de varias líneas poniendo "\" al final de todas las líneas menos la última.

```
RUN echo "\
Hello\
World"
```

Veamos algunas instrucciones. Cuando hablemos de "el contenedor" nos referiremos a "un contenedor basado en la imagen que se está creando".

ARG *nombre=valor*

Define una variable con su valor, que se puede usar en otro punto posterior del Dockerfile. Esta variable es válida solo durante la construcción de la imagen -- no existirá en los contenedores.

LABEL *clave=valor*

Añade metadatos a la imagen.

FROM *imagen*

Establece la imagen de base, la capa inicial. Los comandos del Dockerfile irán añadiendo capas a esa imagen.

RUN *comando*

Ejecuta al crear la imagen una línea de comandos shell. Alternativamente "**RUN** ["ejecutable", "param1", "param2"…] ejecuta un programa arbitrario con los parámetros indicados. Se pueden especificar volúmenes montados con "**RUN --volume**", redes "**RUN --network**", y otras opciones. Un efecto de RUN es añadir una nueva capa a la imagen. El uso habitual es instalar nuevos paquetes ("RUN apt install *paquete*").

CMD *comando*

Donde comando es como antes (una línea para el shell, o una lista de programa y argumentos). CMD indica qué hay que ejecutar por omisión cuando se lanza el contenedor con "docker run". Nótese que el comando como tal no se ejecuta en el momento de construir la imagen. Cuando lancemos un contenedor con "docker run *imagen comando*", se ejecutará el indicado. Pero si no se especifica ("docker run *imagen*") se ejecuta el comando por omisión indicado con esta sentencia CMD.

EXPOSE *puerto/protocolo*

Informa sobre los puertos de escucha que pondrá en marcha el contenedor. Si no se indica protocolo, se asume "tcp". Nótese que EXPOSE *no realiza el mapeo del puerto*, es puramente informativo. El mapeo efectivo hay que hacerlo en el momento de la ejecución del contenedor ("docker run -p").

ENV *clave=valor*

Define variables de entorno. Se aplican (como ARG) durante la construcción de la imagen, pero también estarán disponibles en el contenedor en ejecución. Se pueden cambiar al ejecutar el contenedor, con "docker run --env *key=value*".

ADD *fuente destino*

Copia datos de un fichero *fuente* (local, remoto desde URL) al directorio *destino* del sistema de ficheros de la imagen. Una forma más restringida de hacerlo (no copia desde URLs, no descomprime) es "**COPY** *fuente destino*".

ENTRYPOINT *comando argumentos*

La sintaxis alternativa es "**ENTRYPOINT** ["*ejecutable*", "*param1*", "*param2*"…]" (como en CMD y RUN). Permite especificar un comando que se ejecutará siempre al lanzar el contenedor. También se puede reemplazar con "docker run --entrypoint="*comando*"), pero no es tan fácil hacerlo como en el caso de CMD. Si cuando se ejecuta el contenedor se facilita una lista "docker run *comando arg1 arg2…*", o se ha establecido un CMD, se considerará esa lista (incluyendo el comando) como argumentos del ENTRYPOINT.

VOLUME */volumen*

Crea un punto de montaje en el directorio especificado de la imagen, indicando que será un volumen. Si no existe dicho volumen, se crea.

USER *usuario:grupo*

Establece el usuario para las etapas posteriores.

WORKDIR */path/to/workdir*

Establece el directorio de trabajo para cualquier instrucción RUN, CMD, ENTRYPOINT, COPY y ADD.

Cuando hay que introducir sentencias largas en RUN y COPY, se puede usar una opción del shell de Unix denominada "here-documents". Se empieza un comando, y se indica "<<TERMINADOR" una palabra de terminación. Luego se pueden escribir varias líneas, que se consideran que son parte del mismo comando. Cuando se escribe una línea con el terminador, se entiende que la línea ha terminado. Por ejemplo:

```
FROM debian
RUN <<EOT
```

```
 apt update
 apt install -y vim
EOT
```

Es equivalente a "RUN apt update && apt install -y vim".

Nota: en https://github.com/acpmialj/flask están los ficheros necesarios para los ejemplos 15, 16 y 17 ("git clone https://github.com/acpmialj/flask", "cd flask").

Ejemplo 15

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".

Nota: Como se ha visto, hay diferentes formas de distribuir imágenes que servirán para crear contenedores: (1) Archivos .tar con la imagen, que se puede instalar localmente con "docker load". (2) Almacén de imágenes en registros públicos o privados, que se pueden instalar localmente (con "docker pull") , o instalar y ejecutar (con

"docker run"). (3) Distribuir un Dockerfile con las instrucciones para construir la imagen. Esta última es la más habitual en el caso de proyectos alojados en sitios como GitHub.

Orquestación de contenedores con Docker Compose

Compose sirve para definir y ejecutar aplicaciones que precisan de varios contenedores. Se da información de cada contenedor, los volúmenes que comparten, la red que usan para comunicarse, etc. Todos los contenedores se ejecutan en el mismo host. En definitiva, se trata de un sistema de ***orquestación de contenedores en un único host*.** Mientras que el elemento básico de Docker es el contenedor, el de Compose es el *servicio*.

Compose parte de un fichero de texto con sintaxis YAML que declara los contenedores a usar y sus interrelaciones. Normalmente tendremos un fichero llamado "compose.yaml" en un directorio-proyecto[7]. Al ejecutar "**docker compose up**" dentro de ese directorio-proyecto se realizan todas las operaciones Docker necesarias (incluyendo la construcción de imágenes a partir de Dockerfiles, la creación de redes y de volúmenes) y se pone en marcha un conjunto de contenedores. El sistema se detiene con "**docker compose down**".

Veamos un ejemplo de fichero compose.yaml (es solo un ejemplo, no operativo):

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    networks:
      - backend
    depends_on:
      - redis
  redis:
    image: redis
    networks:
      - backend
volumes:
  logvolume01: {}
networks:
  backend: {}
```

Los elementos más importantes de un fichero docker-compose son:

- **Services**. Declaración de (micro) servicios, cada uno de ellos ofrecido por un contenedor. En el ejemplo se definen dos servicios, "web" y "redis".
    - El servicio "redis" se implementa con contenedores basados en la imagen "redis" de Docker hub. De ahí la sentencia "image".
    - El servicio "web" está basado en contenedores construidos a partir de un Dockerfile que forma parte del proyecto. De ahí la sentencia "build". Además, requiere mapear el puerto 5000 del contenedor en el puerto 8000 del host y montar unos volúmenes. Se indica además que depende de que esté en funcionamiento el servicio "redis".
- **Volumes**. Declaración de los volúmenes usados por los servicios anteriores. En este caso Docker gestionará un volumen para los logs del servicio "web".
- **Networks**. Declaración de las redes usadas por los servicios anteriores. Se puede indicar el *driver* a usar (por omisión es "bridge"). Si no se especifica ninguna red, se usaría una red "bridge" creada ad-hoc para el

[7] El fichero se puede llamar "docker-compose" o "compose", con extensión "yml" o "yaml".

proyecto, a la que estarían conectados todos los contenedores. Nótese que *no* es la red por omisión de Docker, lo que permite la resolución de nombres.

Nota: En YAML la indentación (lo habitual son 2 espacios) es clave para definir sub-secciones dentro de cada sección (por ejemplo, "build" o "ports" dentro de "services". Tras una etiqueta de sección o subsección puede venir un valor único (en la misma línea) o una lista (en líneas sucesivas, precedido con un "-" tras indentación adicional). Muchas subsecciones pueden tener versión corta (una línea, como "image: redis") y versión larga (multilínea, como "volumes:" dentro de "web").

Veamos algunas de las secciones / subsecciones de "docker-compose.yaml":

- **version**: indica el número de versión de Compose que se está usando. Opcional.
- **services**: indica todos los servicios que forman parte de la aplicación. La definición de un servicio puede tener múltiples subsecciones:
  - **image** indica una imagen a usar para lanzar el contenedor que implementa el servicio.
  - **build** indica que hay que construir la imagen a partir de un Dockerfile.
  - **container_name** indica el nombre a dar al contenedor. Si no se indica, se usará uno basado en el proyecto y el servicio.
  - **command** indica el comando inicial a ejecutar por el contenedor.
  - **volumes** (dentro de una definición de servicio) indica normalmente mapeos host/contenedor (como la opción "-v" de "docker run"). También puede indicar mapeos de volúmenes Docker a directorios del contenedor. En el ejemplo, en el servidor "web"
    - se asocia el directorio del proyecto al directorio "/code" del contenedor y
    - se usa un volumen persistente "logvolume01" (se crea si no existe) mapeado al directorio "/var/log" del contenedor.
  - **networks**: indica a qué redes se conectará el contenedor. Si no se indica se usa una red creada ad-hoc para el proyecto.
  - **ports:** indica los puertos que se exponen (como como la opción "-p" de "docker run").
  - **environment**: define variables de entorno, válidas dentro de los contenedores
  - **restart**: indica qué hacer cuando un contenedor termina. "no" = nada. "always" = se reinicia, hasta que se elimine. "on-failure" = se reinicia si el código de terminación indica error. "unless-stopped" = se reinicia cuando termina, a no ser que la terminación sea con stop o se elimine el contenedor.
  - **depends_on**: lista los servicios que tienen que estar activos antes de arrancar este servicio.
  - **deploy**: indica cómo se va a desplegar el servicio. Esto es dependiente de la plataforma host, por lo que puede interpretarse como una serie de meta-datos a utilizar en el momento del despliegue. Entre otras cosas, se puede indicar el número de contenedores replicados a usar para implementar un servicio.
- **volumes**: declara los volúmenes que se crean y usan (entre todos los servicios) – como los manejados vía "docker volume …".
- **network**: indica las redes que se crean y usan (entre todos los servicios) – como las manejadas vía "docker network …".

Hay una lista interesante de ficheros "compose.yaml" en https://github.com/docker/awesome-compose

Ejemplo 16

Este ejemplo es continuación del anterior. Recordemos que la solución con un contenedor no funcionaba porque no había servicio Redis. Luego lo solucionamos a mano creando una red "bridge" y ejecutando, además del contenedor de la aplicación web, otro contenedor con el servicio Redis. Gracias a Compose, se puede automatizar la puesta en marcha de esta aplicación multi-contenedor.

Tenemos en nuestro directorio "$HOME/flask" el Dockerfile que especifica cómo crear el contenedor con la aplicación Python/Flask "app.py". El siguiente fichero "compose.yaml"

- Crea una imagen basada en ese Dockerfile y lanza con ella un contenedor que implementa el servicio llamado "web", cuyo puerto 80 está mapeado en el puerto 4000 del host.
- A partir de una imagen de Redis crea un contenedor que implementa el servicio "redis" (en el puerto 6379, que es el puerto por omisión de Redis). Además se monta el directorio del host "./data" en el directorio "/data" del contenedor Redis. Es ahí donde Redis hace un almacenamiento persistente de sus datos.

Nota: En el directorio flask creado a partir de GitHub, está el fichero "compose_simple.yaml". Puedes cambiarle el nombre para que sea "compose.yaml": "mv compose_simple.yaml compose.yaml".

Nótese que se indica que el contenedor que implementa el servicio "web" requiere construir localmente una imagen a partir de un Dockerfile ("build: ."), y sin embargo, el contenedor que implementa el servicio "redis" se descarga de un repositorio.

```
services:
  web:
    build: .
    ports:
      - "4000:80"
  redis:
    image: redis
    volumes:
      - "./data:/data"
```

El servicio se lanza con "docker compose up -d" (la opción "-d" hace que el terminal quede liberado). Lo pararemos más adelante con "docker compose down". Para ver los contenedores lanzados podemos usar "docker ps", pero es más recomendable usar "docker compose ps", que organiza la salida en base a servicios.

```
$ docker compose up -d
[+] Building 0.2s (9/9) FINISHED
 => [internal] load build definition from Dockerfile                                0.0s
 => => transferring dockerfile: 32B                                                   0.0s
 => [internal] load .dockerignore                                                     0.0s
 => => transferring context: 2B                                                       0.0s
 => [internal] load metadata for docker.io/library/python:2.7-slim                    0.0s
 => [internal] load build context                                                     0.0s
 => => transferring context: 38.44kB                                                  0.0s
 => [1/4] FROM docker.io/library/python:2.7-slim                                      0.0s
 => CACHED [2/4] WORKDIR /app                                                         0.0s

 => CACHED [3/4] COPY . /app                                                          0.0s
 => CACHED [4/4] RUN pip install --trusted-host pypi.python.org -r requirements.txt  0.0s
 => exporting to image                                                                0.0s
 => => exporting layers                                                               0.0s
 => => writing image sha256:8b5b667113ee34783d2730688d2a13a8e9a3150c0f2b6c66634d943811a8b  0.0s
 => => naming to docker.io/library/flask-web                                          0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

[+] Running 3/3
 ⠿ Network flask_default    Created                                                   0.1s
```

```
 ⠿ Container flask-redis-1  Started                                    0.7s
 ⠿ Container flask-web-1    Started                                    0.4s
$ docker compose ps
NAME                COMMAND                  SERVICE             STATUS              PORTS
flask-redis-1       "docker-entrypoint.s…"   redis               running             6379/tcp
flask-web-1         "python app.py"          web                 running             0.0.0.0:4000->80/tcp, :::4000->80/tcp
$ docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
f31b173aee7a   bridge            bridge    local
70658bc9849b   flask_default     bridge    local
25208448122e   host              host      local
f0e94c6a7a03   none              null      local
```

Nótese que se crean dos contenedores, uno llamado "flask-redis-1" y otro llamado "flask-web-1". Los nombres se toman del proyecto. Se monta el directorio del host "./data" en el directorio "/data" del contenedor "flask-redis-1" (es aquí donde Redis guarda sus datos). En este ejemplo no se usan volúmenes persistentes. Se crea además una red que los conecta, "flask_default". La creación de esta red es importante para que los contenedores puedan comunicarse usando sus *nombres de servicio* (columna SERVICE de "docker compose ps").

La aplicación Python/Flask que se ejecuta en "flask-web-1" ("app.py") presenta una página web (http://localhost:4000) que usa los servicios del contenedor "flask-redis-1" (con alias de servicio "redis") como almacén de parejas <clave, valor>. La comunicación entre los dos contenedores se hace a través del puerto 6379, el habitual para Redis, en la red "flask_default", en la que Docker ofrece servicio DNS. Si nos fijamos en el código que la aplicación, vemos que (una vez en ejecución en el contenedor "flask-web-1") se conecta a un servidor Redis sin saber su dirección IP, usando el nombre "redis". Este nombre no corresponde al contenedor "flask-redis-1". Sin embargo, Compose ha creado el alias de servicio "redis" asociado a dicho contenedor, por eso la conexión funciona sin problema.

Al conectarnos al URL anterior obtenemos:

```
Hello World!
Hostname: 052396375fa6
Visits: 3
```

Donde el número de visitas cambia cada vez que se accede.

Destruimos la aplicación con "docker compose down". Si la volvemos a lanzar, el proceso será muy rápido porque no hace falta ni construir una imagen para el servicio "web", ni descargar la imagen de Redis para el servicio "redis". El contenedor que implementa el servicio redis ha dejado un directorio con los ficheros que necesita. Si volvemos a ejecutar el servicio, accederá a estos ficheros ya creados, y seguirá incrementando el contador de visitas.

```
$ docker compose down
[+] Running 3/3
 ⠿ Container flask-redis-1  Removed                                    0.3s
 ⠿ Container flask-web-1    Removed                                   10.3s
 ⠿ Network flask_default    Removed                                    0.1s
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
$ ls
app.py  data  compose.yaml  Dockerfile  requirements.txt
$ ls data
appendonlydir  dump.rdb
```

Ejemplo 17

Vamos un paso más allá: queremos tener una aplicación web multi-servidor, en la que

1. Las peticiones web del usuario las recoja un balanceador de carga
2. El balanceador la reenvíe a uno cualquiera de los contenedores que implementan el servicio web
3. Todas las réplicas del servicio web usan una misma instancia de Redis para mantener un estado consistente

Necesitamos un nuevo elemento en nuestra aplicación: el balanceador de carga. El nuevo fichero, al que vamos a llamar "compose_lb.yaml", quedaría así:

```
services:
  web:
    build: .
    deploy:
      mode: replicated
      replicas: 2
  redis:
    image: redis
    volumes:
      - "./data:/data"
  nginx:
    image: nginx:latest
    volumes:
      - $PWD:/etc/nginx:ro
    depends_on:
      - web
    ports:
      - "4000:4000"
```

Nótese que ahora hay tres servicios, donde "nginx" es un balanceador de carga. El servicio "web" está replicado (se lanzarán dos contenedores para implementarlo). Las peticiones web al puerto 4000 las recibirá el servicio "nginx", y las repartirá entre dos instancias del servicio "web".

Podemos lanzar toda la colección de servicios con "docker compose -f compose_lb.yaml up -d". La opción "-d" es para no bloquear el terminal. Nótese además que hemos especificado con "-f" el fichero de configuración a usar por Compose, en vez de usar el nombre por omisión.

Si ejecutamos "docker compose -f ./compose_lb.yaml up -d --scale web=5" se lanzarían 5 réplicas del contenedor web. Si ya había dos en marcha, se lanzarán 3 adicionales.

Al acceder a http://localhost:4000 obtendremos un mensaje similar al de antes. El número de visitas crecerá en cada acceso. Pero, además, el "Hostname" también variará, en función de cuál de las réplicas del servidor web nos haya atendido.

Falta añadir a nuestro proyecto un fichero de configuración "nginx.conf" que Nginx requiere para hacer el trabajo que deseamos:

```
user  nginx;
events {
    worker_connections   1000;
}
http {
        server {
              listen 4000;
              location / {
                proxy_pass http://web:80;
              }
```

```
        }
}
```

Con todas estas piezas, los pasos a dar son:

```
$ docker compose -f ./compose_lb.yaml up -d --scale web=5 # sin --scale, 2 réplicas
[+] Running 8/8
 ⠿ Network flask_default    Created                                          0.1s
 ⠿ Container flask-redis-1  Started                                          1.5s
 ⠿ Container flask-web-5    Started                                          2.0s
 ⠿ Container flask-web-2    Started                                          2.1s
 ⠿ Container flask-web-1    Started                                          1.8s
 ⠿ Container flask-web-3    Started                                          1.9s
 ⠿ Container flask-web-4    Started                                          1.7s
 ⠿ Container flask-nginx-1  Started                                          2.6s
$ docker compose -f ./compose_lb.yaml
NAME                COMMAND                  SERVICE             STATUS              PORTS
flask-nginx-1       "/docker-entrypoint.…"   nginx               running             80/tcp, 0.0.0.0:4000->4000/tcp,
:::4000->4000/tcp
flask-redis-1       "docker-entrypoint.s…"   redis               running             6379/tcp
flask-web-1         "python app.py"          web                 running             80/tcp
flask-web-2         "python app.py"          web                 running             80/tcp
flask-web-3         "python app.py"          web                 running             80/tcp
flask-web-4         "python app.py"          web                 running             80/tcp
flask-web-5         "python app.py"          web                 running    flask_web_4
$ curl http://localhost:4000
<h3>Hello World!</h3><b>Hostname:</b> 46cfdf5ba655<br/><b>Visits:</b> 21
$ curl http://localhost:4000
<h3>Hello World!</h3><b>Hostname:</b> a28ef7aed479<br/><b>Visits:</b> 22
$ curl http://localhost:4000
<h3>Hello World!</h3><b>Hostname:</b> a6d574bbd3f9<br/><b>Visits:</b> 23
$ docker compose -f ./compose_lb.yaml down
[+] Running 8/8
 ⠿ Container flask-nginx-1  Removed                                          0.6s
 ⠿ Container flask-redis-1  Removed                                          0.6s
 ⠿ Container flask-web-3    Removed                                         10.7s
 ⠿ Container flask-web-1    Removed                                         11.0s
 ⠿ Container flask-web-4    Removed                                         11.1s
 ⠿ Container flask-web-2    Removed                                         11.0s
 ⠿ Container flask-web-5    Removed                                         10.9s
 ⠿ Network flask_default    Removed                                          0.1s
```

Comandos "docker compose"

Cuando se trabaja con Compose tenemos una serie de comandos que empiezan por "docker compose" y que nos permiten interactuar de forma más efectiva con nuestros servicios. Muchos son similares a los que tenemos con "docker" a secas, pero con una funcionalidad ligeramente diferente: trabajan sobre servicios, no sobre contenedores individuales.

docker compose up -d *opciones*

Lanza los servicios indicados en el fichero compose.yaml del directorio de trabajo, incluyendo redes, volúmenes, etc. Una opción interesante es "--scale *servicio*=N", que indica que se lancen N instancias de un servicio dado -- ignorando lo que diga el la subsección de despliegue del fichero compose.yaml. Si la aplicación ya está en marcha, volver a usar "docker compose up" devolvería la aplicación a su funcionamiento normal. Por ejemplo, si hubiese fallado un contenedor, lo volvería a lanzar. También podemos re-escalar un servicio. Por ejemplo, si habíamos lanzado 5 copias del servidor web, ejecutar "docker compose up -d --scale web=3" bajaría el número de instancias a 3.

Nota: Cambiar dinámicamente el número de contenedores que implementan un servicio puede no funcionar siempre correctamente. En este ejemplo, el balanceador de carga se configura al principio con las instancias existentes del servidor web. Si cambian a más, no usará las nuevas. Si cambian a menos, dirigirá tráfico a instancias inexistentes. La solución en estos casos, si no se quiere parar toda la aplicación y volver a lanzarla, es detener únicamente el servicio nginx ("docker compose -f compose_lb.yaml stop nginx") y volver a lanzar la aplicación ("docker compose -f compose_lb.yaml up -d --scale web=7"). El último paso eliminará o creará los contenedores necesarios para el servicio web, y creará una copia nueva del servicio nginx.

docker compose down [--remove orphans]

Detiene todos los servicios del compose.yaml y elimina los contenedores, redes, etc. La opción "--remove orphans" eliminaría contenedores adicionales, no mencionados en el fichero Compose.

docker compose ps

Lista los servicios / contenedores que forman nuestra aplicación.

docker compose run *opciones servicio comando*

Crea un contenedor nuevo basado en el servicio indicado, pero no lo integra en el servicio -- no cuenta como una réplica más.

docker compose stop *opciones servicio comando*

Destruye el servicio indicado.

docker compose exec *opciones servicio comando*

Ejecuta un comando sobre un contenedor existente.

Orquestación de contenedores en Azure

Ya hemos visto que Azure ofrece recursos del tipo "Container instances", que son contenedores Docker. También ofrece formas de orquestar aplicaciones multi-contenedor, entre las que están

1. Docker compose. Tutorial: https://learn.microsoft.com/es-es/azure/container-instances/tutorial-docker-compose
2. Uso del Resource Manager de Azure. Tutorial: https://learn.microsoft.com/es-es/azure/container-instances/container-instances-multi-container-group
3. Uso de archivos YAML, similares a los que usa Kubernetes. Tutorial: https://learn.microsoft.com/es-es/azure/container-instances/container-instances-multi-container-yaml

En todos los casos las aplicaciones se despliegan en un único host. Para orquestación multi-host se puede usar Kubernetes.