

# ADMINISTRACIÓN DE INFRAESTRUCTURAS Y SISTEMAS INFORMÁTICOS (AISI)

## Grado en Ingeniería Informática

Roberto R. Expósito ([roberto.rey.exposito@udc.es](mailto:roberto.rey.exposito@udc.es))

# DOCKER



# Contenidos

3

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm



# Contenidos

4

- **Introducción**
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm



# ¿Qué es Docker?

5

- **Docker** es una plataforma para desarrollar, empaquetar y desplegar aplicaciones y servicios usando virtualización "ligera" basada en contenedores
  - También denominada como virtualización por compartición de *kernel* o virtualización a nivel de sistema operativo
- Actualmente, Docker goza de una enorme popularidad
  - ¿Qué proporcionan los contenedores que no proporcionan otras tecnologías?



<https://www.docker.com>



# Antecedentes

6

- Aplicaciones **monolíticas**: agrupan toda su funcionalidad (lógica de negocio, base de datos, interfaz) en un único bloque de código
  - Toda la aplicación está contenida en un solo proyecto o repositorio
  - La aplicación se despliega/ejecuta como una sola unidad
  - Fuertes dependencias internas
  - Ciclos de desarrollo largos y mantenimiento más complejo
  - Difícil de escalar: toda la aplicación escala a la vez
- La tendencia actual se basa en desarrollar y desplegar múltiples servicios pequeños desacoplados: **arquitectura de microservicios**
  - Microservicios independientes que se comunican a través de APIs
  - Desarrollo más rápido, ágil y flexible: distintos microservicios pueden usar tecnologías distintas
  - Fácil de escalar: es posible escalar solo los microservicios necesarios

microservicio de ventas, p.e



# Contenedores

7

- Encapsulan la ejecución de un servicio/aplicación utilizando características propias que ofrece el *kernel* del sistema operativo
- Se ejecutan como procesos completamente aislados
  - Sistema de ficheros raíz propio
  - CPU
  - Memoria
  - Red
- No son un concepto nuevo: existen desde hace más de 15 años
  - FreeBSD Jails (1999), Linux-Vserver (2001), Solaris Containers (2004)



# Ventajas de Docker

8

- Estandarización y simplificación de formatos
  - Definir cómo construir una imagen de contenedor con *Dockerfiles*
  - Definir un *stack software* completo con Docker Compose
- Consolidación de un clúster de servidores con Docker Swarm
- Escalado, balanceo de carga, replicación...
  - Sin modificar la aplicación
- Usar los mismos contenedores para todos los entornos
  - *Testing*, desarrollo, producción...
- Creación de entornos de desarrollo en segundos/minutos

```
$ git clone ...
$ docker-compose up
```



# Contenidos

9

- Introducción
- **Conceptos básicos**
- Imágenes
- *Dockerfiles*
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm



# Mi primer contenedor

10

```
vagrant@docker:~$ docker run ubuntu echo hello world
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
29202e855b20: Pull complete
Digest: sha256:e6173d4dc55e76b87c4af8db8821b1feae4146dd47341e4d431118c7dd060a74
Status: Downloaded newer image for ubuntu:latest
hello world
vagrant@docker:~$
```

- **docker run** es el comando que permite **crear** y **ejecutar** un **contenedor**
- **ubuntu** es el nombre de una **imagen** de Docker basado en dicho SO
- Docker descarga la imagen desde un repositorio público: el **Docker Hub**
- Luego se crea el contenedor a partir de dicha imagen
- El contenedor ejecuta un binario/comando/aplicación
  - En este ejemplo se ejecuta el comando echo, junto con sus parámetros (*hello world*)



# ¿Qué es una imagen?

11

- Fichero comprimido que contiene todo lo necesario para ejecutar un contenedor
  - Los ficheros que necesita el contenedor (*root file system: rootfs*)
    - Binario/s, librería/s, código, directorios...
  - Metadatos con la configuración del contenedor
- **Un contenedor es una instancia en ejecución de una imagen**
  - De forma similar a un ejecutable (binario) y la aplicación (proceso) que se crea al ejecutarlo
  - Se pueden crear múltiples contenedores independientes a partir de la misma imagen



# ¿Qué es realmente un contenedor?

12

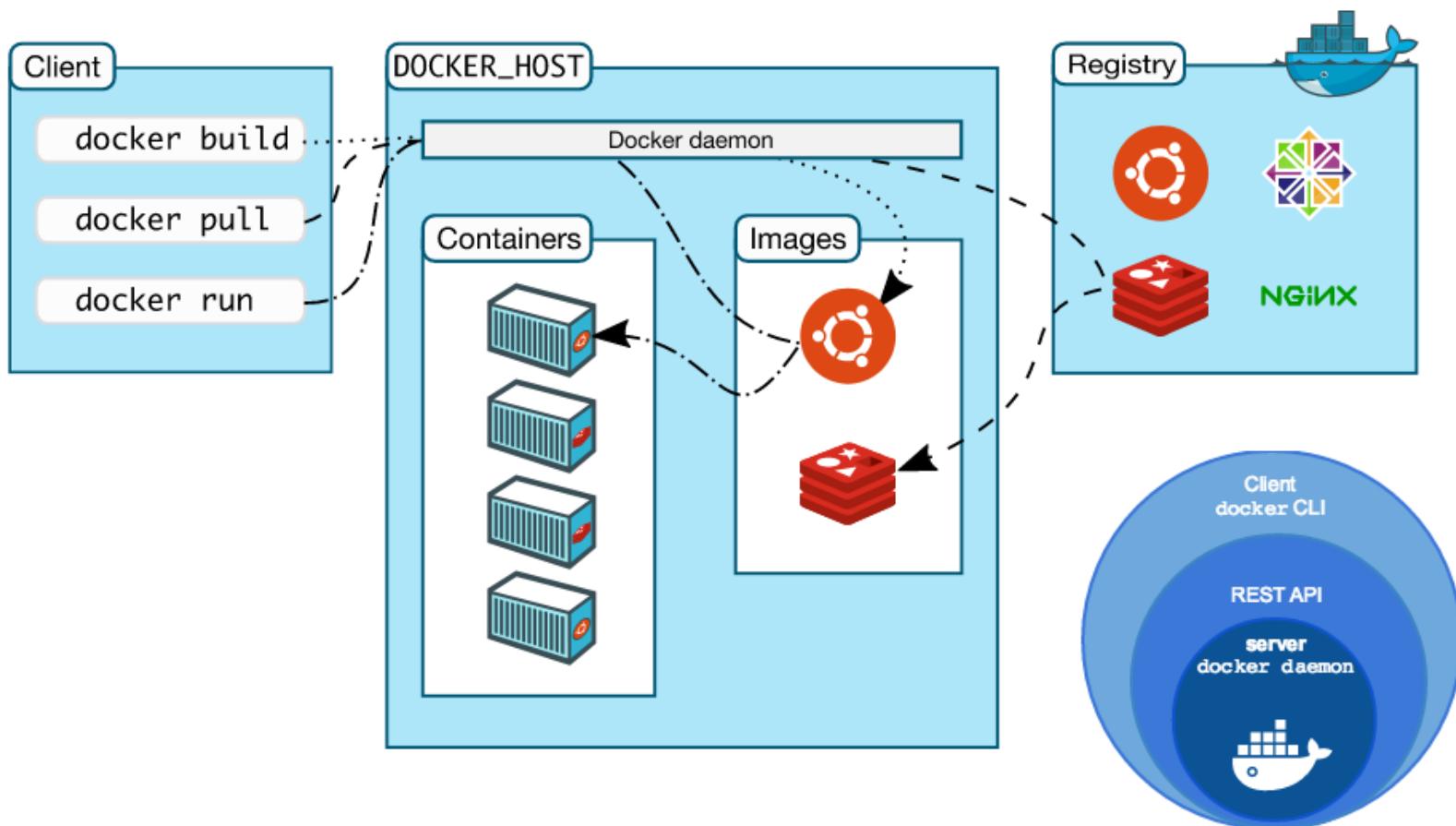
- Un contenedor no es más que un proceso en ejecución **aislado** del resto de procesos del sistema (y de otros contenedores)
  - La imagen es el sistema de ficheros raíz completo (**rootfs**) del proceso
    - La imagen es inmutable, compuesta por capas superpuestas de sólo lectura como veremos más adelante
    - El contenedor “copia” los datos de la imagen cuando es creado y se ejecuta sobre dicha copia en modo escritura
  - Docker permite controlar el nivel de aislamiento de un contenedor
  - **El ciclo de vida de un contenedor está ligado al proceso que ejecuta**



# Arquitectura Docker

13

- Cliente y demonio (Docker daemon) se comunican mediante una API REST





# Creando un contenedor interactivo

14

- Ejemplo usando una imagen de Ubuntu Jammy

```
vagrant@docker:~$ docker run --name interactive --rm -ti ubuntu:jammy bash
Unable to find image 'ubuntu:jammy' locally
jammy: Pulling from library/ubuntu
Digest: sha256:e6173d4dc55e76b87c4af8db8821b1feae4146dd47341e4d431118c7dd060a74
Status: Downloaded newer image for ubuntu:jammy
root@9883c78dc07b:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root
```

- **--name <name>** permite asignar un nombre al contenedor
- **--rm** permite eliminar automáticamente el contenedor cuando termine
- **-t conecta una pseudo-terminal al contenedor**
- **-i habilita modo interactivo (abre la entrada estándar del contenedor)**
- **:jammy** especifica el tag o versión concreta de la imagen ubuntu
- **bash** es el binario que ejecutará el contenedor
- Fíjate en el *prompt* una vez ejecutado el contenedor



# Creando un contenedor interactivo

15

- Otro ejemplo usando una imagen de Rocky Linux 9.5
  - El equipo anfitrión (*host*) ejecuta un SO Ubuntu Jammy v22.04
  - El contenedor funciona como una distribución Rocky Linux pero compartiendo el *kernel* del *host* que ejecuta Ubuntu
  - Virtualización por compartición de *kernel*

```
vagrant@docker:~$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.4 LTS"
vagrant@docker:~$ uname -a
Linux docker 5.15.0-100-generic #110-Ubuntu SMP Wed Feb 7 13:27:48 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
vagrant@docker:~$ 
vagrant@docker:~$ docker run --name rocky -ti rockylinux:9.3 bash
Unable to find image 'rockylinux:9.3' locally
9.3: Pulling from library/rockylinux
446f83f14b23: Pull complete
Digest: sha256:d7be1c094cc5845ee815d4632fe377514ee6ebcf8efaed6892889657e5ddaaa6
Status: Downloaded newer image for rockylinux:9.3
[root@519a1af9b567 ~]#
[root@519a1af9b567 ~]# uname -a
Linux 519a1af9b567 5.15.0-100-generic #110-Ubuntu SMP Wed Feb 7 13:27:48 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
[root@519a1af9b567 ~]#
[root@519a1af9b567 ~]# apt install nano
bash: apt: command not found
[root@519a1af9b567 ~]# dnf install nano
Rocky Linux 9 - BaseOS
Rocky Linux 9 - AppStream
Rocky Linux 9 - Extras
Dependencies resolved.

=====
 Package          Architecture      Version       Repository
 =====
 Installing:
  nano             x86_64          5.6.1-6.el9   baseos

Transaction Summary
=====
 Install 1 Package
```



# Imágenes Docker

16

- En el contenedor se ejecuta el proceso indicado por la línea de comandos
  - O el configurado en la propia imagen
- Las imágenes suelen disponer de software mínimo
- A diferencia de una VM, no incluyen *kernel* ni *drivers* (no los necesitan)

```
vagrant@docker:~$ docker run --name interactive --rm -ti ubuntu:jammy bash
root@dcd6d8f275c9:/#
root@dcd6d8f275c9:/# ps -elf
F S UID          PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root          1      0  80    0 - 1157 do_wai 08:34 pts/0
4 R root          8      1  80    0 - 1766 -        08:34 pts/0
root@dcd6d8f275c9:/#
root@dcd6d8f275c9:/# wget
bash: wget: command not found
root@dcd6d8f275c9:/# nano
bash: nano: command not found
root@dcd6d8f275c9:/#
root@dcd6d8f275c9:/# ls /boot/
root@dcd6d8f275c9:/# exit
exit
vagrant@docker:~$ ls /boot/
System.map-5.15.0-100-generic  grub          initrd.img-5.15.0-100-generic  vmlinuz
config-5.15.0-100-generic     initrd.img   initrd.img.old                  vmlinuz-5.15.0-100-generic
vagrant@docker:~$
```



# Ciclo de vida de un contenedor

17

- El ciclo de vida del contenedor está ligado al proceso que ejecuta
- Para salir del modo interactivo sin detener el contenedor, se usa la combinación ^P^Q para “demonizarlo” y que se siga ejecutando en segundo plano
  - ^P^Q = CTRL+P y CTRL+Q
- Para comprobar todos los contenedores en ejecución se puede usar el comando **docker ps**

```
vagrant@docker:~$ docker run --name interactive --rm -ti ubuntu:jammy bash
root@b5d3944fa2d0:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin
root@b5d3944fa2d0:/#
root@b5d3944fa2d0:/# vagrant@docker:~$
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
b5d3944fa2d0	ubuntu:jammy	"bash"	32 seconds ago	Up 32 seconds		interactive

```
vagrant@docker:~$
```



# Ciclo de vida de un contenedor

18

- Es posible conectarse de nuevo al contenedor
  - `docker attach <CONTAINER>`
- Para **salir y detener** un contenedor se usa `exit`

```
vagrant@docker:~$ docker attach interactive
root@b5d3944fa2d0:/# exit
exit
vagrant@docker:~$
```

- Sin embargo, su sistema de ficheros raíz (`rootfs`) no se elimina por defecto
  - Sigue existiendo en disco, pero los recursos utilizados por el contenedor (p.e. CPU, memoria) **sí son liberados, excepto** que se use la **opción `--rm`** de `docker run`
- Para mostrar **todos** los contenedores (no solo los activos):
  - `docker ps -a`

```
vagrant@docker:~$ docker run --name interactive -ti ubuntu:jammy bash
root@44216fe43ffd:/# exit
exit
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED    STATUS     PORTS      NAMES
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED    STATUS     PORTS      NAMES
44216fe43ffd   ubuntu:jammy   "bash"   8 seconds ago   Exited (0)   3 seconds ago   interactive
vagrant@docker:~$
```

Contenedor ejecutado sin la opción "`--rm`"



# Ciclo de vida de un contenedor

19

- Comandos útiles para controlar el ciclo de vida
  - Iniciar de nuevo un contenedor detenido
    - *docker start <CONTAINER>*

```
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED    STATUS          PORTS     NAMES
44216fe43ffd  ubuntu:jammy "bash"   5 minutes ago Exited (0)  5 minutes ago
vagrant@docker:~$ docker start interactive
interactive
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED    STATUS          PORTS     NAMES
44216fe43ffd  ubuntu:jammy "bash"   5 minutes ago Up 2 seconds
vagrant@docker:~$ docker attach interactive
root@44216fe43ffd:/#
```



# Ciclo de vida de un contenedor

20

- Comandos útiles para controlar el ciclo de vida
  - Iniciar de nuevo un contenedor detenido
    - *docker start <CONTAINER>*
  - Enviar una señal a un contenedor en ejecución
    - *docker kill -s <SIGNAL> <CONTAINER>*
    - Si no se especifica -s, por defecto se envía la señal SIGKILL
  - Reiniciar un contenedor
    - *docker restart <CONTAINER>*
  - Detener un contenedor en ejecución
    - *docker stop <CONTAINER>*
    - Envía la señal SIGTERM, si en 10 segundos no se detiene, le envía SIGKILL
  - Eliminar un contenedor
    - *docker rm <CONTAINER>*
    - Indicando -f es posible forzar la eliminación de un contenedor en ejecución (le envía la señal SIGKILL)



# Ciclo de vida de un contenedor

21

- Comandos útiles para controlar el ciclo de vida
  - Detener un contenedor en ejecución
    - *docker stop <CONTAINER>*
    - Envía la señal SIGTERM, si en 10 segundos no se detiene, le envía SIGKILL
  - Eliminar un contenedor
    - *docker rm <CONTAINER>*
    - Indicando -f es posible forzar la eliminación de un contenedor en ejecución (le envía la señal SIGKILL)

```
vagrant@docker:~$ docker run --name interactive -ti ubuntu:jammy bash
root@b1f9f0cdbbd0:/# vagrant@docker:~$ 
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
b1f9f0cdbbd0   ubuntu:jammy "bash"   32 seconds ago   Up 30 seconds          interactive
vagrant@docker:~$ 
vagrant@docker:~$ docker stop interactive
interactive
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
b1f9f0cdbbd0   ubuntu:jammy "bash"   About a minute ago   Exited (137) 24 seconds ago          interactive
vagrant@docker:~$ docker rm interactive
interactive
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
vagrant@docker:~$
```



# Aislamiento de un contenedor

22

- Por defecto, la entrada estándar del contenedor está cerrada
  - Hemos visto antes que es necesario usar la opción `-i` para "abrirla"
- El contenedor se encuentra aislado del resto de procesos del sistema y del resto de contenedores
- Tiene un sistema de ficheros raíz propio
- Dispone de una *stack* de red propio
- Es posible limitar la cantidad de CPU y memoria (vía `cgroups`)
  - Opciones de `docker run`: `--cpus`, `--memory...`

```
vagrant@docker:~$ echo hello | tail
hello
vagrant@docker:~$ echo hello | docker run --rm --name test ubuntu tail
vagrant@docker:~$ echo hello | docker run --rm --name test -i ubuntu tail
hello
vagrant@docker:~$ echo hello | docker run --rm --name test ubuntu ps -eaf
UID          PID      PPID   C STIME TTY          TIME CMD
root         1          0   2 12:22 ?
root         1          0   2 12:22 ?          00:00:00 ps -eaf
vagrant@docker:~$ ps -eaf | grep init
root         1          0   0 11:23 ?
root         1          0   0 11:23 ?          00:00:01 /sbin/init
vagrant     3097      1683   0 12:24 pts/0          00:00:00 grep --color=auto init
vagrant@docker:~$
```



# Ejemplo: ejecutar el servidor web Nginx

23

- Ejecutamos un contenedor en segundo plano con el servidor web Nginx
  - Opción **-d** de *docker run*
  - Nos devuelve el **identificador del contenedor** creado
- Al comprobar que el servidor se está ejecutando podemos ver los puertos que el contenedor tiene abiertos
- Si intentamos acceder al puerto 80 del servidor web obtenemos un error
  - El servidor web escucha en el puerto 80 del contenedor, no del host

```
vagrant@docker:~$ docker run -d --name web --rm nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
2f44b7a888fa: Pull complete
8b7dd3ed1dc3: Pull complete
35497dd96569: Pull complete
36664b6ce66b: Pull complete
2d455521f76c: Pull complete
dc9c4fdb83d6: Pull complete
8056d2bcf3b6: Pull complete
Digest: sha256:4c0fdcaa8b6341bfdeca5f18f7837462c80cff90527ee35ef185571e1c327beac
Status: Downloaded newer image for nginx:latest
ef154ddd52bcfdb90a4c36d86d8488c1671fdb66f46c137aab84666e04bc48f
vagrant@docker:~$
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS          PORTS     NAMES
ef154ddd52bc   nginx     "/docker-entrypoint..."   3 seconds ago   Up 2 seconds   80/tcp    web
vagrant@docker:~$
vagrant@docker:~$ curl http://localhost:80
curl: (7) Failed to connect to localhost port 80 after 0 ms: Connection refused
vagrant@docker:~$
```



# Inspeccionar contenedores e imágenes

24

- Con el comando **docker inspect** podemos comprobar la configuración de contenedores e imágenes
  - Información en formato JSON
    - Con el parámetro --format se puede parsear el JSON usando plantillas de Go
  - El contenedor hereda la configuración de la imagen
    - La configuración por defecto se puede sobreescibir con opciones de **docker run**

```
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS       PORTS     NAMES
ef154ddd52bc   nginx      "/docker-entrypoint..."   3 minutes ago  Up 3 minutes  80/tcp    web
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web
172.17.0.2
vagrant@docker:~$ curl http://172.17.0.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

El acceso a Nginx desde el host funciona si usamos directamente la IP del contenedor

Inspeccionamos el contenedor para conocer su dirección IP



# Logs de un contenedor

25

- Docker guarda los *streams* de entrada (`stdin`) y salida (`stdout`) del proceso que se ejecuta en el contenedor
- Se pueden obtener mediante el comando `docker logs`
  - Por defecto imprime por pantalla todos los *logs*
  - Se pueden obtener solo los últimos *logs*: `--tail <nlineas>`
  - Es posible visualizar los *logs* en tiempo real: `--follow | -f`

```
vagrant@docker:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ef154ddd52bc nginx "/docker-entrypoint...." 10 minutes ago Up 10 minutes 80/tcp web
vagrant@docker:~$ docker logs --tail 10 web
2024/01/18 12:27:59 [notice] 1#1: using the "epoll" event method
2024/01/18 12:27:59 [notice] 1#1: nginx/1.25.3
2024/01/18 12:27:59 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/01/18 12:27:59 [notice] 1#1: OS: Linux 5.15.0-57-generic
2024/01/18 12:27:59 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024/01/18 12:27:59 [notice] 1#1: start worker processes
2024/01/18 12:27:59 [notice] 1#1: start worker process 30
2024/01/18 12:27:59 [notice] 1#1: start worker process 31
172.17.0.1 - - [18/Jan/2024:12:31:56 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.81.0" "-"
172.17.0.1 - - [18/Jan/2024:12:36:38 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.81.0" "-"

vagrant@docker:~$
```



# Realizar cambios en un contenedor

26

- Creamos un primer contenedor en interactivo y modificamos la página por defecto de Nginx para que muestre “Hello World”
- Iniciamos un segundo contenedor en segundo plano usando la misma imagen y que ejecute un servidor Nginx
- Comprobamos su estado y obtenemos las IPs de ambos contenedores

```
vagrant@docker:~$ docker run -ti --name web1 --rm nginx bash
root@d6fb9ed75c2f:#
root@d6fb9ed75c2f:# echo "Hello World" > /usr/share/nginx/html/index.html
root@d6fb9ed75c2f:#
root@d6fb9ed75c2f:# vagrant@docker:~$
vagrant@docker:~$ docker run -d --name web2 --rm nginx
2746ef402823d81bf9eea5d3e78b61a367615d7957cf3b8bb65200ff472beace
vagrant@docker:~$ docker ps
CONTAINER ID        IMAGE       COMMAND                  CREATED             STATUS              PORTS     NAMES
2746ef402823        nginx       "/docker-entrypoint..."   2 seconds ago      Up 2 seconds          80/tcp    web2
d6fb9ed75c2f        nginx       "/docker-entrypoint..."   About a minute ago   Up About a minute   80/tcp    web1
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web1
172.17.0.2
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web2
172.17.0.3
vagrant@docker:~$
```



# Realizar cambios en un contenedor

27

- Accedemos con *curl* a ambos contenedores
  - El primer contenedor no está ejecutando el servidor Nginx!
  - En el segundo contenedor sí, pero vemos la página por defecto de Nginx
    - Una vez creado, la configuración de un contenedor es immutable
    - Excepto la configuración de red y algún otro dato como su nombre

```
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND       CREATED        STATUS          PORTS     NAMES
2746ef402823   nginx      "/docker-entrypoint..."   2 minutes ago   Up 2 minutes   80/tcp    web2
d6fb9ed75c2f   nginx      "/docker-entrypoint..."   3 minutes ago   Up 3 minutes   80/tcp    web1
vagrant@docker:~$
vagrant@docker:~$ curl http://172.17.0.2
curl: (7) Failed to connect to 172.17.0.2 port 80 after 0 ms: Connection refused
vagrant@docker:~$
vagrant@docker:~$ curl http://172.17.0.3
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

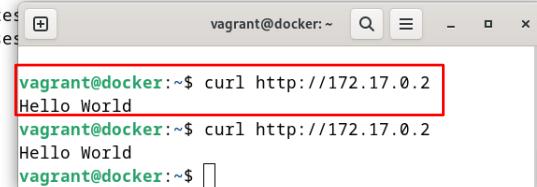


# Realizar cambios en un contenedor

28

- Comprobamos el comando que ejecutan ambos contenedores
- Ejecutamos el mismo comando en el contenedor interactivo para iniciar el servidor Nginx y accedemos al mismo: ahora sí obtenemos "Hello World"

```
vagrant@docker:~$ docker ps --no-trunc
CONTAINER ID        IMAGE               COMMAND
2746ef402823d81bf9eea5d3e78b61a367615d7957cf3b8bb65200ff472beace   nginx
d6fb9ed75c2faaecd0635a084863e05bab116bbdbab65141702f757b60e268d8    nginx
vagrant@docker:~$ docker attach web1
root@d6fb9ed75c2f:#
root@d6fb9ed75c2f:#/ # /docker-entrypoint.sh nginx -g 'daemon off;'
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates: [vagrant@docker:~] vagrant@docker:~ [Q] [-] [x]
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes: [vagrant@docker:~] vagrant@docker:~ [Q] [-] [x]
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/01/18 12:50:02 [notice] 7#7: using the "epoll" event method
2024/01/18 12:50:02 [notice] 7#7: nginx/1.25.3
2024/01/18 12:50:02 [notice] 7#7: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/01/18 12:50:02 [notice] 7#7: OS: Linux 5.15.0-57-generic
2024/01/18 12:50:02 [notice] 7#7: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024/01/18 12:50:02 [notice] 7#7: start worker processes
2024/01/18 12:50:02 [notice] 7#7: start worker process 30
2024/01/18 12:50:02 [notice] 7#7: start worker process 31
172.17.0.1 - - [18/Jan/2024:12:51:07 +0000] "GET / HTTP/1.1" 200 12 "-" "curl/7.81.0" "-"
172.17.0.1 - - [18/Jan/2024:12:51:17 +0000] "GET / HTTP/1.1" 200 12 "-" "curl/7.81.0" "-"


```

- Para guardar los cambios realizados en el contenedor debemos:
  - Crear una imagen a partir del contenedor actual
  - Crear un nuevo contenedor a partir de la nueva imagen



# Contenidos

29

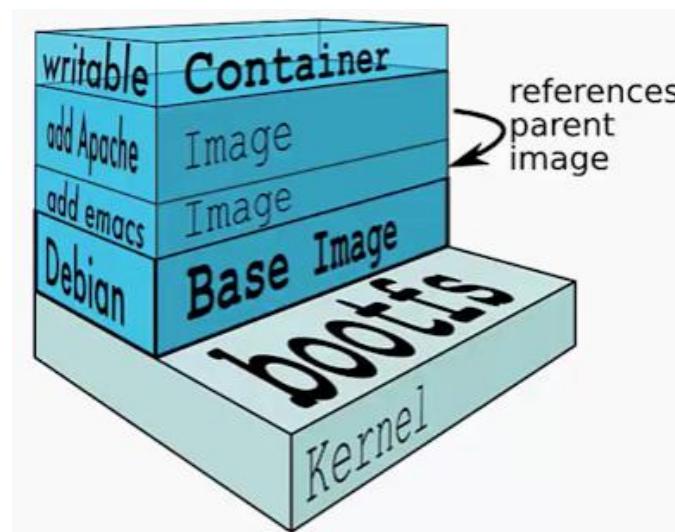
- Introducción
- Conceptos básicos
- **Imágenes**
- *Dockerfiles*
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm



# ¿Qué es una imagen?

30

- Conjunto de ficheros que forman el *rootfs* del contenedor junto con los metadatos necesarios
- Formada por **capas** superpuestas de **sólo lectura**

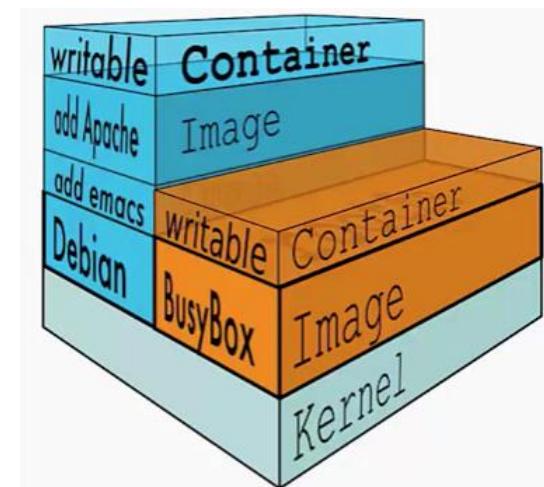
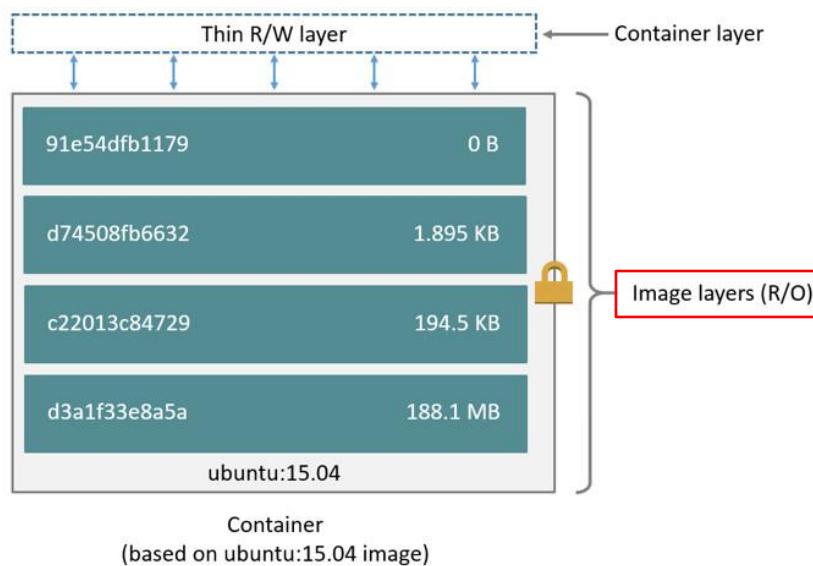




# Capas de una imagen

31

- Cada capa añade, modifica o elimina ficheros de la capa previa
  - Las imágenes **comparten capas** para optimizar disco, memoria y transferencia de datos
    - Estrategia Copy-on-Write (CoW) como optimización
  - El contenedor se ejecuta sobre un sistema de ficheros que es una **nueva capa de lectura/escritura** creada sobre la última capa de la imagen





# Creación y modificación de imágenes

32

1. A partir de un fichero tar
  - **docker import**: imagen sin metadatos
  - **docker load**: imagen con metadatos
2. A partir de un contenedor en ejecución: **docker commit**
  - Convierte en sólo lectura la capa de lectura/escritura del contenedor
  - Guarda todos los cambios hechos en el contenedor hasta ese momento
  - La nueva imagen está formada por la nueva capa y las capas previas de la imagen base
3. A partir de un fichero *Dockerfile*: **docker build**
  - Proceso reproducible y extensible de creación de contenedores
  - Forma estándar de creación de imágenes
  - Paradigma IaC



# Creación de imágenes interactiva

33

- Creamos un contenedor que modifica la página por defecto del servidor web Nginx para que muestre “Hello World”
- Comprobamos los cambios realizados por el contenedor sobre la imagen base con el comando **docker diff**
- Creamos la **nueva imagen** llamada **nginx:hello** mediante el comando **docker commit**
  - Este comando crea una *snapshot* del estado actual del contenedor

```
vagrant@docker:~$ docker run --name hello-nginx nginx bash -c "echo Hello World > /usr/share/nginx/html/index.html"
vagrant@docker:~$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND           CREATED     STATUS      PORTS     NAMES
b0ed440416a5   nginx     "/docker-entrypoint..."   4 seconds ago   Exited (0) 4 seconds ago
vagrant@docker:~$ 
vagrant@docker:~$ docker diff hello-nginx
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
vagrant@docker:~$ 
vagrant@docker:~$ docker commit hello-nginx nginx:hello
sha256:0894cfaddf23897f71a56fef08844a890203ec1efe5948f7f02c0b87afc7b0c5
vagrant@docker:~$ 
vagrant@docker:~$ docker rm hello-nginx
hello-nginx
vagrant@docker:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
vagrant@docker:~$
```

El contenedor no está en ejecución pero su *rootfs* existe en disco, ya que no fue ejecutado con la opción “*--rm*”

Nos devuelve el identificador de la imagen creada

Eliminamos el contenedor



# Creación de imágenes interactiva

34

- Podemos ver la nueva imagen usando: **docker image ls**
  - El tamaño de la nueva imagen es prácticamente el mismo puesto que solo añade sobre la imagen base un nuevo fichero (nuestro *index.html* modificado)
    - Dicho fichero se añade en una nueva capa!
  - Para ver las capas de una imagen tenemos el comando **docker history**

```
vagrant@docker:~$ docker image ls
REPOSITORY      TAG        IMAGE ID      CREATED       SIZE
nginx           hello     0894cfaddf23  16 minutes ago  187MB
ubuntu          jammy    e34e831650c1  7 days ago   77.9MB
ubuntu          latest    e34e831650c1  7 days ago   77.9MB
debian           10       e151bbc09abf  8 days ago   114MB
debian           12       a6916e41aa87  8 days ago   117MB
busybox          latest    9211bbba0dbd  4 weeks ago  4.26MB
nginx           latest    a8758716bb6a   2 months ago  187MB
vagrant@docker:~$
```

Nuestra nueva imagen



```
vagrant@docker:~$ docker history nginx:hello
IMAGE        CREATED      CREATED BY
0894cfaddf23 21 minutes ago bash -c echo Hello World > /usr/share/nginx/...
a8758716bb6a  2 months ago  CMD ["nginx" "-g" "daemon off;"]
<missing>    2 months ago  STOP SIGNAL SIGQUIT
<missing>    2 months ago  EXPOSE map[80/tcp:{}]
<missing>    2 months ago  ENTRYPOINT ["/docker-entrypoint.sh"]
<missing>    2 months ago  COPY 30-tune-worker-processes.sh /docker-ent...
<missing>    2 months ago  COPY 20-envsubst-on-templates.sh /docker-ent...
<missing>    2 months ago  COPY 15-local-resolvers.envsh /docker-entryp...
<missing>    2 months ago  COPY 10-listen-on-ipv6-by-default.sh /docke...
<missing>    2 months ago  COPY docker-entrypoint.sh / # buildkit
<missing>    2 months ago  RUN /bin/sh -c set -x  && groupadd --syst...
<missing>    2 months ago  ENV PKG_RELEASE=1~bookworm
<missing>    2 months ago  ENV NJS_VERSION=0.8.2
<missing>    2 months ago  ENV NGINX_VERSION=1.25.3
<missing>    2 months ago  LABEL maintainer=NGINX Docker Maintainers <d...
<missing>    2 months ago  /bin/sh -c #(nop)  CMD ["bash"]
<missing>    2 months ago  /bin/sh -c #(nop) ADD file:9deb26e1dbc258df4...
vagrant@docker:~$
```

En realidad, nuestra nueva imagen es 12 bytes más grande que la imagen base

Imagen base usada por el contenedor que modificó la página por defecto

Todas estas capas intermedias pertenecen a la imagen base y, por tanto, también a la nuestra



# Creación de imágenes interactiva

35

- Creamos un nuevo contenedor desde la nueva imagen creada (`nginx:hello`) y obtenemos su dirección IP
  - ¿Problema? El contenedor no se está ejecutando!
  - La nueva imagen **hereda la configuración** del contenedor, incluyendo el comando por defecto que ejecuta
- Luego creamos un nuevo contenedor con el comando necesario para poder ejecutar el servidor Nginx, obtenemos su IP y accedemos al mismo

```
vagrant@docker:~$ docker run -d --name web nginx:hello  
51bd583dac8ac3fcb147b1ffbf63c50114c2e324d42554a21dadf0dd3be9d27c  
vagrant@docker:~$  
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
51bd583dac8a	nginx:hello	/dock...r-entrypoint...."	9 seconds ago	Exited (0) 9 seconds ago		web

```
vagrant@docker:~$ docker ps -a  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
51bd583dac8a nginx:hello "/dock...r-entrypoint...." 9 seconds ago Exited (0) 9 seconds ago  
vagrant@docker:~$  
vagrant@docker:~$ docker rm web  
web  
vagrant@docker:~$ docker run -d --name web nginx:hello nginx -g 'daemon off;'  
159a5f794cf18b745406eb3bbe748fab0a1dc4c4fb89e8652031aa2c980e84  
vagrant@docker:~$  
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web  
172.17.0.2  
vagrant@docker:~$ curl http://172.17.0.2  
Hello World  
vagrant@docker:~$
```



# Creación de imágenes interactiva

36

- La aproximación seguida en el ejemplo previo para crear una imagen a partir de una sesión interactiva en la que se realizan todos los cambios necesarios en el contenedor presenta varios problemas:
  - Es un proceso no automático y farragoso de replicar
  - Cierta información como las variables de entorno se pierden
  - Es fácil incluir cambios innecesarios en la imagen
  - Imágenes “opacas”
- Mejor solución siguiendo el paradigma IaC
  - Uso de **Dockerfiles** y comando **docker build**
  - El siguiente apartado del tutorial se centrará en este aspecto



# Eliminación de imágenes

37

- Eliminar una imagen: **docker image rm <image>**
  - La opción **-f** permite forzar la eliminación si hay algún contenedor que todavía referencia la imagen

```
vagrant@docker:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1dc6cdac6d5d nginx:hello "/docker-entrypoint...." 3 minutes ago Up 3 minutes 80/tcp web
vagrant@docker:~$ docker stop web
web
vagrant@docker:~$ docker image rm nginx:hello
Error response from daemon: conflict: unable to remove repository reference "nginx:hello" (must force) - container 1dc6cdac6d5d is using its referenced image 0894cfaddf23
vagrant@docker:~$ docker image rm -f nginx:hello
Untagged: nginx:hello
Deleted: sha256:0894cfaddf23897f71a56fef08844a890203ec1efe5948f7f02c0b87afc7b0c5
vagrant@docker:~$ 
vagrant@docker:~$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1dc6cdac6d5d 0894cfaddf23 "/docker-entrypoint...." 4 minutes ago Exited (0) 38 seconds ago
vagrant@docker:~$ docker rm web
web
vagrant@docker:~$
```



# Repositorios de imágenes

38

- Cuando creamos una imagen como en los ejemplos anteriores, esta se **almacena en local (en el host)**
  - El comando **docker image ls** permite listarlas (también **docker images**)

```
vagrant@docker:~$ docker image ls
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
ubuntu          jammy        e34e831650c1  7 days ago   77.9MB
ubuntu          latest       e34e831650c1  7 days ago   77.9MB
debian          10           e151bbc09abf  8 days ago   114MB
debian          12           a6916e41aa87  8 days ago   117MB
busybox         latest       9211bbaa0dbd  4 weeks ago  4.26MB
nginx           latest       a8758716bb6a  2 months ago 187MB
vagrant@docker:~$ _
```

- Para distribuir una imagen creada en local a otra máquina distinta:
  - Podemos empaquetarla en un fichero con **docker save** y transferirla (por la red, pendrive, etc)
  - Podemos almacenarla en un **registro de imágenes** (Docker Registry) y obtenerla desde ahí
    - Comandos: **docker push** y **docker pull**



# Registros Docker

39

- Un registro Docker (Docker Registry) es un repositorio de imágenes
  - El registro asignado a una imagen está definido en su nombre
- El registro público se llama **Docker Hub**
  - <https://hub.docker.com/>
  - Las imágenes públicas son gratuitas
  - Solo se paga por acceso a un repositorio privado
- Es posible instalar y gestionar registros propios
  - Docker Trusted Registry (de pago)
    - <https://dockerlabs.collabnix.com/beginners/dockertrustedregistry.html>
  - Docker Registry (*open source*)
    - <https://docs.docker.com/registry>
    - Proyecto donado a la *Cloud Native Computing Foundation* (CNCF) y renombrado como *Distribution*
      - <https://github.com/distribution/distribution>
    - Docker Hub es una implementación que cumple la especificación de *Distribution*



# Guardar una imagen en Docker Hub

40

- Es posible subir una imagen a un registro
  - ***docker push <name[:TAG]>***
- Intentamos subir a Docker Hub nuestra nueva imagen `nginx:hello`
  - Error! No tenemos permisos para subir una imagen
  - Debemos tener una cuenta creada en Docker Hub y loguearnos previamente con ***docker login***
  - Sí que podríamos subir la imagen a un registro privado (de tenerlo)
    - Usaríamos ***docker pull*** para obtener/descargar una imagen
    - ***docker run*** hace un ***pull automático*** cuando la imagen no está en local

```
vagrant@docker:~$ docker push nginx:hello
The push refers to repository [docker.io/library/nginx]
bf0a17446bf7: Preparing
009507b85609: Preparing
fbcc9bc44d3e: Preparing
b4ad47845036: Preparing
eddcd06e5ef9: Preparing
b61d4b2cd2da: Waiting
b6c2a8d6f0ac: Waiting
571ade696b26: Waiting
denied: requested access to the resource is denied
vagrant@docker:~$
```



# Contenidos

41

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes de contenedores
- Almacenamiento
- Docker Compose
- Docker Swarm



# ¿Qué es un *Dockerfile*?

42

- Es un fichero de texto plano que contiene todos los pasos necesarios (~instrucciones) para **crear una imagen** Docker
  - Proporciona un mecanismo **automático** y **reproducible** para la creación de imágenes
- Los pasos del *Dockerfile* se ejecutan **secuencialmente**
  - Las instrucciones del *Dockerfile* que modifican el *rootfs* añaden **nuevas capas** a la imagen que pueden contribuir a aumentar su tamaño
    - Aquellas instrucciones que no modifican el *rootfs* (solo añaden metadatos o configuraciones) no aumentan el tamaño ya que crean capas intermedias con tamaño de 0 bytes
- Para generar una imagen a partir de un fichero *Dockerfile* se usa el comando ***docker build***



# Instrucciones básicas

43

- **FROM:** indica la imagen base desde la que se parte
  - Primera instrucción del *Dockerfile* (exceptuando #comentarios iniciales)
    - echo "FROM Ubuntu:jammy" >> *Dockerfile*
- **RUN:** ejecuta el comando indicado
  - Todos los comandos deben ser **no interactivos**
    - echo "RUN apt-get install -y wget unzip" >> *Dockerfile*
  - Puede haber múltiples instrucciones RUN
  - Una instrucción RUN puede modificar ficheros de la imagen, creando una nueva capa



# Creando una imagen

44

## ● Comando **docker build**

- Se le pasa como parámetro el directorio donde está el *Dockerfile*
- El parámetro **-t** permite indicar el nombre y tag de la imagen siguiendo el formato: **nombre[:tag]**

```
vagrant@docker:~$ ls  
ejemplo  
vagrant@docker:~$ cat ejemplo/Dockerfile  
FROM ubuntu:jammmy  
RUN apt-get update  
RUN apt-get install -y wget unzip  
vagrant@docker:~$  
vagrant@docker:~$ docker build -t myubuntu ejemplo  
[+] Building 10.6s (7/7) FINISHED ← Instrucciones del Dockerfile  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 108B  
=> [internal] load .dockerignore  
=> => transferring context: 2B  
=> [internal] load metadata for docker.io/library/ubuntu:jammmy  
=> CACHED [1/3] FROM docker.io/library/ubuntu:jammmy ← Ha tardado casi 11 segundos  
en construir la imagen  
=> [2/3] RUN apt-get update  
=> [3/3] RUN apt-get install -y wget unzip  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:4f12cd956c6ac14277382416e8ac88855b3593d443766ae21f7a7e8fc96fb42d ← Las instrucciones se ejecutan secuencialmente. El  
primer paso sería descargar del Docker Hub la  
imagen base. Este paso está en caché pues ya se  
encuentra la imagen en local por ejemplos previos  
=> => naming to docker.io/library/myubuntu  
vagrant@docker:~$
```

Identificador de la nueva imagen



# Caché

45

- Docker mantiene una cache de los diferentes pasos que se ejecutan
  - Antes de cada paso, se comprueba si la misma secuencia ya fue ejecutada
  - La caché solo analiza las líneas del *Dockerfile*
  - Si se vuelve a crear la imagen del ejemplo previo, la instrucción “*RUN apt-get update*” no se ejecutaría aunque los repositorios se hayan actualizado
  - Se puede ignorar la caché con la opción **--no-cache** de **docker build**

```
vagrant@docker:~$ docker build -t myubuntu ejemplo
[+] Building 0.0s (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 108B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/ubuntu:jammmy
=> [1/3] FROM docker.io/library/ubuntu:jammmy
=> CACHED [2/3] RUN apt-get update
=> CACHED [3/3] RUN apt-get install -y wget unzip
=> exporting to image
=> => exporting layers
=> => writing image sha256 4f12cd956c6ac14277382416e8ac88855b3593d443766ae21f7a7e8fc96fb42d
=> => naming to docker.io/library/myubuntu
vagrant@docker:~$
```

El identificador es el mismo puesto que  
no se ha creado una nueva imagen



# Historial de una imagen

46

- El comando **docker history** muestra las capas de una imagen
  - Cada capa se corresponde con una instrucción del *Dockerfile*
  - Los parámetros de una instrucción RUN se pasan al comando `"/bin/sh -c"`
  - La instrucción FROM se sustituye por las instrucciones de la imagen base
  - La columna "SIZE" muestra el tamaño de cada capa
    - Algunas no afectan al tamaño final (son instrucciones de configuración o que añaden metadatos)

```
vagrant@docker:~$ docker history myubuntu
IMAGE          CREATED     CREATED BY
4f12cd956c6a  14 minutes ago   RUN /bin/sh -c apt-get install -y wget unzip...
<missing>      14 minutes ago   RUN /bin/sh -c apt-get update # buildkit
<missing>      7 days ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]
<missing>      7 days ago      /bin/sh -c #(nop) ADD file:c646150c866c8b5ec...
<missing>      7 days ago      /bin/sh -c #(nop) LABEL org.opencontainers...
<missing>      7 days ago      /bin/sh -c #(nop) LABEL org.opencontainers...
<missing>      7 days ago      /bin/sh -c #(nop) ARG LAUNCHPAD_BUILD_ARCH
<missing>      7 days ago      /bin/sh -c #(nop) ARG RELEASE
vagrant@docker:~$
```



# Instrucción WORKDIR

47

- La instrucción **WORKDIR** permite establecer el **directorio de trabajo** para otras instrucciones (RUN, CMD, ENTRYPOINT, COPY, ADD)

- La instrucción “RUN cd /opt” solo tendría efecto dentro de dicha instrucción, ya que al no modificar ningún fichero no queda reflejada en la imagen resultado
- Otra instrucción RUN que se ejecute a continuación de la anterior partirá del directorio de trabajo por defecto (el directorio raíz /)
  - “RUN cd /opt”
  - “RUN cp file /data” asume que el fichero “file” existe en / (y no en /opt)
- Solución: **anidar comandos en una única instrucción RUN con &&**
  - “RUN cd /opt && cp file /data”
  - Si un comando falla, los siguientes no se ejecutan
  - También se pueden dividir las instrucciones en diferentes líneas con \
  - **Anidar comandos reduce el número de capas de la imagen y su tamaño**
- Como alternativa, también se puede cambiar el directorio de trabajo con WORKDIR
  - “WORKDIR /opt”
  - “RUN cp file /data”
  - Sucesivas instrucciones WORKDIR se sobrescriben entre sí
  - También sobrescribe el WORKDIR de la imagen base (o imagen padre)



# Instrucción ENV

48

- Una variable de entorno que se define en una instrucción RUN solo está disponible de forma local para dicha instrucción
  - `RUN export REDIS_HOME=/opt/redis && cp $REDIS_HOME/file /data`
- Si el contenedor necesitase el valor de la variable de entorno en tiempo de ejecución, es necesario definirla con una instrucción **ENV**
  - Cada instrucción ENV crea una nueva capa pero de 0 bytes
- También es posible definir variables de entorno en tiempo de ejecución usando el parámetro `-e VAR=VALUE` del comando `docker run`



# Instrucción COPY

49

- La instrucción **COPY** permite copiar ficheros locales (del host) al *rootfs* de la imagen resultante
- Los ficheros a copiar deben estar dentro del directorio donde se encuentra el *Dockerfile*
  - Dicho directorio se conoce como el “*build context*”
  - También es posible copiar directorios
- Esta instrucción creará una nueva capa de la imagen y aumentará su tamaño en función de lo que se copie
- Los permisos se mantienen, pero el propietario y el grupo se cambian a UID y GID 0 (*root*)
  - La opción `--chown` de COPY permite establecer el propietario y grupo
  - La opción `--chmod` también permite cambiar los permisos



# Instrucción COPY

50

- Ejemplo

- Creamos una nueva imagen de Ubuntu que incluye el fichero *file*

```
vagrant@docker:~$ cat ejemplo/file
Hello World
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
RUN apt-get update
RUN apt-get install -y wget unzip
ENV DEST=/opt
COPY file $DEST
vagrant@docker:~$ docker build -t myubuntu ejemplo
[+] Building 0.0s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 138B
=> [internal] load metadata for docker.io/library/ubuntu:jammy
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/4] FROM docker.io/library/ubuntu:jammy
=> [internal] load build context
=> => transferring context: 25B
=> CACHED [2/4] RUN apt-get update
=> CACHED [3/4] RUN apt-get install -y wget unzip
=> CACHED [4/4] COPY file /opt
=> exporting to image
=> => exporting layers
=> => writing image sha256:69c938957d6770aadd5c6636d660547e8ce57fc862b2e0464f369ce368a2acf
=> => naming to docker.io/library/myubuntu
vagrant@docker:~$
```



# Instrucción COPY

51

- Ejemplo
  - Mostramos el historial de la nueva imagen

```
vagrant@docker:~$ docker history myubuntu
IMAGE          CREATED      CREATED BY
69c938957d67  57 minutes ago    COPY file /opt # buildkit        12B
<missing>      57 minutes ago    ENV DEST=/opt                0B
<missing>      About an hour ago   RUN /bin/sh -c apt-get install -y wget unzip...
<missing>      About an hour ago   RUN /bin/sh -c apt-get update # buildkit      59MB
<missing>      4 months ago       /bin/sh -c #(nop)  CMD ["/bin/bash"]        0B
<missing>      4 months ago       /bin/sh -c #(nop) ADD file:ebe009f86035c175b...  77.9MB
<missing>      4 months ago       /bin/sh -c #(nop) LABEL org.opencontainers....  0B
<missing>      4 months ago       /bin/sh -c #(nop) LABEL org.opencontainers....  0B
<missing>      4 months ago       /bin/sh -c #(nop) ARG LAUNCHPAD_BUILD_ARCH     0B
<missing>      4 months ago       /bin/sh -c #(nop) ARG RELEASE            0B
vagrant@docker:~$
```

- Creamos un contenedor usando la nueva imagen y mostramos el fichero
  - Fíjate en el tamaño, en el propietario y el grupo del fichero

```
vagrant@docker:~$ docker run -ti --rm myubuntu
root@9983a8f81224:/# ls -l /opt/
total 4
-rw----- 1 root root 12 Jan 31 09:30 file
root@9983a8f81224:/#
root@9983a8f81224:/# echo $DEST/
/opt/
root@9983a8f81224:/# cat /opt/file
Hello World
root@9983a8f81224:/# exit
exit
vagrant@docker:~$ ls -l ejemplo/file
-rw----- 1 vagrant vagrant 12 Jan 31 09:30 ejemplo/file
vagrant@docker:~$
```



# Instrucción CMD

52

- La instrucción **CMD** define el **comando por defecto** que ejecuta un contenedor cuando **no** se especifica ninguno al final del comando `docker run`
- Sucesivos CMD se sobrescriben entre sí (también el de la imagen base)

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
RUN apt-get update
RUN apt-get install -y wget unzip
ENV DEST=/opt
COPY file $DEST
CMD cat /opt/file
```

Se ha omitido el proceso de creación de la imagen *myubuntu* tras añadir la instrucción CMD

```
vagrant@docker:~$ docker run --rm myubuntu
Hello World
vagrant@docker:~$
```

```
vagrant@docker:~$ docker history myubuntu
IMAGE          CREATED      CREATED BY          SIZE
5d05a6144dd8  About an hour ago  CMD ["bin/sh" "-c" "cat /opt/file"]  0B
<missing>      About an hour ago  COPY file /opt # buildkit        12B
<missing>      About an hour ago  ENV DEST=/opt                  0B
<missing>      About an hour ago  RUN /bin/sh -c apt-get install -y wget unzip...  4.31MB
<missing>      About an hour ago  RUN /bin/sh -c apt-get update # buildkit    59MB
<missing>      4 months ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]  0B
<missing>      4 months ago   /bin/sh -c #(nop) ADD file:ebe009f86035c175b...  77.9MB
<missing>      4 months ago   /bin/sh -c #(nop) LABEL org.opencontainers....  0B
<missing>      4 months ago   /bin/sh -c #(nop) LABEL org.opencontainers....  0B
<missing>      4 months ago   /bin/sh -c #(nop) ARG LAUNCHPAD_BUILD_ARCH  0B
<missing>      4 months ago   /bin/sh -c #(nop) ARG RELEASE            0B
vagrant@docker:~$
```

Realmente se ha ejecutado */bin/sh -c "cat /opt/file"*

CMD de la imagen base, que es sobreescrito



# Instrucción ENTRYPPOINT

53

- La instrucción **ENTRYPOINT** define un **comando base** al que se le pasan como parámetros los argumentos indicados al final de *docker run*
  - Si no se especifica un ENTRYPPOINT, por defecto es: **/bin/sh -c** ← En Windows sería: **cmd /s /c**
  - Si no se especifica ningún argumento al final del comando *docker run*, se pasa CMD como parámetro por defecto al ENTRYPPOINT
    - Es decir, se ejecutaría: **/bin/sh -c CMD** (ver ejemplo previo)
- Una instrucción ENTRYPPOINT sobrescribe el de la imagen base
  - Pero se puede invocar su ENTRYPPOINT previamente (si fuese necesario)
  - Con *docker history* podemos ver el ENTRYPPOINT de una imagen
- Se puede sobrescribir en *docker run* con la opción **--entrypoint**
- Ejemplo (sin mostrar de nuevo el proceso de creación de la imagen):

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammmy
RUN apt-get update
RUN apt-get install -y wget unzip
ENV DEST=/opt
COPY file $DEST
CMD cat /opt/file
ENV NAME=Rober
ENTRYPOINT echo "Hello, $NAME"
vagrant@docker:~$ docker run --rm myubuntu
Hello, Rober
vagrant@docker:~$
```

No le pasamos argumentos en *docker run*  
(myubuntu es el nombre de la imagen)



# Argumentos de RUN, CMD, ENTRYPOINT

54

- Se pueden especificar de dos maneras
  - En formato **shell**: <instruction> <command>
    - El comando se ejecuta como un proceso secundario de una **shell**:
      - `/bin/sh -c <command>` ← La shell no transmite señales. Esto significa que el comando que se ejecuta en el contenedor no puede detectar señales del SO como SIGTERM y SIGKILL y responder a ellas correctamente.  
El comando no es el proceso principal con PID 1
    - Ejemplos
      - `RUN apt-get install -y python3`
        - Se ejecutaría: `/bin/sh -c "apt-get install -y python3"`
        - `CMD echo "Hello world"`
        - `ENTRYPOINT echo "Hello world"`
  - En formato **exec**: <instruction> [“command”, “param1”, ...]
    - El comando se ejecuta directamente (sin una **shell**) ← Permite ejecutar el comando como el proceso principal (PID 1) en el contenedor y recibir señales del SO
    - Ejemplos:
      - `RUN ["apt-get", "install", "-y", "python3"]`
      - `CMD ["/bin/echo", "Hello world"]`
      - `ENTRYPOINT ["/bin/echo", "Hello world"]`



# Argumentos de RUN, CMD, ENTRYPOINT

55

- Se pueden especificar de dos maneras
  - En formato **shell**: <instruction> <command>
  - En formato **exec**: <instruction> ["command", "param1", ...]

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
ENV NAME=Rober
ENTRYPOINT echo "Hello, $NAME"
vagrant@docker:~$ docker run --rm myubuntu
Hello, Rober
vagrant@docker:~$
```

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
ENV NAME=Rober
ENTRYPOINT ["/bin/echo", "Hello, $NAME"]
vagrant@docker:~$ docker run --rm myubuntu
Hello, $NAME
vagrant@docker:~$
```

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
ENV NAME=Rober
ENTRYPOINT ["/bin/sh", "-c", "echo Hello, $NAME"]
vagrant@docker:~$ docker run --rm myubuntu
Hello, Rober
vagrant@docker:~$
```



# Argumentos de RUN, CMD, ENTRYPOINT

56

- ENTRYPOINT ignora el CMD y cualquier argumento que se pase al final del comando *docker run* cuando se usa el formato *shell*

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammmy
CMD ["World"]
ENTRYPOINT ["/bin/echo", "Hello"]
vagrant@docker:~$ docker run --rm myubuntu
Hello World
vagrant@docker:~$ docker run --rm myubuntu Rober
Hello Rober
vagrant@docker:~$
```

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammmy
CMD ["World"]
ENTRYPOINT echo "Hello"
vagrant@docker:~$ docker run --rm myubuntu
Hello
vagrant@docker:~$ docker run --rm myubuntu Rober
Hello
vagrant@docker:~$
```



# Proceso inicial de un contenedor

57

- Docker está orientado a la ejecución de un proceso por contenedor
- El proceso con el que se crea el contenedor tiene PID=1, siendo este proceso:
  - El comando definido con ENTRYPOINT | CMD cuando se usa formato exec
  - El proceso *shell* que ejecuta el comando definido con ENTRYPOINT | CMD cuando se usa formato *shell*
  - O el comando especificado en docker run cuando no se definen
- Este proceso tiene un rol especial
  - Define el ciclo de vida del contenedor
  - Recibe las señales enviadas con docker stop y docker kill
  - Su salida estándar y de error es capturada por el demonio Docker



# Instrucción EXPOSE

58

- La instrucción **EXPOSE** permite especificar en qué puerto(s) un contenedor escucha peticiones **pero sin mapearlos** a puertos del host
  - Se considera un tipo de “documentación” entre la persona que crea el *Dockerfile* y/o genera la imagen y la persona que ejecuta el contenedor
  - Ejemplo: EXPOSE 80/tcp
- La opción **-P** de *docker run* mapea (**publica**) todos los puertos declarados con EXPOSE a puertos aleatorios del host
- La opción **-p <hostPort:containerPort>** de *docker run* publica el puerto del contenedor especificado a un puerto concreto del host
- El comando **docker ps** muestra los puertos y *mappings* existentes

```
vagrant@docker:~$ docker run -d --name web --rm -P nginx  
98bc4a237e0e511d41d82ed68a759b8f790fa954c2dc46d366861729d7b8bdf1
```

```
vagrant@docker:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
98bc4a237e0e	nginx	"/docker-entrypoint...."	4 seconds ago	Up 4 seconds	0.0.0.0:32768->80/tcp, :::32768->80/tcp	web
vagrant@docker:~\$						

Puerto 32768 del host mapeado al puerto 80 del contenedor



# Instrucción HEALTHCHECK

59

- Permite especificar un comando para detectar si el contenedor se considera "vivo" (en funcionamiento)
  - `HEALTHCHECK [OPTIONS] CMD <command>`
  - `HEALTHCHECK NONE`
    - Deshabilita cualquier *healthcheck* heredado de la imagen base
- El estado del contenedor se deriva del resultado del comando
  - 0: *success* -> el contenedor está vivo
  - 1: *unhealthy* -> el contenedor no está funcionando correctamente
- Ejemplo: comprobando si un servidor web responde
  - `HEALTHCHECK CMD curl --fail http://127.0.0.1:80 || exit 1`
- Opciones
  - `--interval=DURATION` (30 segundos por defecto)
  - `--timeout=DURATION` (30 segundos por defecto)
  - `--retries=N` (3 por defecto)



# Instrucción ADD

60

- La instrucción **ADD** extiende la funcionalidad de COPY ya que también permite obtener ficheros remotos desde una URL o de un repositorio Git
  - Los ficheros siempre se descargan antes de comprobar si han cambiado
  - Los ficheros descargados tienen permisos 600 y UID/GID 0
  - Soporta las opciones --chown y --chmod como COPY
- Ejemplo:
  - ADD <https://myapp.com/app.jar> /opt/myapp
- ADD también permite descomprimir archivos gzip, xz y tar locales, pero no ficheros remotos
  - ADD fichero.zip /opt/myapp



# Instrucción USER

61

- La instrucción **USER** cambia el identificador de usuario (UID) y/o grupo (GID)
  - `USER <user>[:<group>]`
  - `USER <UID>[:<GID>]`
- El usuario especificado se utiliza para las instrucciones RUN y, en tiempo de ejecución, ejecuta los comandos ENTRYPOINT y CMD
  - Por defecto se ejecutan como *root* (UID 0)
- Ejemplos:
  - `USER rober`
  - `USER 1000`
  - `USER 1000:nogroup`
  - `USER 1000:1010`
- El usuario o grupo especificados **deben existir**
- Recomendable su uso desde el punto de vista de la seguridad
- Es posible cambiar múltiples veces de usuario en un *Dockerfile*



# Buenas prácticas (I)

62

- Aprovechar la caché de Docker
  - El orden de las instrucciones es relevante
  - Comandos no variables mejor al principio del *Dockerfile*
  - Copia de ficheros que se modifican con frecuencia o configuraciones cambiantes mejor hacia el final
- Minimizar el número de capas
  - Tener en cuenta que algunas las instrucciones no aumentan el tamaño
  - Las instrucciones pueden ocupar varias líneas usando \
  - Encontrar balance entre legibilidad y número de instrucciones
- Minimizar el tamaño de la imagen
  - Instalar el mínimo software necesario
  - Eliminar todo lo que no sea necesario
    - Código fuente una vez que ha sido compilado, archivos comprimidos tras ser descomprimidos, dependencias innecesarias, caché de los gestores de paquetes...
    - **OJO con las capas!** Por ejemplo: descomprimir un fichero y eliminar el comprimido en instrucciones RUN distintas no va a reducir el tamaño!



# Buenas prácticas (II)

63

- Cuando sea posible, partir de una imagen oficial como base
- Usar imágenes base lo más ligeras posible
- Es preferible definir las instrucciones CMD y ENTRYPOINT con formato exec
- Usar siempre puertos por defecto en EXPOSE
- Usar USER siempre que sea posible
- Especificar siempre versiones concretas de imágenes y software



# Builds Multi-Stage

64

- La creación de imágenes en múltiples etapas permite optimizar su tamaño al mismo tiempo de poder tener *Dockerfiles* más fáciles de leer y mantener
  - <https://docs.docker.com/build/building/multi-stage/>
- Permiten usar múltiples instrucciones FROM en un *Dockerfile*
  - Cada instrucción FROM puede utilizar una imagen base diferente y cada una de ellas comienza una nueva etapa de la construcción
  - Es posible **copiar selectivamente artefactos de una etapa a otra**, (opción --from de la instrucción COPY) **dejando atrás todo lo que no se desea en la imagen final**
  - Ejemplo típico de uso: tener dos etapas separadas, una para compilar el código y otra donde copiamos el binario compilado previamente en la imagen final
- Ejemplo: creación de una imagen que ejecuta el siguiente código C

```
vagrant@docker:~$ cat multi-stage/main.c
#include <stdio.h>

void main(int argc, char *argv[])
{
    int i;
    printf("Program name and arguments:\n");
    for (i = 0; i < argc; i++)
    {
        // Printing all the Arguments
        printf("%s\n", argv[i]);
    }
}
vagrant@docker:~$
```



# Builds Multi-Stage

65

- Ejemplo: creación de la imagen `myimage` sin *multi-stage*

```
vagrant@docker:~$ cat multi-stage/Dockerfile
FROM debian:bullseye-slim
```

ENV APP\_DIR=/opt/app *Dockerfile "naive" sin multi-stage*

WORKDIR \$APP\_DIR

COPY main.c \$APP\_DIR

RUN apt-get update && apt-get -y install gcc

RUN gcc -o myapp main.c

RUN rm main.c

RUN apt-get purge -y --autoremove gcc && apt-get clean all

ENTRYPOINT ["/opt/app/myapp"]

CMD ["argument1"]

```
vagrant@docker:~$
```

Intentos “fallidos” de minimizar el tamaño

Sobre el tamaño de la imagen base (unos 80 MB) estamos sumando casi 190 MB en total

```
vagrant@docker:~$ docker history myimage
IMAGE          CREATED      CREATED BY                               SIZE
28584d53822d  4 minutes ago  CMD ["argument1"]                   0B
<missing>      4 minutes ago  ENTRYPPOINT ["/opt/app/myapp"]       0B
<missing>      4 minutes ago  RUN /bin/sh -c apt-get purge -y --autoremove...  1.87MB
<missing>      4 minutes ago  RUN /bin/sh -c rm main.c # buildkit        0B
<missing>      4 minutes ago  RUN /bin/sh -c gcc -o myapp main.c # buildkit  16.6kB
<missing>      4 minutes ago  RUN /bin/sh -c apt-get update && apt-get -y ...  187MB
<missing>      4 minutes ago  COPY main.c /opt/app # buildkit           202B
<missing>      4 minutes ago  WORKDIR /opt/app                         0B
<missing>      4 minutes ago  ENV APP_DIR=/opt/app                      0B
<missing>      8 days ago   /bin/sh -c #(nop)  CMD ["bash"]             0B
<missing>      8 days ago   /bin/sh -c #(nop) ADD file:bd961ef3fd78ceb8c...  80.6MB
vagrant@docker:~$
```



# Builds Multi-Stage

66

- Ejemplo: creación de la imagen **myimage-slim** sin **multi-stage**

```
vagrant@docker:~$ cat multi-stage/Dockerfile
FROM debian:bullseye-slim
```

*Dockerfile "optimizado" sin multi-stage*

```
ENV APP_DIR=/opt/app
WORKDIR $APP_DIR
COPY main.c $APP_DIR
RUN apt-get update \
    && apt-get -y install gcc \
    && gcc -o myapp main.c \
    && rm main.c \
    && apt-get purge -y --autoremove gcc \
    && apt-get clean all
```

```
ENTRYPOINT ["/opt/app/myapp"]
CMD ["argument1"]
vagrant@docker:~$
```

Ahora aumentamos solo en 20.1 MB el tamaño de la imagen base

```
vagrant@docker:~$ docker history myimage-slim
IMAGE          CREATED      CREATED BY                               SIZE
70a85c8f24ad  5 minutes ago  CMD ["argument1"]                   0B
<missing>      5 minutes ago  ENTRYPOINT ["/opt/app/myapp"]       0B
<missing>      5 minutes ago  RUN /bin/sh -c apt-get update && apt-get -y...  20.1MB
<missing>      8 minutes ago  COPY main.c /opt/app # buildkit        202B
<missing>      8 minutes ago  WORKDIR /opt/app                         0B
<missing>      8 minutes ago  ENV APP_DIR=/opt/app                      0B
<missing>      8 days ago   /bin/sh -c #(nop)  CMD ["bash"]           0B
<missing>      8 days ago   /bin/sh -c #(nop) ADD file:bd961ef3fd78ceb8c...  80.6MB
vagrant@docker:~$
```



# Builds Multi-Stage

67

- Ejemplo: creación de la imagen **myimage-multi**

```
vagrant@docker:~$ cat multi-stage/Dockerfile
FROM debian:bullseye-slim AS BUILD

ENV APP_DIR=/opt/app
WORKDIR $APP_DIR
COPY main.c $APP_DIR
RUN apt-get update && apt-get -y install gcc
RUN gcc -o myapp main.c
```

**Dockerfile multi-stage:** Etapa BUILD de compilación del código.  
No es necesario hacer "limpieza"

```
FROM debian:bullseye-slim

COPY --from=BUILD /opt/app/myapp /opt/app/myapp
ENTRYPOINT ["/opt/app/myapp"]
CMD ["argument1"]

vagrant@docker:~$
```

**Dockerfile multi-stage:** Segunda etapa y final donde copiamos el binario resultante que ejecutará el contenedor, pues es lo único que necesita. Fíjate en el uso de la opción **--from** en COPY

```
vagrant@docker:~$ docker history myimage-multi
IMAGE          CREATED      CREATED BY
ed90d86dceef  40 seconds ago  CMD ["argument1"]
<missing>      40 seconds ago  ENTRYPOINT ["/opt/app/myapp"]
<missing>      40 seconds ago  COPY /opt/app/myapp /opt/app/myapp # buildkit
<missing>      8 days ago    /bin/sh -c #(nop)  CMD ["bash"]
<missing>      8 days ago    /bin/sh -c #(nop) ADD file:bd961ef3fd78ceb8c...
vagrant@docker:~$
```

Solo aumentamos en 16 KB el tamaño de la imagen base, justo lo que ocupa el binario

	SIZE
ed90d86dceef	0B
<missing>	0B
<missing>	16.6kB
<missing>	0B
<missing>	80.6MB





# Builds Multi-Stage

68

- Comparativa de los tamaños

```
vagrant@docker:~$ docker image ls
REPOSITORY      TAG          IMAGE ID       CREATED        SIZE
myimage-multi   latest        ed90d86dceef  3 minutes ago  80.6MB
myimage-slim    latest        70a85c8f24ad  37 minutes ago 101MB
myimage         latest        28584d53822d  39 minutes ago 270MB
ubuntu          jammy        e34e831650c1  7 days ago   77.9MB
ubuntu          latest        e34e831650c1  7 days ago   77.9MB
debian          10           e151bbc09abf  8 days ago   114MB
debian          bullseye-slim  8d18562eded9  8 days ago   80.6MB
debian          bullseye     a07ba40fa4ab  8 days ago   124MB
debian          12           a6916e41aa87  8 days ago   117MB
busybox         latest        9211bbaa0dbd  4 weeks ago  4.26MB
alpine          3.19         f8c20f8bbcb6  6 weeks ago  7.38MB
nginx           latest        a8758716bb6a  2 months ago 187MB
vagrant@docker:~$
```

3.3 veces menos tamaño



El tamaño de la imagen base usada marca el **mínimo posible** (80.6 MB). Por eso también es muy importante usar imágenes base ligeras. Fíjate la diferencia entre la versión *slim* y la normal de Debian. **Es posible usar imágenes base aún más pequeñas en la etapa final, como las basadas en una distribución Alpine Linux**



# Builds Multi-Stage

69

- Ejemplo: creación de la imagen **myimage-multi-opt** usando Alpine Linux

```
vagrant@docker:~$ cat multi-stage/Dockerfile
FROM debian:bullseye-slim AS BUILD

ENV APP_DIR=/opt/app
WORKDIR $APP_DIR
COPY main.c $APP_DIR
RUN apt-get update && apt-get -y install gcc
RUN gcc -o myapp main.c
```

```
FROM alpine:3.19

RUN apk add libc6-compat
COPY --from=BUILD /opt/app/myapp /opt/app/myapp
ENTRYPOINT ["/opt/app/myapp"]
CMD ["argument1"]
```

```
vagrant@docker:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myimage-multi-opt	latest	584e1a4c622a	33 seconds ago	9.73MB
myimage-multi	latest	ed90d86dceef	11 minutes ago	80.6MB
myimage-slim	latest	70a85c8f24ad	45 minutes ago	101MB
myimage	latest	28584d53822d	47 minutes ago	270MB
ubuntu	jammy	e34e831650c1	7 days ago	77.9MB
ubuntu	latest	e34e831650c1	7 days ago	77.9MB
debian	10	e151bbc09abf	8 days ago	114MB
debian	bullseye-slim	8d18562eded9	8 days ago	80.6MB
debian	bullseye	a07ba40fa4ab	8 days ago	124MB
debian	12	a6916e41aa87	8 days ago	117MB
busybox	latest	9211bbba0dbd	4 weeks ago	4.26MB
alpine	3.19	f8c20f8bbcb6	6 weeks ago	7.38MB
nginx	latest	a8758716bb6a	2 months ago	187MB

La mayoría de las distribuciones GNU/Linux se basan en la biblioteca GNU libc (**glibc**). Alpine Linux sin embargo se basa en la biblioteca musl libc, por eso se debe añadir esta instrucción RUN para instalar el paquete que proporciona compatibilidad con glibc (nuestro binario está compilado contra glibc ya que usamos Debian en la primera etapa)

27.7 veces menos tamaño

La imagen base ahora solo pesa unos 7 MB



# Builds Multi-Stage

70

- Ejecución de contenedores usando todas las imágenes

Todas las imágenes son completamente funcionales

```
vagrant@docker:~$ docker run -ti --rm myimage
Program name and arguments:
/opt/app/myapp
argument1
vagrant@docker:~$ docker run -ti --rm myimage-slim
Program name and arguments:
/opt/app/myapp
argument1
vagrant@docker:~$ docker run -ti --rm myimage-multi
Program name and arguments:
/opt/app/myapp
argument1
vagrant@docker:~$ docker run -ti --rm myimage-multi-opt
Program name and arguments:
/opt/app/myapp
argument1
vagrant@docker:~$
```



# Contenidos

71

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- **Redes**
- Almacenamiento
- Docker Compose
- Docker Swarm



# Publicando puertos de un contenedor

72

- Por defecto, un contenedor solo puede ser accedido desde redes locales
- Como hemos visto, la opción **-P** de `docker run` **publica (mapea)** todos los **puertos expuestos** por el contenedor
  - Los mapea a puertos aleatorios del host
- El comando **`docker port`** lista los *mappings* de un contenedor
  - En el ejemplo inferior, el puerto 80 (TCP) del contenedor se mapea al puerto 32769 del host

```
vagrant@docker:~$ docker run -d --name web --rm nginx
71303a734e2f995684170b67c6a5f5b06bebfc352cb25962fb8e7b579b90c95a
vagrant@docker:~$ docker port web
vagrant@docker:~$ docker stop web
web
vagrant@docker:~$ docker run -d --name web --rm -P nginx
da52a4b4918bb78b311133f5a0fe1d995643f4557f2886413ef65fd0ec472926
vagrant@docker:~$ docker port web
80/tcp -> 0.0.0.0:32769
80/tcp -> [::]:32769
vagrant@docker:~$
```



# Asignación de puertos estática

73

- También es posible especificar una asignación de puertos específica con la opción **-p** de *docker run*
- En este otro ejemplo el puerto 80 (TCP) del contenedor se mapea al puerto 80 del host

```
vagrant@docker:~$ docker run -d --name web --rm -p 80:80 nginx
e3f57cc40bda66ba1660e145868526a0cc11e1bb8838e63847632bd31350a9b2
vagrant@docker:~$ docker port web
80/tcp -> 0.0.0.0:80
80/tcp -> [::]:80
vagrant@docker:~$ curl http://localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```



# Red de un contenedor

74

- Cada contenedor tiene un *stack* de red propio
  - Configuración independiente de interfaces, rutas, etc
- Conceptualmente una red en Docker es un *switch virtual* que se implementa a través de un **driver de red**
  - Un contenedor puede estar conectado a múltiples redes a través de diferentes *drivers*
- Un contenedor solo puede “ver” sus interfaces de red
  - Desde el *host* se puede acceder a todos los contenedores
  - Los contenedores no se pueden acceder desde el exterior
- Las redes se comportan como objetos atómicos
  - Se crean y se destruyen
  - Se añaden y se eliminan a la redes contenedores dinámicamente



# Redes en Docker

75

- El comando **docker network** permite administrar las redes
  - El argumento **ls** permite listar las redes disponibles
  - Docker crea tres redes por defecto: *bridge*, *host* y *none*

```
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
785664dccad6    bridge    bridge      local
1d9cbd8b8509    host      host       local
37738810cf1a    none      null       local
vagrant@docker:~$
```

- Se puede seleccionar la red a la que se conecta un contenedor con la opción **--net <net-name>** del comando **docker run**



# Drivers de red

76

- Docker soporta los siguientes *drivers* de red
  - *bridge* (**driver por defecto**)
  - *host*
  - *null*
  - *overlay*
  - *ipvlan*
  - *macvlan*
  - *Plugins de red*
    - *Drivers* de terceros para proporcionar conectividad de red
    - Permiten integrar Docker con *stacks* de red especializadas



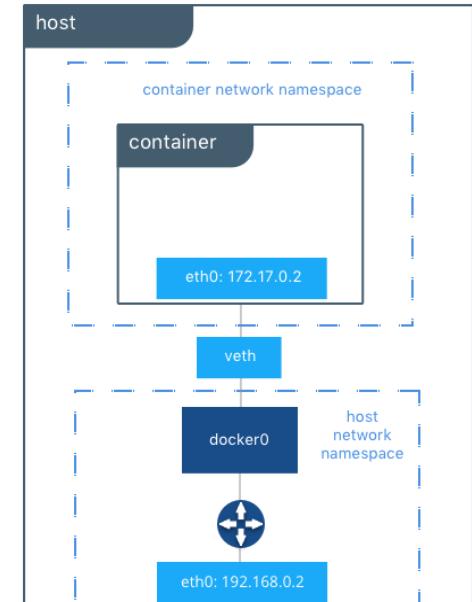
# Driver bridge

77

- Por defecto, un contenedor tiene dos interfaces de red
  - Una interfaz de *loopback* (para comunicación interna)
  - Una interfaz virtual conectada al *bridge docker0* del host
    - El *bridge docker0* tiene asignada la subred privada 172.17.0.0/16
    - El tráfico se gestiona mediante *iptables*

```
vagrant@docker:~$ ip addr show docker0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:f5:af:ce:53 brd ff:ff:ff:ff:ff:ff
        inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
            valid_lft forever preferred_lft forever
        inet6 fe80::42:f5ff:feaf:ce53/64 scope link
            valid_lft forever preferred_lft forever
vagrant@docker:~$
vagrant@docker:~$ docker run --name test --rm busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
134: eth0@if135: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
        inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
            valid_lft forever preferred_lft forever
vagrant@docker:~$
```

No se usa --net, se conectará a la red por defecto



```
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
785664dccad6   bridge    bridge      local
1d9cbd8b8509   host      host       local
37738810cf1a   none      null      local
vagrant@docker:~$
```

La red *bridge*, creada por defecto, se llama igual que el *driver*



# Driver host

78

- El contenedor comparte la stack de red del host

- Puede "ver y usar" las interfaces del host
- Elimina el aislamiento de red entre el contenedor y el host
- El tráfico es directo (no va a través de ningún bridge)
  - Rendimiento de red nativo
- En este modo se ignoran las opciones de mapeo de puertos (-P, -p)
- Se configura con `docker run --net host`

```
vagrant@docker:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP m
    link/ether 02:2d:ad:54:38:ca brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:f5:af:ce:53 brd ff:ff:ff:ff:ff:ff
vagrant@docker:~$ 
vagrant@docker:~$ docker run --name test --rm --net host busybox ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether 02:2d:ad:54:38:ca brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:f5:af:ce:53 brd ff:ff:ff:ff:ff:ff
vagrant@docker:~$
```

NETWORK ID	NAME	DRIVER	SCOPE
785664dcccad6	bridge	bridge	local
1d9cbd8b8509	host	host	local
37738810cf1a	none	null	local

La red *host*, creada por defecto, se llama igual que el *driver*



# Driver null

79

- El contenedor solo posee la interfaz de *loopback*

- De esta forma, el contenedor está completamente aislado del *host* y de otros contenedores
- Útil para ejecutar aplicaciones que no requieren ningún tipo de conectividad de red con máxima seguridad

```
vagrant@docker:~$ docker run --name test --rm --net none busybox ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
vagrant@docker:~$
```

- Se configura con *docker run --net none*
- La red creada por defecto con *driver null* se llama *none*

```
vagrant@docker:~$ docker network ls
NETWORK ID     NAME      DRIVER      SCOPE
785664dcad6   bridge    bridge      local
1d9cbd8b8509  host      host       local
37738810cf1a  none      null       local
vagrant@docker:~$
```



# Driver overlay

80

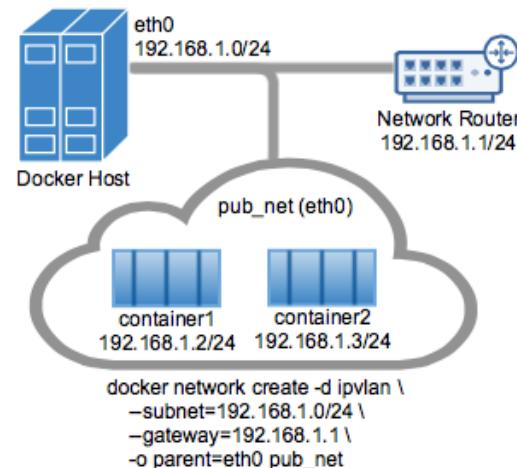
- Permite conectar demonios Docker ejecutándose en diferentes *hosts*
- De esta forma habilita la comunicación segura entre:
  - Contenedores que se ejecutan en diferentes *hosts* (i.e. en diferentes demonios Docker)
  - Servicios creados con Docker Swarm (entorno clúster)
- Por tanto, este *driver* permite crear redes ***multihost***
  - Se crea una red de este tipo automáticamente cuando se inicializa un clúster Swarm
  - Se ubica encima (se superpone) a las redes del *host*
- Docker maneja de forma transparente el enrutamiento de cada paquete hacia y desde el *host* Docker correcto y el contenedor de destino



# Driver ipvlan

81

- Proporciona al usuario control total sobre las direcciones IPv4/IPv6
  - En Linux, las implementaciones IPvlan son extremadamente livianas porque, en lugar de utilizar un *bridge* para el aislamiento, están asociadas a una interfaz Ethernet para imponer la separación entre redes y la conectividad a la red física
- Ventajas
  - Mejor rendimiento al evitar el *bridge* que reside entre el *host* y el contenedor
  - Configuración más simple que únicamente consta de interfaces de contenedor conectadas directamente a la interfaz del *host*
  - Fácil acceso para servicios externos ya que no hay necesidad de mapeos de puertos

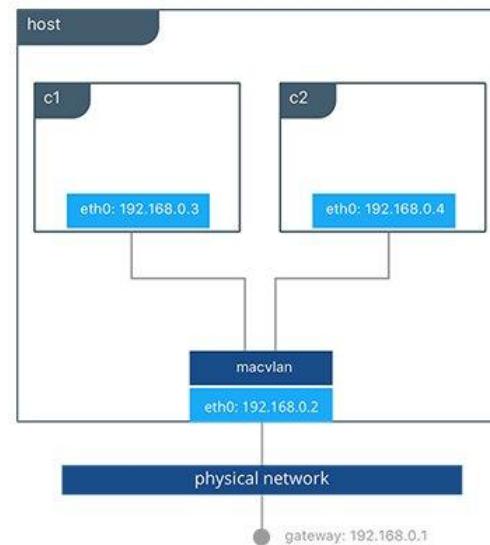




# Driver macvlan

82

- Permite asignar una dirección MAC (dirección física) a una interfaz de red virtual de un contenedor haciendo que parezca una interfaz de red física conectada **directamente** a la red física
  - El demonio Docker dirige el tráfico a los contenedores por sus MACs
  - Es necesario designar una interfaz de red física del host para ser usada con este driver, así como la subred y la puerta de enlace
- Útil cuando se necesita que los contenedores se vean como hosts físicos en una red cada uno con una dirección MAC única





# Creación de redes

83

- Docker permite crear nuevas redes y conectar contenedores a ellas
  - Comando **docker network**
- En el ejemplo inferior, creamos una red llamada “frontend”
  - Por defecto, se crea con el driver bridge (su scope es local)
    - Puede especificarse un driver distinto con la opción -d
  - Existen parámetros de **docker network create** que permiten especificar la subred, puerta de enlace, etc

```
vagrant@docker:~$ docker network create frontend
7952c3c1e1b10eea2a338324fe7e8ac406e0dc7950ffac9bec3bf64756a757b8
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
785664dccad6   bridge    bridge    local
7952c3c1e1b1   frontend  bridge    local
1d9cbd8b8509   host     host     local
37738810cf1a   none     null     local
vagrant@docker:~$
```



# Creación de redes

84

- Inspeccionamos la red “frontend” con **docker network inspect**

```
vagrant@docker:~$ docker network inspect frontend
[{"Name": "frontend",
 "Id": "7952c3c1e1b10eea2a338324fe7e8ac406e0dc7950ffac9bec3bf64756a757b8",
 "Created": "2024-01-22T11:17:48.659459804Z",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
     "Driver": "default",
     "Options": {},
     "Config": [
         {
             "Subnet": "172.18.0.0/16",
             "Gateway": "172.18.0.1"
         }
     ]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
     "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {}, ← Ningún contenedor conectado a la
 "Options": {},
 "Labels": {}
}
]
vagrant@docker:~$
```

Ningún contenedor conectado a la  
nueva red



# Conectar contenedores a redes

85

- Conectamos un contenedor a la red “frontend”
  - Comando **docker network connect**
- Lo inspeccionamos para comprobar a qué redes está conectado
  - A la red *bridge* por defecto (al crearlo con *docker run*)
  - A la red “frontend” creada/definida por el usuario

```
vagrant@docker:~$ docker run -d --name web --rm nginx
66db15fad09b3d9d9b5d53cd09d3d1463ceb7af456ca517aec41f374a6d6d9b8
vagrant@docker:~$
vagrant@docker:~$ docker network connect frontend web
vagrant@docker:~$ docker inspect --format="{{.NetworkSettings.Networks}}" web
map[bridge:0xc000000000 frontend:0xc00000540]
vagrant@docker:~$
```



# Desconectar y eliminar redes

86

- Desconectamos el contenedor de la red por defecto (*bridge*)
  - Comando **docker network disconnect**
  - Comprobamos qué contenedores están conectados a la red por defecto (ninguno en el ejemplo) y a la red “frontend”

```
vagrant@docker:~$ docker network disconnect bridge web
vagrant@docker:~$ docker inspect bridge | jq .[].Containers[].Name
vagrant@docker:~$ docker inspect frontend | jq .[].Containers[].Name
"web"
vagrant@docker:~$
```

- Eliminamos la red “frontend”
  - Comando **docker network rm**
  - No es posible eliminar una red con contenedores conectados a ella

```
vagrant@docker:~$ docker network rm frontend
Error response from daemon: error while removing network: network frontend id 7952c3c1e1b10eea2a338324f
e7e8ac406e0dc7950ffac9bec3bf64756a757b8 has active endpoints
vagrant@docker:~$
```



# Resolución DNS

87

- Los contenedores pueden comunicarse entre ellos a través de sus direcciones IPs
  - Es posible especificar una IP estática a un contenedor
  - Sin embargo, no es cómodo trabajar con IPs directamente
- Los contenedores también pueden referenciarse por su nombre
  - **El nombre de un contenedor Docker es único en el host**
- Docker integra un **servidor DNS** que realiza la resolución de nombres
  - **Pero NO funciona en las redes creadas por defecto!**

```
vagrant@docker:~$ docker run -d --name web --rm nginx ← Estará conectado a la red por defecto
8f923952e09fd9a30ea93b39177f20aaddcdc25293aaaee2547e3a0ef0e14528
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.IPAddress}}" web
172.17.0.2
vagrant@docker:~$ ping -c 1 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.093 ms

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.093/0.093/0.093/0.000 ms
vagrant@docker:~$
vagrant@docker:~$ docker run --name test --rm busybox ping -c 1 172.17.0.2 | Este segundo contenedor hace un ping a IP del primero
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.072 ms

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.072/0.072/0.072 ms
vagrant@docker:~$
vagrant@docker:~$ docker run --name test --rm busybox ping -c 1 web | Aquí hace un ping al nombre del primer contenedor. NO funciona
ping: bad address 'web'
vagrant@docker:~$
```



# Resolución DNS

88

- Docker integra un **servidor DNS** que realiza la resolución de nombres
  - Conectamos ahora el contenedor web a nuestra red “frontend”
  - Inspeccionamos la red para obtener la dirección IP del contenedor
  - Comprobamos con *ping* la conectividad usando su nombre
    - Con el contenedor de prueba conectado a la red bridge falla como antes
    - Con el contenedor de prueba conectado a la red “frontend” funciona, pues es una red definida (creada) por el usuario

```
vagrant@docker:~$ docker network connect frontend web
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.frontend.IPAddress}}" web
172.18.0.2
vagrant@docker:~$ docker run --name test --rm busybox ping -c 1 web
ping: bad address 'web'
vagrant@docker:~$ docker run --name test --rm --net frontend busybox ping -c 1 web
PING web (172.18.0.2) 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.115 ms

--- web ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.115/0.115/0.115 ms
vagrant@docker:~$
```



# Conectividad entre redes

89

- Creamos ahora un contenedor conectado a la red “frontend”
- Creamos otra red con nombre “backend” y un contenedor conectado a ella
- Hacemos *ping* desde un contenedor al otro usando nombre y dirección IP
  - **docker exec** permite ejecutar un comando dentro de un contenedor en ejecución
- Contenedores conectados a redes diferentes no pueden comunicarse directamente
  - Esto permite aislar grupos de contenedores
  - Sí podrían comunicarse a través del host mapeando puertos públicos

```
vagrant@docker:~$ docker run -dti --name ct1 --rm --net frontend busybox  
f80c9ab1254aaf6f0129a4a6ba0a83c4ac22ec2c8210b92465224d67ee384b1c  
vagrant@docker:~$ docker network create backend  
78d7ef20dc0ad805cadab1a133480f06f570d2bde33231a6926ba65ca21726d5  
vagrant@docker:~$ docker run -dti --name ct2 --rm --net backend busybox  
0cb7047b9c94a42e9e1aa2dbfa974596dbdbc49136b598f34602fb600d9354f9  
vagrant@docker:~$ docker exec ct1 ping -c 1 ct2  
ping: bad address 'ct2'  
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.frontend.IPAddress}}" ct1  
172.18.0.2  
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" ct2  
172.19.0.2  
vagrant@docker:~$ docker exec ct1 ping -c 1 172.19.0.2  
PING 172.19.0.2 (172.19.0.2): 56 data bytes  
  
--- 172.19.0.2 ping statistics ---  
1 packets transmitted, 0 packets received, 100% packet loss  
vagrant@docker:~$
```



# Alias de red

90

- El nombre de un contenedor Docker es **único** en el host
  - Creamos un contenedor con nombre "web" conectado a la red "frontend"
  - Creamos otro contenedor "web" conectado a la red "backend"
  - Obtenemos un error aunque sus FQDN serían teóricamente distintos (web.frontend != web.backend)

```
vagrant@docker:~$ docker run -d --name web --rm --net frontend nginx
5b331c9d6f2e7d3c8f15c934903224df06fb8fa62d8b9d76c19d3464d694ec6c
vagrant@docker:~$ 
vagrant@docker:~$ docker run -d --name web --rm --net backend nginx
docker: Error response from daemon: Conflict. The container name "/web" is already in use by container
"5b331c9d6f2e7d3c8f15c934903224df06fb8fa62d8b9d76c19d3464d694ec6c". You have to remove (or rename) that
container to be able to reuse that name.
See 'docker run --help'.
vagrant@docker:~$
```



# Alias de red

91

- Un **alias de red** permite definir el nombre de un contenedor específicamente para una red en concreto
  - Opción `--net-alias` de `docker run`
  - Opción `--alias` de `docker network connect`
  - El nombre solo es válido en la red en la cual se define
  - Los alias en una **misma red no son únicos**

```
vagrant@docker:~$ docker run -d --name web1 --rm --net frontend nginx
ff898d957b6ae4792c25af6c8d33e6ea1a795e5e975a4864ac797bbd848756ed
vagrant@docker:~$ docker run -d --name web2 --rm --net backend --net-alias web nginx
064f37b38933ebd798dea14807ab60e4b066add6d318f35a8224688f76de4984
vagrant@docker:~$ docker run -d --name web3 --rm --net backend --net-alias web nginx
f6561b9bbb9c5341a37042d0bc3dfc74e380de8fd214f705bbde8fa092bbfd30
vagrant@docker:~$
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.frontend.IPAddress}}" web1
172.18.0.2
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" web2
172.19.0.2
vagrant@docker:~$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" web3
172.19.0.3
vagrant@docker:~$
```



# Balanceo de carga

92

- Si hay varios contenedores en la misma red con el mismo alias, el servidor DNS devuelve las direcciones IPs de todos los contenedores que usen dicho alias
- Las direcciones IPs se devuelven en orden *round-robin*
  - Balanceo de carga "simple y barato"

```
vagrant@docker:~$ docker run -d --name web1 --rm --net backend --net-alias web nginx  
4a4516cc0f5c626772f042c7758e33958a576edd3f3ac3c75a6fbc3e97905f33  
vagrant@docker:~$ docker run -d --name web2 --rm --net backend --net-alias web nginx  
db4d1db416039e38859d993a9d9b4c5d72098370acd438383980f092c9e4c78c  
vagrant@docker:~$  
vagrant@docker:~$ docker run -ti --name test --rm --net backend busybox ping -c 1 web  
PING web (172.19.0.3): 56 data bytes  
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.253 ms  
  
--- web ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 0.253/0.253/0.253 ms  
vagrant@docker:~$ docker run -ti --name test --rm --net backend busybox ping -c 1 web  
PING web (172.19.0.2): 56 data bytes  
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.103 ms  
  
--- web ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 0.103/0.103/0.103 ms
```



# Links

93

- Los *links* añaden al fichero `/etc/hosts` una entrada al contenedor referenciado
  - Opción `--link` de `docker run`
  - Se consideran **legacy** (**es recomendable usar redes definidas por el usuario**)
  - No se puede crear un *link* a un contenedor que no está en ejecución
- Es posible definir el nombre con el que se referencia a un contenedor
  - En el tercer ejemplo, `--link web:webserver` significa que se puede referenciar al contenedor "web" con los nombres "web" y "webserver"

```
vagrant@docker:~$ docker run -d --name web --rm nginx
1b06a48d10bf6787e3b81a37f92a9e987d97863cccff21cb90361707bd24fb8
vagrant@docker:~$ docker run --name test --rm busybox cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      98a4de042de3
vagrant@docker:~$ docker run --name test --rm --link web busybox cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      web 1b06a48d10bf
172.17.0.3      d17a172f2f44
vagrant@docker:~$ docker run --name test --rm --link web:webserver busybox cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      webserver 1b06a48d10bf web
172.17.0.3      6dd4a76374fa
vagrant@docker:~$
```

Sin *links* definidos, y con ambos conectados a la red por defecto, este segundo contenedor solo podrá comunicarse con el primero ("web") mediante su dirección IP



# Contenidos

94

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes
- **Almacenamiento**
- Docker Compose
- Docker Swarm



# Introducción

95

- Es posible almacenar datos en la capa de escritura de un contenedor, pero dichos datos **no se persisten** cuando el contenedor se elimina
- Escribir en la capa de escritura requiere usar un *driver* de almacenamiento ([OverlayFS](#)) para manejar el sistema de ficheros
  - Este *driver* proporciona un sistema de ficheros de tipo "Union"
    - Permite montar un sistema de ficheros formado por la unión de otros sistemas de ficheros, de forma que se superponen transparentemente formando así un único sistema de ficheros visible al contenedor
  - La abstracción extra proporcionada por este *driver* **reduce el rendimiento**
- La capa de escritura está estrechamente acoplada al *host* donde se ejecuta el contenedor
  - No es fácil **mover los datos** a otro *host*
- Docker ofrece tres formas de gestionar el almacenamiento
  - Volúmenes
  - Montajes *bind* (*bind mounts*)
  - Montajes *tmpfs* (*tmpfs mounts*)



# Tipos de almacenamiento

96

## • Volúmenes

- Se almacenan en la ruta del sistema de ficheros del host que es gestionada por el demonio Docker (en Linux sería: `/var/lib/docker/volumes`)
- Otros procesos (no-Docker) del sistema **no deberían** modificar dicha ruta
- Se consideran la mejor forma de persistir datos en Docker

## • Montajes bind

- Permiten montar (hacer accesible) un directorio del host en un contenedor
- Se puede usar cualquier ruta del sistema de ficheros del host
- Otros procesos del sistema podrían modificar ficheros en esas rutas
- Importante: un contenedor incluso podría modificar ficheros del host

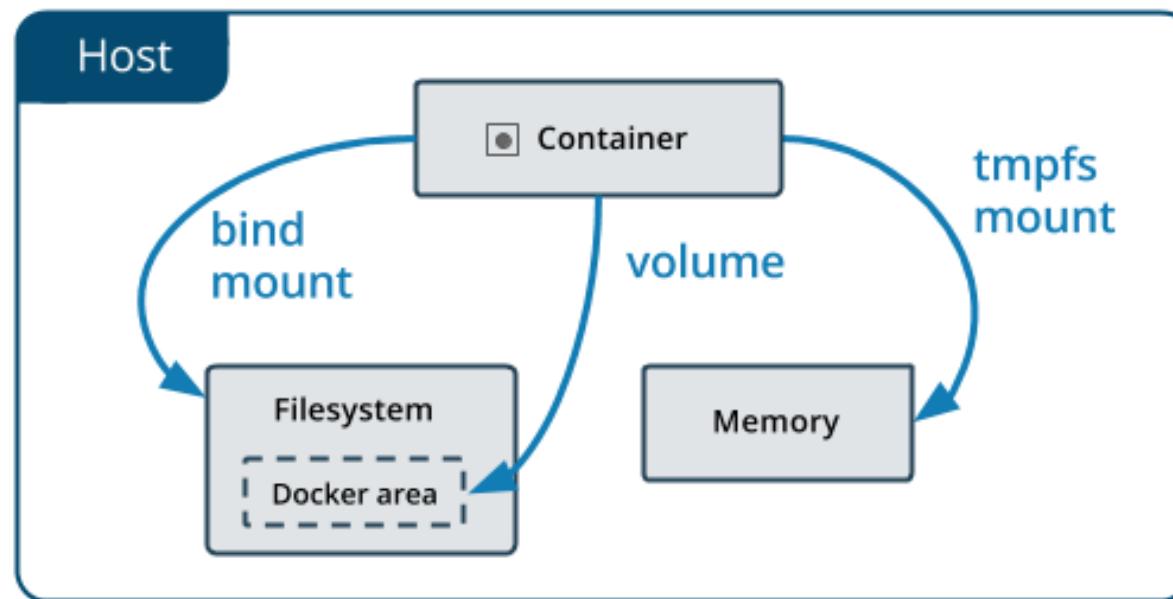
## • Montajes `tmpfs`

- Se almacenan en la memoria del host y por tanto sus datos nunca son escritos a ningún sistema de ficheros del mismo
- Realmente **no permiten persistir datos en disco**, pero son útiles para datos de estado (no persistentes) evitando así usar la capa de escritura y proporcionando además el mejor rendimiento de E/S



# Tipos de almacenamiento

97





# ¿Qué es un volumen?

98

- Es un directorio que reside en el área del host gestionada por Docker y que puede ser montado en un contenedor como un sistema de ficheros adicional o complementario al *rootfs*
- Se puede ver como una forma de compartir directorios y ficheros entre contenedores de forma sencilla
  - Los datos de un volumen no forman parte de la imagen
- Permite persistir datos ya que los volúmenes existen de forma **independiente al ciclo de vida de los contenedores**
  - Se pueden crear explícitamente utilizando el comando **docker volume create** o declararlos durante la creación del contenedor
  - **Un volumen puede crearse con un nombre (*named volume*) o ser anónimo**
- Ventajas
  - Proporcionan mayor rendimiento que usar la capa de escritura
  - Además, no incrementa el tamaño del contenedor que lo utiliza
  - Una ventaja sobre los *bind mounts*, es que un volumen es gestionado completamente por Docker y está aislado del resto del host
    - Los *bind mounts* sí dependen de la estructura de directorios del host y de su SO



# Declaración de volúmenes

99

- Con la opción **-v | --volume** del comando `docker run`
  - `-v [<name>:]<container_dir>[:options]`

```
vagrant@docker:~$ docker run -d --name web --rm -v myvol:/data:ro nginx
cb08f30dbfa7dc27a0436a8535c40c3d057a4904366802d11b9909ef451123d0
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      myvol
```

- Con la opción **--mount** del comando `docker run` (**preferible**)
  - `--mount type=volume, [source=<name>], target=<container_dir>[,readonly|ro]`
    - Ojo, el parámetro `type` también soporta `bind` y `tmpfs`

```
vagrant@docker:~$ docker run -d --name web --rm --mount type=volume,source=myvol,target=/data,ro nginx
1e940de307957b7e0ca2cfcd6a5bddef2615bf403186d94e65a4809bc2b57d58
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      myvol
```

- En el fichero `Dockerfile` usando la instrucción **VOLUME**
  - Cuando se crea un contenedor, `docker run` creará un nuevo volumen con los datos que existan en la imagen en la ruta especificada

```
vagrant@docker:~$ cat ejemplo/Dockerfile
FROM ubuntu:jammy
FROM ubuntu
RUN mkdir /myvol && echo "Hello World" > /myvol/greetings
VOLUME /myvol
ENTRYPOINT cat /myvol/greetings
vagrant@docker:~$
vagrant@docker:~$ docker run --name test --rm myubuntu
Hello World
vagrant@docker:~$
```



# Administración de volúmenes

100

- Las opciones `-v | --mount` de `docker run` crean el volumen si éste no existe cuando se lanza el contenedor
  - Si el contenedor contiene datos en el punto de montaje especificado para el nuevo volumen, éstos se copian en el volumen
- Es posible administrar los volúmenes de forma independiente a los contenedores con el comando **`docker volume`**
  - Listar volúmenes
    - `docker volume ls`
  - Inspeccionar un volumen
    - `docker volume inspect myvol`
  - Inspeccionar volúmenes en una imagen o contenedor
    - `docker inspect --format "{{.Config.Volumes}}" myvol`
  - Eliminar volumen
    - `docker volume rm myvol`



# Administración de volúmenes

101

- Es posible administrar los volúmenes de forma independiente a los contenedores con **docker volume**
  - Ejemplo usando un **volumen anónimo**
    - Su nombre (identificador) es autogenerado

```
vagrant@docker:~$ docker run -dti --name test --rm -v /data busybox
a0e1cf1b892bf7b1c628863cf8c8af33c1fab7a6d2db28bbb9400946d312f82
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      efa92873ca24d37cf0fdee8112c86cae5e8460d9ca7daf47cc4666af5f0609bd
vagrant@docker:~$ docker inspect --format "{{.Config.Volumes}}" test
map[/data:{}]
vagrant@docker:~$ docker exec test sh -c "date > /data/today"
vagrant@docker:~$ docker exec test cat /data/today
Mon Jan 22 14:26:17 UTC 2024
vagrant@docker:~$ sudo ls /var/lib/docker/volumes/efa92873ca24d37cf0fdee8112c86cae5e8460d9ca7daf47cc466
6af5f0609bd/_data
today
vagrant@docker:~$ sudo cat /var/lib/docker/volumes/efa92873ca24d37cf0fdee8112c86cae5e8460d9ca7daf47cc466
6af5f0609bd/_data/today
Mon Jan 22 14:26:17 UTC 2024
vagrant@docker:~$
vagrant@docker:~$ docker stop test
test
vagrant@docker:~$ docker volume ls ← La ejecución del contenedor con la opción --rm
DRIVER      VOLUME NAME
vagrant@docker:~$
```

La ejecución del contenedor con la opción --rm provoca que también se elimine el volumen anónimo de forma automática. No ocurre en el caso de los volúmenes con nombre



# Administración de volúmenes

102

- Es posible administrar los volúmenes de forma independiente a los contenedores con **docker volume**
  - Ejemplo usando un **volumen con nombre**

```
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
vagrant@docker:~$ docker volume create data
data
vagrant@docker:~$ docker run -dti --name test --rm -v data:/data busybox
45205e8769749820f7c2ac5224c7c713eee41175cb529141481ddf1363acad17
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      data
vagrant@docker:~$ docker exec test sh -c "date > /data/today"
vagrant@docker:~$ docker stop test
test
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      data
vagrant@docker:~$ sudo cat /var/lib/docker/volumes/_data/_data/today
Mon Jan 22 14:33:13 UTC 2024
vagrant@docker:~$
vagrant@docker:~$ docker run -dti --name test --rm -v data:/data busybox
c087e15e111459e4f0d298174f7e22e2b63b711fe91c89f1820eb6c1b31d61ac
vagrant@docker:~$ docker exec test cat /data/today
Mon Jan 22 14:33:13 UTC 2024
vagrant@docker:~$
```



# Montajes bind

103

- Permiten montar un directorio (o fichero) del *host* en un contenedor
- Con la opción **-v | --volume** del comando *docker run*
  - **-v <host\_dir>:<container\_dir>[:options]**
  - **El directorio (o fichero) debe ser una ruta absoluta, de lo contrario se consideraría un volumen con nombre**
  - Ejemplo: *docker run -v /www:/data/www nginx*
- Con la opción **--mount** del comando *docker run* (preferible)
  - **--mount type=bind, source=<host\_dir>, target=<container\_dir>[,readonly|ro]**
  - Ejemplo: *docker run --mount type=bind,source=/www,target=/data/www nginx*
- En el fichero *Dockerfile* no está permitido definir un montaje *bind*
  - Por motivos obvios de seguridad
- Si el punto de montaje especificado en el contenedor existe, su contenido se oculta y el contenedor solo “ve” los datos del *host*



# Montajes bind

104

- Permiten montar un directorio (o fichero) del host en un contenedor
  - Importante: un contenedor podría modificar ficheros del host

```
vagrant@docker:~$ cat hello
Hello World
vagrant@docker:~$ pwd
/home/vagrant
vagrant@docker:~$ docker run -ti --name test --rm -v /home/vagrant:/data busybox
/ # ls
bin    data    dev    etc    home   lib    lib64   proc   root   sys    tmp    usr    var
/ # cat data/hello
Hello World
/ # date >> data/hello
/ # cat data/hello
Hello World
Mon Jan 22 14:55:42 UTC 2024
/ # exit
vagrant@docker:~$ cat hello
Hello World
Mon Jan 22 14:55:42 UTC 2024
vagrant@docker:~$
```

```
vagrant@docker:~$ docker run -ti --name test --rm --mount type=bind,source=/home/vagrant,target=/data,ro busybox
/ # cat data/hello
Hello World
Mon Jan 22 14:55:42 UTC 2024
/ # dat >> data/hello
sh: can't create data/hello: Read-only file system
/ # exit
vagrant@docker:~$
```



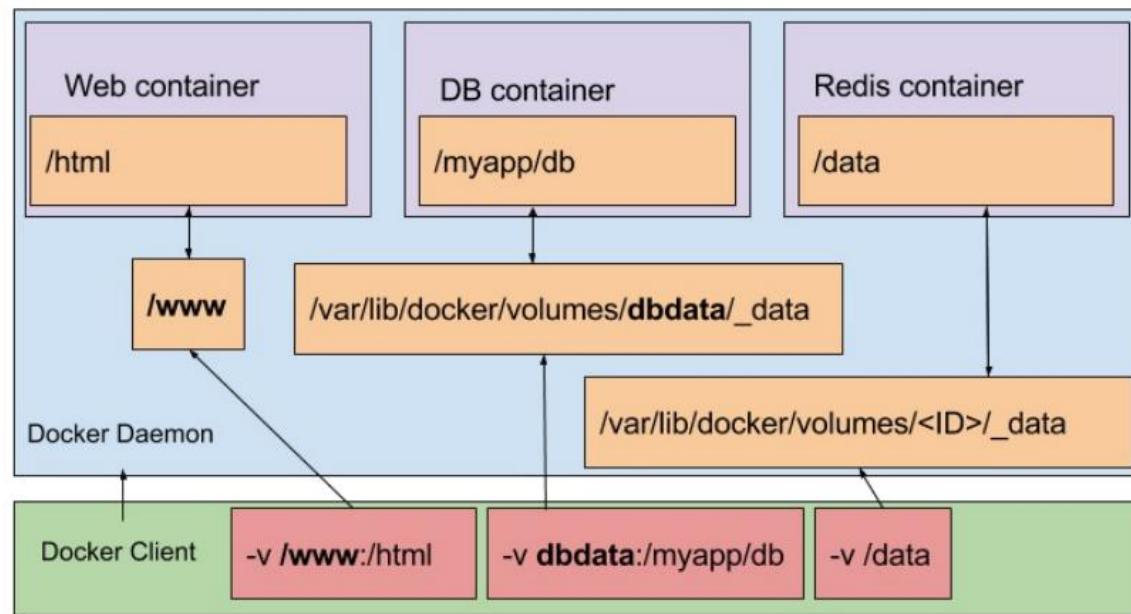
Usando sintaxis alternativa (la recomendada)  
y añadiendo la opción de solo lectura



# Comparativa

105

- Montaje *bind* vs volumen con nombre vs volumen anónimo





# Contenidos

106

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes
- Almacenamiento
- **Docker Compose**
- Docker Swarm



# ¿Qué es Docker Compose?

107

- Es una herramienta que permite definir y ejecutar aplicaciones **multicontenedor** que se ejecutan en el mismo *host*
  - Una aplicación multicontenedor se denomina **stack** en el ámbito Docker
- Los contenedores (**servicios/microservicios**) que componen la aplicación y sus dependencias se definen en un **fichero en formato YAML**
  - Especifican toda la configuración de cada contenedor
  - Descarga automática de imágenes y ejecución de contenedores
  - Creación de los volúmenes, redes, *links*, etc definidos para todos los contenedores
  - Permiten automatizar el despliegue de aplicaciones multicontenedor siguiendo una aproximación declarativa (paradigma IaC)
- Usando dicho fichero de configuración, es posible desplegar la aplicación completa mediante un solo comando
  - **docker compose up**



# Sintaxis docker-compose.yml

108

- Existen diferentes versiones de formato para los ficheros compose
  - <https://docs.docker.com/compose/compose-file/compose-versioning>
  - Especificación más reciente y recomendada
    - <https://docs.docker.com/compose/compose-file/>
  - La versión se puede especificar al inicio del fichero YAML
- En el fichero se definen cinco objetos principales de primer nivel
  - **services**: definición abstracta de un recurso dentro de una aplicación que puede escalarse o reemplazarse independientemente de otros componentes
    - Los servicios están respaldados por un conjunto de contenedores
  - **networks**: se especifican redes definidas por el usuario
  - **volumes**: para definir volúmenes con nombre
  - **configs**: permiten que los servicios adapten su comportamiento sin la necesidad de reconstruir una imagen
  - **secret**: define o hace referencia a datos confidenciales
- Similar a los *Dockerfiles*, el fichero *docker-compose.yml* debe residir en una carpeta la cual es usada por Docker Compose como contexto



# Sintaxis docker-compose.yml

109

- La definición de un servicio requiere al menos de una instrucción
  - **build**: indica la carpeta con el *Dockerfile*
  - **image**: define la imagen base
    - Si ambas están presentes, *image* define el tag de la imagen
- Traducción de algunos parámetros de *docker run*
  - **links** se traduce en --link
  - **ports** se traduce en -p
  - **volumes** se traduce en -v
    - En caso de volúmenes con nombre, deben definirse en su propia sección **volumes**
  - **expose** se traduce en --expose
- Además:
  - **command** substituye el CMD del *Dockerfile*
  - **entrypoint** substituye el ENTRYPOINT del *Dockerfile*
  - **healthcheck** substituye el HEALTHCHECK del *Dockerfile*
  - **depends\_on** permite establecer dependencias entre los servicios para controlar el orden en el que se inician y se detienen



# Sintaxis docker-compose.yml

110

- Ejemplo sencillo definiendo un único servicio

```
vagrant@docker:~$ cat compose/compose.yml
services:
  myapp:
    image: ubuntu:jammy
    command:
      - ls
      - /opt/host_dir
    volumes:
      - type: bind
        source: /home/vagrant/compose
        target: /opt/host_dir
      - type: volume
        source: app-data
        target: /data
    networks:
      - frontend

volumes:
  app-data:

networks:
  frontend:

configs:
  app_config:
    file: ./file.conf

secrets:
  server-certificate:
    file: ./server.cert
vagrant@docker:~$
```



# Desplegar la aplicación

111

- Comando **docker compose up**
  - Con **docker compose stop** detenemos la aplicación
  - Con **docker compose down** además de detener la aplicación, elimina los contenedores, redes y volúmenes creados

```
vagrant@docker:~$ docker compose -f compose/compose.yml up
[+] Running 3/0
  ✓ Network compose_frontend    Created
  ✓ Volume "compose_app-data"   Created
  ✓ Container compose-myapp-1   Created
Attaching to compose-myapp-1
compose-myapp-1 | compose.yml
compose-myapp-1 | file.conf
compose-myapp-1 | server.cert
compose-myapp-1 exited with code 0
vagrant@docker:~$
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      compose_app-data
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
78d7ef20dc0a    backend    bridge      local
785664dccad6    bridge    bridge      local
835af8533466   compose_frontend  bridge      local
7952c3c1e1b1    frontend   bridge      local
1d9cbd8b8509    host      host      local
37738810cf1a    none      null      local
vagrant@docker:~$
```

```
vagrant@docker:~$ docker compose -f compose/compose.yml down
[+] Running 2/0
  ✓ Container compose-myapp-1   Removed
  ✓ Network compose_frontend   Removed
vagrant@docker:~$ docker volume ls
DRIVER      VOLUME NAME
local      compose_app-data
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
78d7ef20dc0a    backend    bridge      local
785664dccad6    bridge    bridge      local
7952c3c1e1b1    frontend   bridge      local
1d9cbd8b8509    host      host      local
37738810cf1a    none      null      local
vagrant@docker:~$
```

Para que borre también el volumen: **docker compose down -v**



# Desplegar la aplicación

112

- Comando **docker compose up -d**
  - El parámetro **-d** permite el despliegue en segundo plano
  - Podemos listar los *stacks* y contenedores con **docker compose ls | ps**
  - Para ver los logs usamos el comando **docker compose logs**

```
vagrant@docker:~$ docker compose -f compose/compose.yml up -d
[+] Running 2/2
 ✓ Network compose_frontend    Created
 ✓ Container compose-myapp-1  Started
vagrant@docker:~$ docker compose ls
NAME          STATUS        CONFIG FILES
compose        running(1)   /home/vagrant/compose/compose.yml
vagrant@docker:~$ docker compose -f compose/compose.yml ps
NAME          IMAGE        COMMAND
compose-myapp-1 nginx      "/dock...r-entrypoint.sh nginx -g 'daemon off;'"    myapp      17 seconds ago  Up 16 seconds  80/tcp
vagrant@docker:~$ docker compose -f compose/compose.yml logs myapp
compose-myapp-1 /dock...r-entrypoint.sh: /dock...r-entrypoint.d/ is not empty, will attempt to perform configuration
compose-myapp-1 /dock...r-entrypoint.sh: Looking for shell scripts in /dock...r-entrypoint.d/
compose-myapp-1 /dock...r-entrypoint.sh: Launching /dock...r-entrypoint.d/10-listen-on-ipv6-by-default.sh
compose-myapp-1 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
compose-myapp-1 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
compose-myapp-1 /dock...r-entrypoint.sh: Sourcing /dock...r-entrypoint.d/15-local-resolvers.envsh
compose-myapp-1 /dock...r-entrypoint.sh: Launching /dock...r-entrypoint.d/20-envsubst-on-templates.sh
compose-myapp-1 /dock...r-entrypoint.sh: Launching /dock...r-entrypoint.d/30-tune-worker-processes.sh
compose-myapp-1 /dock...r-entrypoint.sh: Configuration complete; ready for start up
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: using the "epoll" event method
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: nginx/1.25.3
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: OS: Linux 5.15.0-57-generic
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: start worker processes
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: start worker process 29
compose-myapp-1 2024/01/22 15:49:51 [notice] 1#1: start worker process 30
vagrant@docker:~$
```

Fichero Compose usado  
en este ejemplo

```
vagrant@docker:~$ cat compose/compose.yml
services:
  myapp:
    image: nginx
    networks:
      - frontend

networks:
  frontend:

configs:
  http_config:
    file: ./http.conf
vagrant@docker:~$
```



# Redes en Docker Compose

113

- Si no se especifica ninguna red, se crea una red por defecto
- Si se especifica una red, los contenedores se conectan únicamente a ella y no a la red por defecto
- Existe conectividad y resolución DNS entre los contenedores usando como *hostname* el nombre del servicio
  - No se necesitan *links*, pero se pueden usar para definir aliases extra
  - Ejemplo:
    - A la izquierda, la aplicación web puede conectarse a postgres://db:5432
    - A la derecha, también puede usar postgres://database:5432

```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

```
version: "3"
services:
  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```



# Contenidos

114

- Introducción
- Conceptos básicos
- Imágenes
- *Dockerfiles*
- Redes
- Almacenamiento
- Docker Compose
- **Docker Swarm**



# ¿Qué es Docker Swarm?

115

- Es una herramienta que permite definir y ejecutar aplicaciones **multicontenedor** y **multihost** en un clúster de demonios Docker
  - Por tanto, es una herramienta para la **orquestación de contenedores** Docker en un entorno **clúster**
    - “Similar” a Kubernetes (K8s)
  - Permite **escalar** las aplicaciones basadas en contenedores en tantas instancias y en tantos nodos de red como se requiera
  - Proporciona **balanceo de carga** y **descubrimiento de servicios**



# Stack

116

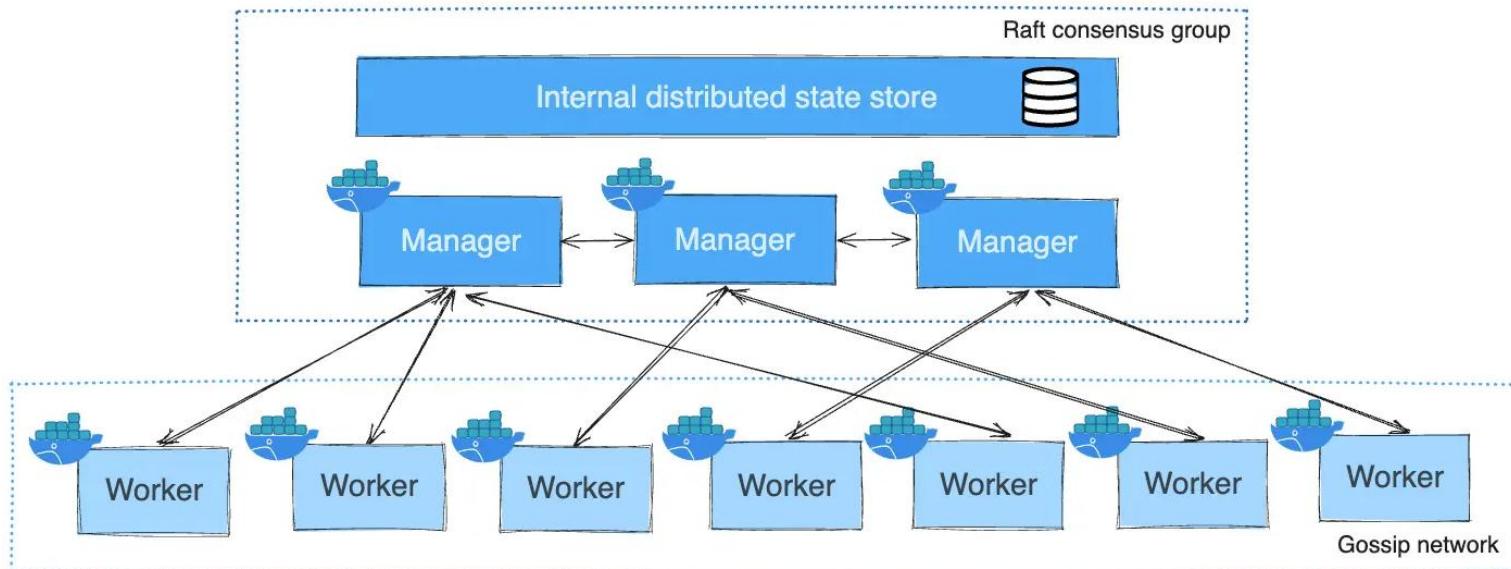
- Un **stack** define un conjunto de servicios posiblemente heterogéneos que componen una aplicación en Swarm
  - *Stack* = aplicación multiservicio
  - Mismo concepto que en Docker Compose pero la aplicación puede ejecutar sus contenedores en diferentes *hosts*
- Los **servicios** de un *stack* se definen en un **fichero en formato YAML**
  - Prácticamente con la misma sintaxis que los ficheros de Docker Compose pero con pequeñas diferencias
  - El mismo fichero YAML funciona para Docker Compose y Swarm!!!
    - Algunas instrucciones son ignoradas por Swarm y otras por Compose
    - Ejemplo: Docker Compose ignora la instrucción **deploy**
- Usando un fichero que define un *stack*, es posible desplegar la aplicación en un clúster Swarm mediante un solo comando
  - **`docker deploy stack`**



# Arquitectura

117

- Swarm se basa en una arquitectura *master/worker*
  - Al menos un nodo maestro o **master**, también llamado **manager**
  - Uno o varios nodos **workers**





# Nodos, servicios y tareas

118

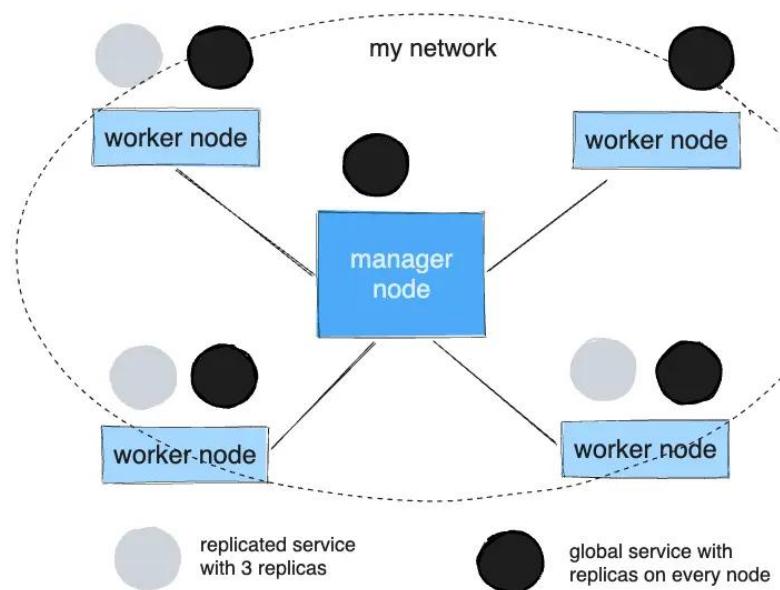
- Un **nodo** es una instancia de Docker Engine (*host*) que forma parte del clúster Swarm (“el enjambre”)
  - El *manager* recibe la definición de un servicio y la despliega en el clúster
- Un **servicio** en Swarm es una estructura abstracta con la que se pueden definir las tareas individuales a ejecutar en el enjambre
  - Cuando se crea un servicio, el usuario determina la imagen en la que se basa y los comandos que se ejecutan en los contenedores
- Una **tarea** es un contenedor junto con los comandos a ejecutar en él
  - Una tarea es la unidad de trabajo básica en Swarm
- Por tanto, el *manager* es responsable de la gestión del clúster y de la coordinación y planificación de tareas
  - Crea y envía tareas para ser ejecutadas en los nodos *worker*
  - Por defecto, un nodo *manager* actúa como *worker* y también ejecuta tareas, pero esto es configurable



# Servicios replicados y globales

119

- **Replicados:** se trata de tareas que se ejecutan en un número de réplicas (i.e. contenedores) definido por el usuario
  - Los servicios replicados se pueden **escalar** creando réplicas adicionales
- **Globales:** cada nodo del enjambre ejecuta una tarea del servicio correspondiente
  - Si se añade un nuevo nodo, Swarm le asigna una tarea de forma inmediata

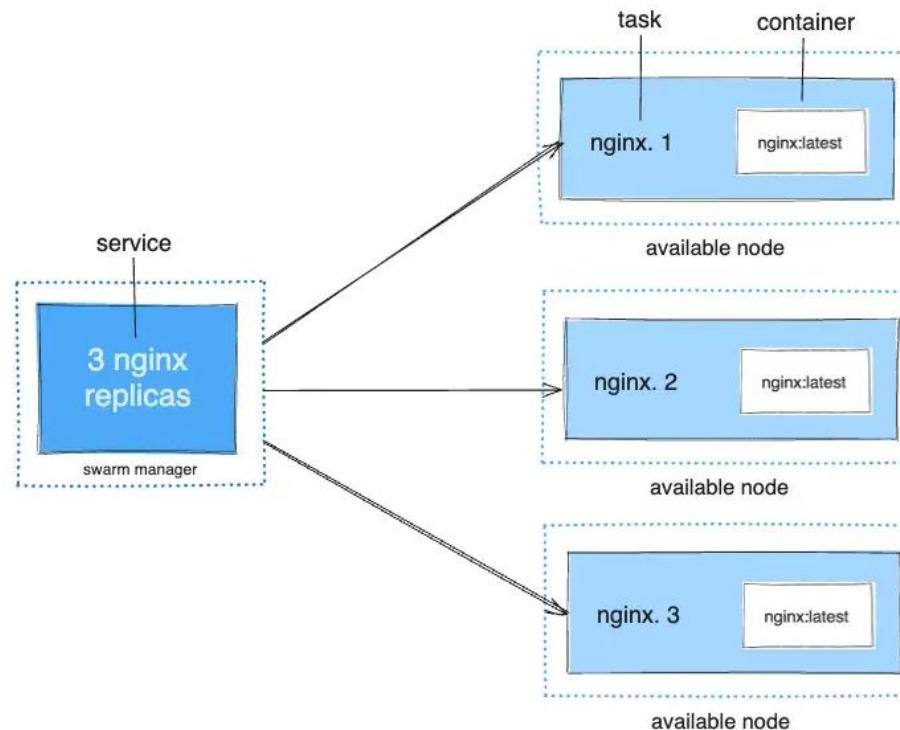




# Servicios replicados y globales

120

- Ejemplo: servicio Nginx replicado con tres instancias
  - Docker distribuye las peticiones entrantes al servicio de forma inteligente entre las instancias disponibles
    - **Balanceo de carga**
  - Cada instancia es una tarea (i.e. contenedor en ejecución)



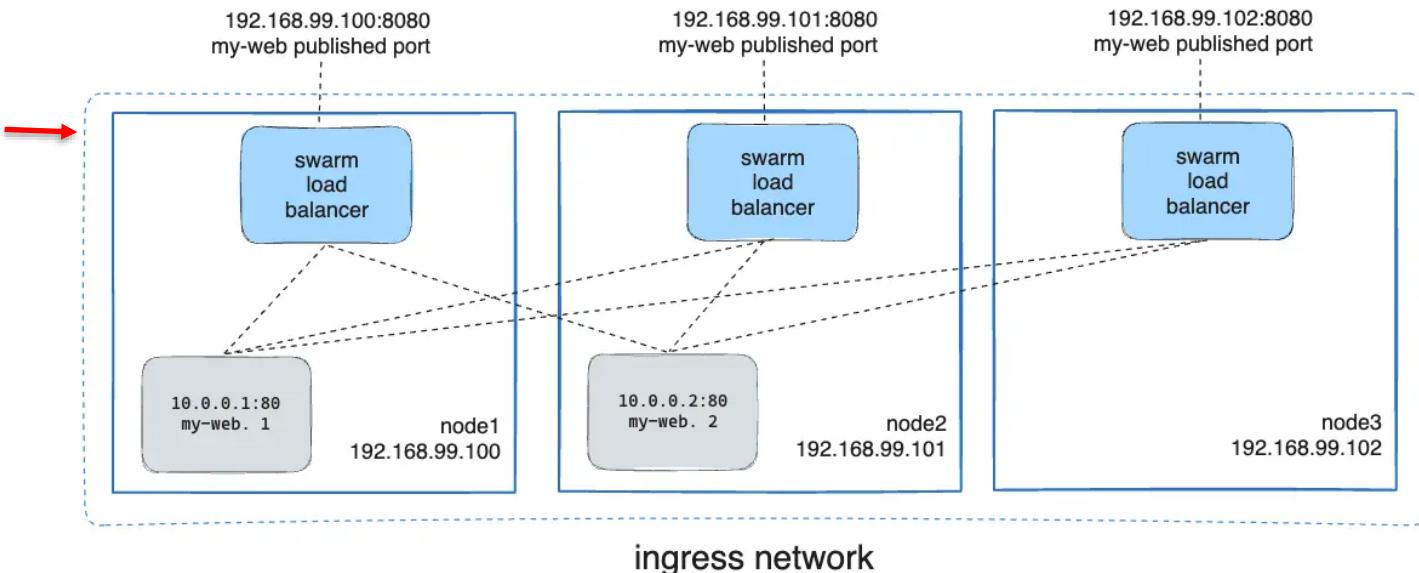


# Ingress routing mesh

121

- Para hacer accesible desde el exterior del clúster un servicio desplegado, Docker Swarm crea una red con *driver overlay* denominada *ingress*
  - Todos los nodos del clúster pertenecen a la red *ingress*
  - Permite publicar un puerto al exterior y que todos los nodos del clúster acepten peticiones a dicho puerto para cualquier servicio Swarm en ejecución
    - Incluso si no hay un contenedor en ejecución en dicho nodo!!
    - Swarm realiza el enrutamiento necesario para encaminar las peticiones al puerto publicado hacia un nodo que disponga de contenedores activos

Cuando se accede al puerto publicado (8080) de cualquier nodo del clúster Swarm, la petición se enruta a un nodo y puerto destino (80) que disponga de un contenedor en ejecución (activo)





# Crear un clúster Swarm

122

- Preparar los nodos del clúster instalando Docker Engine
- Inicializar el clúster Swarm
  - Los nodos con Docker Engine deben ser agrupados en un "enjambre"
  - Primero se debe configurar el nodo que actuará como *manager* ejecutando el comando **docker swarm init** en dicho nodo

Al inicializar el enjambre, Docker crea la red *ingress* con *driver overlay* (scope *swarm*) y la red *docker\_gwbridge* con *driver bridge*, que conecta las redes *overlay* (incluida *ingress*) a la red física del host

```
vagrant@docker:~$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
78d7ef20dc0a    backend    bridge      local
785664dccad6    bridge    bridge      local
a4ea80c30920    docker_gwbridge  bridge      local
7952c3c1e1b1    frontend   bridge      local
1d9cbd8b8509    host      host      local
mq7ms54vpscua  ingress   overlay    swarm
37738810cf1a    none      null      local
vagrant@docker:~$
```

- Integrar nodos en el enjambre
  - Al ejecutar el comando anterior, obtendremos por terminal los comandos necesarios para añadir nuevos nodos al clúster Swarm
  - Si no se dispone de dicha información, primero se debe generar el *token* ejecutando en un nodo *manager* el siguiente comando
    - **docker swarm join-token manager | worker**
  - En el nodo que queremos añadir al clúster Swarm, se ejecuta el siguiente comando usando el *token* generado anteriormente
    - **docker swarm join --token [TOKEN]**



# Crear un clúster Swarm

123

- Comprobando el estado del clúster: **docker node**
  - Listar nodos
    - *docker node ls*
  - Inspeccionar un nodo
    - *docker node inspect <node>*
  - Listar tareas en ejecución en un nodo
    - *docker node ps <node>*
  - Eliminar un nodo
    - *docker node rm <node>*

```
vagrant@docker:~$ docker swarm init
Swarm initialized: current node (24h1nlsmzqpyulyups574ncr3) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-15tgeecq56ke7003awg1v9f79ulv3ky4fc7laomoils16tqtd5-d21qw4zck2t7ovt32gkvth4x9 10.0.2.15:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

```
vagrant@docker:~$
```

```
vagrant@docker:~$ docker node ls
ID                  HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS   ENGINE VERSION
24h1nlsmzqpyulyups574ncr3 *  docker    Ready   Active        Leader        24.0.7
vagrant@docker:~$
```



# Definición y despliegue del stack

124

- Definir los servicios de la aplicación mediante un fichero YAML
  - La instrucción **deploy** permite indicar el tipo de servicio (*global* o *replicated*), el número de réplicas, la política de reinicio, límites en el uso de recursos, etc
- Ejecutar un *stack* en el clúster Swarm
  - `docker stack deploy --compose-file stack.yml <stack-name>`
- Listar los *stacks*
  - `docker stack ls`
- Comprobar el estado de los servicios de un *stack*
  - `docker stack services <stack-name>`
- Ver los logs de un servicio
  - `docker service logs <service-name>`
- Escalar servicios replicados
  - `docker service scale <service-name>=<replicas>`
- Detener un stack
  - `docker stack rm <stack-name>`



# Despliegue del stack

125

- Aplicación con dos servicios replicados: wordpress y db

```
vagrant@docker:~$ cat swarm/compose.yml
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    depends_on:
      - db
  deploy:
    mode: replicated
    replicas: 2
  db:
    image: wordpress
    deploy:
      mode: replicated
      replicas: 1
    volumes:
      - db-data:/var/lib/mysql
  volumes:
    db-data:
vagrant@docker:~$
```

```
vagrant@docker:~$ docker stack deploy -c swarm/compose.yml mystack
Creating network mystack_default
Creating service mystack_wordpress
Creating service mystack_db
vagrant@docker:~$
vagrant@docker:~$ docker stack ls
NAME      SERVICES
mystack   2
vagrant@docker:~$
vagrant@docker:~$ docker stack services mystack
ID        NAME          MODE        REPLICAS      IMAGE           PORTS
agj2to1gfakk  mystack_db     replicated  0/1          wordpress:latest
mf041kr63yca  mystack_wordpress  replicated  0/2          wordpress:latest  *:8080->80/tcp
vagrant@docker:~$
vagrant@docker:~$ docker stack services mystack
ID        NAME          MODE        REPLICAS      IMAGE           PORTS
agj2to1gfakk  mystack_db     replicated  1/1          wordpress:latest
mf041kr63yca  mystack_wordpress  replicated  2/2          wordpress:latest  *:8080->80/tcp
vagrant@docker:~$
```

```
vagrant@docker:~$ docker stack rm mystack
Removing service mystack_db
Removing service mystack_wordpress
Removing network mystack_default
vagrant@docker:~$
```



# Despliegue de servicios

126

- Para el despliegue de servicios Docker de **forma individual** en un clúster Swarm se puede usar el comando **docker service**
- Este comando acepta como parámetros muchas de las instrucciones que contienen los ficheros *stack*
- Ejemplo: servicio web nginx replicado con 3 instancias
  - `docker service create --name mynginx --replicas 3 -p 80:80 nginx`

```
vagrant@docker:~$ docker service create --name mynginx --replicas 3 -p 80:80 nginx
v0wrbjvz4cb0hf5ostphtphoh
overall progress: 3 out of 3 tasks
1/3: running [=====
2/3: running [=====
3/3: running [=====
verify: Service converged
vagrant@docker:~$
vagrant@docker:~$ docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
v0wrbjvz4cb0  mynginx  replicated  3/3        nginx:latest  *:80->80/tcp
vagrant@docker:~$
vagrant@docker:~$ docker service ps mynginx
ID          NAME      IMAGE      NODE      DESIRED STATE  CURRENT STATE      ERROR      PORTS
3e3i6wei3b8  mynginx.1  nginx:latest  docker    Running     Running  23 seconds ago
ttipeve5phz4  mynginx.2  nginx:latest  docker    Running     Running  23 seconds ago
m4nivbmnq55  mynginx.3  nginx:latest  docker    Running     Running  23 seconds ago
vagrant@docker:~$
```



# Escalado de servicios

127

- Es posible escalar **servicios replicados** de forma dinámica mientras están en ejecución con el comando ***docker service scale***
- Este comando acepta como parámetros el nombre del servicio a escalar y el número de réplicas deseado
  - Es posible escalar hacia arriba y hacia abajo el número de instancias de un servicio
- Ejemplo:
  - Escalar el servicio web nginx replicado del ejemplo anterior para que pase a ejecutarse con 5 instancias
    - *docker service scale mynginx=5*