



UT 011. DOCKER

Sistemas informáticos
CFGS DAW

Álvaro

Maceda

a.macedaarranz@edu.gva.es

2022/2023

Versión:230227.0631

Licencia



**Atribución - No comercial - CompartirIgual
(por-nc-sa):**


No se permite el uso comercial de la obra original ni de ninguna obra derivada.

cuya distribución debe realizarse bajo una licencia igual a la que rige la obra original.

Nomenclatura

A lo largo de esta unidad se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

ÍNDICE

1. Qué es Docker	4
1.1 Máquinas virtuales	4
1.2 Contenedores	5
2. Un ejemplo de Docker	6
3. Arquitectura Docker	7
3.1 Cliente y servicio Docker	9
3.2 Imágenes y centros	9
3.3 Contenedores	10
4. Trabajar con contenedores	11
4.1 Creación de contenedores	11
4.2 Arranque, pausa y parada de los contenedores	12
4.3 Docker exec	13
4.4 Ejecución Docker	14
4.5 Cartografía portuaria	14
4.6 Obtener información sobre los contenedores	15
4.7 Datos persistentes	16
4.8 Permisos	17
5. Material complementario	19

UT 0 1. DOCKER

1. QUÉ ES DOCKER

Cuando desplegamos una aplicación web, tenemos que preparar un ordenador con todo el software que necesitaremos para ejecutarla: bibliotecas, paquetes externos, archivos adicionales, etc. Necesitamos poder recrear ese ordenador para fines de desarrollo y pruebas. Por ejemplo, si estamos desarrollando una aplicación web en nuestro ordenador personal tenemos que estar seguros de que todas las librerías son las mismas que las que utilizaremos en el ordenador u ordenadores donde se desplegará posteriormente esa aplicación.

Tal vez podríamos replicar el sistema en nuestro ordenador para una aplicación, pero ¿qué ocurre si estamos trabajando en varias aplicaciones? Será muy difícil y propenso a errores mantener nuestros ordenadores sincronizados.

Necesitamos una forma de crear un ordenador reproducible

1.1 Máquinas virtuales

Una forma de crear un entorno aislado y reproducible es mediante el uso de una máquina virtual (VM). Esencialmente, una máquina virtual es un único ordenador que reside dentro de una máquina anfitriona, en la que pueden coexistir varias máquinas virtuales. La creación de una VM implica la virtualización del hardware subyacente de la máquina anfitriona —procesamiento, memoria y disco— que se emulan utilizando el hardware real de la máquina anfitriona.

Sin embargo, este proceso de virtualización conlleva una importante sobrecarga computacional. Además, cada máquina virtual es una máquina independiente y requiere su propio sistema operativo, cuya instalación suele requerir decenas de gigabytes de almacenamiento y tiempo, un proceso que debe repetirse cada vez que se pone en marcha una nueva máquina virtual.

Aunque las máquinas virtuales (VM) proporcionan muchas ventajas, como la posibilidad de crear un entorno aislado y ejecutar varias instancias de un sistema operativo en un único equipo host, si utilizamos máquinas virtuales para desplegar aplicaciones pueden generar algunos problemas :

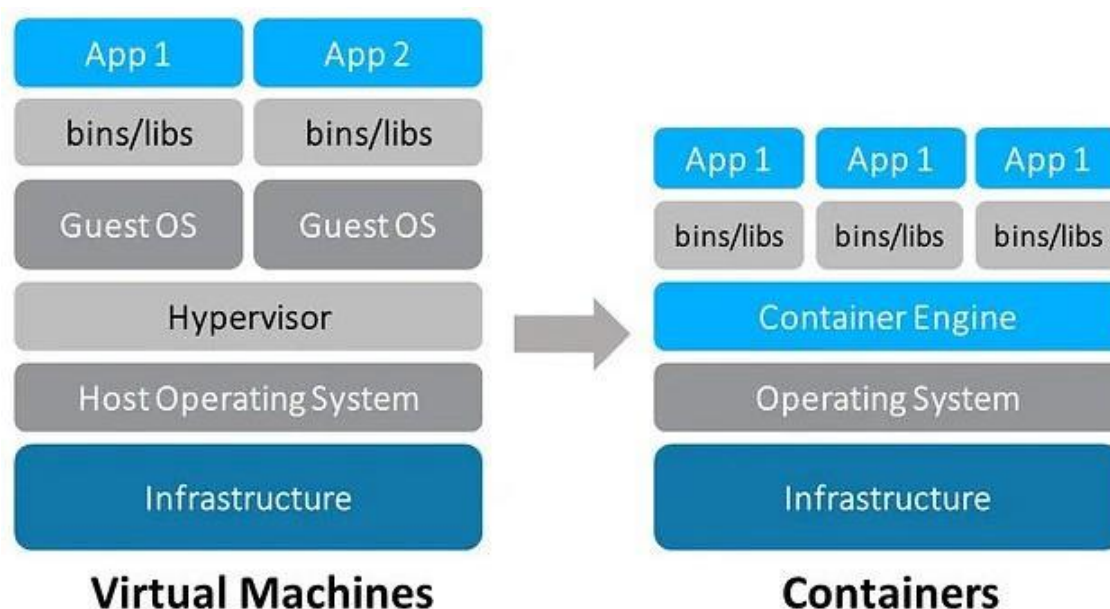
- **Sobrecarga de recursos:** El proceso de virtualización implica la creación de una máquina virtual que tiene su propio sistema operativo, aplicaciones y recursos. Esto requiere una cantidad significativa de recursos informáticos, lo que puede provocar una elevada sobrecarga de recursos y una reducción del rendimiento.
- **Sobrecarga de almacenamiento:** Cada máquina virtual requiere su propio sistema operativo y aplicaciones, lo que puede generar una sobrecarga de almacenamiento significativa. Esto puede ser especialmente problemático cuando se despliegan varias máquinas virtuales, ya que cada una requerirá un conjunto independiente de recursos.
- **Complejidad:** Los entornos de máquinas virtuales pueden ser complejos y difíciles de gestionar. A menudo requieren conocimientos especializados para su configuración y mantenimiento, y pueden ser difíciles de solucionar en caso de problemas.
- **Seguridad:** Las máquinas virtuales pueden introducir riesgos de seguridad adicionales en un entorno. Cada máquina virtual es esencialmente una máquina independiente con su propio

sistema operativo, lo que significa que

la seguridad debe mantenerse para cada máquina virtual individual. Además, las máquinas virtuales pueden ser más vulnerables a los ataques dirigidos a la capa de virtualización.

1.2 Contenedores

Los contenedores intentan resolver algunos de esos problemas, creando una forma ligera y portátil de crear entornos aislados. Los contenedores utilizan un método diferente para lograr el aislamiento: funcionan de forma similar a las máquinas virtuales, residiendo sobre una máquina anfitriona y utilizando sus recursos, pero en lugar de virtualizar el hardware, virtualizan el sistema operativo anfitrión. Como resultado, los contenedores son mucho más ligeros que las máquinas virtuales y pueden ponerse en marcha rápidamente, ya que no necesitan su propio sistema operativo.



El demonio Docker sirve de contrapartida a la capa del hipervisor en el despliegue de contenedores (suponiendo que se esté utilizando Docker), actuando como intermediario entre el sistema operativo anfitrión y los contenedores. Esto se traduce en una menor sobrecarga computacional en comparación con el software del hipervisor, como se ilustra en el recuadro más fino del diagrama anterior, lo que contribuye aún más a la naturaleza ligera de los contenedores en comparación con las máquinas virtuales.

Los contenedores no proporcionan el mismo nivel de aislamiento del sistema anfitrión que las máquinas virtuales. Sin embargo, sí proporcionan suficiente aislamiento para evitar que la configuración del sistema operativo anfitrión afecte al funcionamiento de la aplicación (en su mayor parte).

Un motor de contenedores **sólo** puede ejecutar **el mismo sistema operativo que el host**. Sin embargo, los sistemas operativos Windows y Mac OS permiten ejecutar contenedores Linux. Esto se debe a que estos sistemas operativos ejecutan un kernel Linux ligero y, sobre él, el motor de contenedores.

Algunas de las ventajas de los contenedores son:

- **Ligereza:** Los contenedores son mucho más ligeros que las máquinas virtuales. Los contenedores comparten el sistema operativo anfitrión y no requieren un sistema operativo invitado independiente, lo que los hace más eficientes y rápidos de poner en marcha.
- **Arranque rápido:** Dado que los contenedores no necesitan arrancar un sistema operativo completo, pueden iniciarse en tan solo unos segundos. Esto facilita la rápida puesta en marcha de nuevos contenedores y el escalado de aplicaciones según sea necesario.
- **Utilización más eficiente de los recursos:** Los contenedores son más eficientes en el uso de recursos que las máquinas virtuales. Dado que comparten el sistema operativo del host, necesitan menos recursos para ejecutarse, lo que significa que puede ejecutar más contenedores en un único host que máquinas virtuales.
- **Mayor flexibilidad:** Los contenedores ofrecen más flexibilidad que las máquinas virtuales. Puede personalizar las imágenes de los contenedores para incluir solo los componentes necesarios para su aplicación, lo que reduce la superficie de ataque y minimiza el riesgo de vulnerabilidades.

En este curso aprenderemos Docker, una de las plataformas de contenedorización más populares en la actualidad, aunque existen otras alternativas como Podman o LXD.

2. UN EJEMPLO DE DOCKER

Antes de aprender más sobre Docker, veamos un ejemplo rápido para que te hagas una idea de lo que podremos hacer con un sistema de contenedores. Para ejecutar este ejemplo, necesitarás tener instalado Docker en tu sistema.

Ejecutaremos un servidor web desde nuestra máquina, sirviendo las páginas de un directorio local. Necesitaremos crear un archivo `index.html` en el directorio actual con este contenido:

```
<html>
  <head>
    <title>Hola desde Docker</title>
  </head>
  <body>
    <h1>¡Hola!</h1>
    <p>Esta es una página web servida con nginx</p>
  </body>
</html>
```

Después de crear ese archivo, lo serviremos usando nginx, un servidor web. Pero, en lugar de instalar nginx en nuestra máquina, usaremos un contenedor Docker:

```
docker run --rm --publish 8080:80 \
  --mount type=bind,src=`pwd`,dst=/usr/share/nginx/html,readonly \
  --detach --name docker-example nginx
```

La salida de ese comando será algo así:

```
No se puede encontrar la imagen 'nginx:latest'
localmente latest: Extrayendo de library/nginx
01b5b2efb836: Pull completo
db354f722736: Pull completo
abb02e674be5: Pull completo
214be53c3027: Pull completo
a69afcef752d: Pull complete
```



```
625184acb94e: Pull complete
Digest:
sha256:c54fb26749e49dc2df77c6155e8b5f0f78b781b7f0eadd96ecfabdcdfa5b1ec4
Status: Downloaded newer image for nginx:latest
db032f2b2e159d6ab2d47b07fd7f3789e30e65d979bfd10f23dc993551e49c48
```

Este comando hace un par de cosas:

- Descarga de las imágenes del contenedor en tu ordenador (si vuelves a ejecutarlo, no volverá a extraer las imágenes porque ya las tendrás en tu sistema).
- Arrancando el contenedor. El contenedor, en este caso, está ejecutando nginx
- Compartir el directorio actual con el contenedor
- Redirigiendo el puerto 8080 de su máquina al puerto 80 del contenedor

Después de iniciar el contenedor, podemos abrir la URL <http://localhost:8080> y veremos la página servida por el contenedor:

Hello!

This is a web page served with nginx

Para detener el contenedor, podemos ejecutar

```
docker stop docker-ejemplo
```

Veamos ahora qué ocurre bajo el capó.

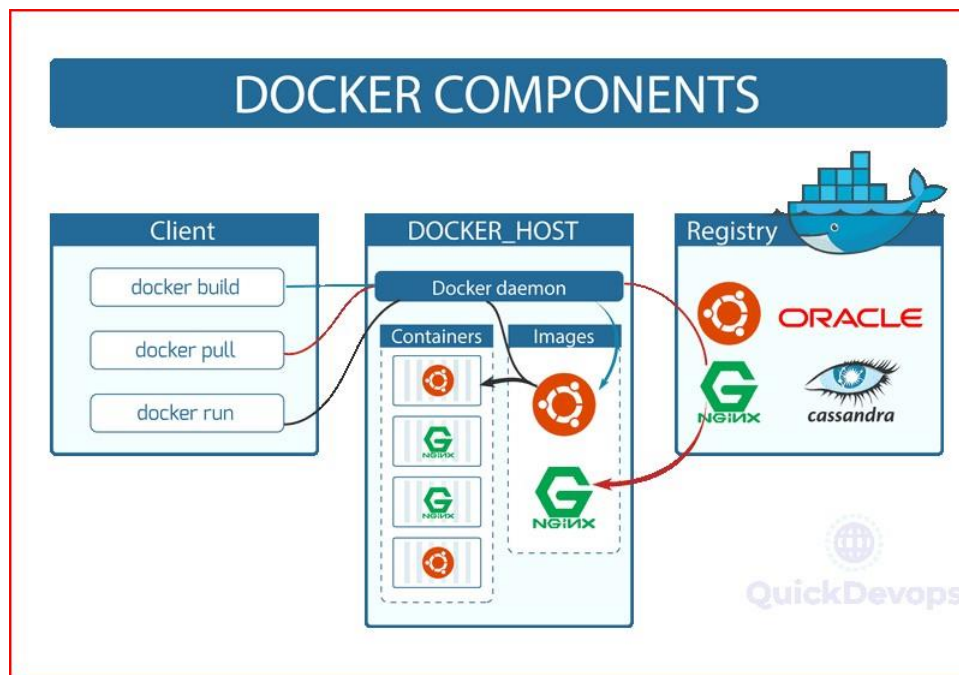
3. ARQUITECTURA DOCKER RE

Docker está formado por múltiples componentes. Trabajaremos principalmente con el cliente Docker (docker) pero es importante conocer todas sus partes.

La arquitectura de Docker se basa en estos componentes:

- **Las imágenes:** Las imágenes son como el "disco duro" del ordenador virtual. A través de Dockerfiles podemos crear esas imágenes que generarán ordenadores virtuales con los mismos archivos. En el ejemplo anterior, estamos utilizando una imagen llamada `nginx:latest` que genera un ordenador que ejecuta un servidor web.
- **Contenedores:** A partir de una imagen, podemos crear y ejecutar "ordenadores virtuales". Esos "ordenadores" se llaman contenedores. Podemos crear múltiples contenedores a partir de la misma imagen (es como instalar múltiples ordenadores con la misma configuración) En el ejemplo anterior, estamos llamando al contenedor `docker-example`. Puedes ejecutar el comando `docker run` varias veces (utilizando diferentes nombres) para lanzar varios contenedores idénticos.
- **Cliente Docker:** El cliente docker es la herramienta de línea de comandos que permite al usuario interactuar con el demonio. Vamos a ejecutar ese comando (`docker`) con diferentes argumentos para utilizar docker.

- **Demonio Docker:** El cliente docker habla con un servicio en segundo plano que se ejecuta en el host y que hace todo el trabajo: gestiona las imágenes y construye, ejecuta y distribuye los contenedores.
- **Docker Hub:** Es un repositorio en internet que contiene las imágenes Docker disponibles. Si no tenemos una imagen en nuestro ordenador, docker la buscará en un repositorio y la descargará a nuestra máquina.



En el ejemplo de `nginx`, esto es lo que ocurrió:

1. Usamos el cliente docker (`docker run...`) para decirle a Docker que queremos crear e iniciar un contenedor desde una imagen llamada `nginx`. El cliente docker es el ejecutable llamado `docker` que lanzamos desde la línea de comandos
2. El cliente docker hablaba con el servicio docker que se estaba ejecutando en nuestra máquina, traduciendo nuestras instrucciones a su propio protocolo
3. El servicio docker vio que no tenía la imagen `nginx` disponible
4. El servicio docker envía una petición por Internet al hub Docker (<https://hub.docker.com/>) buscando esa imagen. El hub responde con la información sobre esa imagen.
5. El servicio docker descargó la imagen en nuestra máquina. Una imagen, como veremos más adelante, se compone de múltiples capas que se pueden descargar de forma independiente
6. Una vez que tuvo la imagen, el servicio docker creó un contenedor con la configuración que le proporcionamos y lo ejecutó.

Veamos un poco más sobre esos componentes.

3.1 Cliente Docker y servicio

3.1.1 Cliente

El cliente Docker es el ejecutable que utilizaremos para realizar operaciones con Docker. Es un ejecutable llamado `docker`.

Tiene múltiples subcomandos, cada uno utilizado para realizar diferentes acciones. Puedes ver todas las diferentes posibilidades en la documentación de Docker:

<https://docs.docker.com/engine/reference/commandline/cli/>

Cada subcomando es como un comando diferente, y tienen múltiples opciones en la línea de comandos. La mayoría de ellos no se utilizan a menudo. En cualquier caso, deberías consultar la documentación de docker para cada comando y echar un vistazo a las diferentes opciones para ser consciente de la variedad de cosas que puedes conseguir con cada comando.

No se sienta abrumado por todas las posibilidades: en este curso sólo utilizaremos un pequeño subconjunto de ellas.

3.1.2 Servidor

El servicio Docker se encarga de realizar las operaciones que enviamos a través del cliente Docker. El cliente Docker no hace nada: solo habla con el servidor para realizar operaciones.

Como el cliente y el servidor son componentes diferentes, podríamos configurar nuestro cliente para hablar con un servidor Docker en otra máquina. No veremos ese tipo de configuración en este curso, pero debes ser consciente de la posibilidad.

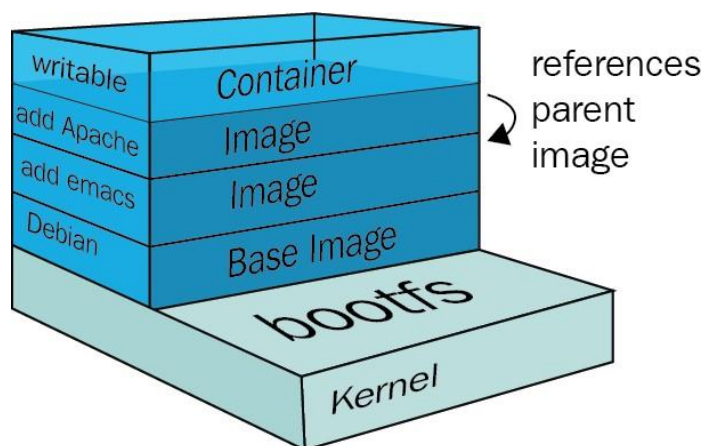
Puedes gestionar el servicio docker como cualquier otro servicio utilizando los comandos `systemctl` y `service`. Por ejemplo, puedes ejecutar `systemctl status docker` para ver el estado del servicio. Por supuesto, si el servicio está detenido no podrás trabajar con docker en tu máquina.

3.2 Imágenes y centros

Una imagen Docker es algo parecido a una imagen de disco para una máquina virtual. Creamos una imagen "instalando" cosas en el ordenador base, y una vez que hemos terminado tenemos un disco duro a partir del cual podemos crear múltiples ordenadores idénticos.

Esos discos duros, o imágenes, se estructuran en capas. A partir de una capa base (normalmente proporcionada por Docker) creamos nuevas capas añadiendo, eliminando o modificando archivos. Esto nos permite compartir capas entre distintos contenedores.

Una vez que iniciamos un contenedor utilizando una imagen, añadimos una nueva capa sobre ella. La única capa escribible es la última, por lo que las otras capas pueden ser compartidas por los otros contenedores e imágenes:



Desde el punto de vista del contenedor, tiene un disco duro "normal". La arquitectura en capas es algo interno de Docker y la mayor parte del tiempo no necesitarás preocuparte por ello.

Teniendo esto en cuenta, podemos clasificar las imágenes en imágenes base e imágenes hijo:

- **Las imágenes base** son imágenes que no tienen una imagen padre: no se basan ni derivan de otra imagen. Suelen ser imágenes que representan un sistema operativo (por ejemplo, Ubuntu, busybox...). No crearás contenedores usando esas imágenes, porque son inútiles por sí mismas.
- **Las imágenes hijas** son imágenes que se basan en las imágenes base y añaden funciones adicionales; la mayoría de las imágenes que crees serán imágenes hijas.

También es importante saber que hay algunas imágenes (imágenes oficiales) que son más fiables que otras:

- **Las imágenes oficiales** son imágenes mantenidas y soportadas oficialmente por la gente de Docker. Suelen tener una sola palabra. Algunos ejemplos son `nginx`, `ubuntu` y `hello-world`.
- **Las imágenes de usuario** son imágenes creadas y compartidas por personas que utilizan Docker. Normalmente se basan en imágenes base y añaden funcionalidad. Normalmente tienen el formato `usuario/nombre-imagen`.

A veces tendrás que crear tus propias imágenes, pero la mayoría de las veces utilizarás imágenes ya creadas. Un Docker Hub es un lugar donde buscar imágenes útiles, y el más utilizado es el Docker Hub:

<https://hub.docker.com/>

Allí encontrarás imágenes listas para usar con algunas de las aplicaciones y servicios más útiles. Hablaremos más adelante sobre el uso de un hub.

3.3 Contenedores

Un contenedor es como un ordenador virtual. Utilizar una imagen (el "disco duro") para crear un contenedor es como configurar el ordenador: asignar recursos compartidos, redes, puertos compartidos...

Una vez configurado el contenedor, podemos arrancarlo y pararlo tantas veces como queramos, y

destruirlo cuando ya no necesitemos ese "ordenador".

Los contenedores pueden estar en uno de esos estados:

- **creado:** Docker asigna el estado creado a los contenedores que nunca fueron iniciados desde que fueron creados. Los contenedores en este estado no utilizan CPU ni memoria: es como una máquina virtual parada.
- **en ejecución:** El contenedor se está ejecutando: algunos procesos se están ejecutando en el entorno aislado dentro del contenedor, utilizando memoria y CPU.
- **salido:** Una vez que el contenedor termina de ejecutar su comando principal, sale. Es como si se cerrara una vez terminada la tarea. La mayoría de los contenedores no salen, porque ejecutan un servicio que sigue funcionando.

Por ejemplo, si ejecutamos un contenedor nginx no terminará hasta que detengamos manualmente el contenedor. Sin embargo, otros simplemente ejecutarán un comando y terminarán. Si ejecutamos un contenedor `hello-world`, se detendrá cuando termine.

- **en pausa:** Los contenedores también pueden estar en pausa, consumiendo memoria pero no CPU.
- **muerto:** si el contenedor tiene un error estará en estado muerto.

Comprenderás mejor estos estados cuando trabajemos con contenedores en una sección posterior

4. TRABAJAR CON CONTENEDORES

4.1 Creación de contenedores

Lo primero que necesitas para trabajar con un contenedor es crear uno. La forma más sencilla de crear un contenedor es utilizar `docker create`:

<https://docs.docker.com/engine/reference/commandline/create/> El

uso del comando es:

```
docker create [OPCIONES] IMAGEN [COMANDO] [ARG...]
```

La parte más importante aquí es la IMAGEN. Siempre creamos un contenedor a partir de una imagen, por lo que es un argumento necesario para `docker create`. Por ejemplo, para crear un contenedor a partir de la imagen nginx podemos ejecutar:

```
docker create nginx
```

Se mostrará el ID del contenedor recién creado (podemos listar todos los contenedores con `docker container ls --all`)

La segunda parte más relevante son las OPCIONES. Aquí tenemos muchas opciones; con esas opciones podemos configurar la "máquina virtual" añadiendo límites de memoria, direcciones IP, cuotas de CPU... Por ejemplo, para crear un contenedor llamado `ubuntu_container` con un límite de memoria de 1GB podemos ejecutar:

```
docker create --name ubuntu_container --memory 1g -ti ubuntu
```

La tercera parte es el comando que se ejecutará al iniciar el contenedor (recuerda que los contenedores no se están ejecutando todavía) Las imágenes suelen venir con un comando inicial útil, por lo que no vamos a modificarlo con `docker create` en este curso.

4.2 Puesta en marcha, pausa y parada de los contenedores

4.2.1 Contenedores iniciales

Una vez que tenemos el contenedor creado, necesitamos iniciarlo para que comience a ejecutarse:

```
docker start <nombre del contenedor>
```

Este comando mostrará el nombre del contenedor y volverá a la consola. Si examinamos los contenedores en ejecución, podemos ver que el contenedor se está ejecutando:

ID DEL CONTENEDOR	COMANDO	CREADO	ESTADO	PUERT	NOMBR
IMAGE	"/docker-	hace 2	minutosHacia	arriba 31	segundos

Un contenedor se ejecutará hasta que su proceso principal termine. Los contenedores suelen proporcionar un servicio como un servidor de base de datos, un servidor web... por lo que a menudo seguirán ejecutándose hasta que los detengamos. Sin embargo, en algunos casos, el contenedor ejecutará algo y terminará, por lo que no podremos verlo ejecutándose.

4.2.2 Fijación a un contenedor

Cuando inicias un contenedor no puedes ver la salida que genera, ni enviarle comandos. Eso es porque, normalmente, los contenedores proporcionan servicios y no necesitamos interactuar con ellos.

Para ver la salida del contenedor, podemos utilizar el siguiente comando:

```
docker logs micontenedor
```

Enviaré la `salida estándar del contenedor` a nuestra consola. Podemos usar el parámetro `--follow` para estar atentos a nuevas salidas del contenedor. Por ejemplo, si creamos e iniciamos un contenedor nginx, la salida de `docker logs` podría ser algo como esto:

```
...
...
/docker-entrypoint.sh: Lanzando /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Iniciando /docker-entrypoint.d/30-tune-worker-
processes.sh
/docker-entrypoint.sh: Configuración completa; listo para arrancar
2023/02/21 12:13:43 [aviso] 1#1: utilizando el método de eventos "epoll
2023/02/21 12:13:43 [aviso] 1#1: nginx/1.23.3
2023/02/21 12:13:43 [aviso] 1#1: compilado por gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/02/21 12:13:43 [aviso] 1#1: SO: Linux 5.15.0-60-generic
2023/02/21 12:13:43 [aviso] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/02/21 12:13:43 [aviso] 1#1: iniciar procesos de trabajo
2023/02/21 12:13:43 [aviso] 1#1: iniciar proceso de trabajador 22
2023/02/21 12:13:43 [aviso] 1#1: iniciar proceso de trabajador 23
2023/02/21 12:13:43 [aviso] 1#1: iniciar proceso de trabajador 24
2023/02/21 12:13:43 [aviso] 1#1: iniciar proceso de trabajador 25
```

A veces necesitaremos interactuar con el contenedor. Podemos adjuntar el stdin, stdout y stderr de nuestra consola al contenedor para que, en la práctica, sea como si estuviéramos trabajando en la consola del contenedor. Podemos hacerlo con:

```
docker attach micontenedor
```

También podemos iniciar el contenedor en modo interactivo utilizando las opciones `--interactive` o `--attach` en el comando `docker start`.

4.2.3 Contenedores de parada

Si el contenedor no se ha terminado, podemos pararlo con:

```
docker stop mycontainer
```

No utilizará CPU ni memoria hasta que lo iniciemos de nuevo, pero utilizará parte del espacio de nuestro disco duro.

También puede pausar un contenedor con `docker pause mycontainer`. No utilizará CPU pero seguirá utilizando memoria de su host.

4.2.4 Retirada de contenedores

Al eliminar un contenedor se destruirá toda su configuración y sus datos internos. Ya no utilizará espacio en el disco duro:

```
docker rm mycontainer
```

Podemos usar la orden `docker container prune` para eliminar todos los contenedores parados. Úsala con cuidado.

4.3 Docker exec

Podemos ejecutar comandos adicionales en un contenedor ya en ejecución. Uno de los usos más comunes de esa utilidad es abrir un intérprete de comandos en el contenedor para ejecutar algunos comandos administrativos, comprobar el contenido de archivos, etc:

```
docker exec -ti micontenedor /bin/sh
```

Una vez que hayamos terminado con la consola, podemos terminarla con `exit` o Control+D para volver al ordenador anfitrión.

Para poder interactuar con el comando, podemos usar las opciones `--tty` y `--interactive` (o su abreviatura `-ti`) Si queremos ejecutar un comando desatendido podemos usar la opción `--detach`: lanzará el comando en el contenedor y volverá a nuestra consola directamente.

El ejecutable debe existir en el contenedor. Los contenedores son una versión sobresimplificada de los ordenadores Linux, por lo que a menudo no dispondremos de algunos de los comandos más utilizados, como `ps`, herramientas de red o el intérprete de comandos `bash`. Podemos instalar lo que queramos en el contenedor, pero todo lo que hagamos se perderá cuando destruyamos el contenedor.

Ese tipo de operación debería usarse sólo mientras se desarrolla el contenedor: la filosofía de docker es que los contenedores puedan crearse y descargarse fácilmente desde la imagen así que, una vez tengamos las cosas claras, deberíamos modificar nuestra imagen para reflejar los cambios que queremos para el contenedor. Hablaremos de las imágenes más adelante.

4.4 Docker ejecutar

En lugar de crear un contenedor e iniciarlo más tarde, podemos realizar ambas operaciones en un único comando con `docker run`. Esta opción es más utilizada que la combinación `docker create + docker start`:

```
docker run [OPCIONES] IMAGEN [COMANDO] [ARG...]
```

Como puedes ver, el comando es similar a `docker create` pero también iniciará el contenedor. Las opciones son las mismas que en `docker create`. También se adjuntará al contenedor. Si queremos ejecutar el contenedor en segundo plano, podemos utilizar la opción `--detach`.

Así, por ejemplo, para crear e iniciar un contenedor `nginx` con un solo comando podrías usar:

```
docker run nginx
```

Debes tener en cuenta que `docker run` **no destruirá el contenedor una vez que se haya detenido**. Si no has especificado un nombre al ejecutar el comando, Docker creará uno aleatorio. Así que si, por ejemplo, ejecutas el comando `docker run nginx` y luego detienes el contenedor con `Ctrl+C` tres veces, puedes terminar con tres contenedores detenidos como estos, y necesitarías eliminarlos manualmente:

ID DEL CONTENEDOR	IMAGEN	COMANDO	CREADO	ESTADO	PUERTOS	NOMBRES
a0dba86b4936	nginx	"/docker-entrypoint..."	Hace 7 segundos	Salió (0) hace 5 segundos		enfocado_faraday
17113cd4cd60	nginx	"/docker-entrypoint..."	hace 9 segundos	Salió (0) hace 7 segundos		wonderful_jang
6b4485de9248	nginx	"/docker-entrypoint..."	hace 21 segundos	Salió (0) hace 18 segundos		nostalgic_euclid

Pero si utiliza la opción de línea de comandos `--rm` el contenedor se destruirá automáticamente una vez que haya finalizado su ejecución.

También podemos ejecutar un comando en el contenedor. Por ejemplo, podemos abrir un shell en un contenedor `nginx` con este comando. El servidor `nginx` no se ejecutará todavía:

```
docker run -ti --rm nginx /bin/sh
```

Luego podemos modificar archivos, iniciar el servicio con `service start nginx`, etc, pero el contenedor se detendrá (y será destruido, porque hemos usado la opción `--rm`) cuando salgamos del shell.

También podemos establecer un nombre para el contenedor con la opción `--name` (tiene sentido utilizar esa opción junto con la opción `--rm` si separamos el contenedor y queremos interactuar con él más tarde)

4.5 Puerto mapping

A menudo, utilizamos contenedores que proporcionan servicios de red. Veremos las redes de Docker más adelante, pero Docker proporciona una forma rápida de hacer que esos servicios estén disponibles en nuestro host: la bandera `--publish` o `-p`. Usando esa bandera *redirigimos* un puerto en nuestra máquina a un puerto en el contenedor. Así, por ejemplo, cuando nos conectamos al puerto 8080 en nuestra máquina en realidad nos estaríamos conectando al puerto 80 en un contenedor.

La asignación de puertos se realiza al crear el contenedor. La bandera tiene un argumento de múltiples valores separados por dos puntos. El primer valor es el puerto en nuestra máquina, y el segundo el puerto en el contenedor. Con este mapeo de puertos:

```
CC docker run --rm --publish 8080:80 \
    --detach --name docker-example nginx
```

Cuando nos conectemos al puerto 8080 en nuestra máquina (navegando a <http://localhost:8080>, por ejemplo) nos estaremos conectando al puerto 80 del contenedor.

Sólo podemos usar un puerto en nuestra máquina para una redirección. Si queremos redirigir a varios contenedores, tendremos que utilizar puertos diferentes. Por ejemplo, podemos redirigir el puerto 80 al puerto 80 de un contenedor y el puerto 81 al puerto 80 de otro contenedor.

También puede especificar la IP de su host que desea redirigir, y el protocolo (TCP, por defecto o UDP) con esta sintaxis: `-p 192.168.1.100:8080:80/udp`

4.6 Obtener información sobre los contenedores

Hay dos comandos para obtener información sobre los contenedores:

`docker ps` mostrará información sobre los contenedores en ejecución. Por ejemplo:

```
→ ~ docker ps
ID DEL CONTENEDOR    IMAGEN    COMANDO    CREADO    ESTADO    PUERTOS    NOMBRES
2e4fb80e36da        nginx    "/docker-entrypoint... " hace 22 segundosHace 3    80/tcp    otro_contenedor
f36dc40fad64        nginx    "/docker-entrypoint... " hace 16 minutosHacia arriba    80/tcp    mycontaine
6 minutos (En pausa)
```

Sólo mostrará los contenedores en ejecución y en pausa. Para ver todos los contenedores podemos usar la opción `--all` parámetro:

```
→ ~ docker ps --all
ID DEL CONTENEDOR    IMAGEN    CREADO    ESTADO    PUERTOS    NOMBRES
aeecl55bd7wordpress "docker-entrypoint.s... Creado    un_contenedor_más
2e4fb80e36da        nginx    "/docker-entrypoint... "hace 2 minutos    Arriba 2 minutos    80/tcp    otro_contenedor
f36dc40fad64        nginx    "/docker-entrypoint... "hace 18    Arriba 8 minutos (En    80/tcp    mi_contenedor
```

Esto nos muestra información básica sobre los contenedores. Podemos obtener toda la información de un contenedor, incluyendo el estado, con `docker inspect`. Nos devolverá un objeto grande en formato JSON con toda la información sobre el contenedor:

```
→ ~ docker inspect mycontainer
[
  {
    "Id":
"f36dc40fad64887d0b50148508ce2a7be83e8ed098d2d8fa24e6ebc7e0d916e5",
    "Created": "2023-02-17T11:32:06.105645545Z",
    "Path": "/docker-entrypoint.sh",
    "Args": [
      "nginx",
      "-g",
      "daemon off;"
    ],
    ...
    ...
    ...
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02",
    "DriverOpts": null
  }
]
```

Podemos filtrar la información que queremos ver usando el parámetro `--format`. Por ejemplo, `docker inspect -f '{{.State.Status}}' mycontainer` devolverá solo el estado del contenedor.

4.7 Persistencia de los datos

Veamos qué ocurre con los datos que tenemos dentro de los contenedores. En primer lugar, vamos a ejecutar un nuevo contenedor Ubuntu:

```
docker run -ti --name mycontainer ubuntu /bin/bash
```

Esto abrirá un shell bash dentro de nuestro nuevo contenedor. Podemos crear un nuevo archivo dentro del directorio raíz:

```
root@3153657f9594:/# echo "Algún contenido" > /a_fichero_nuevo.txt
root@3153657f9594:/# cat a_fichero_nuevo.txt
Algunos contenidos
```

Entonces, podemos teclear "exit": que terminará el shell. Si iniciamos el contenedor de nuevo:

```
docker start --interactive mycontainer
```

Podemos ver que el fichero sigue ahí, con el mismo contenido. Así que una forma de persistir los datos será mantener el contenedor hasta que ya no necesitemos los datos. Sin embargo, la filosofía de Docker es que los contenedores son desechables, por lo que necesitamos algún tipo de sistema de persistencia de datos que almacene los datos en algún lugar fuera de los contenedores. Docker proporciona dos maneras de hacer esta persistencia de datos: bind mounts y volúmenes. Ambas soluciones funcionan montando un espacio de almacenamiento de datos en un directorio del contenedor, de forma similar a como se monta un sistema de archivos en Linux.

Debes configurar los bind mounts (y los volúmenes) al crear el contenedor con `docker create` o `docker run`. **Una vez creado el contenedor, no podremos añadirle bind mount o volúmenes.**

Hay dos parámetros diferentes para la persistencia de datos: `-v` o `--volume`, y `--mount`. En los ejemplos utilizaremos la sintaxis `--mount`, más explícita. Ambas opciones funcionan casi de la misma manera.

Hay algunas advertencias con Linux y Mac cuando se utilizan volúmenes, por favor echa un vistazo a la documentación si tienes algún problema con ellos.

4.7.1 Fijar soportes

Con un bind mount, montamos una carpeta host dentro de la carpeta del contenedor. Esta es la solución más simple para persistir los datos del contenedor, y se utiliza a menudo cuando usamos contenedores para desarrollar aplicaciones.

Para el ejemplo del principio de la unidad hemos utilizado un soporte de fijación:

```
docker run --rm
  --mount type=bind,src=`pwd`,dst=/usr/share/nginx/html,readonly \
  --detach --name docker-example nginx
```

El parámetro `--mount` consiste en una serie de opciones separadas por una coma. El orden de las opciones no es relevante:

- Para el montaje bind, necesitaremos incluir la opción `type=bind`
- La opción `source` es la ruta al archivo o directorio en la máquina anfitriona. Puede montar tanto archivos individuales como directorios (Docker lo detectará, dependiendo de la ruta, y actuará en consecuencia) La opción `source` **debe ser una ruta absoluta**.

En el ejemplo anterior, usamos ``pwd`` porque esa expresión ejecutará `pwd` y sustituirá la expresión con los resultados de ese comando antes de ejecutar `docker run`.

- Las opciones de destino es la ruta dentro del contenedor. También se puede escribir como `destino` o `dst`, es el mismo en todos los casos.
- La opción `readonly` es opcional. La utilizamos cuando no queremos que el contenedor tenga permisos de escritura en esa carpeta.

Puede enlazar la carpeta del mismo host a más de un contenedor. Esto les permitirá compartir datos.

4.7.2 Volúmenes

Los volúmenes son una solución más avanzada para la persistencia de datos. Se almacenan en una parte del sistema de archivos del host gestionada por Docker.

Para crear un volumen, utilice el siguiente comando:

```
docker volume create mi_volumen
```

Esto creará un nuevo volumen llamado `mi_volumen`. Para utilizar ese volumen en un nuevo contenedor:

```
docker run --rm -d -ti --name ubuntu_container4 \
  --mount type=volume,src=my_volume,dst=/mountpoint \
  ubuntu /bin/bash
```

Podemos listar los volúmenes actuales con `docker volume ls`, y obtener información detallada sobre el volumen con `docker volume inspect my_volume`.

Los contenedores no se eliminan cuando detenemos o destruimos el contenedor. Para eliminar un contenedor podemos ejecutar:

```
docker volume rm mi_volumen
```

También podemos eliminar todos los volúmenes no utilizados con `docker volume prune`.

4.7.3 montajes tmpfs

En hosts Linux, también podemos crear `montajes tmpfs`. Los datos de `los montajes tmpfs` son temporales (se borrarán cuando detengamos el contenedor) y se almacenarán en la memoria del host.

A diferencia de los volúmenes y los montajes bind, no puedes compartir montajes `tmpfs` entre contenedores.

4.8 Permisos

El usuario por defecto dentro de un contenedor es `root` (`id=0`). Así, por ejemplo, si creamos un contenedor con un montaje bind:

```
docker run --rm -ti --name ubuntu_container \
```

```
--mount type=bind,src=/tmp/docker,target=/docker \
ubuntu /bin/bash
```

El usuario en el contenedor es root. Si creamos un archivo

```
root@8d2cd626b49c:/# cd /docker/
root@8d2cd626b49c:/docker# whoami
root
root@8d2cd626b49c:/docker# touch a file
```

El archivo será creado en el host como si fuera creado por root. Se puede cambiar el usuario que ejecuta el contenedor con la opción `--user`:

```
docker run --rm -ti --name ubuntu_container \
--user `id -u`:`id -g`
--mount type=bind,src=/tmp/docker,target=/docker \
ubuntu /bin/bash
```

``id -u`:`id -g`` se sustituye por el usuario y grupo actuales en el host.

Podemos utilizar números para los parámetros `--user` y no es necesario que esos usuarios existan en el contenedor. Si usamos nombres, deben existir en el contenedor (el desarrollador de la imagen puede crear usuarios adicionales).

5. MATERIA SUPLEMENTARIA L

Introducción a Docker - Lo que necesita saber para empezar a crear contenedores

<https://medium.com/zero-equals-false/docker-introduction-what-you-need-to-know-to-start-crear-contenedores-8ffaf064930a>

Introducción a los contenedores y Docker

<https://endjin.com/blog/2022/01/introduction-to-containers-and-docker>

Un tutorial de Docker para principiantes <https://docker-curriculum.com/>

Contenedores:

<https://www.digitalocean.com/community/tutorials/working-with-docker-containers>

<https://www.baeldung.com/ops/docker-container-states>

Hojas de trucos:

<https://phoenixnap.com/kb/docker-commands-cheat-sheet>

<https://dockerlabs.collabnix.com/docker/cheatsheet/>