unir
LA UNIVERSIDAD DE INTERNET

UNIDAD FORMATIVA 3

Visualización, Cloud y Contenedores

Contenedores

Índice

Contenedores

Un **contenedor** es una unidad de software estandarizada que empaqueta el código de una aplicación y todas sus dependencias para permitir su ejecución de manera rápida y consistente, independientemente del entorno en el que se ejecute.

Los contenedores tienen la característica de ser livianos, ya que no necesitan la carga adicional de un hipervisor como ocurre con las máquinas virtuales, sino que se ejecutan directamente dentro del núcleo de la máquina host. Esto significa que con un determinado hardware podremos ejecutar mayor número de contenedores que si estuviéramos utilizando varias máquinas virtuales.

Docker es una plataforma de código abierto para desarrollar, distribuir y desplegar aplicaciones en entornos aislados y seguros denominados *contenedores*.

En está lección veremos qué es la tecnología Docker y qué ventajas supone su uso habitual. Aprenderemos cómo ejecutar imágenes ubicadas en cualquier registro (público o privado), así como lo más básico para construir un Dockerfile de nuestra aplicación.

A continuación se introducirá una herramienta que, basándose en Docker, va un paso más allá: Docker Compose nos permitirá definir la **orquestación de los contenedores** especificando cómo se relacionan entre sí y con el sistema host uno o más contenedores, que podrán existir ya o ser construidos al vuelo.

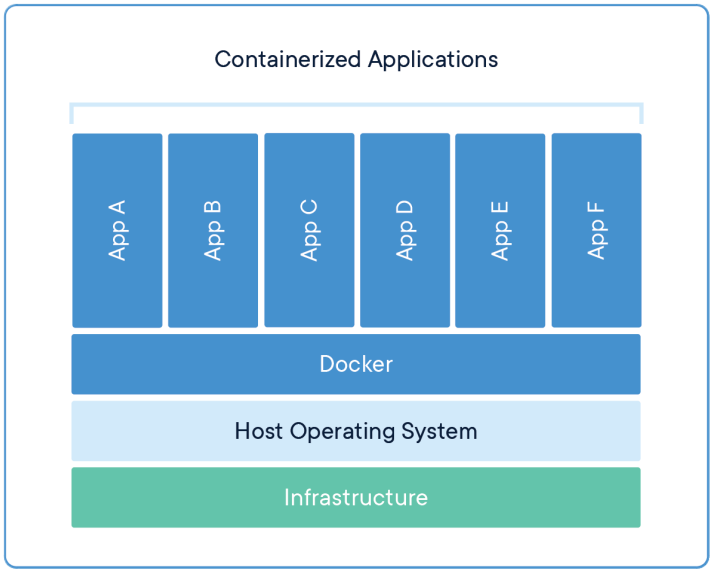
Es precisamente está versatilidad la que hace de Docker Compose una **herramienta multitarea** que expande las posibilidades de Docker varios órdenes de magnitud, pues permite que las aplicaciones declaren, a través de un fichero en el control de versiones, maneras de ejecutarse localmente, probarse en CI, simular un entorno de despliegue… Las posibilidades son infinitas, lo único que tenemos que hacer para poder sacar ventaja de esta tecnología es probar, probar y probar.

Objetivos

1. Conocer la tecnología Docker y qué supone frente a la virtualización basada en hipervisores.
2. Aprender a lanzar imágenes Docker para complementar el trabajo en local.
3. Definir, construir y subir a un registro nuestras propias imágenes.
4. Utilizar Docker Compose para orquestar el despliegue en local de varios contenedores.

Docker

Los contenedores tienen una filosofía totalmente diferente a la de las máquinas virtuales. En lugar de tener un sistema operativo completo, los contenedores comparten el mismo kernel o núcleo del sistema operativo host sobre el que se ejecutan.



Los contenedores comparten los recursos del host.

En resumen, los contenedores comparten los recursos subyacentes del host sobre el que se ejecutan. Además, los desarrolladores construyen una imagen que incluye exactamente lo que necesitan para ejecutar su aplicación o servicio, comenzando por lo básico y agregando solo lo necesario.

Las **máquinas virtuales** se construyen en la dirección opuesta, se empieza por un sistema operativo completo y, dependiendo de las necesidades de aplicación, iremos añadiendo o eliminando componentes.

La plataforma Docker nos permite **empaquetar nuestras aplicaciones con todo lo necesario para su ejecución** (librerías, código, herramientas, configuraciones), eliminando así dependencias del sistema operativo de la máquina host y facilitando un despliegue muy rápido de nuestras aplicaciones.

El motor de Docker será el encargado de **desplegar y gestionar los contenedores de nuestras aplicaciones**, pero, en lugar de reservar parte de los recursos de hardware de la máquina para cada contenedor, lo que hace es compartirlos entre todos los contenedores, permitiendo optimizar su uso y eliminando la necesidad de tener sistemas operativos separados para conseguir el aislamiento.

Ventajas de utilizar contenedores

En la administración tradicional de servidores de aplicaciones nos encontramos con una serie de **tareas que se repiten constantemente** en cada máquina: instalación del sistema operativo, actualizaciones y parches necesarios, instalar la última versión de la aplicación y sus dependencias.

A medida que aumenta el número de máquinas a gestionar, crece la dificultad de mantenerlas actualizadas con nuevas actualizaciones de seguridad, nuevas versiones de la aplicación, cambios en las dependencias, etc. Se trata de un proceso bastante propenso a errores.

Con la llegada de Docker tenemos la posibilidad de empaquetar en una imagen todo el código de nuestra aplicación, su sistema operativo, sus dependencias, binarios, ficheros de configuración (y solamente hacer este proceso una vez).

Si alguno de nuestros contenedores falla y termina no haría falta sacar a ese servidor del sistema para reparar el problema, sino que simplemente volveremos a desplegar una nueva instancia del contenedor de nuestra aplicación. De este modo, todas las instancias de la aplicación serían copias idénticas que no haría falta configurar individualmente.

Podríamos resumir el proceso de esta nueva visión en tres fases:

- **Construir la imagen**. Empaquetar consistentemente todo lo que nuestra aplicación necesita para ser ejecutada.
- **Distribuir la imagen**. Haremos disponible la imagen para utilizarla en nuestro datacenter, en la nube o en la máquina local del desarrollador.
- **Ejecutar la imagen**. Desplegar contenedores a partir de la imagen de una manera rápida, sencilla y consistente.

Veamos algunos escenarios en los que los contenedores nos ayudarán.

1. Despliegues portables y escalables

   Podremos ejecutar nuestros contenedores sabiendo que tendrán el mismo comportamiento en diferentes entornos, ya sea en el portátil de un desarrollador, en máquinas físicas o virtuales de nuestro datacenter, en servidores en la nube o en una combinación de ellos.

   La portabilidad de los contenedores y su reducido tamaño nos permiten gestionar las distintas cargas de trabajo, escalando las aplicaciones y servicios según lo exijan las necesidades casi en tiempo real.

2. Mejor aprovechamiento del hardware disponible

   Los contenedores son más ligeros y rápidos al desplegarse que las máquinas virtuales, lo que nos permite ejecutar más cargas de trabajo en el mismo hardware.

3. Trabajando con prototipos

   Los contenedores nos ofrecen entornos aislados con los que podremos probar diferentes configuraciones para nuestras aplicaciones sin necesidad de desplegar nuevas máquinas virtuales ni reconfigurar los sistemas existentes.

4. Empaquetando y versionando nuestro software

   Las imágenes de Docker nos aseguran que, al no tener dependencias con el sistema subyacente, nuestra aplicación se comportará de la misma manera en cualquier servidor que tenga Docker instalado. Además, podremos gestionar diferentes versiones de nuestras imágenes.

5. Actualizando las aplicaciones

   Cuando lanzamos una nueva versión de nuestro software o detectamos un defecto no es necesario parchear o actualizar las aplicaciones existentes, simplemente distribuiremos una nueva imagen con las modificaciones necesarias y desplegaremos un nuevo contenedor que reemplazará al actual. El tiempo de despliegue de un contenedor se mide en segundos.

6. Facilitando arquitecturas basadas en microservicios

   Trabajar con contenedores nos ayudará a descomponer nuestros sistemas complejos en componentes más pequeños y manejables, creando imágenes para cada uno.

7. Modelado de redes

   Podremos modelar grandes redes para simular escenarios de pruebas más cercanos a los entornos de producción, ya que podremos ejecutar cientos de contenedores en un mismo servidor con la configuración necesaria.

8. Facilitando la entrega continua

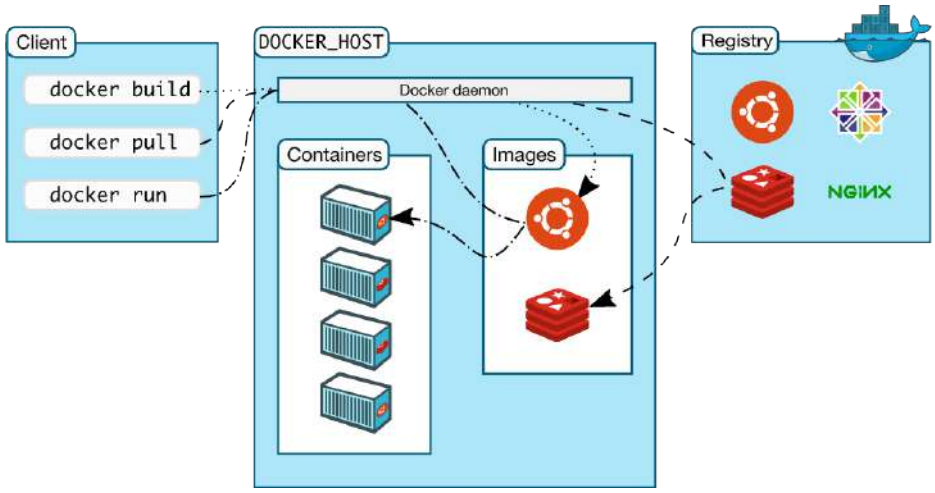
   Los sistemas basados en contenedores son más fácilmente reproducibles y replicables que los sistemas tradicionales, lo que nos permitirá probar de manera consistente nuestro código en los diferentes entornos (stage, uat, prod).

9. Entrega rápida y consistente de sus aplicaciones

   Se optimiza el ciclo de vida del desarrollo al permitir que los desarrolladores trabajen en entornos estandarizados utilizando contenedores locales que proporcionan sus aplicaciones y servicios. Los contenedores son excelentes para la integración continua y los flujos de trabajo de entrega continua (CI / CD).

Arquitectura de Docker

La arquitectura de Docker sigue un modelo cliente-servidor. En la siguiente imagen vemos los principales componentes de la arquitectura de Docker.



El demonio de Docker

El demonio de Docker (proceso *dockerd*) escucha las solicitudes del API y es el responsable de administrar los objetos de Docker, como imágenes, contenedores, redes y volúmenes. Un demonio también puede comunicarse con otros demonios para administrar los servicios de Docker de forma distribuida.

El cliente Docker

El cliente Docker (comando docker) permite a los usuarios interactuar con el demonio de Docker utilizando la línea de comandos. El cliente Docker y el demonio de Docker se comunican mediante un API REST, ya sea a través de sockets UNIX o mediante una interfaz de red. Ambos no tienen por qué ejecutarse en la misma máquina, ya que el cliente Docker podrá conectarse a un demonio de Docker de un sistema remoto.

Imágenes

Las imágenes son objetos de Docker que nos servirán para empaquetar una aplicación o servicio junto a todo lo necesario para su funcionamiento: código de la aplicación, librerías dependientes, configuraciones, etc.

Contenedores

Los contenedores son instancias o ejecuciones de la imagen de nuestra aplicación o servicio. Los contenedores se comportan como entornos aislados y seguros, por lo que en una misma máquina podemos tener varios contenedores en ejecución de la misma imagen.

Para entender mejor la **diferencia entre imágenes y contenedores** podemos ver su paralelismo con un fichero ejecutable y el proceso que lo ejecuta. La imagen sería el equivalente al ejecutable, y el contenedor similar a los distintos procesos que hay en ejecución del ejecutable.

Registros de Docker

Los registros de Docker permiten almacenar y distribuir imágenes de Docker que luego serán utilizadas para la creación de los contenedores. **Docker Hub** es un registro público que puede ser utilizado por los usuarios para compartir imágenes.

Por defecto, Docker buscará imágenes en Docker Hub, aunque, como veremos, existen otros repositorios públicos en la nube. Además, podremos disponer de nuestro propio registro de imágenes privado.

Tecnología detrás de Docker

Docker está escrito en el lenguaje de programación Go y utiliza varias características del kernel de Linux para ofrecer su funcionalidad. Recordemos que todos los contenedores en ejecución y la máquina host comparten el kernel.

1. Espacios de nombres (namespaces)

Docker utiliza una tecnología llamada espacios de nombres o *namespaces* para proporcionar un espacio de trabajo aislado a los contenedores. Cuando se ejecuta un contenedor, Docker crea un conjunto de espacios de nombres para ese contenedor.

Estos espacios de nombres crean una capa de aislamiento. Cada aspecto de un contenedor se ejecuta en un espacio de nombres separado y su acceso está limitado a ese espacio de nombres.

Algunos de los **namespaces de Linux utilizados por Docker** son:

- Espacio de nombres pid: aislamiento de los procesos.
- Espacio de nombres net: gestión de las interfaces de red.
- Espacio de nombres ipc: gestión del acceso a los recursos de IPC.
- Espacio de nombres mnt: gestión de puntos de montaje del sistema de archivos.
- Espacio de nombres uts: Aislamiento del núcleo y los identificadores de versión.

2. Grupos de control (cgroups)

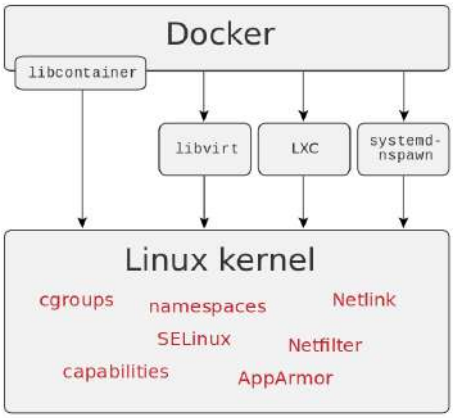
Docker Engine en Linux también se basa en otra tecnología llamada **grupos de control** (*cgroups*). Un grupo de control limita una aplicación a un conjunto específico de recursos. Los grupos de control permiten que Docker Engine comparta los recursos de hardware disponibles con los contenedores y, opcionalmente, imponga límites y restricciones. Por ejemplo, puede limitar la memoria disponible para un contenedor específico.

3. Sistemas de archivos de unión (UnionFS)

En Linux, el servicio de sistemas de archivos de unión (o UnionFS) permite montar un sistema de archivos formado por la unión de otros sistemas de archivos. Funciona creando capas, haciéndolos muy livianos y rápidos. Docker Engine utiliza UnionFS para proporcionar los bloques de construcción para contenedores. Docker Engine puede usar múltiples variantes de UnionFS, incluidas AUFS, btrfs, vfs y DeviceMapper.

4. Formato de contenedor

Docker combina servicios de Linux, como son los espacios de nombres, grupos de control o los sistemas de archivos de unión, en un envoltorio denominado *formato contenedor*. El formato de contenedor predeterminado de Docker es *libcontainer*.



libcontainer ofrece abstracción para virtualización y aislamiento de contenedores.

Instalación de Docker

En Linux existen varias formas de instalar Docker:

- **Utilizar los repositorios de Docker.** Facilitan la instalación y las tareas de actualización. Es la opción recomendada y elegida por la mayoría de los usuarios.
- **Descargar e instalar el paquete manualmente**. La administración de las actualizaciones también será manual. Este método es útil en sistemas sin acceso a internet.
- **Utilizar los scripts de instalación que Docker ofrece**. No recomendable para entornos de producción, pero comúnmente utilizado para la automatización desatendida en entornos de prueba y desarrollo.

En Windows y Mac OS esto no es posible pues, como sabemos, no existe soporte nativo para contenedores Docker: deberemos utilizar la aplicación Docker Desktop para poder trabajar con Docker.

Para más detalles sobre estos procesos se recomienda encarecidamente consultar la documentación oficial o consultar las guías que se han dejado en el Anexo I: Instalación de Docker, el que encontrarás más adelante en este mismo fastbook.

Ejecución de una imagen

Vamos a aprender ahora cómo ejecutar localmente una imagen con Docker. La utilidad de esto es enorme, pues, como veremos, nos permitirá descargar imágenes de un tamaño mucho menor que el de una máquina virtual, preconfiguradas en muchos casos, y empezar a usarlas para multitud de casos:

- Probar si una aplicación que estamos desarrollando funciona en una versión específica de un sistema operativo.
- Ejecutar la última versión de la herramienta que deseemos sin tener que descargarla y configurarla.
- Lanzar una base de datos preconfigurada y usarla durante el desarrollo.

Las utilidades de las imágenes son casi ilimitadas pero, ¿de dónde salen? ¿Dónde puedo encontrarlas? La respuesta a esto son los ***Docker registries***.

Registros Docker

Los registros Docker nos permiten almacenar las imágenes de Docker que vayamos creando de manera que estén disponibles para su descarga, cuando sea necesario, desde nuestros sistemas. Docker por defecto utilizará su registro Docker Hub, que ofrece repositorios tanto públicos como privados.

Sin embargo, Docker Hub no es el único registro disponible públicamente, aunque sí el más popular. Existen otros registros con sus propias características y funcionalidades. Según nuestras necesidades podemos decantarnos por uno u otro atendiendo al coste, rendimiento, seguridad, geo-replicación, etc. Además, también podríamos crearnos un registro privado en nuestro datacenter o nuestra cloud privada y conseguir lo mismo pero con el añadido de la seguridad para la organización.

En el caso de que usemos un registro privado que requiera autenticación o pretendamos subir una imagen a un registro público, lo primero que debemos hacer es un comando de *login:*

$ docker login -u <user> -p <password> url.to/my_registry

Login Succeeded!

Ejecución de imágenes

Una vez sabemos qué imagen de Docker queremos ejecutar, tenemos varias opciones:

- *docker pull* si tan solo queremos descargarla para ejecutarla después o para poder usarla en una construcción.
- *docker run* si queremos ejecutar un comando disponible dentro de la imagen.

Si por el contrario existe uno o más contenedores ejecutándose, y queremos meternos en uno de ellos para inspeccionar el estado, podemos usar *docker exec* para hacerlo.

Vamos a ver todo esto con un **ejemplo** en el que se usarán las opciones de configuración más comunes de estos comandos, que se pueden consultar en la web de Docker o usando el modificador *--help* de cada comando docker empleado.

- *Ejecutar última versión de Ubuntu de forma interactiva:*

```
$  docker run -it ubuntu:latest
root@68f711cd7f24:/# ls
bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp
usr  var
root@68f711cd7f24:/# cat /etc/de
debconf.conf  debian_version  default/      deluser.conf
root@68f711cd7f24:/# cat /etc/debian_version
bullseye/sid
```

- *Ejecutar nginx de forma no interactiva:*

```
$ docker run nginx:latest
docker/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: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

- *Ejecutar nginx en segundo plano y conectarse al contenedor en marcha:*

```
$  docker run -d nginx:latest
946c55f139ff69726e815e80b78ae14b946fc59f61bf26c3c61c201d16703395
$  docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS         PORTS     NAMES
946c55f139ff   nginx:latest   "/docker-entrypoint...."   3 seconds ago   Up 1 second   80/tcp
beautiful_herschel
$ docker exec -it 946c55f139ff bash
root@946c55f139ff:/# curl 127.0.0.1:80
<!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>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
```

- *Matar ejecuciones pendientes:*

```
$ docker kill 946c55f139ff
```

- *Enlazar puerto del contenedor con puerto local:*

```
$ docker run -d  -p 3000:80 nginx:latest
$ curl 127.0.0.1:3000
<!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>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Este ejemplo muestra cómo podríamos levantar una serie de contenedores y dejarlos ejecutándose en un puerto local, para lo que los necesitemos: bases de datos, servicios web...

- *Montar sistema de ficheros local:* Docker solo es capaz de montar rutas absolutas, por lo que, si queremos montar una carpeta dentro del directorio donde estamos, debemos anteponer el path. Una manera sencilla con bash sería así:

```
 $  ls src/
main.tf

$  docker run -ti -v $(pwd)/src:/code ubuntu:latest
root@8260a98a5fb3:/# ls /code/
main.tf
root@8260a98a5fb3:/#
```

Este ejemplo muestra cómo podríamos usar Ubuntu, como máquina de desarrollo o de pruebas, estando en otro sistema operativo diferente. Muy útil si estamos tratando de reproducir un error en una máquina remota.

- *Actualizar estado imagen:* si quisiéramos guardar el estado del contenedor que acabamos de crear podríamos ejecutar desde un nuevo terminal de nuestra máquina el comando **docker commit** para persistir el estado actual del contenedor. Será necesario indicarle el identificador o nombre del contenedor y, opcionalmente, el nombre que queremos darle a nueva imagen:

```
$ docker commit 0ef5d5b9795d myapp:version1
sha256:cd6827d1d4f348ea41fcafa55c49cd76354706f8860f244852329df16008e86a
```

Ahora podríamos hacer los cambios que quisiéramos probar directamente en el contenedor. A modo de ejemplo, podríamos actualizar el gestor de paquetes e instalar o actualizar alguna librería:

```
root@0ef5d5b9795d:/# apt-get update
root@0ef5d5b9795d:/# apt-get install nmap
```

Si ahora volviéramos a guardar el estado del contenedor en una nueva imagen y listamos las imágenes disponibles veríamos las dos imágenes creadas y cómo la segunda tiene un tamaño mayor al haber realizado algunos cambios en el contenedor:

```
$ docker commit 0ef5d5b9795d myapp:version2
sha256:e5849e3ac4b085045dd46d5ea427290a0a1be19cb20ea6da54a639dec0a145ca
$ docker images myapp
REPOSITORY      TAG          IMAGE ID       CREATED         SIZE
myapp           version1     cd6827d1d4f3   4 minutes ago   64.2MB
myapp           version2     e5849e3ac4b0   14 seconds ago  153MB
```

Una vez que tenemos nuestras imágenes creadas, podríamos ejecutar un contenedor a partir de cualquiera de ellas indicando su nombre y etiqueta:

```
$ docker run -it myapp:version1 /bin/bash
```

Consideraciones

- Los parámetros '-it' casi siempre van juntos, pero en realidad son dos distintos:
  - -i especifica que la sesión va a ser interactiva, dejando STDIN abierto.
  - -t (--tty) prepara una sesión TTY dentro del contenedor que lee de STDIN.
  - Entre las dos, se consigue de manera efectiva el poder enviarle comandos al contenedor.

- En caso de no querer que se ejecute el comando establecido por defecto, existe la opción ***--entrypoint*** que permite elegir el binario: un ejemplo usual es ejecutar *bash* en contenedores que, de otra manera, levantan una base de datos o un servicio web.
- El comando ***docker attach*** se comporta de forma muy similar a ***exec***, pero no crea un nuevo terminal, sino que se conecta a lo se esté ejecutando.

Aplicación: Registro Docker privado

Los registros privados de contenedores nos permitirán crear y publicar imágenes en un host al que solo pueda acceder nuestro equipo, empresa o clientes. Docker nos proporciona la imagen ***registry*** para crearnos nuestro propio registro.

Esto nos permite controlar el acceso al mismo mediante mecanismos de autenticación propios, certificados SSL privados o, incluso, usar un backend de *storage* personalizado dentro de nuestra cloud de elección (por ejemplo, un *bucket* de AWS S3). En el caso de servicios como ***Kubernetes***, un *registry* privado es la única manera de utilizar imágenes propias dentro del clúster.

El siguiente comando crea un Docker Registry privado en un contenedor:

- Accesible a través del puerto 5000 del host.
- Con almacenamiento un directorio local.
- Disponible como servicio, siempre levantado mientras el *demonio* Docker esté funcionando.

```
$ docker run -d -p 5000:5000 -v $HOME/registry:/var/lib/registry \
  --restart always -name registry registry:2
```

En la documentación oficial de Docker podemos consultar más opciones y casos de uso de los registros privados: Deploy a registry server.

Volúmenes

Como hemos visto, es posible montar uno o más directorios locales a los contenedores: esto es lo que se conoce como un volumen de tipo ***bind mount.***

```
$ docker inspect 8260a98a5fb3 | jq .[0].Mounts
[
 {
  "Type": "bind",
  "Source": "/data/applications/edix/pyapp_example/src",
  "Destination": "/code",
  "Mode": "",
  "RW": true,
  "Propagation": "rprivate"
 }
]
```

Procesamiento json en CLI

En el comando anterior se ha introducido una herramienta de línea de comandos nueva: el procesador de json jq. Está potentísima utilidad está disponible precompilada en los principales sistemas operativos y su uso se recomienda **siempre** que vayamos a procesar contenido en JSON desde scripts/líneas de comandos.

Su aprendizaje a fondo está fuera del alcance de este curso, aunque, en la medida de lo posible, se mostrarán ejemplos sencillos que demuestran su utilidad. Para más detalles sobre la herramienta se recomienda consultar su propio tutorial.

Existen más tipos de volúmenes, como *tmpfs* (volumen muy rápido, pues se monta desde la memoria RAM del host, pero no persiste) o los volúmenes propios de Docker.

Estos volúmenes se pueden crear desde la CLI de Docker y se pueden montar en cualquier número de contenedores, que compartirán su contenido entre ellos. Como desventaja, no pueden utilizar una carpeta local.

Veamos un ejemplo de creación y uso de un volumen de Docker:

```
$ docker volume create mis-datos
mis-datos

$ docker volume ls
DRIVER              VOLUME NAME
local               mis-datos

$ docker volume inspect mis-datos
[
    {
        "CreatedAt": "2020-07-12T09:15:03Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/mis-datos/_data",
        "Name": "mis-datos",
        "Options": {},
        "Scope": "local"
    }
]
```

Este último tipo de volumen es la manera recomendada en Docker de persistir los datos de nuestros contenedores. Algunas de sus **principales ventajas** son:

- Las copias de seguridad y migración a otros hosts son sencillas.
- Podemos gestionarlos tanto con comandos de Docker como mediante su API.
- Funcionan en Linux y Windows.

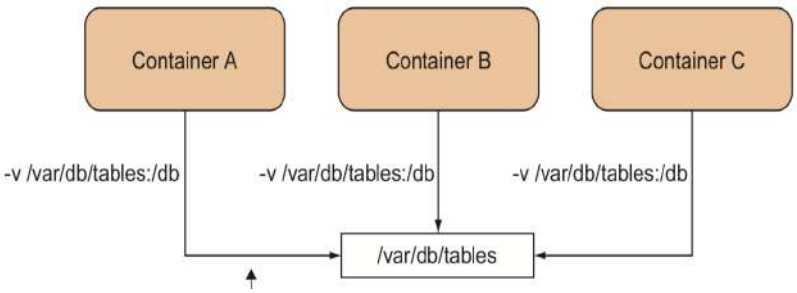
- Se pueden compartir de manera más segura entre múltiples contenedores.
- Existen controladores que nos permitirán tenerlos en hosts remotos o añadir características como, por ejemplo, cifrar su contenido.

Contenedores de datos

Cuando utilizamos muchos volúmenes de datos puede ser complicado gestionar el arranque de los contenedores. En estos casos podemos utilizar el patrón de diseño de contenedor de datos, que nos permitirá simplificar la gestión de nuestros volúmenes de datos.

Un contenedor de datos será un contenedor que **solamente tendrá volúmenes montados y no ejecutará ninguna aplicación**. De hecho, no es necesario que el contenedor esté en ejecución para poder montar sus volúmenes en otros contenedores, puede estar detenido perfectamente.

La siguiente imagen muestra cómo montaríamos el mismo volumen en varios contenedores sin utilizar un contenedor de datos. Vemos que es necesario indicar la ruta en cada uno de ellos.



Contenedores montando el mismo directorio del host.

Utilizando volúmenes de datos, cuando un contenedor monta los volúmenes **no necesitará saber dónde se encuentran los datos** en el disco. Solamente necesitará saber el nombre del contenedor de datos, lo que lo hace mucho más portable.

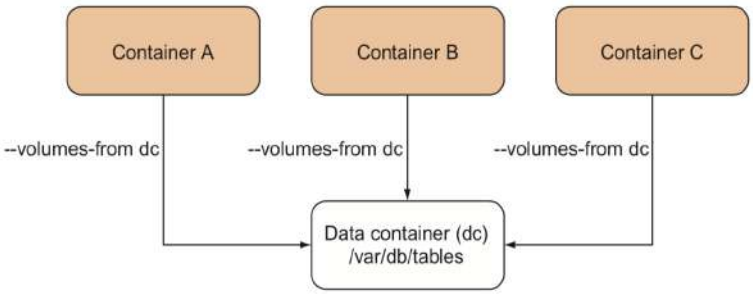
Para crear nuestro contenedor de datos, lo habitual es utilizar como imagen base ***busybox***: se trata de una imagen muy reducida, con funcionalidades básicas de Linux, pero más ligera que ***alpine***. Ya que no vamos a ejecutar nada en el contenedor, nos será suficiente:

```
$ docker run -v /datos-compartidos --name contenedorDatos busybox \
  touch /datos-compartidos/ficheroVacio
```

Si queremos copiar archivos en el contenedor de datos tendremos que utilizar el comando **docker cp**, ya que no es accesible directamente desde el host. El siguiente comando copiaría un fichero de configuración en el directorio */config* del contenedor de datos:

```
$ docker cp config.conf contenedorDatos:/config/
```

Una vez creado y configurado nuestro contenedor de datos, montaremos sus volúmenes al ejecutar otros contenedores con la opción ***–volumes-from***, indicando el identificador o nombre del contenedor de datos.



Contenedores enlazados a un contenedor de datos.

```
$ docker run -it --name ContenedorA \
   --volumes-from contenedorDatos busybox /bin/sh
/ # ls /datos-compartidos
ficheroVacio
```

Debemos tener en cuenta que, **si ya existía un directorio** */config* en la imagen del contenedor que estamos ejecutando, los volúmenes montados **anularían su contenido** y sería el contenido del volumen del contenedor de datos el que prevalece.

Importar y exportar contenedores de datos

Cuando utilizamos contenedores de datos, el proceso de moverlos de una máquina a otra se simplifica muchísimo. Podemos exportarlos a un fichero TAR y después importarlo en otra máquina:

```
$ docker export contenedorDatos > contenedorDatosExport.tar
$ docker import contenedorDatosExport.tar
```

Redes en Docker

Los contenedores, además de compartir almacenamiento por medio de volúmenes, también pueden relacionarse unos con otros compartiendo redes. En esta sección veremos los distintos tipos de redes disponibles en Docker y cuáles son sus **principales usos y funcionalidades**:

- **bridge**: es el controlador por defecto. Permite la comunicación entre dos contenedores independientes ejecutándose en el mismo anfitrión o host. Los contenedores creados se asociarán por defecto a la red existente llamada *bridge*. Pero podremos crear tantas redes de este tipo como necesitemos.

- **host:** permite eliminar el aislamiento de red entre el contenedor y el anfitrión.

- **overlay**: nos permitirá conectar contenedores y/o servicios de Docker Swarm corriendo en diferentes demonios Docker, es decir, en distintos nodos.

- **macvlan**: nos permite asignar una dirección MAC a un contenedor, permitiendo así redirigir directamente tráfico de red hacia ellos.

- **none**: deshabilita todas las redes del contenedor.

Si listamos las redes iniciales que tenemos tras la instalación de Docker, veremos solamente tres redes: la red por defecto de tipo bridge, la red que utilizará un contenedor con el controlador host y, por último, la red none que simplemente tiene la interfaz de bucle invertido.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1620617a10d6        bridge              bridge              local
707918bf2b62        host                host                local
f12a0ea3db2b        none                null                local
```

Si no indicamos explícitamente ningún tipo de red a nuestros contenedores al crearlos, estos se crearán por defecto en una red de tipo bridge. Podemos comprobarlo creando un par de contenedores en segundo plano (*detach*) e interactivos para poder conectarnos a ellos:

```
$ docker run -dit --rm --name contenedorA alpine sh
$ docker run -dit --rm --name contenedorB alpine sh
```

Si inspeccionamos la red bridge veremos que tiene asociados nuestros dos contenedores, cada uno en su propia subred:

```
$ docker network inspect bridge | jq .[0].Containers
{
  "55622a13407b3f4d7f144c2e3287e971a476ea7d8281185e96624a84c58b1815": {
    "Name": "contenedorB",
    "EndpointID": "e335702f2028c8fbea45cd00b2e63487eafb8c7e6d84831c2985191d536c9681",
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
  },
  "bb017f0c56e9fa59cfce5ef166337120115c4e0adb4fd1c6e2d1782d7bf37b32": {
    "Name": "contenedorA",
    "EndpointID": "e0af55f5b7b199f1b22d0cb215d653cd373b55f57ed133ef925e5f5e18182885",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
}
```

Si nos conectamos a uno de los contenedores veremos que podemos llegar al otro contenedor por su dirección IP, pero no resuelve el nombre de su red.

Importante

Podemos salir del contenedor con Ctrl^P^Q sin terminarlo; si salimos con Ctrl^D terminamos el proceso en marcha y docker lo mataría.

```
$ docker attach contenedorA
/ # ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.157 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.137 ms
--- 172.17.0.3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.137/0.147/0.157 ms
/ # ping contenedorB
ping: bad address 'contenedorB'
```

Enlazando contenedores directamente

Ahora vamos a recrear el *contenedorB,* pero esta vez enlazándolo al contenedorA con la opción ***--link***. Esto nos proporcionará un alias para acceder al contenedor enlazado. Realmente, Docker añadirá una entrada en el fichero /etc/hosts con el nombre del contenedor enlazada y el alias:

```
$ docker stop contenedorB
$ docker run -dit --rm --name contenedorB \
   --link contenedorA:aliasA alpine sh
```

Si nos conectamos al *contenedorB* podemos ver la entrada en el fichero ***/etc/hosts*** añadida y, además, comprobaremos que ahora sí que resuelve por nombre al contenedor A. Sin embargo, desde el *contenedorA* **seguiremos sin resolver** el *contenedorB*:

```
$ docker attach contenedorB
/ # cat /etc/hosts
...
172.17.0.2        aliasA d8f56632f3c2 contenedorA
/ # ping contenedorA
PING alpine1 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.228 ms
--- contendorA ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.131/0.165/0.228 ms
/ # read escape sequence

$ docker attach contenedorA
/ # ping contenedorB
ping: bad address 'contenedorB'

$ docker stop $(docker ps -q)
```

Reto

¿Qué hace ese último comando? Parece que para todos los contenedores en marcha pero ¿cómo lo consigue?

Enlazando contenedores por red

Esta es la opción recomendada, pues el comando ***--link*** está en proceso de desaparecer. Para enlazar contenedores vía red vamos a crear una nueva red de tipo *bridge* y adherirlos a ella:

```
$ docker network create --driver bridge miRed
$ docker run -dit --rm --name contenedorA --network miRed alpine sh
$ docker run -dit --rm --name contenedorB --network miRed alpine sh
$ docker network inspect miRed  | jq .[0].Containers
  contenedorB - 172.18.0.3/16
  contenedorA - 172.18.0.2/16

$ docker attach contenedorA
/ # ping contenedorB
PING contenedorB (172.18.0.3): 56 data bytes
...
4 packets transmitted, 4 packets received, 0% packet loss
```

```
$ docker attach contenedorB
/ # cat /etc/hosts
...
172.17.0.2        aliasA d8f56632f3c2 contenedorA
/ # ping contenedorA
PING alpine1 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.228 ms
--- contendorA ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.131/0.165/0.228 ms
/ # read escape sequence

$ docker attach contenedorA
/ # ping contenedorB
ping: bad address 'contenedorB'

$ docker stop $(docker ps -q)
```

Reto

¿Qué hace ese último comando? Parece que para todos los contenedores en marcha pero ¿cómo lo consigue?

Enlazando contenedores por red

Esta es la opción recomendada, pues el comando ***--link*** está en proceso de desaparecer. Para enlazar contenedores vía red vamos a crear una nueva red de tipo *bridge* y adherirlos a ella:

```
$ docker network create --driver bridge miRed
$ docker run -dit --rm --name contenedorA --network miRed alpine sh
$ docker run -dit --rm --name contenedorB --network miRed alpine sh
$ docker network inspect miRed  | jq .[0].Containers
  contenedorB - 172.18.0.3/16
  contenedorA - 172.18.0.2/16

$ docker attach contenedorA
/ # ping contenedorB
PING contenedorB (172.18.0.3): 56 data bytes
...
4 packets transmitted, 4 packets received, 0% packet loss
```

La gran ventaja de esta opción es que escala con el número de contenedores y de redes. Por ejemplo, si creamos otra red más, podemos adherir un contenedor **también** a ella, creando de este modo una conexión entre redes distintas a nivel de IP:

```
$ docker network create --driver bridge miRed2
$ docker network connect miRed2 contenedorB

# Si inspeccionamos el contenedor lo veremos conectado a las dos redes
$ docker inspect contenedorB | jq .[0].Containers
    miRed - 172.18.0.3
    miRed2 - 172.19.0.2
```

Creación de una imagen

Los contenedores de Docker están basados en imágenes previamente generadas que contienen todas las dependencias necesarias para nuestra aplicación o servicio. Aunque existen de manera pública multitud de imágenes listas para ser utilizadas, la mayoría de veces necesitaremos crear nuestras propias imágenes, habitualmente basadas en alguna ya disponible.

Para automatizar la creación de imágenes de Docker describiremos los pasos necesarios para la construcción en un manifiesto de Docker llamado ***Dockerfile***.

Un **fichero Dockerfile** es un archivo de texto que contiene las instrucciones que se usarán para compilar y ejecutar una imagen de Docker. En el Dockerfile se definen los siguientes aspectos de la imagen:

- La imagen base sobre la que se creará la nueva imagen.
- Comandos necesarios para actualizar el sistema operativo e instalar el software y dependencias necesarias.
- El código o empaquetado de nuestra aplicación.
- Puertos que expondrá el contenedor, así como la configuración de red y almacenamiento.
- El comando que se ejecutará al iniciarse el contenedor.

Para obtener información más completa y detallada de todas las instrucciones disponibles consulta la referencia en la documentación oficial de Docker, o bien, en la propia documentación de la herramienta:

```
$ docker build --help
```

Ejemplo. Construyendo una imagen para una aplicación de Python

Veamos con un ejemplo cómo podemos generar una imagen de Docker para una aplicación sencilla de Python utilizando un Dockerfile. Supongamos que tenemos nuestra aplicación Python según la siguiente estructura de directorios:

```
app
├── Dockerfile
├── requirements.txt
└── src
    ├── ...
    └── server.py
```

El Dockerfile para nuestra aplicación podría tener un aspecto similar al siguiente:

```
# Establecemos la imagen base sobre la que construiremos nuestra imagen
FROM python:3.8.5

# Establecemos el directorio de trabajo del contenedor
WORKDIR /code

# Copiamos el fichero de dependencias al directorio de trabajo
COPY requirements.txt .

# Instalamos las dependencias definidas en el fichero
RUN pip install -r requirements.txt

# Copiamos nuestra aplicación de directorio local src al dir. de trabajo
COPY src/ .

# Configuramos los requerimientos de red
EXPOSE 5000

# Comando que se ejecutará al iniciarse el contenedor
CMD [ "python", "./server.py" ]
```

Una vez que tenemos creado nuestro fichero Dockerfile, podremos generar nuestra imagen con el comando ***docker build***, indicando el nombre de la imagen y opcionalmente una etiqueta que, por defecto, será *latest*. Si nos fijamos en la salida del comando veremos que cada instrucción la ejecuta en un paso separado:

```
$ docker build -t python_app .
Sending build context to Docker daemon  5.12kB
Step 1/7 : FROM python:3.8.5
3.8.5: Pulling from library/python
...
Status: Downloaded newer image for python:3.8.5
 ---> 79cc46abd78d
Step 2/7 : WORKDIR /code
 ---> Running in a7cc669c40a4
Removing intermediate container a7cc669c40a4
 ---> 34a49c2912d5
Step 3/7 : COPY requirements.txt .
 ---> 0a1810e7a142
Step 4/7 : RUN pip install -r requirements.txt
 ---> Running in 4cad7d8b2c1f
...
---> 030d148118da
Step 5/7 : COPY src/ .
 ---> f82da4308992
Step 6/7 : EXPOSE 5000
 ---> Running in d47dbd078182
Removing intermediate container d47dbd078182
 ---> 965a5b1775dd
Step 7/7 : CMD [ "python", "./server.py" ]
 ---> Running in 4f1e1641d47b
Removing intermediate container 4f1e1641d47b
 ---> f855c94ad5df
Successfully built f855c94ad5df
Successfully tagged python_app:latest
```

Ya tenemos generada nuestra imagen. Si hiciéramos cambios en el código fuente de la aplicación o en el fichero de dependencias podríamos regenerar la imagen volviendo a ejecutar el mismo comando *docker build*. Además, podemos ver las diferentes capas que componen una imagen utilizando el comando ***docker history***:

```
$ docker image history python_app:latest
IMAGE          CREATED          CREATED BY                                      SIZE
859fb1db4a90   3 minutes ago    /bin/sh -c #(nop)  CMD ["python" "./server.p...   0B
867e1b296ff9   3 minutes ago    /bin/sh -c #(nop)  EXPOSE 5000:5000             0B
2cf8552ed9ca   3 minutes ago    /bin/sh -c #(nop) COPY dir:9b3e3bb266ad8957c...   164B
45dd0cfd49a7   3 minutes ago    /bin/sh -c pip install -r requirements.txt      9.63MB
9c7c03f19b8a   3 minutes ago    /bin/sh -c #(nop) COPY file:dcf08683799a1e65...   13B
5a3811ed82ae   3 minutes ago    /bin/sh -c #(nop) WORKDIR /code                 0B
79cc46abd78d   5 days ago       /bin/sh -c #(nop)  CMD ["python3"]              0B
<missing>      5 days ago       /bin/sh -c set -ex;   wget -O get-pip.py "$P...   7.24MB
```

Ya solamente nos quedaría ejecutar un contenedor a partir de nuestra imagen, exponiendo el puerto del contenedor al host:

```
$ docker run -d -p 5000:5000 python_app
$ curl http://localhost:5000
```

Argumentos y variables de entorno

Es muy usual que necesitemos ciertos valores dentro del Dockerfile que no se conocen a la hora de escribirlo o que son credenciales privadas que, lógicamente, no deben estar en el fichero y se le deben inyectar en tiempo de creación de imagen.

Para **solucionar estos problemas** tenemos dos opciones (complementarias, no excluyentes) que son los argumentos de entrada y las variables de entorno.

Argumentos de entrada

Se trata de parámetros que se pasan al Dockerfile a través de la sentencia de build:

```
$ docker build --build-arg arg1 [--build-arg arg2 ...]
```

Dentro del Dockerfile, estos parámetros se leen usando la palabra reservada ***ARGS***, que admite un valor por defecto en caso de que el parámetro no se encuentre presente.

Variables de entorno

El contenedor tiene acceso a todas las variables de entorno disponibles en el momento de ejecutar la construcción, pero solo puede usar aquellas que se declaren explícitamente con la palabra reservada ***ENV,*** que también admite un valor por defecto.

Veamos un ejemplo de ambos casos:

```
FROM debian:buster-slim

ARG USER_NAME=tomcat
ARG WORKDIR="/home/$USER_NAME"
ARG USER_PROFILE_FILE="$WORKDIR/.bashrc"

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Elevamos permisos para crear un usuario
USER root
RUN set adduser --uid 1000 --disabled-password --system $USER_NAME

# Restringimos permisos para evitar crear recursos como root
USER $USER_NAME
ENV PATH=${WORKDIR}/.tomcat/bin:$PATH
```

Construcción multietapa

La construcción en múltiples etapas, o multi-stage builds, es una nueva funcionalidad introducida en la versión 17.05 que nos permite optimizar y reducir la complejidad de nuestros Dockerfiles.

Para definir las etapas utilizaremos múltiples sentencias FROM en el mismo fichero Dockerfile. Cada sentencia FROM iniciará un nuevo proceso de construcción y puede utilizar una imagen base distinta. Además, podremos copiar ficheros de una etapa anterior a la actual con la instrucción COPY, referenciando la etapa origen con --from. Las etapas se numeran partiendo desde 0, pero podemos darles un nombre añadiendo AS name al final de la instrucción FROM.

En el siguiente Dockerfile de ejemplo podemos ver cómo se utiliza una primera etapa para compilar una aplicación en Go, utilizando la imagen golang como base. En una segunda etapa simplemente copiamos el ejecutable sobre la imagen base alpine. Esto nos permite quedarnos solamente con los artefactos que queramos de la primera etapa, reduciendo así el tamaño final de la imagen.

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Buenas prácticas en el diseño de Dockerfiles

Veamos algunas buenas prácticas que nos ayudarán a crear ficheros Dockerfiles que nos permitan reducir el tiempo de construcción, mantener un tamaño de imagen reducido y que, a su vez, sean mantenibles y reproducibles.

Tiempo de construcción incremental

Durante el ciclo de vida de nuestras aplicaciones iremos realizando cambios y actualizaciones que harán que tengamos que regenerar nuestras imágenes. Saber utilizar caché nos ayudará a no repetir pasos de nuestro Dockerfile durante la reconstrucción, optimizando el tiempo destinado a esta tarea.

Ordena los pasos de menor a mayor frecuencia de cambio

El orden de las instrucciones o pasos de nuestro Dockerfile es importante, ya que cuando la caché se invalida por un cambio en una instrucción o en algunos ficheros, todos los pasos posteriores también son invalidados en esta y se volverán a ejecutar.

Un ejemplo **muy típico** es el de los comandos de instalación de paquetes: una instrucción del tipo ***apt get install*** o ***pip install -r requirements.txt*** siempre va a tardar una cantidad de tiempo nada despreciable. Si tenemos comandos **antes** de esas sentencias que invalidan la caché, cada vez que los modifiquemos volveremos a descargar e instalar paquetes, una y otra vez.

Se lo más específico posible con las instrucciones COPY

Evita copiar directorios completos copiando solamente lo necesario. Cualquier cambio en los ficheros que copiamos con COPY invalidará la caché.

Identifica conjuntos de instrucciones 'cacheables'

Deberíamos evitar el uso intensivo de la instrucción RUN, así como encadenar demasiados comandos en una misma instrucción. Cada instrucción RUN es susceptible de ser cacheada, por ello deberemos identificar correctamente qué comandos deberían ejecutarse como un conjunto y ejecutarlos en la misma instrucción RUN.

Al ser comandos Linux, se pueden concatenar usando el operador && y separar las líneas (para facilitar la lectura) usando el operador \

```
# set -ex se usa para tener un output más detallado de los pasos
RUN set -ex \
  && apt update --assume-no \
  && apt-get install -y \
    build-essential libssl-dev zlib1g-dev libbz2-dev \
    libreadline-dev libsqlite3-dev wget curl llvm libncurses5-dev libncursesw5-dev \
    xz-utils tk-dev libffi-dev liblzma-dev python-openssl \
    \
    git curl bash \
  && apt clean && rm -rf /var/lib/apt/lists/*
```

Utiliza imágenes oficiales siempre que sea posible

Las imágenes oficiales publicadas ya incluyen todos los pasos necesarios de la instalación y, además, están generadas siguiendo buenas prácticas. Todo ello nos ahorrará tiempo de mantenimiento.

Utiliza etiquetas específicas en las imágenes base

Las imágenes etiquetadas como *latest* se van actualizando regularmente con los últimos cambios, lo cual podría provocar efectos no deseados e incluso fallos al regenerar nuestras imágenes. Es una buena práctica utilizar una etiqueta más específica para nuestra imagen base de nuestro Dockerfile.

Revisa las imágenes publicadas mínimas

Normalmente, las imágenes oficiales para una versión específica tienen varias publicaciones con diferentes etiquetas como, por ejemplo, slim o alpine. Deberíamos buscar cuál es la distribución mínima compatible con nuestra aplicación.

Elimina las dependencias de compilación utilizando múltiples etapas

Al hacer nuestras imágenes reproducibles hemos introducido algunos componentes necesarios para la compilación, pero no para la ejecución. Utilizando Dockerfiles que se construyan utilizando múltiples etapas podremos eliminar de nuestra imagen las dependencias de compilación moviéndolas a una etapa previa.

Docker Compose

Docker Compose es una herramienta que nos permite simplificar el despliegue de aplicaciones multicontenedor como un único servicio, permitiéndonos gestionar fácilmente todo el ciclo de vida de los contenedores y otros objetos de la aplicación como volúmenes y redes.

En lugar de crearnos nuestros propios scripts con llamadas al cliente de Docker para configurar y desplegar todos los objetos de nuestra aplicación (contenedores, redes, volúmenes, etc.), Docker Compose nos permite definir en ficheros YAML toda nuestra aplicación, su configuración y cómo se relacionan los objetos y sus dependencias.

Será la propia herramienta Docker Compose la encargada de enviar las tareas necesarias a Docker Engine para crear los objetos y ejecutar los contenedores.

Algunas de las ventajas de los servicios de Docker Compose son:

- Permite el despliegue en el mismo host de **múltiples entornos** aislados de nuestra aplicación multicontenedor. Cada uno de estos entornos será un servicio en Docker Compose con un nombre de proyecto asociado.
- Al iniciar un servicio de Docker Compose se buscan ejecuciones anteriores de los contenedores manteniendo sus volúmenes de datos, de manera que no perdamos información.
- Al reiniciar un servicio de Docker Compose, se reutilizan aquellos contenedores cuya configuración no ha sido modificada, recreando únicamente aquellos que han sido modificados, acelerando así el reinicio.

Instalación de Docker Compose

Antes de instalar Docker Compose deberemos tener instalado previamente Docker Engine. En los instaladores de Docker Desktop para Windows y Mac ya viene incluido Docker Compose, por lo que no será necesario hacer nada más. La instalación en Linux consistirá simplemente en descargar la herramienta y darle permisos de ejecución de la siguiente manera:

```
$ sudo curl -L \
  "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s)-
$(uname -m)" -o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

Creación de servicios

El uso de Docker Compose para la creación y despliegue de servicios se basa en los siguientes pasos:

- En un directorio para el proyecto, crear el código y ficheros necesarios para nuestra aplicación, tal como haríamos habitualmente.
- Crear los ficheros Dockerfile para las imágenes de nuestros contenedores en el directorio del proyecto.
- Crear un fichero de definición de los servicios, volúmenes y redes en formato YAML llamado docker-compose.yml
- Utilizar la herramienta docker-compose para crear los servicios y objetos Docker y gestionar su ciclo de vida.

Formato del fichero Docker Compose

Los ficheros de Docker Compose están escritos en formato YAML y, por defecto, tendrán el nombre de *docker-compose.yml*. En ellos definiremos los servicios, redes y volúmenes de nuestra aplicación.

Existen varias versiones del formato de fichero de Docker Compose. Dependiendo de la versión del motor de Docker que estemos ejecutando soportaremos hasta una versión determinada. La última versión disponible de Docker Compose es la 3.8, soportada a partir de la versión 19.03 de Docker.

Se recomienda comprobar que las opciones de configuración utilizadas en los ficheros YAML son soportadas por la versión de Docker que disponemos.

Un fichero de Compose puede incluir las siguientes secciones a nivel raíz:

- *version*. Indica la versión de Compose utilizada en el fichero.
- *services*. Lista de servicios que componen nuestra aplicación.
- *volumes*. Permite crear volúmenes que serán utilizados por los servicios.
- *networks*. Permite crear volúmenes que serán utilizados por los servicios.

En el siguiente ejemplo se muestra el contenido de un sencillo fichero de Compose en el que se definen dos servicios:

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
  redis:
    image: "redis:alpine"
```

Definición de servicios en Compose

Dentro de la key ***services*** se definirá cada uno de los servicios de los que consta nuestra aplicación multicontenedor, con una serie de parámetros adicionales. El nombre del servicio no es meramente informativo: se inyecta al ***resolver*** de cada contenedor de la misma red, por lo que un contenedor puede conectarse con otros usando su nombre, en lugar de tener que buscar su IP usando algún artificio.

Esto es algo **muy potente**, pues permite simplificar enormemente la configuración de una aplicación: por ejemplo, podemos prefijar la URL de conexión a la base de datos de nuestra aplicación como **"redis_database:8080"** si el servicio de Redis se llama *redis_database* y expone el puerto 8080.

A continuación, veremos algunas de las principales **opciones de configuración** de las secciones de Compose. Recomendamos consultar la documentación oficial para ver todas las opciones disponibles y ejemplos de uso.

Opciones de configuración para construcción

La configuración **build** nos permite especificar la ruta del contexto de nuestro fichero Dockerfile. El siguiente ejemplo utilizaría el fichero Dockerfile de un directorio:

```
version: "3.8"
services:
  webapp:
    build: ./dir
```

La configuración build soporta además otras configuraciones internamente. Por ejemplo, podemos especificar un nombre específico del Dockerfile:

```
  build:
    context: ./dir
    dockerfile: Dockerfile.dev
```

Con la configuración **image** podemos indicar directamente la imagen que utilizará el contenedor, ya sea especificando su nombre o su identificador.

```
  image: ubuntu
  image: ubuntu:18.04
  image: portainer/portainer-ce:alpine
```

Configuración de puertos

La opción **expose** permite exponer puertos sin publicarlos en el host. Solamente serán accesibles por los servicios enlazados. Veamos un par de formas de definirlos:

```
# en yaml, ambas expresiones son equivalentes
 expose: ["3000", "8000"]
 expose:
  - "3000"
  - "8000"
```

La opción **ports** permite publicar al host puertos del contenedor. Admite dos sintaxis, una corta especificando ***puerto_host:puerto_contenedor***, y una larga con alguna opción adicional. Veamos algunos ejemplos:

```
ports:
   - "3000"
   - "8000:80"
   - "9090-9091:8080-8081"
   - "127.0.0.1:8001:8001"
   - "6060:6060/udp"

 ports:
   - target: 80
     published: 8080
     protocol: tcp
     mode: host
```

Ejecución de comandos

Las opciones ***command*** y ***entrypoint*** de Compose son equivalentes a las instrucciones CMD y ENTRYPOINT de los Dockerfiles que ya vimos. Recordemos que *entrypoint* sobrescribe a *command*. Veamos algún ejemplo de uso:

```
 command: /app/entrypoint.sh
 command: ["php", "-d", "vendor/bin/phpunit"]

 entrypoint: /app/start-service.sh
 entrypoint: [php, -d, vendor/bin/phpunit]
```

Variables de entorno

Existen varias formas de pasar variables de entorno a los contenedores de los servicios. Podemos utilizar la configuración ***environment*** para indicar una lista de variables, o bien la configuración ***env_file*** para múltiples variables de entorno definidas en un fichero externo:

```
$ cat ./Docker/api/api.env
NODE_ENV=test

$ cat docker-compose.yml
version: '3'
services:
  api:
    image: 'node:6-alpine'
    env_file:
     - ./Docker/api/api.env
    environment:
     - NODE_ENV=production
```

En caso de tener varias definiciones de la misma variable de entorno, Compose utilizará el siguiente orden para elegir qué valor utilizar:

- Fichero de Compose.
- Variables de entorno del Shell.
- Fichero de variables de entorno.
- Variables definidas en el Dockerfile.

Dependencias entre servicios

La opción ***links*** permite enlazar otros servicios al contenedor permitiendo establecer un *alias*. Esta opción será eliminada en un futuro; en su lugar se recomienda utilizar la configuración de redes y establecer dependencias entre servicios.

```
links:
   - "db:database"
   - "redis"
```

Para establecer dependencias entre servicios utilizaremos la opción ***depends_on***, permitiendo que un servicio espere a que otros estén arrancados antes de empezar su ejecución. Veamos un ejemplo:

```
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

Montar volúmenes en un servicio

La opción **volumes**, dentro de la definición de un servicio, nos permite montar rutas del host en el contenedor, o bien volúmenes a partir de su nombre.

La sintaxis corta de esta opción nos permite utilizar el formato [ruta_host:]ruta_contenedor[:modo], mientras que la sintaxis larga permite configuraciones adicionales como, por ejemplo, el tipo de montaje.

Veamos algunos ejemplos:

```
volumes:
  # Indicando solo la ruta del contenedor, se creará un nuevo volumen
  - /var/lib/mysql
  - /opt/data:/var/lib/mysql
  - ./cache:/tmp/cache
  - ~/configs:/etc/configs/:ro
  - datavolume:/var/lib/mysql

volumes:
  - type: volume
    source: mydata
    target: /data
    volume:
      nocopy: true

  - type: bind
    source: ./static
    target: /opt/app/static
```

Definición de volúmenes en Compose

Cuando referenciamos por nombre un volumen en la sección de los servicios, estos deben estar definidos en la sección ***volumes*** a nivel raíz del fichero de Compose. Esta sección nos permite la creación y definición de volúmenes asignando un nombre para poder ser referenciado en los servicios definidos.

```
services:
  db:
    image: postgres
    volumes:
      - datavolume:/var/lib/postgresql/data

volumes:
  datavolume:
    external: true
```

Definición de redes en Compose

Al igual que con los volúmenes, podemos definir a nivel raíz las redes que utilizan los servicios. Podemos crear nuevas redes o utilizar alguna ya existente, indicar el driver a utilizar y sus opciones, etc. Veamos un ejemplo de uso:

```
services:
  app:
    build: ./app
    networks:
      - frontend
      - backend
  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    name: frontend-network
  backend:
    external:
      name: existing-backend-network
```

El comando docker-compose

Utilizaremos la herramienta **docker-compose** para generar las imágenes y crear objetos definidos (volúmenes y redes) en el fichero de Compose. Por defecto usará el fichero con el nombre docker-compose.yaml del directorio actual, y como nombre del proyecto asignará el nombre del directorio en el que nos encontramos. Tanto el nombre del fichero como la ruta podemos modificarlos con los argumentos -f y -p respectivamente.

Si nuestra aplicación es bastante compleja podríamos querer tener la definición de nuestra aplicación en varios ficheros YAML. Para ello usaremos tantas veces el parámetro -f como ficheros tengamos.

Principales comandos de Docker Compose

docker-compose build	Construye los servicios a partir de los Dockerfiles.
docker-compose up	Crea e inicia los contenedores del servicio.

docker-compose stop	Detiene un servicio, parando sus contenedores.
docker-compose start	Inicia un servicio parado previamente.
docker-compose down	Detiene y elimina un servicio, parando y eliminando los recursos asociados (contenedores, redes, imágenes, volúmenes).
docker-compose ps	Lista los contenedores de los servicios.
docker-compose exec	Ejecuta un comando en un contenedor del servicio.

Comandos de Docker Compose para la gestión de servicios.

Recuerda

En la documentación oficial podemos consultar todos estos comandos y muchos más.

Anexo I: Instalación de Docker

Instalación de Docker Desktop en Windows 10

Docker Desktop for Windows es la versión Community de Docker y está diseñada para ejecutarse en Windows 10.

La instalación de Docker en Windows 10 es realmente sencilla, ya que simplemente debemos ejecutar el instalador descargado. El asistente de instalación se encargará de descargar de internet todo lo necesario para su puesta en funcionamiento, no obstante, nos preguntará si queremos utilizar contenedores Windows en lugar de contenedores Linux. Por defecto hace uso de contenedores Linux, aunque estemos usando Windows 10. Esto podremos cambiarlo posteriormente sin problemas.

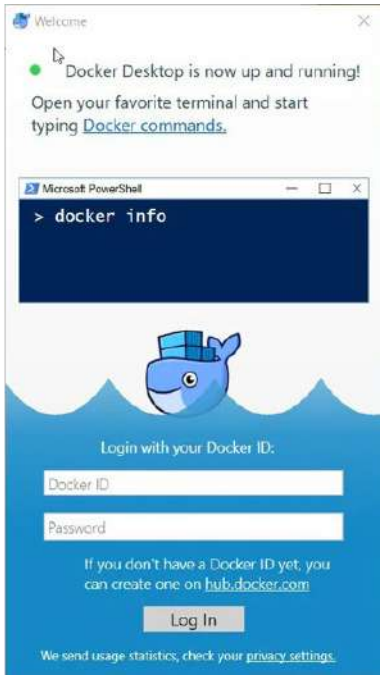
Empezaremos por descargar e instalar Docker Desktop for Windows desde Docker Hub.

Docker Desktop for Windows.

El demonio de Docker no es compatible con Windows de forma nativa, por lo que, para utilizar el cliente Docker de forma transparente, es necesario instalar algún tipo de virtualización. En concreto, Docker Desktop para Windows utiliza la virtualización Hyper-V nativa de Windows, que nos permitirá la ejecución de contenedores Linux y Windows.

Si aún no tenemos habilitado Hyper-V en las características de nuestro Windows, al finalizar la instalación se nos avisará de que es recomendable hacer uso de Hyper-V para que Docker funcione correctamente en Windows 10. Es importante saber que, si lo habilitamos, Virtual Box dejará de funcionar.

Una vez terminada la instalación, lo podemos ejecutar desde el menú de inicio. Un icono de Docker aparecerá en la barra de estado. Si lo pulsamos podemos ver que se está ejecutando:



Pantalla de bienvenida de Docker Desktop en Windows.

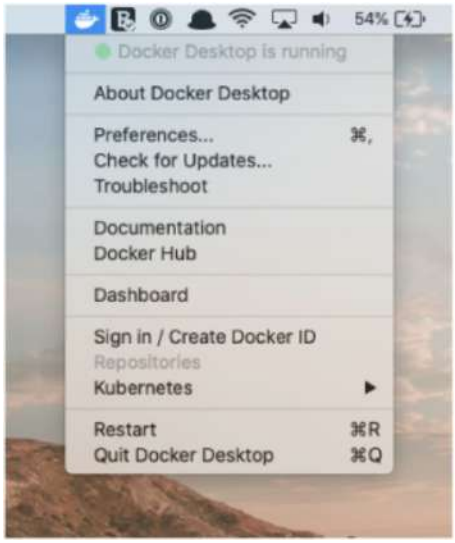
En las opciones de Docker vamos a poder acceder a las opciones del propio software, comprobar actualizaciones o cambiar de contenedores Linux a contenedores Windows fácilmente. Además, también vamos a poder hacer uso de nuestro Docker ID para acceder a los repositorios de Docker Hub.

Instalación de Docker Desktop en Mac OS

El demonio de Docker no es compatible con OS X directamente. Además, los contenedores no podrían ejecutarse nativamente, ya que requieren llamadas al sistema del kernel de Linux.

Sin embargo, la aplicación **Docker Desktop for Mac** nos permitirá ejecutar Docker como si fuera nativo. Esto funciona mediante una liviana máquina virtual sobre la virtualización de Mac. De este modo, cuando la aplicación Docker Desktop está en ejecución, el comando docker puede acceder al demonio de Docker ejecutándose en la máquina virtual y dar una experiencia casi nativa.

Empezaremos por descargar e instalar Docker Desktop for Mac desde Docker Hub. Una vez instalado, ejecutamos la aplicación y veremos el icono de Docker en la barra de estado superior. Si pinchamos sobre él, podremos acceder a varias opciones y ver el estado de Docker Desktop:



Opciones de Docker Desktop for Mac.

Instalación de Docker en Ubuntu

Se puede instalar Docker en Ubuntu desde la versión 18.04 en adelante sin problema. Versiones anteriores a esa puede que funcionen, pero no cuentan con soporte ya. Derivados de Ubuntu, como Kubuntu, también pueden usar este mismo mecanismo.

Antes de nada, y por si acaso el sistema tiene instalada alguna de las versiones de Docker más antiguas, se recomienda hacer una limpieza:

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

El siguiente paso será configurar el repositorio de Docker:

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
   sudo apt-key add -

$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) stable"
```

Ahora ya podemos instalar la última versión de Docker desde el repositorio:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

En entornos de producción es recomendable instalar el entorno fijando una versión para evitar sorpresas en las actualizaciones de Docker, aunque actualmente es bastante compatible hacia atrás:

```
$ sudo apt-get install docker-ce=18.06.1~ce~3-0~ubuntu \
    docker-ce-cli=18.06.1~ce~3-0~ubuntu containerd.io
```

Instalación de Docker en otras distribuciones

Podríamos extendernos con tutoriales para distribuciones Debian, Centos, Arch… pero no sería práctico. Si necesitamos instalar Docker en un sistema que no esté en este documento, lo ideal es seguir las instrucciones de la página oficial.

Pasos post instalación para Linux

Usando Docker con un usuario no root

El demonio de Docker utiliza un socket de Unix que requiere permisos de administración, por lo que se ejecutará como root. El socket creado por Docker es accesible para los miembros del grupo de Unix *docker*.

Si queremos evitar tener que ejecutar el comando docker con *sudo*, podemos añadir nuestro usuario al grupo docker de la siguiente manera:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Ojo: deberemos iniciar una nueva sesión con nuestro usuario para que se aplique el cambio de grupo.

Iniciar el demonio de Docker

Una vez instalado Docker en Linux, será necesario iniciar el demonio de Docker:

```
$ sudo systemctl start docker
```

Si queremos podemos configurarlo para que se inicie automáticamente al arrancar la máquina:

```
$ sudo systemctl enable docker
```

Verificando la instalación

El comando ***docker version*** nos muestra las versiones para cada uno de los componentes de Docker instalados:

```
$ docker version
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 ...

Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.12)
  ...
```

También podemos obtener información de sistema referente a la instalación de Docker. El comando ***docker info*** nos proporciona la versión del kernel, el número de contenedores e imágenes que tenemos, drivers y plugins instalados, etc.

```
$ docker info
 ...
 Images: 33
 Server Version: 19.03.4
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 ...
 Kernel Version: 4.9.184-linuxkit
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 1.952GiB
 ...
```

Adicionalmente, podemos comprobar que Docker Engine está ejecutándose y funcionando correctamente lanzando un contenedor a partir de la imagen *hello-world*.

Si ejecutamos el siguiente comando veremos cómo el contenedor muestra varios mensajes autoexplicativos y termina su ejecución.

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
...
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

unir LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA