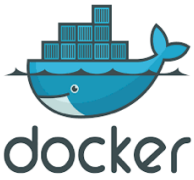
Despliegue de aplicaciones con Docker



En este documento vamos a explicar cómo empezar a trabajar con Docker, un gestor de contenedores open source, que automatiza el despliegue de aplicaciones en dichos contenedores. Un **contenedor** es una pieza de software con un sistema de ficheros completo, que contiene todo lo necesario para poder funcionar. Los contenedores sirven para distribuir y desplegar aplicaciones, de forma portable, estandarizada y autocontenida (sin necesidad de otras dependencias externas). Para ser más concretos, Docker permite gestionar contenedores Linux, permitiendo crear distintos entornos virtuales y ejecutar aplicaciones en ellos.

1. ¿Qué es Docker?

Podríamos ver Docker a priori como una alternativa a otras herramientas de virtualización, como VirtualBox o VMWare, pero existen ciertas ventajas y diferencias entre una cosa y otra, como veremos a continuación.

1.1. Ventajas de usar Docker frente a máquinas virtuales

Entre las principales ventajas que ofrece Docker frente a utilizar una máquina virtual convencional, podemos citar las siguientes:

- Se tiene un **uso ligero de recursos**, al no ser necesario instalar ningún sistema operativo *guest* completo sobre el sistema anfitrión (*host*). Esto es posible gracias a que, al gestionar únicamente contenedores Linux, se pueden aislar las características básicas de dicho kernel, compartidas por varias distribuciones (Ubuntu, Red Hat, CentOS...).
- Es **open source y multiplataforma**, puede instalarse sobre anfitriones Windows, Mac OS X e incluso algunos en la nube como AWS o Azure, aunque es preferible que dicho anfitrión sea una distribución Linux para tomar el kernel de él. En estos apuntes supondremos que vamos a utilizar Docker sobre una distribución Linux Debian.
- Es **portable**, ya que todas las dependencias de las aplicaciones se empaquetan en el propio contenedor, pudiendo llevarlo y ejecutarlo en cualquier *host*.
- La forma de **comunicar** contenedores entre sí es más sencilla que la comunicación entre máquinas virtuales. Mientras que para comunicar máquinas virtuales complejas necesitamos tener habilitados puertos entre ellas, en Docker se establecen puentes de comunicación (*bridges*), que hacen el proceso mucho más automático.

Así, por ejemplo, podríamos tener un ordenador con un sistema Debian instalado, y sobre él, mediante Docker, tener un contenedor corriendo CentOS, otro corriendo RedHat, o el mismo Debian, entre otras opciones. Pero no habrá que instalar físicamente ni CentOS, ni RedHat ni ninguna otra distribución como máquina virtual. De esta forma, el tamaño de las imágenes que se generan para los contenedores es mucho más reducido que el de una máquina virtual convencional.

1.2. ¿Para qué se utiliza Docker?

El principal uso de Docker se da en arquitecturas orientadas a servicios, donde cada contenedor ofrece un servicio o aplicación, facilitando así su escalabilidad. Por ejemplo, podemos tener un contenedor ofreciendo un servidor Nginx, otro con un servidor MariaDB/MySQL, otro con MongoDB, y establecer comunicaciones entre ellos para compartir información.

¿Quién utiliza Docker?

Actualmente, Docker se utiliza en algunas empresas o webs de relevancia, como Ebay o Spotify. Además, tiene por detrás el soporte de empresas importantes, como Amazon o Google.

1.3. Principales elementos de Docker

Antes de comenzar con la instalación y primeras pruebas con Docker, veamos qué elementos principales lo componen.

El motor de Docker (*Docker engine*)

Esta herramienta nos permitirá crear, ejecutar, detener o borrar contenedores. Necesita de un kernel de Linux para funcionar, e internamente se compone de una aplicación cliente y de un demonio. Con la primera, y mediante comandos, podemos comunicarnos con el segundo para enviar órdenes concretas (crear contenedor, ponerlo en marcha, etc.).

El archivo *Dockerfile*

Para almacenar la configuración y puesta en marcha de un determinado contenedor, se utiliza un archivo de texto llamado *Dockerfile*, aunque, como veremos, también se pueden crear y lanzar contenedores sin este archivo. En él se incluyen distintas instrucciones para, entre otras cosas:

- Determinar el sistema operativo sobre el que se va a basar el contenedor
- Instalar y poner en marcha aplicaciones
- Definir ciertas variables de entorno
- ... etc.

Aquí podemos ver un ejemplo muy sencillo de archivo Dockerfile:

```
FROM ubuntu:latest
MAINTAINER Juan Pérez <juan.perez@gmail.com>
RUN apt-get update
RUN apt-get install -y apache2
ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Analizando con algo de detalle sus líneas, podemos ver que:

- La primera línea especifica que el sistema operativo a emplear para el contenedor será Ubuntu. La versión se indica tras los dos puntos, y en este caso es la última, pero podríamos indicar alguna concreta por su nombre de pila, como por ejemplo "trusty" (versión 14) o "precise" (versión 12).
- La segunda línea alude al creador del archivo Dockerfile y su contacto, en caso de tener alguna duda o problema con el mismo.
- Las siguientes dos líneas actualizan los repositorios e instalan Apache, respectivamente. Se pueden poner comandos en líneas separadas, o enlazados en una sola línea.
- Las siguientes dos líneas añaden el virtual host por defecto a la carpeta de sitios disponibles de Apache, y le cambian su propietario a root.
- La línea con la instrucción EXPOSE hace público el puerto que se indique (el 80, en este caso), de forma que pueda ser accesible desde aplicaciones fuera del contenedor e incluso del sistema anfitrión.
- La última línea (CMD) debe ser única en cada contenedor. Indica lo que se va a ejecutar al iniciarse. En este caso, arrancamos apache en modo FOREGROUND (es decir, en primer plano). Si lo arrancáramos como servicio (por ejemplo, con CMD["service apache2 start"]), entonces el contenedor se cerraría inmediatamente, al no tener ningún proceso ejecutando en primer plano. Si hubiera más de una instrucción CMD definida en el archivo *Dockerfile*, sólo se considerará la última de ellas.

Existen otras instrucciones, como ENV para definir variables de entorno, pero la idea de la estructura y utilidad de un archivo *Dockerfile* puede haber quedado clara con este ejemplo sencillo.

2. Instalación y puesta en marcha

Veamos ahora cómo instalar Docker. En la web oficial podemos encontrarlo para los principales sistemas operativos:

- En el caso de *Windows* y *Mac* instalamos una herramienta llamada *Docker Desktop*, que nos permite configurar y poner en marcha los distintos contenedores que necesitemos desde estos sistemas. Es necesario, no obstante, algún sistema que simule un kernel de Linux como *host*, sobre el que apoyar Docker. Las últimas versiones de *Docker Desktop* ya incorporan este sistema Linux virtualizado. Una vez instalado, podremos usar diferentes instrucciones de Docker desde línea de comandos.
- En el caso de *Linux* (especialmente si estamos utilizando un servidor remoto sin interfaz gráfica), lo que se instala es directamente el *Docker Engine*, para poder configurar y lanzar contenedores directamente desde el terminal. La instalación es mucho más ligera, al no necesitar una máquina virtual adicional sobre la que ejecutar Docker (ya que utilizará la infraestructura del propio sistema Linux en que se instala).
- De forma adicional, también puede instalarse y configurarse Docker en la nube (sistemas AWS, Azure...).

2.1. Instalar *Docker Engine* en Linux Debian

En estos apuntes nos centraremos en la instalación de *Docker Engine* sobre Linux, por ser lo más habitual. En concreto lo instalaremos sobre un sistema Debian. Podemos encontrar más información respecto a la instalación en la documentación oficial de Docker. Podemos comprobar la versión actual de nuestro sistema Linux con el comando `cat /etc/os-release` o bien con el comando `lsb_release -cs`. También

podemos determinar la arquitectura de procesador con el comando `uname -m`. Así podremos determinar si nuestro sistema es compatible con Docker actualmente o no, consultando el listado de sistemas disponibles en la web anterior de instalación.

Siguiendo los pasos de la documentación oficial que mencionamos, primero debemos desinstalar versiones antiguas de Docker (llamadas *docker*, *docker-engine* o *docker.io*), si las hubiera, con este comando :

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

Después ya podemos instalar Docker, de varias formas: desde repositorio (recomendado), instalando manualmente el paquete *.deb*, o desde algunos scripts que automatizan la instalación, y que suponen la única vía de instalación en sistemas como Raspberry Pi (Raspbian). En nuestro caso haremos una instalación desde repositorio, por lo que podemos seguir los pasos y ejecutar los comandos que se indican aquí.

Una vez que ya hemos instalado Docker, podemos probar que todo ha ido correctamente ejecutando la imagen de prueba `hello-world`, que mostrará un mensaje de saludo por pantalla y finalizará.

```
sudo docker run hello-world
```

> **NOTA**: observad en los mensajes que se muestran al ejecutar el comando que la imagen no está disponible de forma local, pero Docker la descarga automáticamente.

También podemos verificar que esté correctamente instalado ejecutando el comando `docker` a secas (que sacará una lista con las opciones a indicar junto con el comando), o `docker version` (que mostrará la versión actualmente instalada).

Por defecto, el demonio de Docker se inicia automáticamente tras la instalación. Podemos habilitar o deshabilitar que se inicie con el sistema con estos comandos, respectivamente:

```
sudo systemctl enable docker
sudo systemctl disable docker
```

2.2. Desinstalar Docker

Para desinstalar Docker del sistema, ejecutamos el comando:

```
sudo apt-get purge docker-ce docker-ce-cli containerd.io
```

También podemos eliminar las imágenes, contenedores, etc del sistema, borrando manualmente la carpeta `/var/lib/docker`:

```
sudo rm -rf /var/lib/docker
```

3. Creación y uso de contenedores

Ya hemos visto qué es Docker y cómo instalarlo. Veamos ahora cómo crear distintos tipos de contenedores y qué operaciones útiles podemos emplear en ellos. Conviene tener presente que la mayoría de comandos `docker` que se indican a continuación requerirán de los permisos adecuados (*root/sudo*) para poderlos utilizar.

3.1. Las imágenes: el *hub* de Docker

Para crear un contenedor necesitamos disponer de una imagen Docker. Existen multitud de sitios desde donde descargar imágenes de sistemas utilizables en Docker, pero quizá el lugar más interesante de todos sea el **hub de Docker** (*Docker Hub*), en esta URL.

En esta web los usuarios suben imágenes creadas o personalizadas por ellos mismos, para que el resto de la comunidad las pueda utilizar y/o adaptar. Existen algunos repositorios importantes, como el de Apache, Nginx, Node, MongoDB o MySQL, con estas aplicaciones ya instaladas y listas para poderse utilizar. Veremos más tarde algún ejemplo sobre ello.

Para buscar alguna imagen que nos pueda interesar, podemos emplear el comando `docker search` con la palabra clave (por ejemplo, "mongodb"), y ver qué imágenes hay disponibles sobre este elemento, junto con su valoración (*STARS*), y alguna información adicional. Las imágenes aparecen ordenadas por valoración:

```
docker search mongodb

NAME            DESCRIPTION           STARS  OFFICIAL
mongo           MongoDB document...   10099  [OK]
mongo-express   Web-based MongoDB...   1411  [OK]
bitnami/mongodb Bitnami MongoDB...      245
...
```

Desde el propio cliente/terminal de Docker podemos descargar las imágenes con:

```
docker pull nombre_imagen
```

Para ver las imágenes que tenemos disponibles en el sistema, podemos escribir:

```
docker images
```

Y aparecerá un listado con el nombre de la imagen, su versión, y el tamaño que ocupa, entre otros datos. Por ejemplo:

```
REPOSITORY    TAG       IMAGE ID      CREATED       SIZE
busybox       latest    5b0d59026729  3 weeks ago   1.15MB
alpine        latest    05455a08881e  5 weeks ago   7.38MB
hello-world   latest    f2a91732366c  3 months ago  1.85kB
```

Docker permite albergar un número elevado de imágenes en local, pero aún así, si queremos eliminar una imagen del listado porque ya no la vayamos a utilizar (y siempre que no esté siendo utilizada por ningún contenedor), podemos hacerlo con `docker rmi`, seguido del id de la imagen (según el listado anterior):

```
docker rmi f2a91732366c
```

También podemos escribir nada más el principio del id (en éste y otros comandos de *docker*), siempre que sirva para diferenciarlo del resto:

```
docker rmi f2a
```

3.2. Descarga de imágenes

Vamos a probar a descargar una imagen para unas primeras pruebas de comandos sencillos. Probaremos con una imagen muy ligera, llamada ***alpine***, que contiene un conjunto de herramientas básicas para trabajo sobre un kernel de Linux. Descargamos la imagen con este comando:

```
docker pull alpine
```

Al no indicar versión, nos informará por terminal de que se descargará la última (*latest*). En el caso de querer especificar versión, se puede indicar a modo de *tag*, poniendo dos puntos tras el nombre de la imagen, y después el nombre de la versión:

```
docker pull alpine:3.5
```

La imagen de Alpine es una imagen muy ligera (unos pocos MB), que podemos comprobar que está descargada con el comando `docker images` visto anteriormente:

```
docker images

REPOSITORY  TAG     IMAGE ID     CREATED      SIZE
alpine      latest  05455a08881e 5 weeks ago  7.38MB
...
```

3.3. Ejecución sencilla de contenedores

Veamos ahora cómo crear y lanzar algún que otro contenedor sencillo a partir de las imágenes que obtengamos de los repositorios anteriores. El siguiente comando **lanza un contenedor** con la imagen *alpine* descargada previamente, **y ejecuta un comando** `echo` sobre dicho contenedor:

```
docker run alpine echo "Hola mundo"
```

> **NOTA**: si al ejecutar `docker run` la imagen no se encuentra disponible en nuestro equipo, se descargará como paso previo, antes de poner en marcha el contenedor sobre ella.

En este caso, obtendremos por pantalla "Hola mundo" y finalizará el proceso. Podemos consultar los contenedores activos con el comando `docker ps`. Si lo ejecutamos no obtendremos ninguno en la lista, puesto que el comando anterior ya finalizó. Pero podemos consultar los contenedores creados (detenidos o en ejecución) con `docker ps -a`:

```
CONTAINER ID  IMAGE   COMMAND               CREATED       ...
3fd9065eaf02  alpine  "echo 'Hola mundo'"   2 minutes ago ...
```

Con este ejemplo que hemos visto, podemos empezar a entender cómo se usan y para qué sirven los contenedores. Su filosofía es muy sencilla: ejecutar un comando o programa y finalizar (o dejarlo en ejecución). De este modo, podemos lanzar varios contenedores sobre una misma imagen, y cada uno se encargará de ejecutar una tarea o proceso diferente (o el mismo varias veces).

Este otro ejemplo lanza un contenedor con la imagen *alpine*, y **abre un terminal interactivo** en él, de forma que podemos escribir comandos sobre el terminal que actuarán sobre la imagen almacenada en el contenedor y su sistema de archivos. Para finalizar la ejecución del contenedor, basta con que escribamos el comando `exit` en el terminal:

```
docker run -it alpine /bin/sh
```

Podemos **especificar el nombre** (*name*) que se le da, para no tener que estar buscando después el id o el nombre que Docker genera automáticamente. Para ello, utilizaremos el parámetro `--name`:

```
docker run -it --name alpine_nacho alpine /bin/sh
```

Otra opción que puede resultar útil es **ejecutar un contenedor sin almacenar su estado en la lista**, con lo que se borra automáticamente al finalizar. Esto se consigue con la opción `-rm`:

```
docker run -rm alpine echo "Hola mundo"
```

En el caso de querer **ejecutar un contenedor en segundo plano** (como si fuera un demonio), se hace con el parámetro `-d`. En este caso, no mostrará ningún resultado por pantalla, y liberará el terminal para poder hacer otras operaciones. Esta opción es útil para lanzar contenedores relacionados con servidores (bases de datos, servidores web...), y dejarlos así en marcha sin bloquear el terminal.

```
docker run -d imagen
```

Asociado con el comando anterior, si por lo que fuera se produjera algún mensaje de error o de salida, para poderlo consultar emplearemos el comando `docker logs`, seguido del id del contenedor ejecutado:

```
docker logs f2e5a1629beb
```

3.4. Parada y borrado de contenedores

Para finalizar un contenedor actualmente en ejecución podemos lanzar el comando `docker stop` junto con el id del contenedor (o el nombre), obtenido con el propio `docker ps` anterior:

```
docker stop f2e
```

Podemos eliminar el contenedor de la lista de contenedores, siempre que no esté activo, con el comando `docker rm`, pasándole el id del contenedor (o el principio del mismo):

```
docker rm f2e
```

En el caso de que el contenedor esté en la lista de contenedores y haya finalizado su ejecución (visible con `docker ps -a`), podemos volverlo a lanzar fácilmente con el comando `docker start`, seguido del id o del nombre del contenedor:

```
docker start f2e
```

3.5. Asociar recursos del host al contenedor

Es muy habitual en ciertas ocasiones asociar algún recurso del sistema *host* al contenedor, de forma que desde el contenedor se pueda acceder o comunicar con dicho recurso del *host*. En concreto, veremos a continuación cómo asociar puertos y carpetas.

3.5.1. Asociar puertos entre *host* y contenedor

Para lanzar un contenedor y **asociar un puerto** del sistema *host* con un puerto del contenedor, se emplea el parámetro `-p`, indicando primero el puerto del *host* que se comparte, y después, seguido de dos puntos `:`, el puerto del contenedor por el que se comunica. Podemos repetir este parámetro más de una vez para asociar más de una pareja de puertos.

Por ejemplo, el siguiente comando asocia el puerto 80 del *host* con el puerto 80 del contenedor, y el puerto 8080 del host con el 3000 del contenedor, de forma que las peticiones que lleguen al *host* por los puertos 80 y 8080, respectivamente, se enviarán al contenedor.

```
docker run -p 80:80 -p 8080:3000 alpine:apache
```

3.5.2. Asociar carpetas entre *host* y contenedor

Si queremos **asociar una carpeta del *host*** a una carpeta del contenedor, se emplea el parámetro `-v`, indicando primero la carpeta del *host*, seguida de dos puntos `:`, y luego la carpeta del contenedor. Por ejemplo, el siguiente comando asocia la carpeta `~/data` del *host* con la carpeta `/data/db` del contenedor asociado a una imagen `mongo`, comunicando los dos puertos por defecto, de forma que los datos que se añadan sobre el contenedor se van a almacenar en la carpeta `~/data` de nuestro sistema *host*. Además, lanza el contenedor en segundo plano para que el servidor quede ejecutando sin bloquear el terminal.

```
docker run -d -p 27017:27017 -v ~/data:/data/db mongo
```

3.6. Arrancar contenedores Docker con el sistema

Ahora que ya sabemos cómo lanzar contenedores Docker para alojar nuestros servicios (servidores web, bases de datos, etc), es conveniente también saber cómo ponerlos en marcha al arrancar el sistema, por si éste sufre cualquier reinicio inesperado.

Para ello, haremos uso del parámetro `--restart`, pasándole la opción de reinicio que queramos:

- `no`: para no reiniciar el contenedor automáticamente (opción por defecto)
- `on-failure`: se reiniciará si se cerró debido a un error
- `always`: se reiniciará siempre que se detenga (aunque lo detengamos nosotros manualmente con `docker stop`)
- `unless-stopped`: se reiniciará siempre que no lo hayamos detenido nosotros manualmente.

Así, por ejemplo, reiniciaríamos automáticamente un contenedor con *alpine*:

```
docker run --restart unless-stopped alpine
```

> **NOTA:** si se reinicia el sistema y tenemos el demonio de `docker` habilitado como servicio, dicho demonio se reiniciará también, y pondrá en marcha todos los contenedores que tengan configurada apropiadamente su opción de `--restart`.

3.7. Resumen de comandos útiles

A continuación mostramos los comandos más relevantes que hemos utilizado, a modo de guía resumen. Recuerda anteponer la palabra *sudo* si no tienes permisos suficientes de ejecución:

```
# Buscar imágenes en Hub
docker search nombre_imagen

# Descargar imagen del Hub
docker pull nombre_imagen

# Ver imágenes disponibles en local
docker images

# Eliminar imagen (no utilizada por ningún contenedor)
docker rmi id_imagen

# Lanzar contenedor desde imagen (forma simple)
# La imagen se descarga del Hub si no está en local
docker run nombre_imagen

# Lanzar contenedor en modo demonio (servidores)
docker run -d nombre_imagen

# Listado de contenedores activos
docker ps

# Listado de todos los contenedores
docker ps -a

# Detener contenedor activo
docker stop id_contenedor

# Reanudar contenedor
docker start id_contenedor

# Eliminar contenedor (previamente detenido)
docker rm id_contenedor

# Asociar puertos a contenedor
docker run -p 80:80 -p 8080:3000 nombre_imagen

# Asociar carpetas locales (volúmenes) a contenedor
docker run -v carpeta_local:carpeta_contenedor

# Iniciar contenedor indicando que arranque con sistema
docker run --restart unless-stopped nombre_imagen
```

Ejercicio 1:

Descarga la imagen de *mongo* con Docker y pon en marcha un contenedor, mapeando los puertos por defecto y asociando la carpeta */data/db* del contenedor con la carpeta *~/data* de tu sistema *host*. Hazlo

> de forma que se ejecute como demonio, y se reinicie con el sistema salvo que lo paremos manualmente. Prueba a conectar con el servidor desde algún cliente MongoDB (Compass, extensión de VS Code...).

> **Ejercicio 2:**
>
> Utiliza el proyecto *LibrosWebSesiones* que has realizado en esta sesión. Súbelo a tu VPS (a través de GitHub), y ponlo en marcha. Puedes utilizar la pasarela *Passenger* con Apache, como hicimos también en esta otra sesión. Comprueba que conecta adecuadamente con el servidor Mongo lanzado en el ejercicio anterior con Docker.
>
> **NOTA**: deberás crear a mano la carpeta *public/uploads* y darle permisos adecuados de escritura para probar la subida de imágenes

4. Creación de imágenes

En la sección anterior hemos visto cómo lanzar de distintas formas contenedores de terceras partes. Pero una interesante utilidad de Docker consiste en poder instalar el software que necesitemos sobre un contenedor, ejecutarlo, y también hacer una imagen personalizada del contenedor modificado, para poderla reutilizar las veces que queramos.

Ahora vamos a ver cómo podemos crear nuestras propias imágenes, para poderlas lanzar en contenedores, o distribuir a otros equipos. Veremos dos formas de hacerlo: ejecutando el contenedor e instalando el software necesario, o a través de archivos de configuración *Dockerfile*.

4.1. Instalar aplicaciones y crear imágenes personalizadas

Vamos a hacer un ejemplo sencillo a partir de la imagen Alpine utilizada antes. Esta imagen por defecto no viene con el editor *nano* instalado, así que vamos a instalarlo, y a crear una nueva imagen con ese software ya preinstalado.

Lo primero que haremos será ejecutar la imagen original en modo interactivo, como en algún ejemplo anterior:

```
docker run -it alpine /bin/sh
```

Una vez dentro del terminal, instalamos *nano* con estos comandos:

```
apk update
apk upgrade
apk add nano
```

Tras esto, podemos probar a abrir el editor con el comando `nano`, y ya podemos cerrar el contenedor con `exit`.

Una vez definido el contenedor, para crear una imagen a partir de él emplearemos el comando `docker commit`. Este comando acepta una serie de parámetros, donde podemos especificar el autor de la imagen, la versión, etc. Antes de nada, debemos averiguar el id del contenedor que queremos utilizar para la imagen (con `docker ps -a`). Supongamos para este ejemplo que el id es *f2e5a1629beb*. En este caso, podemos crear una imagen del mismo con un comando como éste:

```
docker commit -a "Nacho Iborra <nachoiborra@iessanvicente.com>" \
f2e5a1629beb nacho/alpine:nano
```

En el parámetro `-a` indicamos el autor y e-mail del mismo, después indicamos el id del contenedor a utilizar, y finalmente indicamos un nombre de repositorio con unos tags opcionales que ayuden a identificar la imagen. En este caso indicamos que es una versión particular de la imagen *alpine* con *nano* instalado en ella.

Tras este comando, podemos localizar la imagen en el listado de imágenes con `docker images`:

```
REPOSITORY    TAG       IMAGE ID     ...
nacho/alpine  nano      1a3e882f228a ...
alpine        latest    3fd9065eaf02 ...
```

A partir de este momento, podemos utilizar esta imagen para lanzar contenedores, como en este ejemplo:

```
docker run -it nacho/alpine:nano /bin/sh
```

Así podemos crear imágenes a medida con cierto software base preinstalado.

4.2. Crear imágenes a partir de archivos *Dockerfile*

Veamos en esta sección cómo emplear el archivo de configuración *Dockerfile* para crear imágenes Docker personalizadas de una forma más automatizada, sin tener que entrar por terminal en el contenedor e instalar los programas, como hemos hecho en el apartado anterior.

Vamos a crear una imagen similar a la del apartado anterior: partiendo de una base Alpine, instalaremos *nano* en ella y crearemos la imagen. En primer lugar, debemos crear un archivo llamado *Dockerfile* en nuestra carpeta de trabajo. El contenido del archivo será el siguiente:

```
FROM alpine:latest
MAINTAINER Nacho Iborra <nachoiborra@iessanvicente.com>
RUN apk update
RUN apk upgrade
RUN apk add nano
CMD ["/bin/sh"]
```

En la primera línea (FROM) indicamos la imagen base sobre la que partir (última versión de la imagen alpine), y en la segunda (MAINTAINER) los datos de contacto del creador de la imagen. Después, con la instrucción RUN podemos ejecutar comandos dentro de la imagen, como hemos hecho previamente. En este caso, actualizamos el sistema e instalamos *nano*. Finalmente, la instrucción CMD es obligatoria para crear contenedores, e indica lo que van a ejecutar (todos los contenedores deben ejecutar algo). En este caso indicamos que ejecute el terminal.

A partir de este archivo *Dockerfile*, en la misma carpeta donde está, ejecutamos el comando `docker build` para crear la imagen asociada.

```
docker build -t nacho/alpine:nano .
```

El parámetro `-t` sirve para asignar tags a la imagen. En este caso, indicamos que es la versión con *nano* incorporada de la imagen alpine. También podemos utilizar este tag múltiples veces, para asignar diferentes. Por ejemplo, indicar el número de versión, y que además es la última disponible:

```
docker build -t nacho/alpine:1.0 -t nacho/alpine:latest .
```

Una vez creada la imagen, la tendremos disponible en nuestro listado `docker images`:

```
REPOSITORY      TAG         IMAGE ID      ...
nacho/alpine    nano        6800a48a4d68  ...
alpine          latest      3fd9065eaf02  ...
```

Y podremos ejecutarla en modo interactivo. En este caso no hace falta que indiquemos el comando a ejecutar, ya que lo hemos especificado en el archivo *Dockerfile* (el terminal):

```
docker run -it nacho/alpine:nano
```

4.2.1. Ejemplo: aplicación Node

Veamos ahora un ejemplo concreto de cómo configurar un archivo *Dockerfile* para crear un contenedor para una aplicación Node.js. Tenemos que añadir un fichero `Dockerfile` en la raíz de nuestro proyecto Node. Podría tener una apariencia como ésta:

```
FROM node:20
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package*json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD ["npm", "start"]
```

- La primera línea `FROM node:20` especifica que necesitamos incorporar al contenedor la imagen de *Node*, en concreto de su versión 20.
- La línea `RUN mkdir -p /usr/src/app` indica que se creará esa carpeta dentro del contenedor (creando también las subcarpetas necesarias, a través de la opción `-p`). No tiene por qué ser esa ruta específicamente, podemos cambiarla por cualquier otra que prefiramos.
- La instrucción `WORKDIR /usr/src/app` nos ubica en esa carpeta como carpeta de trabajo, para que los comandos que ejecutemos partan de ahí.
- La instrucción `COPY package*.json ./` copia los ficheros de configuración JSON (*package.son* y *package-lock.json*) en la carpeta indicada antes en *WORKDIR*.
- La instrucción `RUN npm install` instalará las dependencias de esos archivos en la subcarpeta *node_modules*, dentro de *WORKDIR*.
- La instrucción `COPY . .` copia todo el contenido de nuestra carpeta actual dentro de la carpeta indicada en *WORKDIR*. Esto puede suponer un problema, ya que podríamos tener una carpeta *node_modules*, por ejemplo, u otros elementos que no queramos copiar. Para solucionar esto podemos crear un fichero `.dockerignore` que indique qué elementos de la carpeta del proyecto deben ignorarse. El formato es similar a los ficheros *.gitignore* de *Git*, y podría tener un contenido como éste:

```
node_modules
```

- La instrucción `EXPOSE 8080` exporta el puerto 8080, de forma que el *host* se va a comunicar con el contenedor a través de este puerto. Cambiaremos este número de puerto por el que hayamos puesto a escuchar a nuestra aplicación Node.
- La línea `CMD` indica el comando que finalmente ejecutará el contenedor. En este caso ejecuta `npm start`, suponiendo que tenemos definido este *script* en nuestro *package.json* para poner en marcha la aplicación. También podríamos ejecutar el comando `node index.js`.

Para generar la imagen del proyecto ejecutamos este comando desde la raíz del mismo. Cambiaremos el parámetro *app_node* por el nombre que queramos dar a dicha imagen:

```
docker build -t app_node .
```

Una vez finalizada la ejecución del comando (puede tardar un tiempo si necesita descargar la imagen de *node*), podremos ver nuestra nueva imagen listada en el comando `docker images`.

4.3. Ubicación de las imágenes. Copia y restauración.

Las imágenes que descargamos, y las que creamos nosotros de forma manual, se ubican por defecto en la carpeta `/var/lib/docker`, aunque dependiendo de la fuente y el formato de la imagen, se localizan en una u otra subcarpeta, y con uno u otro formato.

La principal utilidad que puede tener conocer la ubicación de las imágenes es el poderlas llevar de una máquina *host* a otra. Para ello, podemos emplear el comando `docker save`, indicando el nombre del archivo donde guardarla, y el nombre de la imagen a guardar:

```
docker save -o <path_a_archivo> <nombre_imagen>
```

Por ejemplo:

```
docker save -o /home/alumno/mi_imagen.tar nacho/alpine:nano
```

Se generará un archivo TAR que podremos llevar a otra parte. En la nueva máquina, una vez copiado el archivo, podemos cargar la imagen que contiene en la carpeta de imágenes de Docker con `docker load`:

```
docker load -i <path_a_archivo>
```

5. Comunicación entre contenedores

Hemos visto cómo crear y poner en marcha contenedores de diversas formas. En ejemplos y ejercicios anteriores hemos puesto en marcha, por un lado, un contenedor MongoDB (*Ejercicio 1*) y, por otro, una aplicación Node (ejemplo anterior). ¿Cómo podríamos comunicar estos dos contenedores para que desde la aplicación podamos acceder a la base de datos, por ejemplo? En este apartado veremos algunos mecanismos que podemos emplear para comunicar contenedores.

5.1. Compartir una misma red virtual

Una forma sencilla de comunicar contenedores es añadiéndolos a una misma red virtual. Primero creamos la red virtual:

```
docker network create nombre_red
```

donde cambiaremos el parámetro *nombre_red* por el nombre que queramos darle a la red. El siguiente paso es conectar a la red virtual cada uno de los contenedores implicados:

```
docker network connect nombre_red id_contenedor1 id_contenedor2 ...
```

> **NOTA**: también se puede usar el nombre del contenedor, si lo tenemos accesible.

> **Ejercicio 3**:
>
> Vamos a comunicar el proyecto *tareas-nest* que desarrollamos en este apartado con la base de datos Mongo que pusimos en marcha en el *Ejercicio 1*. Sigue estos pasos:
>
> 1. Detén y elimina el contenedor de Mongo. No te preocupes por los datos, porque estarán guardados en la carpeta *data* de tu carpeta de usuario. Suponiendo que el contenedor Mongo tiene el *id* 111a, los comandos serían
>
> ```
> sudo docker stop 111a
> sudo docker rm 111a
> ```
>
> 2. Crea una red donde convivirán los contenedores que crearemos (Mongo y Nest). La llamaremos por ejemplo *node_mongo*:
>
> ```
> sudo docker network create node_mongo
> ```
>
> 3. Pon de nuevo en marcha un contenedor Mongo. Pero esta vez lo crearemos asociado a esa red, y le daremos un nombre (por ejemplo "mi_bd_mongo").
>
> ```
> sudo docker run -d --name mi_bd_mongo --network node_mongo \
> --restart unless-stopped -p 27017:27017 -v ~/data:/data/db mongo
> ```
>
> 4. Modifica el proyecto *tareas-nest-jwt* que mencionábamos antes. Añade un fichero *Dockerfile* en la raíz del proyecto con un contenido como este:

```
FROM node:20
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package*json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

5. Añade también al proyecto un fichero *.dockerignore* para que no se tenga en cuenta la carpeta *node_modules* en los comandos de Docker. Tendrá este contenido:

```
node_modules
```

6. Modifica también el fichero *app.module.ts* del proyecto para que no conecte a MongoDB en *localhost*, sino en el nombre que le has dado al contenedor creado en el paso 3:

```
MongooseModule.forRoot('mongodb://mi_bd_mongo/tareas-nest'),
```

7. Sube el proyecto *tareas-nest-jwt* modificado a un repositorio GitHub, y clónalo en tu carpeta del VPS.
8. Accede a la carpeta de tu proyecto, y crea la imagen con el correspondiente comando Docker (recuerda añadir el punto al final para indicar la carpeta origen):

```
sudo docker build -t app_tareas .
```

9. Lanza la app Nest en la misma red que el contenedor Mongo, y exponla por el puerto que quieras (por ejemplo, el 3000):

```
sudo docker run -d --name mi_app_tareas --network node_mongo \
--restart unless-stopped -p 3000:3000 app_tareas
```

10. Ya puedes acceder a la app con la URL *vpsXXXXX.vps.ovh.net:3000/tarea* (para el listado de tareas).

5.2. Conexión a través de *docker-compose*

Una segunda alternativa que tenemos para conectar contenedores es definirlos juntos en un archivo de configuración YAML (extensión `.yml`), llamado `docker-compose.yml` (normalmente en la raíz del

proyecto a desarrollar). Aquí vemos un ejemplo:

```
version: "3"

services:
    web:
        container_name: app_node
        restart: always
        build: .
        ports:
            - "8080:3000"
        depends_on:
            - "mongo"
    mongo:
        container_name: mi_bd_mongo
        restart: always
        image: mongo
        ports:
            - "27017:27017"
```

Analicemos algunas líneas del fichero:

- La primera línea `version: "3"` hace referencia a la versión de *docker-compose* que se quiere utilizar, a efectos de sintaxis permitida.
- Dentro de la sección *services* definimos cada uno de los servicios o contenedores que queremos poner en marcha. En este ejemplo tenemos dos: *web* para la aplicación Node y *mongo* para el servidor Mongo.
- En cuanto al servicio *web*, indicamos un nombre de contenedor, el modo de reinicio, la carpeta desde la que construir la aplicación (carpeta actual, si hemos definido el fichero *docker-compose.yml* en la raíz del proyecto Node), mapeo de puertos y, en *depends_on*, a qué otros servicios necesitamos conectar desde éste (al servicio de base de datos posterior)
- En lo que respecta al servicio *mongo*, lo crearemos a partir de la imagen *mongo*, con el mapeo de puertos indicado.

Para construir todos los servicios ejecutamos este comando desde la raíz del proyecto Node, en este caso:

```
docker compose build
```

Para poner luego en marcha los servicios construidos ejecutamos este otro comando:

```
docker compose up
```