Guía

Esta documentación se ha ido realizando durante el progreso de cursos sobre Docker y Kubernetes en los que se ha trabajado con diferentes versiones de Docker y Docker Compose. Está publicada en https://github.com/falces/docker-docs, por lo que en caso de encontrar errores se agradece un Pull Request con la corrección.

Conceptos

Image / Imagen

Una **imagen** en Docker es la representación estática de una aplicación o servicio con su configuración y todas sus dependencias. Las imágenes se utilizan para crear contenedores y nunca cambian.

Por ejemplo una imagen podría contener un sistema Ubuntu con un servidor Apache y una aplicación web.

Las imágenes pueden almacenarse localmente o remotamente en un repositorio conocido como **registro**, donde están disponibles por nombre y normalmente en diferentes versiones etiquetadas, por ejemplo `ubuntu:latest` o `mysql:5.7`. El más utilizado es Docker Hub, un repositorio en la nube para crear, probar, guardar y distribuir imágenes. También proporciona a los usuarios un espacio para crear repositorios privados, automatizar funciones de compilación, crear *webhooks* o compartir espacios de trabajo.

Las imágenes son referenciadas por su id (que provee Docker) o su etiqueta, que creamos nosotros.

Pueden ser repositadas en Docker Hub y consumidas por cualquier usuario.

Las imágenes constan de una serie de capas:

- Imagen base
- Ejecución
- Directorio de trabajo
- Exposición
- Comando

Dockerfile

Es un archivo de configuración que se utiliza para crear imágenes. En dicho archivo indicamos qué es lo que queremos que tenga la imagen, y los distintos comandos para instalar las herramientas.

Containers / Contenedores

Son instancias en ejecución de una imagen. Al ejecutar una imagen se crea un contenedor. Son los que ejecutan cosas, los que ejecutarán nuestra aplicación. El concepto de contenedor es como si restauráramos una máquina virtual a partir de un snapshot. A partir de una única imagen, podemos ejecutar varios contenedores.

Como las imágenes no cambian, las modificaciones realizadas durante la ejecución de un contenedor no serán persistentes al detenerlo y volver a ejecutarlo. Pero es posible crear una nueva imagen, una nueva versión, con los cambios realizados. Y si algo va mal podríamos volver de forma sencilla a una versión anterior del contenedor.

Almacenamiento / Storage

La información generada durante la vida del contenedor se pierde cuando este se detiene a no ser que configuremos algún tipo de persistencia de datos. Para esto, Docker nos ofrece dos (tres, si utilizamos Linux como host) formas de anclar puntos del host con el contenedor:

Volúmenes

Docker configura un path dentro del host y se le asigna un nombre de volumen. El contenedor únicamente conoce este nombre de volumen. Por lo tanto, aplicaciones externas que accedan al contenedor no tendrán forma de acceder a la ubicación real de la información. Este asilamiento mantiene la integridad y seguridad de host y contenedor.

Los volúmenes son el mecanismo preferido para mantener la persistencia de datos. Es posible definir volúmenes en modo «*sólo lectura*». Y volúmenes que pueden compartirse por más de un contenedor, algunos en modo «*lectura/escritura*» y otros en modo «*sólo lectura*».

Los volúmenes se almacenan en el host en una carpeta gestionada por Docker (Linux: `/var/lib/docker/volumes`, Windows: `C:\ProgramData\docker\volumes`).

Bind Mounts

Su funcionamiento es prácticamente igual que el de los volúmenes, salvo por el detalle de que nos permite seleccionar cualquier ubicación del host para persistir la información. Esto en muchos casos puede ser útil, pero por otro lado expone información del contenedor fuera de éste, lo que puede representar un problema de seguridad.

Los bind mounts se almacenan en cualquier carpeta del host. Cualquier proceso o programa puede acceder a esta carpeta y modificar la información en cualquier momento.

TMPFS

Si estamos ejecutando Docker en Linux podemos utilizar esta tercera vía. Con este método, el contenedor puede crear archivos fuera de su capa de escritura, en el host, siendo accesible por éste. Es temporal y sólo persiste en la memoria del host, por lo que cuando el contenedor para (o reinicia), la información se pierde.

Links

Sirven para enlazar contenedores entre sí, que están dentro de una misma máquina, sin exponer a los contenedores cuáles son los datos de la máquina que los contiene.

Docker CLI

Herramienta para gestionar Docker desde el terminal.

Docker Hub

Es un servicio cloud gestionado por Docker que nos permite construir, enlacar y gestionar nuestras imágenes Docker.

Además de nuestras imágenes, podemos acceder a miles de imágenes de otros usuarios. Muchas de estas imágenes son lo que se conoce como imágenes oficiales: imágenes creadas por los fabricantes de productos, listas para consumir. Algunos ejemplos de imágenes oficiales:

- PHP
- Ubuntu
- Nginx
- MySQL
- Redis
- Node

Instalación

Linux / Windows / Mac OS

Desde la web oficial https://www.docker.com/products/docker-desktop podemos descargar Docker para Linux, Windows y Mac.

Instalación manual en Linux

Si no encontramos un instalador óptimo para nuestra distribución Linux, podemos hacer una instalación manual:

```
# Borrar posible versión anterior:
$  sudo apt-get remove docker docker-engine docker.io containerd runc

# Actualizar orígenes de software
$ sudo apt-get update

# Instalar prerequisitos:
# Curso:
$   sudo apt-get install \
        apt-transport-https \
        ca-certificates \
        curl \
        software-properties-common
# Docker:
$  sudo apt-get install \
        ca-certificates \
        curl \
        gnupg \
        lsb-release

# Descargar GPG Key oficial:
$  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /usr/share/keyrings/docker-archive-keyring.gpg

# Añadir repositorio a APT:
$ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null

# Actualizar e instalar:
$ sudo apt-get update
```

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

# Probar:
$ sudo docker run hello-world

# Acciones post instalación
# Ejecutar Docker sin privilegios root
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
# Cerrar y abrir sesión para recargar los privilegios del usuario.
# Activar cambios a los grupos:
$ newgrp docker
# Hacer prueba sin sudo:
$ docker run hello-world

# Configurar Docker para arrancar al iniciar en distribuciones Linux que no lo
hagan por defecto (Ubuntu y Debian sí lo hacen y esto no es necesario):
$ sudo systemctl enable docker.service
$ sudo systemctl enable containerd.service
# Si queremos desactivar esto, usamos disable:
$ sudo systemctl disable docker.service
$ sudo systemctl disable containerd.service
```

Comandos: Docker CLI

Docker Hub

Login

Una vez creada nuestra cuenta en Docker Hub, podemos logar nuestro Docker-CLI:

```
$ docker login
```

```
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: nombre_de_usuario_docker_hub
Password:
Login Succeeded
```

Podemos salir:

```
$ docker logout
```

```
Removing login credentials for https://index.docker.io/v1/
```

En ocasiones se corrompe la información de sesión en nuestro Docker y obtenemos errores cuando tratamos de descargar imágenes. En ese caso, renombramos el archivo de registro y volvemos a logarnos:

```
$ docker logout
$ mv ~/.docker/config.json ~/.docker/config_old.json
$ docker login
```

Buscar imágenes

Podemos localizar las imágenes a través de Docker Hub que, además de ser mucho más fácil, nos permite consultar toda la documentación relativa al uso de las imágenes. Otra opción es buscar imágenes desde el terminal:

```
$ docker search php
$ docker search php:latest
$ docker search php:7.1.34-fpm
$ docker search --filter "is-official=true" php:7.1.34-fpm
```

Crear imagen y ejecutar

Con el comando `run` levantamos un contenedor específico. Los contenedores están disponibles en *DockerHub*. Por ejemplo, podemos levantar el contenedor con la imagen *hello-world*:

```
$ docker run hello-world
```

O *busybox*:

```
$ docker run busybox
```

El comando `run` es lo mismo que ejecutar los comandos `create` y `start`. Para la imagen `hello-world`, el comando anterior es lo mismo que ejecutar:

```
$ docker create hello-world # nombre de la imagen
0ccf62a4aaff2bb25b9d21e78d423251e17d710cf5ddaabc5858353b167431e9 # Nos devuelve
el id del contenedor
# Con el id del contenedor levantamos la imagen:
$ docker start -a
0ccf62a4aaff2bb25b9d21e78d423251e17d710cf5ddaabc5858353b167431e9
```

El comando `create` crea una imagen del contenedor, mientras que `start` ejecuta el comando de inicio con el que se creó el contenedor. Con el atributo -a hacemos que se envíen las respuestas a nuestro terminal, si no lo usáramos no veríamos nada. Para únicamente ejecutar un contenedor previamente creado, usamos el comando `start`, y de esta forma no crearemos un nuevo contenedor. No podemos especificar un nuevo comando de inicio con el comando `start`.

Podemos hacer lo mismo especificando un nombre para el contenedor con el atributo `name`, lo que nos permite no usar los id de contenedor:

```
# Atributo: --name [NOMBRE_DEL_CONTENEDOR]
$ docker create --name HelloWorld hello-world
$ deb4d7ca1ecd350c455c35363f2cfff1b67743a57f5752aef5ce5db08b60a43d
$ docker start -a HelloWorld
```

Podemos especificar comandos a ejecutar dentro del contenedor en el momento de su ejecución:

```
$ docker run busybox ls
```

Nos mostrará el contenido del directorio raíz del contenedor. Esto es porque la imagen *busybox* tiene instalado el ejecutable `ls`. Si ejecutamos:

```
$ docker run hello-world ls
```

Tendremos un error, ya que la imagen hello-world no tiene el ejecutable `ls` instalado.

Listar contenedores e imágenes

Listar contenedores en ejecución:

```
$ docker ps
```

Listar contenedores aunque no estén en ejecución:

```
$ docker ps -a
$ docker container ls -a
$ docker ps --all
```

Listar imágenes:

```
# Listar todas las imágenes
$ docker images
$ docker image ls

# Listar las imágenes de un repositorio
# docker images [NOMBRE]
$docker images mysql

REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
mysql        8.0.27    ecac195d15af   2 weeks ago     516MB
mysql        5.7.33    450379344707   7 months ago    449MB

# Filtrar búsqueda:
# docker images --filter "<key>=<value>"
# "reference" : localizar imágenes por nombre o etiqueta
# "before" : filtrar imágenes creadas antes de una fecha específica
# "since" : filtrar imágenes creadas a partir de una fecha específica
# "label" : filtrar imágenes con una etiqueta específica
# "dangling" : imágenes no utilizadas
$ docker images --filter "dangling=true"
```

Borrar contenedores e imágenes

Debemos revisar las imágenes y contenedores que quedan en nuestro Docker, especialmente porque podemos ocupar mucho espacio de nuestro ordenador con datos que no estamos usando.

Borrar contenedores detenidos:

```
$ docker rm <container_id>
$ docker rm <container_id> <container_id> <container_id>
$ docker container rm <container_id>
```

Borrar contenedor en ejecución:

```
$ docker rm -f <container_id>
```

Borrar imagen sin instancias en ejecución:

```
$ docker rmi <image_id>
$ docker image rm <image_id>
```

Borrar imagen con instancias en ejecución:

```
$ docker rmi -f <image_id>
```

Borrar imágenes no asociadas a contenedor

```
# Listar:
$ docker images -f dangling=true

# Eliminar:
$ docker rmi -f $(docker images -f dangling=true)
```

Borrar todas las imágenes:

```
$ docker rmi -f $(docker images -q)
```

Borrar todo: contenedores parados, redes no utilizadas por ningún contenedor, imágenes colgadas, caché:

```
$ docker system prune
$ docker system prune -a
```

Acceder a los logs

Acceder a los logs de un contenedor. Por ejemplo, si no hemos puesto el parámetro -a y no tenemos respuesta en nuestro terminal. Los logs se almacenan con cada ejecución del contenedor:

```
$ docker logs <container_id>
```

Parar contenedores

Envía una SIGNTERM (señal de terminal) para que el proceso pare por sí solo:

```
$ docker stop <container_id>
```

Envía una SIGKILL (señal de matar) al proceso primario que esté ejecutando el contenedor:

```
$ docker kill <container_id>
```

Si tras el comando `stop` el contenedor no se detiene en 10 segundos, Docker cancela el `stop` y envía un `kill` al contenedor

Múltiples comandos

Con `exec` lanzamos procesos sobre un contenedor en ejecución. Para enviar comandos a un contenedor en ejecución, con el atributo `it` abrimos la comunicación con el contenedor. Este atributo equivale a los parámetros `i` (cualquier cosa que escriba que vaya al canal de entrada del contenedor STDIN) y `t` (formatea texto, identación y muestra ayudas visuales).

Por ejemplo, enviamos el comando `sh` para ejecutar un terminal (también podría ser bash, zsh, Powershell...):

```
$ docker exec -it <container_id> sh
```

Volúmenes

Los volúmenes se utilizan para persistir la información generada en los contenedores. Esto es especialmente importante ya que si no persistimos esta información, cuando el contenedor se para la información se pierde.

Podemos elegir dónde persistir esta información (host local, cloud, etc.) Para persistir esta información usamos un objeto de Docker llamado volúmen. Hay dos formas de declarar volúmenes:

- Imperativa: Usando Docker-CLI, a través de terminal.
- Declarativa: Usando Dockerfile o Docker Compose.

Vamos a ver la forma imperativa, la forma declarativa la trataremos en el apartado Docker Compose:

Crear volumen

```
# docker volume create [NOMBRE_DE_VOLUMEN]
$ docker volume create db_data
```

```
db_data
```

Listar volúmenes

```
$ docker volumes ls
```

```
DRIVER    VOLUME NAME
local     db_data
```

Detalle de volumen:

```
# docker volume inspect [NOMBRE_DE_VOLUMEN]
$ docker volume inspect db_data
```

```
[
    {
        "CreatedAt": "2021-11-07T15:04:16Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/db_data/_data",
        "Name": "db_data",
        "Options": null,
        "Scope": "local"
    }
]
```

Eliminar volumen

Hasta que no borremos los contenedores que usen ese volumen, no podremos borrarlo.

```
# docker volume rm [NOMBRE_DE_VOLUMEN]
$ docker volume rm db_data
```

```
db_data
```

Eliminar volúmenes no usados por contenedores:

```
$ docker volume prune
```

```
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
577d214e014ec59f7e612f55ef8b7d56ee3f961f26ad378c92738af54ad16930
957ef00f87adf9d1eb23471f519d72ea4fc7aa4c54e81aacdb6e505650ef8c10
c8534bacc6424ce3e834a1f88c13b13eae151cf9030a3adc06345d3f48c634e7
...

Total reclaimed space: 5.21GB
```

Ejecutar contenedor asignando volumen

Al ejecutar un contenedor podemos crear un volumen y asignarlo a una carpeta del contenedor. De esta forma, lo que suceda en esta carpeta lo tendremos reflejado en una ubicación del host:

```
# docker run --volume [NOMBRE_DE_VOLUMEN]:[UBICACIÓN_EN CONTENEDOR]
$ docker run --volume db_data:/var/log nginx:latest
```

Si inspeccionamos un contenedor al que le hemos asignado un volumen:

```
# docker container inspect [NOMBRE_DE_CONTENEDOR]
$ docker container inspect my-nginx

# Filtrar y formatear (con python) únicamente la información relativa al volúmen (clave Mounts del JSON)
# docker container inspect --format "{{json .Mounts}}" [NOMBRE_DE_CONTENEDOR] | python -m json.tool
$ docker container inspect --format "{{json .Mounts}}" my-nginx | python -m json.tool
```

```
[
    {
        "Type": "volume",
        "Name": "db_data",
        "Source": "/var/lib/docker/volumes/db_data/_data",
        "Destination": "/var/log",
        "Driver": "local",
        "Mode": "z",
        "RW": true,
        "Propagation": ""
    }
]
```

Redes

Dos contenedores levantados de forma independiente no tienen comunicación entre ellos, son completamente independientes.

Supongamos que tenemos levantado un contenedor con Redis, funcionando y escuchando peticiones en su puerto. Por otro lado, tenemos levantado un contenedor Node en el que hemos instanciado una conexión con Redis. Pues bien, este último contenedor nos dará error al conectar con Redis, ya que aunque ambos contenedores estén funcionando no tienen capacidad para comunicarse entre ellos. Esto lo resolvemos con Docker Compose.

Esto lo resolvemos con Docker Compose. Cuando creamos estos dos contenedores en el mismo Docker Compose éstos se pueden comunicar entre sí haciendo referencia únicamente al nombre que se les ha dado dentro del servicio, ya que se crea una red para albergar todos los contenedores indicados. Así, en nuestra aplicación, si usamos Node la conexión con el servidor Redis se hará así:

```
const redisClient = redis.createClient({
    host: 'redis-server', // Nombre del servicio en docker-compose.yaml
    port: 6379
});
```

Estas conexiones son gestionadas por objetos conocidos como network drivers:

- Software que gestiona la red del contenedor. Se gestionan con el comando `docker network`, sin necesidad de archivos o imágenes.
- Gestionan la comunicación entre contenedores y la comunicación con el host: direcciones IP y puertos.

Crear una red

Con el atributo `--driver` indicaremos el tipo, en este caso, `bridge`:

```
$ docker network create --driver brigde mi-red
```

Docker nos devuelve el ID de la red creada:

```
0d8f3d31fbdf31c570cbfa32e60be9e5d27a399ac4f38861cced6cf2ab0d116d
```

Creamos una red tipo bridge con subnet y rango de IPS:

```
$ docker network create --driver bridge --subnet=192.168.0.0/16 --ip-
range=192.168.5.0/24 mi-red-1
```

Listar redes

```
$ docker network ls
```

```
NETWORK ID     NAME                                DRIVER    SCOPE
0feb1fdcc786   bridge                              bridge    local
565485a2b694   host                                host      local
0d8f3d31fbdf   mi-red                              bridge    local
b67ad0f12708   mi-red-1                            bridge    local
c2359f9ff803   none                                null      local
```

Como vemos, no sólo tenemos las redes que acabamos de crear, también tenemos redes que por defecto Docker ha creado. Podemos listar redes usando un filtro:

```
$ docker network ls --filter driver=bridge
```

```
NETWORK ID     NAME                                DRIVER    SCOPE
0feb1fdcc786   bridge                              bridge    local
0d8f3d31fbdf   mi-red                              bridge    local
b67ad0f12708   mi-red-1                            bridge    local
```

Conectar un contenedor a una red

```
# docker network connect [NOMBRE_DE_RED] [NOMBRE_DE_CONTENEDOR]
$ docker network connect mi-red my-nginx
```

Una vez hecho, inspeccionamos el contenedor para ver su información:

```
$ docker container inspect my-nginx
```

Lo que nos devolverá un JSON con todos los datos del contenedor, nos fijamos en la información de redes:

```
"Networks": {
    "bridge": {
```

```
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID":
"0feb1fdcc7867f0e764af49b07d7a6776238082a41be44f7219d768b86b9e16b",
        "EndpointID":
"c5754da97dfc31d84f2c7180646c9b1563269dab11ec23a1906a33082e730d40",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02",
        "DriverOpts": null
    },
    "mi-red-1": {
        "IPAMConfig": {},
        "Links": null,
        "Aliases": [
            "2b0ca24fb098"
        ],
        "NetworkID":
"b67ad0f12708a50e3f792dd6848f2062a295291ec65b096dc4daadf0d0de8b92",
        "EndpointID":
"c049d6f853c8f54853721750e0d3109ffead44a703d600075e3c2bb1537c2a75",
        "Gateway": "192.168.5.0",
        "IPAddress": "192.168.5.1",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:c0:a8:05:01",
        "DriverOpts": {}
    }
```


Y vemos la información de la red a la que hemos conectado. Veremos una red tipo bridge, esto es porque de no especificar red, Docker asigna una red bridge por defecto.

Inspeccionar red

```
# docker network inspect [NOMBRE_DE_RED]
$ docker network inspect mi-red-1
```

Y dentro del JSON de salida, vemos nuestro contenedor asignado:


```
"Containers": {
    "2b0ca24fb0989be5cbf90840bbe203600f3278ea22ec4f85c22c2a26a4277e19": {
        "Name": "my-nginx",
        "EndpointID":
"c049d6f853c8f54853721750e0d3109ffead44a703d600075e3c2bb1537c2a75",
        "MacAddress": "02:42:c0:a8:05:01",
        "IPv4Address": "192.168.5.1/16",
        "IPv6Address": ""
    }
}
```

Desconectar un contenedor de una red

```
# docker network disconnect [NOMBRE_DE_RED] [NOMBRE_DE_CONTENEDOR]
$ docker network disconnect mi-red-1 my-nginx
```

Crear nuestras propias imágenes

Dockerfile

Es un archivo de texto plano que contiene líneas con configuración: qué programas va a contener y qué va a hacer cuando se inicie su ejecución. Tiene este flujo:

1. Especificar una imagen base
2. Ejecutar comandos para instalar programas adicionales, dependencias, etc.
3. Especificar un comando para ejecutar al iniciar el contenedor

La estructura de Dockerfile es:

```
# Dockerfile

# Instrucciones base
ARG # Opcional
FROM
LABEL

# Instrucciones de configuración
WORKDIR
RUN
ADD | COPY
ENV

# Instrucciones de ejecución
CMD
ENTRYPOINT
EXPOSE
```

Ejemplo: vamos a crear nuestra propia imagen con Redis:

```
# Dockerfile

# Con FROM decimos qué imagen vamos a usar como base:
FROM alpine

# Ejecutamos comandos mientras se prepara nuestra imagen
RUN apk add --update redis

# Qué instrucción se ejecuta cuando nuestra imagen se instancia
CMD ["redis-server"]
```

Una vez hecho esto, nos situamos en el directorio y creamos el contenedor:

```
$ docker build .
```

Podemos añadir una etiqueta a la imagen y así referenciar a la imagen por esta etiqueta y no por el id:

```
$ docker build -t redis-test .
```

Descarga y crea el contenedor:

```
[+] Building 1.8s (6/6) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 249B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/alpine:latest
 => [1/2] FROM docker.io/library/alpine
 => [2/2] RUN apk add --update redis
 => exporting to image
 => => exporting layers
 => => writing image
sha256:38903694787e77ff8b67dc422c273d29be43a8ff643abbe6e2dd787c78ad6336
```

¿Qué hemos hecho? El primer paso: hemos usado como base Alpine, una distribución Linux muy ligera (apenas 5 megas) y completa. En el segundo paso nos hemos descargado e instalado Redis y, por último, el tercer paso, ejecutamos el comando `redis-server` para iniciar el servidor Redis.

Y lo iniciamos:

```
$ docker run --name RedisApp
38903694787e77ff8b67dc422c273d29be43a8ff643abbe6e2dd787c78ad6336
```

Y ya tenemos nuestra propia imagen con Redis funcionando y con el nombre RedisApp.

Publicar imagen en Docker Hub

Desde Docker Hub, dentro de nuestra cuenta, creamos un repositorio. El nombre del repositorio tendrá este formato:

```
[NOMBRE_DE_USUARIO]/[NOMBRE_DE_REPOSITORIO]
```

Desde Docker-CLI, una vez logados, creamos un nuevo tag para una imagen con el nombre del repositorio creado:

```
# docker tag [NOMBRE_DE_IMAGEN] [USUARIO]/[REPOSITORIO]:[TAG]
# Si no especificamos [TAG] tomará "latest" por defecto
$ docker tag redis-test falces/redis:redis-test
```

Una vez hecho esto, si listamos las imágenes con `$ docker images`, veremos las dos imágenes pero compartiendo `IMAGE ID`, ya que la nueva es tan sólo un alias de la anterior.

Ahora publicamos la imagen:

```
# docker image push [USUARIO]/[REPOSITORIO]:[TAG]
$ docker image push falces/redis:redis-test
```

Instrucciones base

`FROM`: Instrucción obligatoria. Especifica la imagen base, de la que vamos a partir, para crear la imagen definitiva. Es el sistema operativo de la imagen. La única instrucción que puede ser escrita antes que `FROM` es `ARG`. Lo normal es que `FROM` contenga una imagen de sistema operativo o de una aplicación.

En la primera línea de nuestro Dockerfile:

```
FROM alpine
```

Instala una imagen base, un sistema operativo. Una imagen es como un ordenador sin sistema operativo, por lo tanto lo primero que hay que hacer es instalar el sistema operativo. *Alpine* es una distribución Linux que viene con una serie de programas preinstalados que nos ayudan a crear nuestra imagen.

En una imagen base podemos especificar, usando tags, diferentes versiones. En *Docker Hub* podemos consultar todas las etiquetas que una imagen admite. Dado que *Alpine* es una distribución muy completa y ligera, muchas imágenes tienen versiones con esta distribución. Por ejemplo, si quisiéramos instalar *Node* como base, esta imagen tiene una etiqueta para instalar una versión de *Alpine* con *Node*:

```
FROM node:alpine
```

Otros comandos base:

`ARG`: Opcional. Define argumentos que usará la sentencia `FROM`. Ayuda a mantener parámetros (como, por ejemplo, versiones, bajo control):

```
ARG CODE_VERSION=16.04
```

```
# Sin ARG:
FROM ubuntu:16.04

# Con ARG:
ARG CODE_VERSION=16.04
FROM ubuntu:${CODE_VERSION}
```

`LABEL`: Opcional. Añade información metadata adicional de la imagen (ver apartado "listar imágenes"):

```
LABEL Creator: "John Doe"
```

Instrucciones de configuración

`RUN`: hace que Docker ejecute el comando especificado en la imagen base (especificada en `FROM`). Se usan para la instalación de dependencias y ejecución de otros comandos necesarios previos al uso de la imanen.

En la segunda línea de nuestro Dockerfile:

```
RUN apk add --update redis
```

El comando `apk` (Alpine Package Keeper), es el gestor de paquetes de esta distribución Linux. En estas líneas (no tiene por qué ser una sola línea) irán instrucciones aptas para el sistema operativo elegido. Por ejemplo, podríamos tener:

```
RUN apk add --update redis
RUN apk add --update gcc
```

Con lo que además de Redis habríamos instalado el GCC, el compilador de colecciones GNU. Otros ejemplos de uso del comando `RUN`:

```
# Actualizar orígenes de instalación e instalar:
RUN apt-get update && apt-get install -y curl \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Crear un directorio:
RUN mkdir /home/Codes
```

`WORKDIR`: especificamos cuál será el directorio de trabajo dentro del contenedor y partir de este momento todo se hará de forma relativa a este directorio

```
WORKDIR /usr/app
```

`COPY` : copia archivos desde nuestro ordenador al contenedor. Dado que hemos determinado el directorio de trabajo, si usamos `./` como destino los archivos se copiarán en el directorio de trabajo. Si no hubiéramos especificado directorio de trabajo, los archivos se habrían copiado en la raíz del contenedor o nos habría dado error, dependiendo de la versión de la imagen base.

El resultado sería:

```
WORKDIR /usr/app
COPY ./ ./
RUN npm install

# Alternativo
WORKDIR /usr/app
COPY . .
RUN npm install
```

`ADD`: Funciona de la misma forma que `COPY`, pero nos ofrece la posibilidad de utilizar como origen una URL o gestionar archivos comprimidos. Las buenas prácticas con Dockerfile (https://docs.docker.com/develop/develop-images/dockerfile_best-practices/) sugieren utilizar `COPY` siempre que no sea necesaria una funcionalidad específica de `ADD`.

`ENV`: declara variables de entorno dentro del contenedor:

```
ENV USER Nombre-Usuario
ENV SHELL /bin/bash
ENV LOGNAME usuario-log
```

Caso de uso

En ocasiones es posible que necesitemos instalar dependencias de nuestro proyecto. Por ejemplo, para una aplicación Node, usamos:

```
RUN npm install
```

Lo que leería nuestro archivo package.json para descargar las dependencias. Pero dado que este archivo no está dentro del contenedor temporal con el que se está creando la imagen, tendríamos un error. Lo que tenemos que hacer es copiar los archivos necesarios dentro del contenedor antes de instalar las dependencias. Primero especificamos un directorio de trabajo dentro del contenedor (si no lo especificamos intentaría copiar en el directorio raíz y tendríamos error) y luego copiamos los archivos.

Antes de instalar las dependencias, copiamos nuestros archivos al contenedor:

```
COPY ./ ./
# También válido:
COPY . .
```

Pero, ¿qué pasa si modifico algún archivo de mi aplicación? No vería reflejado el cambio en el contenedor, ya que el comando `build` lee Dockerfile y es en este momento cuando se copian los archivos, por lo que una modificación posterior no se vería reflejada. Lo primero que nos viene a la cabeza es volver a ejecutar `build`, que nos funcionaría. En aplicaciones grandes la instalación de dependencias puede llevar mucho tiempo, por lo que no sería ágil que cada vez que modifique un archivo volviera a ejecutar `build` para reconstruir el contenedor. Esto lo podemos solucionar con una pequeña estrategia en la copia:

```
COPY ./package.json ./
RUN npm install
COPY ./ ./

# También válido
COPY package.json .
RUN npm install
COPY . .
```

Con esto añadimos un paso más, muy corto, la copia de un archivo. Pero si este archivo no es modificado, cuando volvemos a construir el contenedor Docker no ejecuta el siguiente comando, ya que nada ha cambiado y, después, copia los archivos. Dado que no volvemos a instalar dependencias, el proceso `build` es mucho más rápido.

Instrucciones de ejecución

`EXPOSE`: informa a Docker del puerto en el que nuestra imagen estará escuchando peticiones. Importante: NO PUBLICA EL PUERTO, no lo hace accesible desde nuestro PC, tan sólo informa. El mapeo del puerto se debe especificar cuando levantamos el contenedor, por lo tanto la instrucción `EXPOSE` tan sólo documenta (en ocasiones es utilizado por algún proceso automático como veremos más adelante) el puerto que la aplicación de la imagen usará para escuchar peticiones:

Para mapear el puerto usamos el parámetro `-p` del comando `run`. Este parámetro funciona a modo de documentación, ciertas aplicaciones (por ejemplo Travis para integración continua) leen esta configuración y gestionan los puertos:

```
# Es lo mismo, por defecto el protocolo es TCP:
EXPOSE 80/tcp
EXPOSE 80

EXPOSE 80/udp
```

`CMD`: el comando que queremos ejecutar al iniciar el contenedor, por ejemplo:

```
CMD ["nginx", "-g", "daemon off;"]
```

Ejecutaría el comando:

```
$ nginx -g daemon off;
```

La tercera línea de nuestro Dockerfile:

```
CMD ["redis-server"]
```

Es la que ejecuta la aplicación en sí. Qué tiene que ejecutar la imagen cuando es iniciada como un contenedor.

Custom Dockerfile

Podemos tener diferentes archivos Dockerfile, por ejemplo, según en el entorno en el que se vaya a desplegar la aplicación. Si vamos a crear una imagne para un entorno de desarrollo, podemos crear un archivo llamado Dockerfile.dev. Para construir una imagen usando este nombre de archivo, usamos el parámetro `-f`, donde especificamos el nombre del archivo:

```
$ docker build -f Dockerfile.dev .
```

Build: construir imágenes

Para crear nuestra imagen usaremos el comando:

```
$ docker build .
```

Este comando toma el contenido de Dockerfile y compone la imagen. También le hemos puesto un punto al final: el contexto. Este contexto es dónde está la ubicación donde tendremos los archivos y carpetas que vamos a encapsular en nuestra imagen y dónde está nuestro archivo Dockerfile.

Siguiendo el flujo de ejecución, Docker va creando imágenes y contenedores temporales de los que crea snapshots para crear nuevas imágenes temporales y nuevos contenedores con cada vez más programas (dependencias), tantas como hayamos especificado. Cuando no hay más instrucciones que ejecutar, devuelve la última imagen generada, que será nuestra imagen.

Mientras tengamos en nuestro Dockerfile detalladas imágenes o aplicaciones que hayan sido instaladas previamente, éstas se instalarán desde la caché y no se descargarán, lo que hará el proceso mucho más rápido.

Etiquetas

Podemos etiquetar nuestra imagen para no tener que usar el container_id para su ejecución:

```
$ docker build -t myredis .
```

Podemos crear la etiqueta usando el formato:

```
nombre_de_usuario/repo_project_name:version
```

Run: levantar contenedor

Con el comando run levantamos un contenedor partiendo de una imagen previamente construida. Con este comando podemos ver qué imágenes tenemos:

```
$ docker image ls
```

Donde vemos las imágenes, sus identificadores y, si tienen, su etiqueta.

Una vez hecho esto, podemos arrancar el contenedor con un nombre, haciendo referencia a la etiqueta y no al container_id:

```
$ docker run --name MyRedisApp myredis
```

Podemos ejecutar el contenedor en segundo plano y no bloquear el terminal, usando `-d`:

```
$ docker run -d --name MyRedisApp myredis
```

También podemos añadir comandos a ejecutar dentro del contenedor al final de la instrucción. En este caso añadimos el atributo `-it` para tener interacción con el terminal y ejecutamos el terminal `sh`:

```
$ docker run -it MyRedisApp sh
```

O ejecutar un comando que ejecuta test sobre un contenedor levantado usando su `id_container`:

```
$ docker exec -it 2568d84a681c npm run test
```

Commit

Podemos crear imágenes de forma manual partiendo de contenedores que estén funcionando. Por ejemplo, iniciamos un contenedor con Alpine:

```
$ docker run -it alpine sh
```

E instalamos alguna aplicación en este contenedor:

```
# apk add --update redis
```

Esto habrá modificado el file system del contenedor, ya que ha instalado *Redis* en él. Sin cerrar este terminal para no cerrar el contenedor, en otro terminal podemos crear una nueva imagen a partir del estado actual de este contenedor:

```
$ docker ps # Para ver y copiar el id_container del contenedor que queremos copiar
$ docker commit -c 'CMD ["redis-server"]' d3fc85816e85 # -c => especificar el comando de inicio
sha256:b48d00800affea196a167d3dcc01f7248276e0be9a00757d072b76cdb646528d
# No es necesario copiar todo el hash, con un trozo es suficiente:
$ docker run --name MiInstancia b48d00800af
```

Y tendremos nuestra instancia de *Alpine* con *Redis* instalado corriendo con el nombre `MiInstancia`.

Mapeo de puertos

Si levantamos un contenedor con un servicio que escucha peticiones en un puerto determinado, si intentamos acceder a dicho servicio desde nuestro ordenador (desde fuera del contenedor) no obtendremos respuesta, ya que la aplicación está escuchando peticiones dentro del contexto del contenedor. Para esto tenemos que levantar el contenedor especificando un mapeo de puertos, que no es más que una redirección de un puerto externo al contenedor a un puerto interno del contenedor.

```
# Atributo: -p puerto_externo:puerto_interno
$ docker run --name NodeAPP -p 8000:8080 simpleweb
```

Con esto levantamos un contenedor llamado NodeAPP y las peticiones externas al puerto 8000 serán atendidas por el puerto 8080 de nuestro contenedor.

Volúmenes: mapear carpetas

Cada vez que modificamos un archivo de nuestro proyecto tenemos que reconstruir la imagen para que el archivo modificado se copie y quede dentro del contenedor que levantamos. Con los volúmenes ya no necesitamos esto, ya que lo que hacemos es, en lugar de copiar los archivos dentro del contenedor, creamos una referencia entre el directorio de trabajo del contenedor y el de nuestro ordenador local. Para configurar los volúmenes usamos el atributo `-v`:

```
$ docker run -p puerto_local:puerto_contenedor -v
carpeta_local:carpeta_contenedor <id_imagen>

# Según el sistema operativo (pwd = directorio actual)
$ docker run -p 3000:8080 -v $(pwd):/app --name MyApp b48d00800af
$ docker run -p 3000:8080 -v /usr/name/myapp:/app --name MyApp b48d00800af
```

En ocasiones tenemos carpetas dentro del contenedor pero no fuera (por ejemplo, eliminamos node_modules para ahorrar tiempo en el build). Esto hará que run devuelva error ya que la referencia node_modules no encuentra carpeta en nuestro equipo; para corregirlo debemos añadir un volumen con la carpeta que no está en nuestro equipo:

```
$ docker run -p 3000:8080 -v /app/node_modules -v /usr/name/myapp:/app --name
MyApp b48d00800af
```

Al indicar una carpeta del contenedor, sólo le decimos a Docker que no trate de mapear la unidad contra nada.

Multi Step Build Proccess

Dentro de un único fichero Dockerfile podemos instanciar diferentes imágenes. Por ejemplo, para un servidor de producción no usaremos el servidor Node, usaremos Nginx,

```
# Dockerfile

# Creamos la imagen con el alias 'builder'
FROM node:alpine AS builder
WORKDIR '/app'
COPY 'package.json' .
RUN npm install
COPY . .
RUN npm run build

FROM nginx
# Con el parámetro --from indicamos la imagen origen, usamos el alias 'build'
# y de la documentación de NGINX sacamos la carpeta donde se sirve por defecto:
COPY --from=builder /app/build /usr/share/nginx/html
```

Ahora ejecutaremos el comando:

```
$ docker build .

[+] Building 22.3s (14/14) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 296B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/nginx:latest
 => [internal] load metadata for docker.io/library/node:alpine
 => [internal] load build context
 => => transferring context: 5.99kB
 => [stage-1 1/2] FROM docker.io/library/nginx
 => [builder 1/6] FROM
docker.io/library/node:alpine@sha256:417b3856d2e5d06385123f3924c36f5735fb1f69028
9ca69f2ac9
 => CACHED [builder 2/6] WORKDIR /app
 => CACHED [builder 3/6] COPY package.json .
 => CACHED [builder 4/6] RUN npm install
 => [builder 5/6] COPY . .
 => [builder 6/6] RUN npm run build
 => [stage-1 2/2] COPY --from=builder /app/build /usr/share/nginx/html
 => exporting to image
 => => exporting layers
 => => writing image
sha256:9b49b4e173bb56e668ada14988fc6371d3b6310cfebc236b9389bb3520a3f314
```

Para construir la imagen tomamos el id de la imagen y la levantamos:

```
$ docker run -p 8080:80 9b49b4e173bb
```

Dado que hemos copiado el contenido de `/app/build` en la carpeta que Nginx sirve por defecto, sólo necesitamos mapear el puerto local que queremos utilizar (8080) con el puerto por defecto de Nginx (80).

Stop: parar contenedores

Con el comando stop y el id o nombre del contenedor, podemos pararlo:

```
$ docker stop 852b5021a89b
$ docker stop nombre_del_contenedor
```

Docker Compose

Qué es

Nos permite definir una aplicación multi contenedor en un único archivo y ejecutar nuestra aplicación y sus contenedores en una única instrucción, ya que tiene toda la información necesaria para ejecutarse.

Es una herramienta que nos sirve para compartir y compenetrar aplicaciones de diferentes contenedores. También se utiliza para levantar contenedores al mismo tiempo y nos simplifica el uso que hemos estado haciendo hasta ahora con el comando `run` a la hora de pasar argumentos, mapear puertos, asignar nombres, etc.

En este archivo definimos múltiples objetos: contenedores, redes, servicios, volúmenes, etc. y sus parámetros.

El primer paso para utilizar Docker Compose es crear un archivo `docker-compose.yaml` en el que escribir la configuración que queramos. A partir de ese momento podemos utilizar el comando `$ docker-compose` en nuestro terminal para realizar diferentes tareas.

En este archivo es vital mantener una correcta indentación de los elementos, ya que para indicar que una instrucción está dentro de otra lo hacemos únicamente con este método. Veamos cómo escribir el archivo `docker-compose.yaml`:

Versión

La primera línea del archivo docker-compose.yaml contiene la versión de Docker Compose que vamos a utilizar. La estructura de este archivo ha ido evolucionando y es necesario que indiquemos a Docker qué versión va a leer para que se adapte a cómo recoger la información. Toda la info sobre versiones la podemos encontrar en https://docs.docker.com/compose/compose-file/compose-versioning/.

```
# docker-compose.yaml
version: '3'
```

Servicios

Lo primero que vamos a hacer es meter los mismos comandos que ejecutamos en terminal (`build`, `run`, etc) y los vamos a encapsular en un archivo `docker-compose.yaml` que ejecutaremos desde Docker Compose CLI:

```yaml
# docker-compose.yaml
version: '3'          # Línea obligatoria: versión del formato Docker Compose
services:             # Contenedores que necesitamos
  redis-server:       # Contenedor redis-server
    image: 'redis'    # Usando la imagen redis
  node-app:           # Contenedor node-app
    build: .          # Construído a partir de su Dockerfile
    ports:            # Mapeo de puertos:
      - "4001:8081"   # [puerto de host]:[puerto de contenedor]
```

Para cada imagen creamos un servicio, añadiéndolo dentro de `services`. Dentro de cada servicio podemos añadir diferentes instrucciones. En el ejemplo anterior hemos definido los servicios redis-server y node-app, y dentro de cada servicio le hemos dado su configuración con diferentes instrucciones. Algunas de estas instrucciones son:

image

Especifica el nombre de la imagen que será utilizada para crear el contenedor. Puede ser una imagen local o remota (Docker Hub).

```yaml
# docker-compose.yaml
version: '3'
services:
  [NOMBRE_DE_SERVICIO]:
    image: '[NOMBRE_DE_LA_IMAGEN]'
```

En la imagen podemos especificar la versión. Esto se hace añadiendo `:[VERSION]` después del nombre de la imagen. En caso de no especificar versión, Docker usará `:latest` para usar la versión más reciente.

```yaml
# docker-compose.yaml
version: '3'
services:
  redis-server:
    # [IMAGEN]:[VERSION]
    image: redis:latest
```

container_name

Establece un nombre para el contenedor

```yaml
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
```

build

Se utiliza para declarar el contexto (carpeta raíz) y el archivo Dockerfile que se utilizará. Aquí ejemplificamos dos formas de usar la instrucción `build`:

1. Ejemplo 1: archivo Dockerfile que se encuentre en la misma ubicación.
2. Ejemplo 2: dando un contexto (carpeta raíz), una ubicación y un nombre de archivo (si especificamos `conext: .` indicamos la carpeta donde se encuentre el archivo `docker-compose.yaml`)

Si no se especifica nombre de archivo, por defecto Docker trata de localizar un archivo con el nombre `Dockerfile` (sin extensión).

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    # Ejemplo 1
    build: .

    # Ejemplo 2
    build:
    context: ./redis_conf
    dockerfile: Dockerfile.dev
```

command

Ejecuta un comando en el contenedor. Es un array, por lo que deberán ir los comandos listados, indentados y precedidos con un guion:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
```

restart

Con Docker Compose podemos controlar los procesos: qué hacer cuando nuestra aplicación falla y devuelve un código de error. Esto lo hacemos con la instrucción `restart`. Tenemos cuatro posibles actuaciones:

- `'no'` : no se reinicia el contenedor. Con comillas, ya que en yaml un no equivale a false, y no sería interpretado.
- `always` : se reinicia el contenedor siempre
- `on-failure` : se reinicia el contenedor sólo si se devolvió un código de error:

    - 0 = se detuvo el proceso, pero es ok, está controlado
    - (Resto de códigos) = código de error
- `unless-stopped` : levantar siempre a no ser que los desarrolladores, a través de la línea de comandos, paren el contenedor.

Se añade la instrucción `restart` seguida de la opción justo después de declarar el servicio en `docker-compose.yaml`:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
    restart: always # Se reiniciará el contenedor siempre
```

volumes

Docker Compose nos ayuda a simplificar los parámetros que añadimos a `docker run`. Uno de ellos es `volumes`, que nos sirve para mapear unidades entre nuestro ordenador y el contenedor. El formato es [CARPETA_DE_HOST]:[CARPETA_DE_CONTENEDOR]

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
    restart: always
    volumes:
      - /home/user/data:/data
```

Este ejemplo está hecho con una base de datos. En el caso de las aplicaciones que hemos visto anteriormente, tenemos en Dockerfile una sentencia `COPY` para copiar al contenedor nuestros archivos locales, lo que si mapeamos unidades no sería necesario. Es interesante mantener esta línea en Dockerfile de cara a futura referencia, aunque queda inservible ya que cualquier acceso a /app irá directamente a nuestra carpeta local.

Para ver los puntos de montaje de un contenedor levantado:

```
$ docker inspect --format='{{json .Mounts}}' redis-server | python -m json.tool
```

environment

Declararemos variables de entorno para nuestro contenedor:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
    restart: always
    volumes:
      - /home/user/data:/data
    environment:
      REDIS_USER: master
      REDIS_PASSWORD: masterpass
```

Si entramos en el contenedor con un terminal y escribimos `$ echo $[NOMBRE_DE_VARIABLE]` veríamos su valor.

depends_on

Indica qué contenedores deberán estar levantados antes de levantar el contenedor en el que se declara esta sentencia. Por ejemplo, si a nuestro docker-compose.yaml añadimos un servicio de una aplicación que estamos desarrollando, podemos indicarle que hasta que no esté levantado el contenedor de la base de datos no levante el de la aplicación, ya que depende de la base de datos para su funcionamiento:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
    restart: always
    volumes:
      - /home/user/data:/data
    environment:
      REDIS_USER: master
      REDIS_PASSWORD: masterpass
  # Segundo servicio:
  my-app:
    image: my-react-app
    build: .
    volumes:
      - ./:/var/www
```

```
    # Dependencia:
    depends_on:
      - redis-server
```

ports

Expone los puertos del contenedor hacia el host. El formato es `"[PUERTO_DE_HOST]:[PUERTO_DE_CONTENEDOR]"`:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
      - --protected-mode no
    restart: always
    volumes:
      - /home/user/data:/data
    environment:
      REDIS_USER: master
      REDIS_PASSWORD: masterpass
  my-app:
    image: my-react-app
    build: .
    volumes:
      - ./:/var/www
    depends_on:
      - redis-server
    ports:
      - "8080:80"
```

Cuando el contenedor esté levantado podremos acceder a nuestra aplicación a través de http://localhost:8080: el servidor web está escuchando en el puerto 80, pero hemos mapeado nuestro puerto local 8080 contra ese puerto 80 del contenedor. Si necesitáramos acceder al servicio de la base de datos también podríamos añadir una instrucción `ports` en su servicio.

Volúmenes

En el apartado Docker CLI vimos la forma imperativa de crear volúmenes. Vamos a ver la forma declarativa, usando Docker Compose. Existen estos tipos de volúmenes:

Volúmenes anónimos

Cualquier ejecución de un archivo docker-compose.yaml crea, por defecto, un volumen anónimo. La información persistida en este tipo de volumen se pierde si eliminamos el contenedor y no es accesible por otros contenedores. En Linux la información se almacena en la carpeta `/var/lib/docker/volume`, en Windows es en `C:\ProgramData\docker\volumes`:

```yaml
# docker-compose.yaml
version: '3'
services:
  redis-server:
    # ...
    volumes:
      # Ruta del contenedor
      - /data
```

Docker Named Volumes

La información almacenada en un volumen de este tipo persiste incluso si eliminamos el contenedor y es accesible por otros contenedores. En Linux la información se almacena en la carpeta `/var/lib/docker/volume`, en Windows es en `C:\ProgramData\docker\volumes`.

Por un lado, en nuestro docker-file, declaramos el volumen añadiendo la instrucción `volumes` al mismo nivel de indentación que `services`, ya que es un objeto diferente:

```yaml
# docker-compose.yaml
version: '3'
services:
  redis-server:
    # ...
    volumes:
      # Indicamos el nombre del volumen declarado en volumes
      - db_data:/data
volumes:
  - db_data:
```

Este tipo de volúmenes pueden ser:

- Interno: es la forma por defecto si no se especifica lo contrario. Este tipo de volúmenes tiene el alcance dentro del docker-compose.yaml y Docker los crea si no existen. Los volúmenes creados en el ejemplo anterior son internos.

> Desde la versión 3.4 de Docker File el nombre del volumen puede venir de una variable de entorno declarada en un archivo `.env` situado en la misma carpeta que `docker-compose.yaml`.
>
> Es posible declarar el volumen como sólo lectura para aumentar la seguridad.

- Externo: los volúmenes externos son los que hemos creado de forma imperativa usando Docker-CLI y tenemos que indicarlo de esta forma:

```yaml
# docker-compose.yaml
version: '3'
services:
  redis-server:
    # ...
    volumes:
      - db_data:/data
volumes:
  db_data:
    external: true
```

Bind Mounts

Sólo tienen una diferencia con los Named Volumes: podemos especificar cualquier carpeta del host para persistir la información.

En este ejemplo, en el servicio my-app dejamos mapeado el volumen desde la carpeta de host donde tenemos nuestros archivos de la aplicación a la carpeta del contenedor desde la que el servidor web sirve la aplicación (bind mount), mientras que para el servicio de base de datos le asignamos el volumen declarado en `volumes` (named volume):

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    # ...
    volumes:
      - db_data:/data
    # ...
  my-app:
    # ...
    volumes:
      - ./myApp:/var/www
    # ...
volumes:
  - db_data:
```

Redes

Si no se especifica nada, por defecto Docker crea una red tipo bridge y añade a ella los contenedores que esté levantando. Podemos configurar una red en Compose y añadir a ella los servicios, configurando sus características. Creamos la red con la clave de primer nivel `networks`:

```
# docker-compose.yaml
version: '3'
# ...
networks:
  my-net:
    driver: bridge
    ipam: # IP Address Management
      config:
        - subnet: 100.0.0.0/24 # Rango de IPs
        gateway: 100.0.0.1 # Opcional
```

Ahora, indicamos a los servicios a qué red pertecerán:

```
# docker-compose.yaml
version: '3'
services:
  redis-server:
    image: redis:latest
    container_name: redis-server
    build:
      context: ./redis_conf
      dockerfile: Dockerfile.dev
    command:
```

```
      - --protected-mode no
    restart: always
    volumes:
      - /home/user/data:/data
    environment:
      REDIS_USER: master
      REDIS_PASSWORD: masterpass
    networks:
      # Nombre de la red asignado en networks
      my-net:
        ipv4_address: 100.0.0.2 # Dirección IPv4
  my-app:
    image: my-react-app
    build: .
    volumes:
      - ./:/var/www
    depends_on:
      - redis-server
    ports:
      - "8080:80"
    networks:
      # Nombre de la red asignado en networks
      my-net:
        ipv4_address: 100.0.0.3 # Dirección IPv4
networks:
  my-net:
    driver: bridge
    ipam: # IP Address Management
      config:
        - subnet: 100.0.0.0/24 # Rango de IPs
          gateway: 100.0.0.1 # Opcional
```

Podemos añadir una serie de nombres DNS y asignarlos a una dirección IP, de la misma forma que hacemos en nuestro fichero hosts en Windows, por ejemplo:

```
# docker-compose.yaml
version: '3'
services:
  # ...
  my-app:
    # ...
    networks:
      my-net:
        ipv4_address: 100.0.0.3
    extra-hosts:
      # Añadimos con el formato [NOMBRE_DNS]:[IP]
      - "otro.servicio.local:100.0.0.4"
networks:
  my-net:
    driver: bridge
    ipam:
      config:
        - subnet: 100.0.0.0/24
          gateway: 100.0.0.1
```

Ejecución

Como hemos comentado, Docker Compose nos ayuda a ejecutar los comandos y parámetros que ejecutamos en terminal (`build`, `run`). Así:

```
$ docker run myimage
# Se corresponde con:
$ docker-compose up
$ docker-compose up -d # Ejecuta en segundo plano

$ docker build .
$ docker run myimage
# Se corresponde con:
$ docker-compose up --build

$ docker stop nombre_contenedor
# Se corresponde con
$ docker-compose down # Para y borra los contenedores
```

Como vemos, con `docker-compose up` no especificamos imagen, ya que lo que hacemos es leer el archivo `docker-compose.yaml` y ejecutar sus instrucciones, allí es donde están los nombres de las imágenes.

Comprobar el estado de los contenedores

Con el comando

```
$ docker-compose ps
```

vemos el estado de los contenedores que están en nuestro archivo `docker-compose.yaml`.

Docker Compose CLI

Disponemos de comandos para trabajar con Docker Compose desde la línea de comandos

- Ver el contenido de docker-compose.yaml

  ```
  $ docker-compose config
  ```

- Ver los servicios declarados

  ```
  $ docker-compose config --services
  ```

  ```
  api
  client
  nginx
  postgres
  redis
  worker
  ```

- Ver las imágenes que se usarán para crear los contenedores:

  ```
  $ docker-compose images
  ```

- Ver los logs:

```
$ docker-compose logs
```

- Ver los últimos 10 logs:

```
$ docker-compose logs --tail=10
```

- Ver los contenedores que están en ejecución:

```
$ docker-compose ps
```

- Ver los procesos en ejecución de todos los contenedores:

```
$ docker-compose top
```

- Parar contenedores y eliminar recursos:

```
$ docker-compose down
```

Kubernetes

Comandos

Iniciar un servicio

```
$ kubectl apply -f nombre-archivo.yaml
```

Listar servicios por tipo

```
$ kubectl get pods
$ kubectl get services
$ kubectl get deployments

$ kubeclt get pods,services

$ kubectl get all
```

Lista servicios con namespaces

```
$ kubectl get deployments --all-namespaces
$ kubectl get all --all-namespaces
```

```
NAMESPACE     NAME                READY   UP-TO-DATE   AVAILABLE   AGE
default       client-deployment   1/1     1            1           28m
default       web                 5/5     5            5           5m50s
kube-system   coredns             2/2     2            2           32h
```

> Fuente: https://stackoverflow.com/questions/40686151/kubernetes-pod-gets-recreated-when-deleted

Descripción de un servicio

```
# kubectl describe [type]/[name]
$ kubectl describe deplyment/client-deployment
```

Borrar despliegue

```
# kubectl delete -n [NAMESPACE] [type] [DEPLOYMENT]
$ kubectl delete -n default deployment web
```

> Fuente: https://stackoverflow.com/questions/40686151/kubernetes-pod-gets-recreated-when-deleted

Cambiar una propiedad de un servicio

```
# kubectl set [property] [type]/[name] [container_name] = [new image to use]

# Forzar la actualización de un servicio:
$ kubectl set image deployment/client-deployment client=stephengrinder/multi-client:v5
```

Recursos

Docker

Documentación oficial

https://docs.docker.com/

https://docs.docker.com/get-started/overview/

Dockerfile

Documentación: https://docs.docker.com/engine/reference/builder/

Docker Compose

Referencia: https://docs.docker.com/compose/compose-file/compose-file-v3/#volume-configuration-reference

Versionado: https://docs.docker.com/compose/compose-file/compose-versioning/

Recursos exernos

AWS: ¿Qué es Docker?

https://aws.amazon.com/es/docker/

Artículos

- Running a Docker container as a non-root user: https://medium.com/redbubble/running-a-docker-container-as-a-non-root-user-7d2e00f8ee15
- Running Docker Containers as Current Host User: https://jtreminio.com/blog/running-docker-containers-as-current-host-user/
- Using volumes in Docker Compose: https://devopsheaven.com/docker/docker-compose/volumes/2018/01/16/volumes-in-docker-compose.html

Kubernetes

Official cheatsheet

https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Recursos externos

Cursos IBM

https://www.ibm.com/es-es/cloud/kubernetes-service/kubernetes-tutorials?utm_content=SRCWW&p1=Search&p4=43700066871613664&p5=p&gclid=Cj0KCQjw_fiLBhDOARIsAF4khR39sNFIo5Ofe5sAc6FG527DTbobS-8ZCiYXDLAa4NB_c5SCxdzicZoaAlTVEALw_wcB&gclsrc=aw.ds

Integración continua

Travis

https://www.travis-ci.com/