Dockerfiles

La mejor fuente de información de dockerfiles se encuentra en el sitio oficial de docker: https://docs.docker.com/engine/reference/builder/

A continuación puedes ver la lista completa de instrucciones que podemos usar para crear un Dockerfile:

FROM — Especifica la base para la imagen.

ENV — Establece una variable de entorno persistente.

ARG — Permite definir una variable usable en el resto del Dockerfile con la sintaxis ${NOMBRE_DEL_ARG}

RUN — Ejecuta el comando especificado. Se usa para instalar paquetes en el contenedor.

COPY — Copia archivos y directorios al contenedor.

ADD — Lo mismo que COPY pero con la funcionalidad añadida de descomprimir archivos .tar y la capacidad de añadir archivos vía URL.

WORKDIR — Indica el directorio sobre el que se van a aplicar las instrucciones siguientes.

ENTRYPOINT — Docker tiene un Entrypoint por defecto, /bin/sh -c, que se ejecuta cuando el contenedor está listo. Este comando permite sobreescribirlo.

CMD — Especifica el comando y argumentos que se van a pasar al contenedor. Se ejecutarán junto con lo indicado en el Entrypoint.

EXPOSE — Indica que puerto del contenedor se debe exponer al ejecutarlo. No lo expone directamente.

LABEL — Nos permite aportar meta-datos a la imagen.

VOLUME — Crea un directorio sobre el que se va a montar un volumen para persistir datos más allá de la vida del contenedor.

USER — Establece el usuario que se va a usar cuando se ejecute cualquier operación posterior con RUN, CMD y ENTRYPOINT.

SHELL — Permite especificar el uso de otra terminal, como zsh, csh, tcsh, powershell, u otras..

STOPSIGNAL — Indica una señal que va a finalizar el contenedor.

HEALTHCHECK — Indica a Docker una manera de testear el contenedor para verificar que sigue funcionando correctamente.

ONBUILD — Cuando la imagen donde se encuentra se use como base de otra imagen, va a actuar de trigger y va a ejecutar el comando que le indiquemos.

Comando **FROM**

Un archivo Dockerfile solo puede empezar por dos tipos de instrucción, una es ARG y la otra es FROM.

Todas las imágenes Docker necesitan de otra imagen que les sirva de **base**. FROM sirve para indicar que imagen queremos usar.

Modo de empleo:

```
FROM ubuntu:18.04
```

En el ejemplo anterior usamos la imagen Ubuntu, disponible en el registro oficial de Docker.

Normalmente las imágenes oficiales solo indican un nombre. Si quisiéramos usar una imagen no oficial deberíamos especificar a qué usuario pertenece:

```
nginx/nginx-ingress
```

En el ejemplo de Ubuntu, además, especificamos que queremos la imagen con el Tag 18.04. En caso de no indicar ningún tag, la versión que Docker va a descargar será la “latest”.

Comando **ENV**

Se usa para **declarar variables de entorno** que van a ser visibles para las siguientes instrucciones y para el contenedor resultante.

Modo de empleo:

```
ENV <key> <value>
ENV <key>=<value>
```

Las dos formas son completamente equivalentes.

Comando **RUN, CMD y ENTRYPOINT**

La sintaxis de Dockerfile a menudo nos permite conseguir el mismo resultado de muchas formas diferentes. Este es el caso con RUN, CMD y ENTRYPOINT, todas estas instrucciones **permiten ejecutar comandos**. Aunque cada uno de ellos sirve para una situación diferente.

RUN — Se usa para ejecutar comandos relacionados con la instalación de paquetes.

CMD — Indica el comando y argumentos que va a ejecutar el entrypoint.

ENTRYPOINT — Se encarga de ejecutar un comando cuando el contenedor arranca. Por defecto el entrypoint es “/bin/sh -c”.

Ejemplo:

```
FROM ubuntu
ENTRYPOINT ["sleep"]
CMD ["10"]
```

En este caso, cambiamos el entrypoint por defecto para que se ejecute el comando sleep y usamos CMD para indicar el argumento “10”.

RUN no solo ejecuta una instrucción, sino que además **crea una imagen** después de haberse ejecutado.

Docker internamente cachea todas las imágenes con las que trata, una vez crea una imagen y la tiene almacenada, nunca más la va a crear.

Cuando creamos un contenedor, y se generan todas sus capas, es probable que la primera ejecución tarde algo de tiempo. La segunda vez, Docker ya tendrá todas las imágenes que necesita para crear el contenedor en un instante.

Si usamos RUN, aprovechamos este mecanismo para guardar el resultado de su ejecución. Por lo que cuando creemos un contenedor por segunda vez ya no se tendrá que calcular más.

Esto significa que RUN no solo se puede usar para instalar paquetes, pero es especialmente útil para eso.

```
RUN pip install pip
```

Ahora lo importante, **¿cuál es la diferencia entre ENTRYPOINT y CMD?**

Cuando iniciamos un contenedor, podemos indicar una lista de argumentos que le queremos proporcionar:

```
docker run tu_imagen arg1 arg2
```

Los argumentos que usemos sobrescribirán el valor de CMD en caso de haberlo y serán ejecutados junto con el comando del ENTRYPOINT.

Lo que se ejecute en el contenedor será el resultado de los valores que tenga ENTRYPOINT + los valores que tenga CMD, que pueden ser sustituidos con los argumentos con los que ejecutemos el contenedor.

Como el ENTRYPOINT por defecto es “/bin/sh -c”, nos permite ejecutar cualquier comando de terminal directamente con CMD. En la práctica solemos ver lo siguiente:

```
CMD ["ejecutable", "param1", "param2"]
Es equivalente a:
CMD ejecutble param1 param2
```

Aunque si cambiamos el ENTRYPOINT, el modo de empleo de CMD pasa a ser:

```
CMD ["param1","param2"]
```

ENTRYPOINT permite indicarle argumentos, aunque estos no van a ser sobreescribibles vía terminal:

```
ENTRYPOINT [“/bin/echo”, “Hello”]
```

Comando **COPY y ADD**

La instrucción **COPY**, indica a Docker que coja los archivos de nuestro equipo y los añada a la imagen con la que estamos trabajando. COPY crea un directorio en caso de que no exista.

```
Modo de empleo:
COPY . ./app
COPY <fuente> <destino>
```

Indica que queremos coger todos los archivos del directorio actual (.) y trasladarlos a al directorio ./app de la imagen.

ADD hace exactamente lo mismo que COPY, pero además añade dos funcionalidades. La primera es que permite indicar contenidos vía URL, y la segunda es la extracción de archivos TAR.

```
ADD http://example.com/directorio /usr/local/remote/
ADD recursos/jdk-7u79-linux-x64.tar.gz /usr/local/tar/
```

Comando **WORKDIR**

WORKDIR se usa para **cambiar** el **directorio actual** para las siguientes instrucciones: COPY, ADD, RUN, CMD y ENTRYPOINT.

Modo de empleo:

Para dar claridad a tus archivos Dockerfile, es preferible usar WORKDIR a otra instrucción para cambiar el directorio actual de la imagen. Si el directorio especificado no existe, se creará automáticamente. Además, podemos usar este comando varias veces en el mismo archivo, si tenemos que trabajar con varios directorios a la vez.

```
FROM ubuntu:18.04
WORKDIR /proyecto
RUN npm install
WORKDIR ../proyecto2
RUN touch archivo1.cpp
```

Comando **EXPOSE**

La instrucción **EXPOSE** muestra que puerto queremos publicar para acceder al contenedor y opcionalmente el protocolo. Es muy importante tener claro que esta instrucción **NO publica directamente el puerto**. Su función es **comunicar** a la persona que va a usar la imagen qué puerto se deben usar.

Modo de empleo:

```
EXPOSE 80/udp
```

Comando **VOLUME**

La instrucción **VOLUME** crea un punto en el sistema de archivos de la imagen que se va a usar para montar un volumen externo. Eso significa que la información que guardemos en el directorio indicado va a perdurar en caso de que el contenedor deje de ejecutarse.

Modo de empleo:

```
FROM ubuntu
RUN mkdir /mivol
RUN echo “hello world” > /mivol/hola
VOLUME /mivol
```

Actividad Práctica: Ejercicios con Dockerfiles

Mi primer Dockerfile

Los *Dockerfile* son los archivos que contienen las instrucciones que crean las imagenes. Deben estar guardados dentro de un *build context*, es decir, un directorio. Este directorio es el que contiene todos los archivos necesarios para construir nuestra imagen, de ahí lo de *build context*.

Creamos nuestro *build context*

```
1   mkdir -p  ~/Sites/hello-world
2   cd ~/Sites/hello-world
3   echo "hello" > hello
```

Dentro de este directorio crearemos un archivo llamado *Dockerfile* con este contenido:

```
1   FROM busybox
2   COPY /hello /
3   RUN cat /hello
```

FROM	Indica la imagen base sobre la que se basa esta imagen
COPY	Copia un archivo del *build context* y lo guarda en la imagen
RUN	Ejecuta el comando indicado durante el proceso de creación de imagen.

Ahora para crear nuestra imagen usaremos `docker build`.

```
docker build -t helloapp:v1 .
```

El parámetro `-t` nos permite etiquetar la imagen con un nombre y una versión. El `.` indica que el *build context* es el directorio actual.

El resultado de ejecutar lo anterior sería:

```
$ docker build -t helloapp:v1 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM busybox
latest: Pulling from library/busybox
8c5a7da1afbc: Pull complete
Digest:
sha256:cb63aa0641a885f54de20f61d152187419e8f6b159ed11a251a09d115fdff9
bd
Status: Downloaded newer image for busybox:latest
 ---> e1ddd7948a1c
Step 2/3 : COPY /hello /
 ---> 8a092965dbc9
Step 3/3 : RUN cat /hello
 ---> Running in 83b5498790ca
hello
Removing intermediate container 83b5498790ca
 ---> f738f117d4b6
Successfully built f738f117d4b6
Successfully tagged helloapp:v1
```

Y podremos ver que una nueva imagen está instalada en nuestro equipo:

```
$ docker images
REPOSITORY   TAG  IMAGE ID      CREATED         SIZE
helloapp     v1   f738f117d4b6  40 seconds ago  1.16MB
```

Creando aplicaciones en contenedores

Vamos a crear una aplicación en python y la vamos a guardar en un contenedor. Comenzamos creando un nuevo *build context*:

```
mkdir -p  ~/Sites/friendlyhello
cd ~/Sites/friendlyhello
```

El código de la aplicación es el siguiente, lo guardaremos en un archivo llamado `app.py`:

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2,
socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"),
hostname=socket.gethostname(),  visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Nuestra aplicación tiene una serie de dependencias (librerías de terceros) que guardaremos en el archivo *requirements.txt*:

```
Flask
Redis
```

Y por último definimos nuestro *Dockerfile*:

```
# Partimos de una base oficial de python
FROM python:2.7-slim

# El directorio de trabajo es desde donde se ejecuta el contenedor al
iniciarse
WORKDIR /app

# Copiamos todos los archivos del build context al directorio /app
del contenedor
COPY . /app

# Ejecutamos pip para instalar las dependencias en el contenedor
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Indicamos que este contenedor se comunica por el puerto 80/tcp
EXPOSE 80

# Declaramos una variable de entorno
ENV NAME World

# Ejecuta nuestra aplicación cuando se inicia el contenedor
CMD ["python", "app.py"]
```

Para conocer todas las directivas visita la documentación oficial de Dockerfile.

En total debemos tener 3 archivos:

```
$ ls
app.py  Dockerfile  requirements.txt
```

Ahora construimos la imagen de nuestra aplicación:

```
docker build -t friendlyhello .
```

Y comprobamos que está creada:

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
friendlyhello       latest              88a822b3107c        56 seconds
ago      132MB
```

Probar nuestro contenedor

Vamos a arrancar nuestro contenedor y probar la aplicación:

```
docker run --rm -p 4000:80 friendlyhello
```

Tip

Normalmente los contenedores son de usar y desechar, sobre todo cuando hacemos pruebas. El parámetro `--rm` borra automáticamente un contenedor cuando se para. Recordemos que los datos volátiles siempre se deben guardar en volúmenes.

Lo que arranca la aplicación Flask:

```
$ docker run --rm -p 4000:80 friendlyhello
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Comprobamos en el puerto 4000 si efectivamente está iniciada o no: http://localhost:4000.

Obtendremos un mensaje como este:

```
Hello World!

Hostname: 0367b056e66e
Visits: cannot connect to Redis, counter disabled
```

Ya tenemos una imagen lista para ser usada. Pulsamos `Control+C` para interrumpir y borrar nuestro contenedor.

Creando la aplicación

En este caso nuestro contenedor no funciona por sí mismo. Es muy habitual que dependamos de servicios para poder iniciar la aplicación, habitualmente bases de datos. En este caso necesitamos una base de datos *Redis* que no tenemos.

Como vimos en el apartado anterior, vamos a aprovechar las características de *Compose* para levantar nuestra aplicación.

Vamos a crear el siguiente archivo *docker-compose.yaml*:

```
version: "3"
services:
    web:
        build: .
        ports:
            - "4000:80"
    redis:
        image: redis
        ports:
            - "6379:6379"
        volumes:
            - "./data:/data"
        command: redis-server --appendonly yes
```

La principal diferencia con respecto al capítulo anterior, es que en un servicio podemos indicar una imagen (parámetro `imagen`) o un *build context* (parámetro `build`).

Esta es una manera de integrar las dos herramientas que nos proporciona *Docker*: la creación de imágenes y la composición de aplicaciones con servicios.

Balanceo de carga

Vamos a modificar nuestro *docker-compose.yaml*:

```
version: "3"
services:
    web:
        build: .
    redis:
        image: redis
        volumes:
            - "./data:/data"
        command: redis-server --appendonly yes
    lb:
        image: dockercloud/haproxy
        ports:
            - 4000:80
        links:
            - web
        volumes:
            - /var/run/docker.sock:/var/run/docker.sock
```

En este caso, el servicio web no va a tener acceso al exterior (hemos eliminado el parámetro `ports`). En su lugar hemos añadido un balanceador de carga (el servicio `lb`).

Vamos a arrancar esta nueva aplicación, pero esta vez añadiendo varios servicios web:

```
docker-composer up -d --scale web=5
```

Esperamos a que terminen de iniciar los servicios:

```
$ docker-compose up -d --scale web=5
Creating network "friendlyhello_default" with the default driver
Creating friendlyhello_redis_1 ... done
Creating friendlyhello_web_1   ... done
Creating friendlyhello_web_2   ... done
Creating friendlyhello_web_3   ... done
Creating friendlyhello_web_4   ... done
Creating friendlyhello_web_5   ... done
Creating friendlyhello_lb_1    ... done
```

Podemos comprobar como del servicio web nos ha iniciado 5 instancias, cada uno con su sufijo numérico correspondiente. Si usamos `docker ps` para ver los contenedores disponibles tendremos:

```
$ docker ps
CONTAINER ID  IMAGE               [...]   PORTS
NAMES
77acae1d0567  dockercloud/haproxy  [...]   443/tcp, 1936/tcp,
0.0.0.0:4000->80/tcp  friendlyhello_lb_1
5f12fb8b80c8  friendlyhello_web    [...]   80/tcp
friendlyhello_web_5
fb0024591665  friendlyhello_web    [...]   80/tcp
friendlyhello_web_2
a20d20bdd129  friendlyhello_web    [...]   80/tcp
friendlyhello_web_4
53d7db212df8  friendlyhello_web    [...]   80/tcp
friendlyhello_web_3
41218dbbb882  friendlyhello_web    [...]   80/tcp
friendlyhello_web_1
06f5bf6ed070  redis               [...]   6379/tcp
friendlyhello_redis_1
```

Vamos a fijarnos en el `CONTAINER ID` y vamos a volver a abrir nuestra aplicación: http://localhost:4000.

Si en esta ocasión vamos recargando la página, veremos como cambian los *hostnames*, que a su vez coinciden con los identificadores de los contenedores anteriores.

Info

Esta no es la manera adecuada de hacer balanceo de carga, puesto que todos los contenedores están en la misma máquina, lo cual no tiene sentido. Solo es una demostración. Para hacer balanceo de carga real necesitaríamos tener o emular un cluster de máquinas y crear un enjambre (*swarm*). Pero veremos luego esto con kubernetes.

Compartir imágenes

Si tenemos una imagen que queramos compartir, necesitamos usar un registro. Existe incluso una imagen que nos permite crear uno propio, pero vamos a usar el repositorio público de *Docker*.

Los pasos son:

1. Crear una cuenta de usuario en el repositorio oficial de *Docker*.
2. Pulsar sobre el botón "*Create Repository +*".
3. En el formulario hay que rellenar solo un dato obligatoriamente: el nombre. Usaremos el de la imagen: *friendlyhello*.
   Nuestro nombre de usuario es el namespace y es obligatorio que tenga uno. Si estuvieramos en alguna organización podríamos elegir entre varios. El resto de campos lo dejamos como está por el momento. La cuenta gratuita solo deja tener un repositorio privado, asi que no lo malgastaremos aquí.
4. Ahora tenemos que conectar nuestro cliente de *Docker* con nuestra cuenta en el *Hub*. Usamos el comando `docker login`.

```
$ 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: username
Password: ********
WARNING! Your password will be stored unencrypted in
/home/sergio/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#creden
tials-store
```

5. **Danger**
   Las claves se guardan sin cifrar. Hay que configurar un almacen de claves o recordar hacer `docker logout` para borrarla.
   Visita la web de referencia para saber como crear un almacen.
6. Para que las imágenes se puedan guardar, tenemos que etiquetarla con el mismo nombre que tengamos en nuestro repositorio más el *namespace*. Si nuestra cuenta es '*username*' y el repositorio es '*friendlyhello*', debemos crear la imagen con la etiqueta '*username/friendlyhello*'.

```
$ docker build -t username/friendlyhello .
```

7. **Tip**
   Por defecto ya hemos dicho que la etiqueta si no se indica es *latest*. Podemos indicar más de una etiqueta para indicar versiones:

```
$ docker build -t username/friendlyhello -t
username/friendlyhello:0.1.0 .
```

8. En la próxima que hagamos le subimos la versión en la etiqueta:

```
$ docker build -t username/friendlyhello -t
username/friendlyhello:0.2.0 .
```

9. De esta manera nuestra imagen aparecerá con tres etiquetas: *latest* y *0.2.0* que serán la misma en realidad, y *0.1.0*.
10. Ahora ya podemos enviar nuestra imagen:

```
$ docker push username/friendlyhello
```

Para practicar entre pares:

1. Cambia el *docker-compose.yaml* para usar tu imagen en vez de hacer *build*.
2. Cambia el *docker-compose.yaml* para usar la imagen de algún compañero.

Historial de Versiones

Versión	Fecha	Autor	Descripción
1.0	07/10/2021	Javier Dorigoni	Dockerfile sintaxis y Ejercicios prácticos de uso y escritura de dockerfiles.