

Programa DevOps



Working Time #3.3

≡ Dockerfile

Es hora de que pongas en práctica todo lo aprendido. 🤖

Este apartado tiene el objetivo de ayudarte a seguir potenciando tus habilidades, por lo que a continuación encontrarás diferentes **desafíos** que podrás resolver de forma independiente y a tu ritmo.

¡Manos a la obra!

Práctica Crear imágenes propias

Ya hemos visto cómo usar imágenes de terceros para crear aplicaciones y servicios. Pero ¿si no hay ninguna imagen que tenga lo que queremos? ¿O si queremos hacer una imagen de nuestra aplicación para distribuirla?

Docker permite crear imágenes propias. Aunque podríamos hacerla partiendo de cero, es un esfuerzo que no tiene sentido. Existen ya imágenes base para crear las nuestras y es mucho más fácil crear una imagen basándose en otra que hacerlo todo nosotros. Podemos partir de una imagen base que parte de un lenguaje de programación (python, php) o de alguna distribución (ubuntu, debian).

Mi primer Dockerfile

Los *Dockerfile* son los archivos que contienen las instrucciones que crean las imágenes. 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*.

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

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

COPY /hello /
 RUN cat /hello

Directiva	Explicación
FROM	Indica la imagen base sobre la que se basa esta imagen
COPY	Copia un archivo del <i>build context</i> 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:cb63aa0641a885f54de20f61d152187419e8f6b159ed11a251a09d115fdff9bd
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"]
```

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 tirar, 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 a lo que vimos anteriormente, es que en un servicio podemos indicar una imagen (parámetro `image`) 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-compose 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 cómo cambian los hostnames, que a su vez coinciden con los identificadores de los contenedores anteriores.

 **Nota:** 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 clustes de máquinas y crear un enjambre (swarm).

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 estuviéramos en alguna organización podríamos elegir entre varios. El resto de los campos lo dejamos como está por el momento. La cuenta gratuita solo deja tener un repositorio privado, así que no lo malgastamos 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/#credentials-store
```

 Nota: Las claves se guardan sin cifrar. Hay que configurar un almacén de claves o recordar hacer docker logout para borrarla.

Visita [la web de referencia para saber cómo crear un almacén](#).

- 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 .
```

 **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 .
```

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

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

De esta manera nuestra imagen aparecerá con tres etiquetas: latest y 0.2.0 que serán la misma en realidad, y 0.1.0.

- Ahora ya podemos enviar nuestra imagen:

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

Ejercicios adicionales

- Cambia el docker-compose.yaml para usar tu imagen en vez de hacer *build*.

Cambia el docker-compose.yaml para usar la imagen de algún compañero.

