edix
Digital Workers

# Actividad de refuerzo: Docker

## DEVOPS - UF3

# Docker

Veamos cómo utilizar docker-compose con una sencilla aplicación. Utilizaremos uno de los ejemplos de Compose que nos ofrece Docker en su repositorio de GitHub. En concreto, desplegaremos la aplicación nginx-flask-mongo.

Se trata de una aplicación sencilla en Python y Flask con Nginx y MongoDB. En estructura de directorios vemos una carpeta llamada flask que contiene la propia aplicación de Python junto con un Dockerfile para generar su imagen y una carpeta para la configuración del servidor Nginx.

```
.
├── docker-compose.yaml
├── flask
│   ├── Dockerfile
│   ├── requirements.txt
│   └── server.py
└── nginx
    └── nginx.conf
```

En el fichero docker-compose.yaml se definen tres servicios:

- Web. Servidor web Nginx basado en la imagen oficial. Se define un volumen para su fichero de configuración. Al tener una dependencia el servicio no se iniciará hasta que el servicio de backend este corriendo. Además, se publica el puerto 80.
- Backend. Servicio para nuestra aplicación en Python. Su imagen se generará según el Dockerfile del directorio flask. A su vez depende del servicio de mongo para su ejecución.
- Mongo. Simplemente utiliza la imagen oficial de Mongo.

Veamos un extracto del fichero Compose de la aplicación:

```
version: "3.7"
services:
  web:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/tmp/nginx.conf
    ...
    ports:
      - 80:80
    depends_on:
      - backend
  backend:
    build: flask
    ...
    volumes:
      - ./flask:/src
    depends_on:
      -  mongo
  mongo:
    image: mongo
```

Para generar las imágenes de los servicios definidos en Compose, ejecutaremos el comando docker-compose build. Cada vez que hagamos algún cambio en el código podemos ejecutarlo de nuevo para reconstruir las imágenes. Para aquellos servicios que no necesitan generar imágenes, simplemente las ignoran, pero no las descargan:

```
$ docker-compose build
mongo uses an image, skipping
web uses an image, skipping
Building backend
Step 1/5 : FROM python:3.7
Step 2/5 : WORKDIR /src
Step 3/5 : COPY . .
Step 4/5 : RUN pip install -r requirements.txt
Step 5/5 : CMD ["./server.py"]
---> 5da980efd2f1
Successfully built 5da980efd2f1
Successfully tagged python-app_backend:latest
```

Ya podemos desplegar nuestros servicios con docker-compose up. Si aún no tenemos descargadas las imágenes de los repositorios referenciadas en el fichero de Compose, se descargarán antes de iniciar ningún servicio.

```
$ docker-compose up
Creating network "python-app_default" with the default driver
Pulling mongo (mongo:)...
latest: Pulling from library/mongo
Status: Downloaded newer image for mongo:latest
Pulling web (nginx:)...
latest: Pulling from library/nginx
Status: Downloaded newer image for nginx:latest
Creating python-app_mongo_1 ... done
Creating python-app_backend_1 ... done
Creating python-app_web_1     ... done
```

Los contenedores y objetos creados se han nombrado con el prefijo python-app que es el nombre del proyecto, que por defecto usa el del directorio donde se encuentra la aplicación. Ahora, ya podemos listar los contenedores creados, podemos hacerlo con docker ps como hasta ahora, o bien con docker-compose ps:

```
$ docker-compose ps
        Name                      Command        State     Ports
-------------------------------------------------------------------------
-
python-app_web_1       /bin/bash -c 'envsu...    Up
0.0.0.0:80->80/tcp
python-app_backend_1   ./server.py              Up
python-app_mongo_1     docker-entrypoint.sh...   Up    27017/tcp

Por último, podemos probar la aplicación publicada en el puerto 80, y
finalizar los servicios y objetos creados:

$ curl localhost:80
Hello fom the MongoDB client!

$ docker-compose down
Stopping python-app_mongo_1 ... done
Removing python-app_backend_1 ... done
Removing python-app_mongo_1   ... done
Removing network python-app_default
```