Iker Solozabal

PRACTICA 3. PROYECTO COMPLETO

Comenzamos con el siguiente diseño del proyecto:

```
Proyect/
├── db
│   └── init.sql
└── flask
├── app
│   ├── __init__.py
│   └── view.py
└── run.py
```

- ***flask/***: contiene la aplicación Flask que se conecta a la base de datos y expone un *endpoint* de la API REST

- ***init.sql***: un script SQL para inicializar la base de datos antes de que se ejecute la aplicación por primera vez.

Creando una imagen de Docker para nuestra aplicación.

Necesitamos crear un **Dockerfile** en el directorio de la aplicación, este archivo contiene un conjunto de instrucciones que describe como construir nuestra imagen y permite su compilación automática.

```
# Dockerfile-flask
# We simply inherit the Python 3 image. This image does
# not particularly care what OS runs underneath
FROM python:3.11.2
RUN apt -qq -y update \
    && apt -qq -y upgrade
# Set an environment variable with the directory
# where we'll be running the app
ENV APP /app
# Create the directory and instruct Docker to operate
# from there from now on
RUN mkdir $APP
WORKDIR $APP
# Expose the port uWSGI will listen on
# EXPOSE 56734
# Copy the requirements file in order to install
# Python dependencies
COPY requirements.txt .
# Install Python dependencies
RUN pip install --upgrade pip -r requirements.txt
RUN pip install uwsgi
# We copy the rest of the codebase into the image
COPY . .
# Finally, we run uWSGI with the ini file we
# created earlier
CMD [ "uwsgi", "app.ini" ]
# CMD [ "python", "app.py"]
```

Lo que hace esto es:

- La imagen de **Docker** se crea a partir de una imagen existente, *python:3.11*, correspondiente a una imagen ligera de *python3*.

- Se selecciona el puerto de trabajo, en este caso el *8003*, primero asegúrese de disponer de un puerto abierto para usarlo en la configuración. Para verificar si hay un puerto libre, ejecute el siguiente comando:

```
sudo nc localhost 8003< /dev/null; echo $?
```

Si el resultado del comando anterior es 1, el puerto estará libre y podrá utilizarse. De lo contrario, deberá seleccionar un puerto diferente y repetir el procedimiento.

- Se copia el archivo **requirements.txt** al contenedor para que pueda ejecutarse y luego se analiza el archivo **requirements.txt** para instalar las dependencias especificadas en la aplicación. También se copia todo el directorio de trabajo del repositorio dentro de la imagen para posteriormente se comparte como volumen externo.

- Se crea un directorio de trabajo en el que se copia todo el repositorio.

Necesitamos que nuestras dependencias (**Flask** y **mysql-connector**) se instalen y entreguen con la imagen, por lo que debemos crear el archivo **requirements.txt** antes mencionado como:

requirements.txt

```
Flask==2.2.2
mysql-connector
typing
```

Creamos el archivo requiriments.txt en la carpeta Flask/app

```
ikersolozabal@ISG-Linuxmint:~/Desktop/practica3/flask/app$ cat requirements.txt
Flask==2.2.2
mysql-connector
typing
ikersolozabal@ISG-Linuxmint:~/Desktop/practica3/flask/app$
```

En el caso de la imagen de ***MySQL***, no hará falta crear un ***Dockerfile*** puesto que utilizaremos la imagen descargada directamente del ***Docker Hub***.

El archivo ***app.ini*** contendrá las configuraciones de *uWSGI* para nuestra aplicación. *uWSGI* es una opción de implementación para *Nginx,* que es tanto un protocolo como un servidor de aplicaciones. El servidor de aplicaciones puede proporcionar los protocolos *uWSGI, FastCGI y HTTP.*

app.ini

```
[uwsgi]
wsgi-file = run.py
; This is the name of the variable
; in our script that will be called
callable = app
; We use the port 8080 which we will
; then expose on our Dockerfile
socket = :8080
; Set uWSGI to start up 5 workers
processes = 4
threads = 2
master = true
chmod-socket = 660
vacuum = true
die-on-term = true
```

Este código define el módulo desde el que se proporcionará la aplicación de Flask, en este caso es el archivo ***run.py.*** La opción *callable* indica a *uWSGI* que use la instancia de app exportada por la aplicación principal. La opción master permite que su aplicación siga ejecutándose, de modo que haya poco tiempo de inactividad incluso cuando se vuelva a cargar toda la aplicación.

Con las integraciones comentadas, el directorio del proyecto quedaría de la siguiente manera:

```
Proyect/
├── db
│   └── init.sql
├── flask
├── app
│   ├── __init__.py
│   └── view.py
├── .dockerignore
├── app.ini
├── Dockerfile
├── requirements.txt
└── run.py
```

Construcción imagen de Nginx en Docker

Antes de implementar la construcción de la imagen del contenedor Nginx, crearemos nuestro archivo de configuración que le dirá a Nginx cómo enrutar el tráfico a uWSGI en nuestro otro contenedor. El archivo nginx.conf reemplazará el /etc/nginx/conf.d/default.conf que el contenedor Nginx incluye implícitamente.

```
server {
  listen 80;
  location / {
    include uwsgi_params;
    uwsgi_pass flask:8080;
  }
}
```

La línea se ***uwsgi_pass flask:8080*** está utilizando flask como host para enrutar el tráfico. Esto se debe a que configuraremos ***docker-compose*** para conectar nuestros contenedores *Flask* y *Nginx* a través del ***flask*** como nombre de host.

Nuestro Dockerfile para *Nginx* simplemente heredará la última imagen de Nginx del registro de Docker, eliminará el archivo de configuración predeterminado y agregará el archivo de configuración que acabamos de crear durante la compilación.

```
# Dockerfile-nginx
FROM nginx:latest# Nginx will listen on this port
# EXPOSE 80
# Remove the default config file that
# /etc/nginx/nginx.conf includes
RUN rm /etc/nginx/conf.d/default.conf# We copy the
requirements file in order to install
# Python dependencies
COPY nginx.conf /etc/nginx/conf.d/
```

Con ello el directorio del proyecto queda de la siguiente manera:

Creamos el Dockerfile en la carpeta Nginx

```
ikersolozabal@ISG-Linuxmint:~/Desktop/practica3/nginx$ cat Dockerfile
FROM nginx:latest

RUN rm /etc/nginx/conf.d/default.conf

COPY nginx.conf /etc/nginx/conf.d/
```

```
Proyect/
├── db
│   └── init.sql
├── flask
│   ├── app
│   │   ├── __init__.py
│   │   └── view.py
│   ├── .dockerignore
│   ├── app.ini
│   ├── Dockerfile
│   ├── requirements.txt
│   └── run.py
└── nginx
├── Dockerfile
└── nginx.conf
```

Creando el orquestador de contenedores docker-compose.yml

Se crea el archivo ***docker-compose.yml*** en el directorio raíz de nuestro proyecto:

```
version: "3.7"
services:
  flask:
    build: ./flask
    container_name: flask
    restart: always
    environment:
      - APP_NAME=MyFlaskApp
    expose:
      - 8080
```

Estamos utilizando dos servicios, uno es un contenedor que expone la *API (Flask)* y otro contiene la base de datos *MySQL (db).*

- ***build***: especifica el directorio que contiene el *Dockerfile,* el cual contiene las instrucciones para construir este servicio.

- ***environment***: Agregación de variables de entorno. La variable especificada la utilizaremos en la aplicación *Flask* para mostrar un mensaje desde la API.

- ***ports***: asignación de *<Host>: <Container>* puertos.

```
db:
  image: mysql:5.7
  ports:
```

```
    - "32000:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
  volumes:
    - ./db:/docker-entrypoint-initdb.d/:ro
```

- ***image***: En lugar de escribir un nuevo *Dockerfile*, usamos una imagen existente de un repositorio. Es importante especificar la versión, si su cliente *mysql* instalado no es de la misma versión, pueden aparecer problemas.

- ***environment***: Agregación de variables de entorno. La variable especificada es necesaria para esta imagen. Como sugiere su nombre, configura la contraseña para el usuario raíz de *MySQL* en este contenedor. Aquí se especifican más variables.

- ***ports***: Como ya tengo una instancia de mysql en ejecución en mi host usando este puerto, lo estoy mapeando a uno diferente.

- ***volumes***: Dado que queremos que el contenedor se inicialice con nuestro esquema, conectamos el directorio que contiene nuestro script ***init.sql*** al punto de entrada para este contenedor, que según la especificación de la imagen ejecuta todos los scripts ***.sql*** en el directorio dado.

A continuación se muestra el código que se conecta a la base de datos (***app/views.py***):

```
config = {
  'user': 'root',
  'password': 'root',
  'host': 'db',
  'port': '3306',
  'database': 'knights'
}connection = mysql.connector.connect(**config)
```

Se conecta como ***root*** con la contraseña configurada en el archivo ***docker-compose***. Observe que definimos explícitamente el host (que es *localhost* por defecto) ya que el servicio *SQL* está en un contenedor diferente al que ejecuta este código. Podemos (y debemos) usar el nombre '*db*' ya que este es el nombre del servicio que definimos y vinculamos anteriormente. El puerto es *3306* y no *32000* ya que este código no se está ejecutando en el host.

Para el servicio *nginx*, hay algunas cosas a tener en cuenta:

```
nginx:
  build: ./nginx
  container_name: nginx
  restart: always
  ports:
    - "8003:80"
```

Esta pequeña sección le dice a ***docker-compose*** que asigne el puerto *8003* de su máquina local al puerto *80* del contenedor *Nginx* (puerto que *Nginx* sirve por defecto).

Tal y como se ha implementado en el ***nginx.conf***, enrutamos el tráfico de *Nginx* a *uWSGI* y viceversa enviando datos a través del *flask* como nombre de host. Lo que hace esta sección es crear un nombre de host virtual *flask* en nuestro contenedor *nginx* y configurar la red para

que podamos enrutar los datos entrantes a nuestra aplicación *uWSGI* que vive en un contenedor diferente.

Creamos en la carpeta raiz del proyecto el archivo docker-compose.yml

```
ikersolozabal@ISG-Linuxmint:~/Desktop/practica3$ cat docker-compose.yml
version: "3.7"
services:
  flask:
    build: ./flask
    container_name: flask
    restart: always
    environment:
      - APP_NAME=MyFlaskApp
    expose:
      - 8080
  db:
    image: mysql:5.7
    ports:
      - "32000:3306"
    environment:
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - ./db:/docker-entrypoint-initdb.d/:ro
  nginx:
    build: ./nginx
    container_name: nginx
    restart: always
    ports:
      - "8003:80"
ikersolozabal@ISG-Linuxmint:~/Desktop/practica3$ S
```

Desplegar Aplicación

La secuencia de comandos ***start.sh*** es una secuencia de comandos de shell que nos permitirá ejecutar la construcción del ***docker-compose.yml***, para que los contenedores se ejecuten en modo *background*.

```
#!/bin/bash
docker-compose up -d
```

La primera línea se denomina *shebang*. Especifica que este es un archivo bash y se ejecutará como comandos. El indicador ***-d*** se utiliza para iniciar un contenedor en el modo de demonio, o como proceso en segundo plano.

Para probar la creación de las imagen de Docker y los contenedores a partir de las imágenes resultantes, ejecute:

```
sudo bash start.sh
```

Una vez que la secuencia de comandos termine de ejecutarse, utilice el siguiente comando para enumerar todos los contenedores en ejecución:

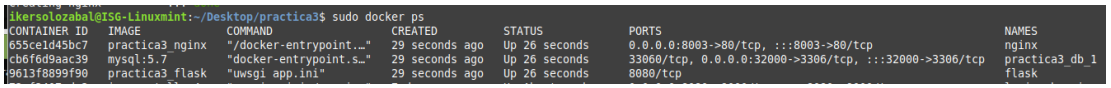
```
sudo docker ps
```

Verá los contenedores en ejecución en ejecución sobre un mismo servicio. Ahora que se está ejecutando, visite la dirección IP pública de su servidor en el puerto especificado de su navegador *http://IP:8003*. o accediendo desde su dominio personal *http://your-domain:8003.*

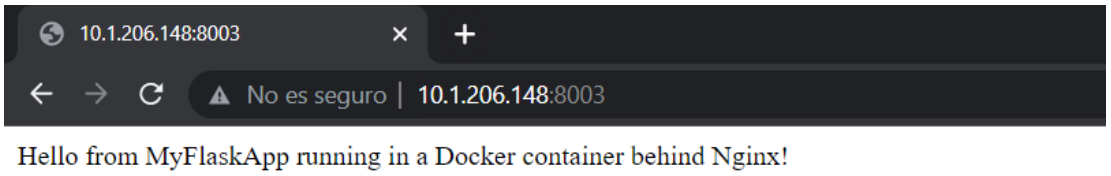
Ahora ya puede visitar su aplicación en *http://your-domain:8003* desde un navegador externo al servidor para ver la la aplicación en ejecución.

Una vez hecho el docker-compose up -d en la carpeta raiz del proyecto

comprobamos los contenedores creados



Y comprobamos en el navegador que funciona correctamente



Actualizar la aplicación

A veces, deberá realizar en la aplicación cambios que pueden incluir instalar nuevos requisitos, actualizar el contenedor de Docker o aplicar modificaciones vinculadas al *HTML* y a la lógica. A lo largo de esta sección, configurará *touch-reload* para realizar estos cambios sin necesidad de reiniciar el contenedor de Docker.

Autoreloading de Python controla el sistema completo de archivos en busca de cambios y actualiza la aplicación cuando detecta uno. No se aconseja el uso de *autoreloading* en producción porque puede llegar a utilizar muchos recursos de forma muy rápida. En este paso, utilizará *touch-reload* para realizar la verificación en busca de cambios en un archivo concreto y volver a cargarlo cuando se actualice o sustituya.

Para implementar esto, abra el archivo ***uwsgi.ini*** e incorpore la siguiente línea de código:

```
touch-reload = /app/uwsgi.ini
```

Esto especifica un archivo que se modificará para activar una recarga completa de la aplicación.

A continuación, si hace una modificación en cualquier *template* y abre la página de inicio de su aplicación en *http:// your-domain:8003* observará que los cambios no se reflejan. Esto se debe a que la condición para volver a cargar es un cambio en el archivo ***uwsgi.ini.*** Para volver a cargar la aplicación, use touch a fin de activar la condición:

```
sudo touch uwsgi.ini
```