

Introducción a Docker

UD 06. Caso práctico

02 - Django con Docker Compose



Autor: Sergi García Barea

Actualizado Febrero 2025

Licencia



Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

Importante

Atención

Interesante

1. Introducción	3
2. Ficheros “Dockerfile” y “requirements.txt” del caso práctico	3
3. Fichero “docker-compose.yml” del caso práctico	3
4. Paso 1: Poniendo en marcha el sistema	4
5. Paso 2: Creando proyecto Django y conectando a la base de datos	5
6. Paso 3: Reiniciando el sistema	5
7. Bibliografía	7

UD06. CASO PRÁCTICO 02

1. INTRODUCCIÓN

En este caso práctico vamos a poner en marcha un servidor web en Python con Django, conectado a una base de datos PostgreSQL. Construiremos la imagen del servidor a partir de un Dockerfile y estableceremos que tenga persistencia tanto el código de la aplicación como el contenido de la base de datos.

2. FICHeros “DOCKERFILE” Y “REQUIREMENTS.TXT” DEL CASO PRÁCTICO

El contenido del fichero “**Dockerfile**” que incluimos comentado, es el siguiente:

```
#Imagen base
FROM python:3.7
#Definimos la variable de entorno PYTHONBUFFERED
ENV PYTHONUNBUFFERED=1
#Establecemos como directorio de trabajo /code
WORKDIR /code
#Copiamos requirements.txt del anfitrión a la imagen
COPY requirements.txt /code/
#Instalamos las dependencias de Python indicadas en requirements
RUN pip install -r requirements.txt
```

Básicamente, este Dockerfile, a partir de la versión 3 de la imagen “**python**”, establece un directorio de trabajo (/code), copia del anfitrión “**requirements.txt**” (que contiene dependencias que deseamos instalar de Python) y las instala usando “pip”.

El contenido del fichero “**requirements.txt**” es el siguiente:

```
Django>=3.0,<4.0
psycopg2-binary>=2.8
```

El resultado de construir esta imagen, será una imagen con Python 3, Django con una versión de la rama 3.X y la biblioteca psycopg2 con una versión superior o igual a la 2.8.

3. FICHERO “DOCKER-COMPOSE.YML” DEL CASO PRÁCTICO

El contenido del fichero “**docker-compose.yml**” que incluimos comentado, es el siguiente:

```
#Versión del fichero docker-compose 3.9. No obligatorio desde la versión de docker-compose 1.27.0
version: "3.9"

#Indicamos los servicios
services:
  #Base de datos
  db:
    #Se basa en Postgres
    image: postgres
    #Guarda la persistencia de la base de datos en el directorio
```

```

#./datos/db de donde lancemos Docker Compose
volumes:
- ./datos/db:/var/lib/postgresql/data
#Establece variables de entorno para indicar base de datos, usuario y password
environment:
- POSTGRES_DB=postgres
- POSTGRES_USER=postgres
- POSTGRES_PASSWORD=postgres
#Crea una aplicación web con Django
web:
#Construye la imagen a partir de un Dockerfile del directorio actual
build: .
#Comando por defecto al crear contenedor, lanzar manage.py para que
#lance el servidor web con Django en el puerto 8000
command: python manage.py runserver 0.0.0.0:8000
#Mapea el código del proyecto Django
# dentro de la carpeta ./codigo del anfitrión
volumes:
- ./codigo:/code
#Enlaza puerto 8000 de contenedor con puerto 8000 de anfitrión
ports:
- "8000:8000"
#Este contenedor depende de "db"
depends_on:
- db

```

En este caso concreto, lo que estamos haciendo es:

- Poner en marcha la base de datos.
- Enlazar la persistencia de la base de datos a la carpeta local “./datos/db”, que se creará en el directorio local donde lancemos “**Docker Compose**”.
- Crear una imagen a partir del “**Dockerfile**” de nuestro directorio actual y una vez creada:
 - Si se cumple la dependencia con “**db**”, lanzar un contenedor con dicha imagen.
 - Establecer como comando de inicio del contenedor el comando para iniciar el servidor web Python con “**Django**”.
 - Enlazar la persistencia del código del servidor a una carpeta “./codigo” que se creará en el directorio local donde lancemos “**Docker Compose**”-
 - Mapee puerto 8000 de contenedor con puerto 8000 del anfitrión.

4. PASO 1: PONIENDO EN MARCHA EL SISTEMA

Previamente a poner en marcha el sistema y de manera opcional, podemos usar el comando

```
docker compose build
```

para que construya la imagen del Dockerfile previamente a lanzar el servicio, obteniendo:

```

sergi@casa:~/Desktop/django$ docker compose build
[+] Building 26.0s (5/9)
=> [internal] load .dockerignore                                0.7s
=> => transferring context: 2B                                  0.0s
=> [internal] load build definition from Dockerfile              0.9s
=> => transferring dockerfile: 384B                              0.0s
=> [internal] load metadata for docker.io/library/python:3      2.3s
=> [auth] library/python:pull token for registry-1.docker.io    0.0s
=> [1/4] FROM docker.io/library/python:3@sha256:0d753a7365274cef746b34dde9b4aaa27644f64e1567ed8f40ccd191ac4bd530 22.6s

```

Sí queremos descargar imágenes ya creadas antes de poner en marcha el sistema, con:

```
docker compose pull
```

Si las descargamos, obtenemos algo similar a:

```
sergi@casa:~/Desktop/django$ docker compose pull
[+] Running 7/15
  # web Skipped - No image to be pulled          0.0s
  # db Pulling                                  18.7s
    # 3f9582a2cbe7 Already exists                0.0s
    # 0d9d08fclala Pull complete                 2.3s
    # ecae4ccb4dlb Pull complete                 2.5s
    # e75693e0d7a5 Pull complete                 2.8s
```

Para poner en marcha el sistema, simplemente nos situamos en el directorio donde tengamos el fichero “**docker-compose.yml**” de este caso práctico y escribimos:

```
docker compose up -d
```

La opción “**-d**” indica que “**Docker Compose**” se ejecute en segundo plano.

La opción “**up**”, descarga y construye imágenes (si no estaban ya). Tras ello lanza los contenedores asociados, siguiendo orden de dependencia.

Si todo ha ido bien, obtendremos un mensaje similar a este:

```
sergi@casa:~/Desktop/django$ docker compose up -d
[+] Running 2/2
  # Container django-db-1   Running
  # Container django-web-1  Started
```

5. PASO 2: CREANDO PROYECTO DJANGO Y CONECTANDO A LA BASE DE DATOS

Si tras el paso anterior, intentamos acceder a <http://localhost:8000>, veremos que no se puede acceder. Eso es porque no se está sirviendo ningún proyecto Django y deberemos crear una base.

Podremos crearla con el siguiente comando

```
docker compose run web django-admin startproject ejemplodjango .
```

En este comando las opciones indicadas son:

- “**run**” indica que ejecutaremos un comando.
- “**web**” indica para qué servicio es el comando.
- “**django-admin startproject ejemplodjango .**” es un comando que crea un proyecto con nombre “**ejemplodjango**” en el directorio actual del contenedor (indicado por “**.**”). Recordemos que el directorio actual es “**/code**”, que fue definido en el “**Dockerfile**”.

Obtendremos algo similar a:

```
sergi@casa:~/Desktop/django$ docker compose run web django-admin startproject ejemplodjango .
[+] Running 1/0
  # Container django-db-1   Running
```

Tras esto aún no veremos nada, comentamos en el punto siguiente el porqué de esto y como solucionarlo.

6. PASO 3: REINICIANDO EL SISTEMA

Si lanzamos el siguiente comando

```
docker compose ps
```

Observamos lo siguiente:

```
sergi@casa:~/Desktop/django$ docker compose ps
NAME                IMAGE              COMMAND                                SERVICE    CREATED        STATUS
django-db-1         postgres          "docker-entrypoint.s..."            db         29 seconds ago Up 25 seconds
5432/tcp
```

Vemos que el servidor de la web no está en ejecución (está detenido). Eso es porque en el momento de lanzarlo no existía el proyecto Django y el comando por defecto del contenedor que habíamos definido en “**docker-compose.yml**” (“**python manage.py runserver 0.0.0.0:8000**”) no podía lanzarse, ya que “**manage.py**” no existía.

Si ahora visitamos nuestro directorio “**/codigo**” del anfitrión, observamos que tenemos mapeado el proyecto Django ahí y podemos modificarlo desde nuestra máquina anfitriona.

Como el usuario del contenedor es “**root**”, todos los ficheros mapeados pertenecen a “**root**”, pero podemos cambiarlos a un usuario local nuestro para facilitar la tarea con un comando similar a:

```
sudo chown -R $USER:$USER ./codigo
```

Al crear el proyecto, por defecto Django utiliza una base de datos **SQLite** en un fichero, por lo cual el servidor de bases de datos que hemos lanzado, no tiene ningún uso.

Antes de relanzar el proyecto, podemos modificar el fichero “**./codigo/emplodjango/settings.py**” y comentar/eliminar la conexión a base de datos **SQLite** e indicar una conexión a la base de datos **PostgreSQL**. Para ello, el siguiente código lo comentaremos o eliminaremos:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    }  
}
```

y en su lugar colocaremos:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'postgres',  
        'USER': 'postgres',  
        'PASSWORD': 'postgres',  
        'HOST': 'db',  
        'PORT': 5432,  
    }  
}
```

Los datos de este fichero, deben coincidir con los indicados como variables de entorno que hemos colocado en “**docker-compose.yml**”.

Tras este cambio, paramos el sistema y lo arrancamos de nuevo con

```
docker compose down; docker compose up -d
```

Podemos comprobar, que al existir ya el proyecto **Django**, el contenedor no está parado:

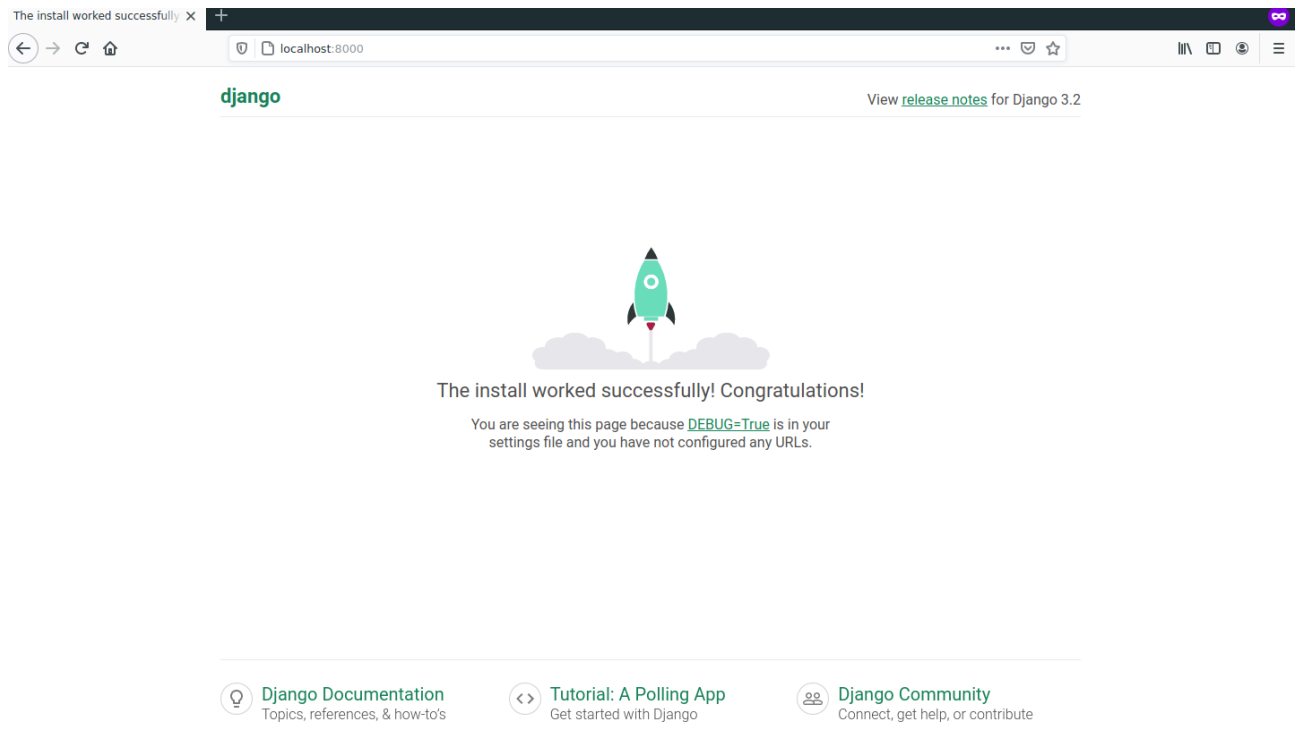
```
docker compose ps
```

Y observamos que así es con algo similar a

```
sergi@casa:~/Desktop/django$ docker compose ps
```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS
django-db-1	postgres	"docker-entrypoint.s..."	db	19 seconds ago	Up 16 seconds
django-web-1	django-web	"python manage.py ru..."	web	19 seconds ago	Up 15 seconds

Tras ello, accedemos a <http://localhost:8000> y observamos que todo funciona adecuadamente:



Finalmente, recordar que la persistencia de la base de datos la tenemos enlazada a nuestro directorio “**./datos/bd**” y el código que podemos manipular dentro de “**./codigo**”.

7. BIBLIOGRAFÍA

- [1] Docker Docs <https://docs.docker.com/>
- [2] Docker Compose Docs <https://docs.docker.com/compose/>