Nombre: Micael Acosta Dóniz

Módulo 2. Backend con Python

Tarea 3: Desarrollo de una API completa con Django

Objetivos de la tarea

La tarea consistía básicamente en trasladar lo que habíamos hecho en la tarea 1 a Django. Para ello he tenido en cuenta también las correcciones que tuve en dicha tarea, que básicamente fueron:

- Introducir más validación de entradas.
- Usar un fichero **.env** para almacenar las credenciales.
- Simplificación del uso de la API de Spotify.
- Añadir referencias visuales en la documentación.
- Añadir un fichero **requirements.txt** con las dependencias del proyecto.

CRUD de Usuarios:

En primer lugar, y para simplificar muchísimo toda la gestión del CRUD, he usado un **viewset** para esta parte.

En el fichero **models.py** he definido los tres objetos básicos que voy a necesitar: el *usuario*, el *artista* y la *canción*.

Cada usuario tendrá un nombre (*username*), una colección de artistas favoritos (*favorite_artists*) y una colección de canciones favoritas (*favorite_songs*).

```python
# Definición del modelo de usuario
class User(models.Model):
    username = models.CharField(max_length=150, unique=True)
    favorite_artists = models.ManyToManyField('Artist', related_name='users', blank=True)
    favorite_songs = models.ManyToManyField('Song', related_name='users', blank=True)

    def __str__(self):
        return self.username
```

Los artistas tendrán solamente un nombre (*name*)

```python
# Definición de los artistas favoritos del usuario
class Artist(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return self.name
```

Y las canciones tendrán un título (*name*) y una FK al artista que la interpreta.

```python
# Definición de las canciones favoritas del usuario
class Song(models.Model):
    name = models.CharField(max_length=255)
    artist = models.ForeignKey(Artist, on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.artist.name} - {self.name}"
```

En el fichero **serializer.py** he definido la forma de visualizarse los datos y algunas validaciones.

Para los **artistas** he decidido mostrar id y nombre, y comprobar que el nombre no esté vacío.

Para las **canciones** he decidido mostrar el id del artista, el nombre del artista y el nombre de la canción. Me pareció necesario añadir el nombre del artista, y no solo su id, ya que podría darse el caso de que un usuario tenga una canción favorita de un artista, pero ese artista no sea uno de sus artistas favoritos. En ese caso, si no mostráramos el nombre del artista junto al título de la canción, solo tendríamos un id que no nos daría ninguna información, ya que no tendríamos al artista en la lista de artistas favoritos del usuario, como en este ejemplo:


```
[
    {
        "id": 1,
        "username": "Michel",
        "favorite_artists": [
            {
                "id": 1,
                "name": "Jamiroquai"
            }
        ],
        "favorite_songs": [
            {
                "id": 1,
                "name": "Virtual Insanity",
                "artist_id": 1,
                "artist_name": "Jamiroquai"
            },
            {
                "id": 2,
                "name": "My name is",
                "artist_id": 3,
                "artist_name": "Eminem"
            }
        ]
    }
]
```

Para las canciones he decidido comprobar que tanto el nombre del artista como el de la canción estén informados.

Para los **usuarios** he decidido que se muestre su nombre y sus colecciones de artistas y canciones. La comprobación que he añadido es que el nombre de usuario no pueda estar vacío.

Para la gestión de los artistas y canciones de cada usuario he necesitado dos métodos, que he añadido en un fichero **services.py**. Estos dos métodos se encargan de añadir un artista al usuario y de añadir una canción al usuario.

Ambos siguen la misma estructura básica: reciben un id de usuario y un nombre de artista/canción, obtienen el usuario de la base de datos y posteriormente tratan de obtener un objeto ya existente o lo crean, mediante la función **get_or_create**. En el caso de la canción, primero se obtiene o crea el artista y posteriormente se hace lo mismo con la canción.

```
song, created = Song.objects.get_or_create(
    name=song_name.strip(),
    artist=artist
)
```

Finalmente, el objeto obtenido se añade a la colección correspondiente del usuario.

Me parecen muy interesantes estas funciones de Python que asignan múltiples variables, ya que es algo que no había utilizado anteriormente en los lenguajes que estoy acostumbrado a utilizar. Realmente es muy cómodo.

En el fichero **urls.py** he introducido la estructura básica que vimos en la clase número seis.

Finalmente, en el fichero **views.py** he definido el **UserViewSet** básico, con 4 acciones que nos dan los endpoints necesarios para realizar las operaciones básicas que he querido añadir:

- *Añadir un artista al usuario*
- *Añadir una canción al usuario*
- *Eliminar un artista del usuario*
- *Eliminar una canción del usuario*

Conexión con la API de Spotify

Esta parte ya la teníamos resuelta en la práctica uno, así que me preocupé básicamente de dos cambios fundamentales:

1. Traducir la estructura del código al formato de Django
2. Reemplazar mi gestión de datos con MySQL por un modelo de Django

Para ello, en el fichero **models.py** definí una clase para el Token de Spotify, que contiene únicamente el propio token y su fecha de caducidad

```python
from django.db import models

# Modelo para almacenar un token de Spotify y una fecha de expiración del mismo
class SpotifyToken(models.Model):
    access_token = models.CharField(max_length=255)
    expires_at = models.DateTimeField()

    def __str__(self):
        return f"Token expira en {self.expires_at}"
```

Toda la lógica de comunicación con la API de Spotify y con la base de datos me la llevé al fichero **services.py**

Aquí tengo métodos para obtener un token de la base de datos (siempre habrá solamente uno), guardar un token a la base de datos (primero elimina todo lo que hay y luego inserta el token) y obtener un nuevo token desde la API.

En este punto estoy usando las variables **SPOTIFY_CLIENT_ID** y **SPOTIFY_CLIENT_SECRET**, que he definido en el fichero **api_server/settings.py** y que se rellenan desde el **.env**

```python
from pathlib import Path
from dotenv import load_dotenv
import os

load_dotenv() # Cargar las variables de entorno desde el archivo .env

SPOTIFY_CLIENT_ID = os.getenv('SPOTIFY_CLIENT_ID')
SPOTIFY_CLIENT_SECRET = os.getenv('SPOTIFY_CLIENT_SECRET')
```

Además de estas funciones relacionadas con el token de Spotify, me he traído mi función *spotify_get*, por la que pasan las llamadas *search_artists* y *search_songs*, que también están definidas en este mismo fichero.

En el archivo **urls.py** he definido las tres rutas que vamos a usar en la parte de Spotify, que son *token*, *artists* y *songs*.

Y por último, en **views.py** he definido los tres GET que vamos a necesitar, para obtener el token, los resultados de la búsqueda de artistas y los resultados de la búsqueda de canciones.

Conclusiones y observaciones

Quedé muy sorprendido con la cantidad de trabajo que nos ahorran los viewsets.

Por otra parte, como comenté anteriormente, estoy descubriendo funciones de Python que nos facilitan muchísimo el trabajo, como por ejemplo get_or_create.

También he podido llevar a la práctica el uso de los ficheros .env y requirements.txt.

Esta práctica ha sido una muy buena toma de contacto con Django.