Tarea API Spotify (Flask)

Módulo 2: Fundamentos de Backend con Python

Alumno: Iñigo Uribarri Parada

1. Introducción y objetivos del trabajo

Este documento describe de forma ordenada y entendible el funcionamiento de una pequeña API desarrollada en Flask que se integra con el servicio de Spotify. El objetivo es explicar cómo está organizado el Proyecto desarrollado, qué tecnologías se utilizan, cómo se gestionan los datos de usuario y cómo se puede probar la API mediante Swagger/OpenAPI.

2. Arquitectura del proyecto

El proyecto sigue una estructura de carpetas sencilla pero clara, separando la lógica principal de la aplicación, la configuración, los servicios externos, los repositorios de datos y los esquemas de validación.

2.1 Estructura de carpetas

Los archivos y directorios principales son:

- main.py: punto de entrada; crea la aplicación Flask mediante la función create_app y la arranca en modo debug.
- app/__init__.py: factoría de la app; carga la configuración, inicializa Flask-Smorest, registra los blueprints y define rutas básicas como "/" y "/health".
- app/config.py: contiene la configuración general y las credenciales de Spotify, que se leen de las variables de entorno SPOTIFY_CLIENT_ID y SPOTIFY_CLIENT_SECRET (con valores de demostración por defecto).
- app/extensions.py: instancia la Api de Flask-Smorest y otros componentes compartidos.
- app/api/: carpeta que agrupa los diferentes blueprints de la API.
- app/api/spotify_api.py: endpoints relacionados con la búsqueda y el detalle de canciones, álbumes y artistas en Spotify.
- app/api/users.py: endpoints de gestión de usuarios (CRUD) y detalle de favoritos.
- app/services/spotify_service.py: servicio que actúa como cliente HTTP hacia Spotify usando el flujo Client Credentials.
- app/repositories/user_repository.py: capa de acceso a datos que lee y escribe en data/users.json (base de datos simulada).
- app/schemas/: esquemas Marshmallow para validar entradas y estructurar las respuestas.
- data/users.json: archivo JSON con usuarios de ejemplo y sus listas de favoritos.

2.2 Dependencias principales

Las dependencias clave (definidas en requirements.txt) incluyen:

- Flask: framework principal para construir la API.
- Flask-Smorest: gestión de blueprints, validación y generación automática de documentación OpenAPI.
- Marshmallow: definición de esquemas y validación de datos de entrada y salida.
- Requests: cliente HTTP para consumir la API externa de Spotify.
- python-dotenv: carga de variables de entorno desde archivos .env durante el desarrollo.

3. Integración con la API de Spotify

La API local actúa como intermediaria entre el cliente que la consume y los servicios ofrecidos por Spotify. Para ello se implementa un flujo de autenticación y una serie de llamadas a endpoints públicos de Spotify.

3.1 Autenticación y obtención del token

Para autenticarse frente a Spotify se usa el flujo Client Credentials:

- Se leen las variables de entorno SPOTIFY_CLIENT_ID y SPOTIFY_CLIENT_SECRET.
- Con esos valores se genera un header Authorization de tipo Basic (client_id:client_secret codificado en Base64).
- Se envía una petición a https://accounts.spotify.com/api/token para obtener un token de acceso.
- El token devuelto se cachea hasta que expira, para evitar pedirlo en cada llamada.

En las peticiones posteriores hacia Spotify se añade siempre el header "Authorization: Bearer <token>" para que la plataforma reconozca al cliente.

3.2 Endpoints externos consumidos

Los principales endpoints que se utilizan de la API pública de Spotify son:

- GET /v1/search: búsqueda de recursos con parámetros q (texto), type (track, album, artist), limit y market.
- GET /v1/tracks/{id}: detalle de una canción concreta.
- GET /v1/albums/{id}: detalle de un álbum concreto.
- GET /v1/artists/{id}: detalle de un artista concreto.

De estos endpoints se extraen datos relevantes como nombre, artistas, álbum, fecha de publicación, número de pistas, géneros, seguidores, enlaces externos y, en el caso de las canciones, la duración y la preview_url si está disponible.

3.3 Gestión de secretos y entorno local

Las credenciales de Spotify no se guardan en el código fuente, sino que se esperan en variables de entorno. En un entorno local de desarrollo se pueden definir, por ejemplo, con:

- set SPOTIFY_CLIENT_ID=...
- set SPOTIFY_CLIENT_SECRET=...

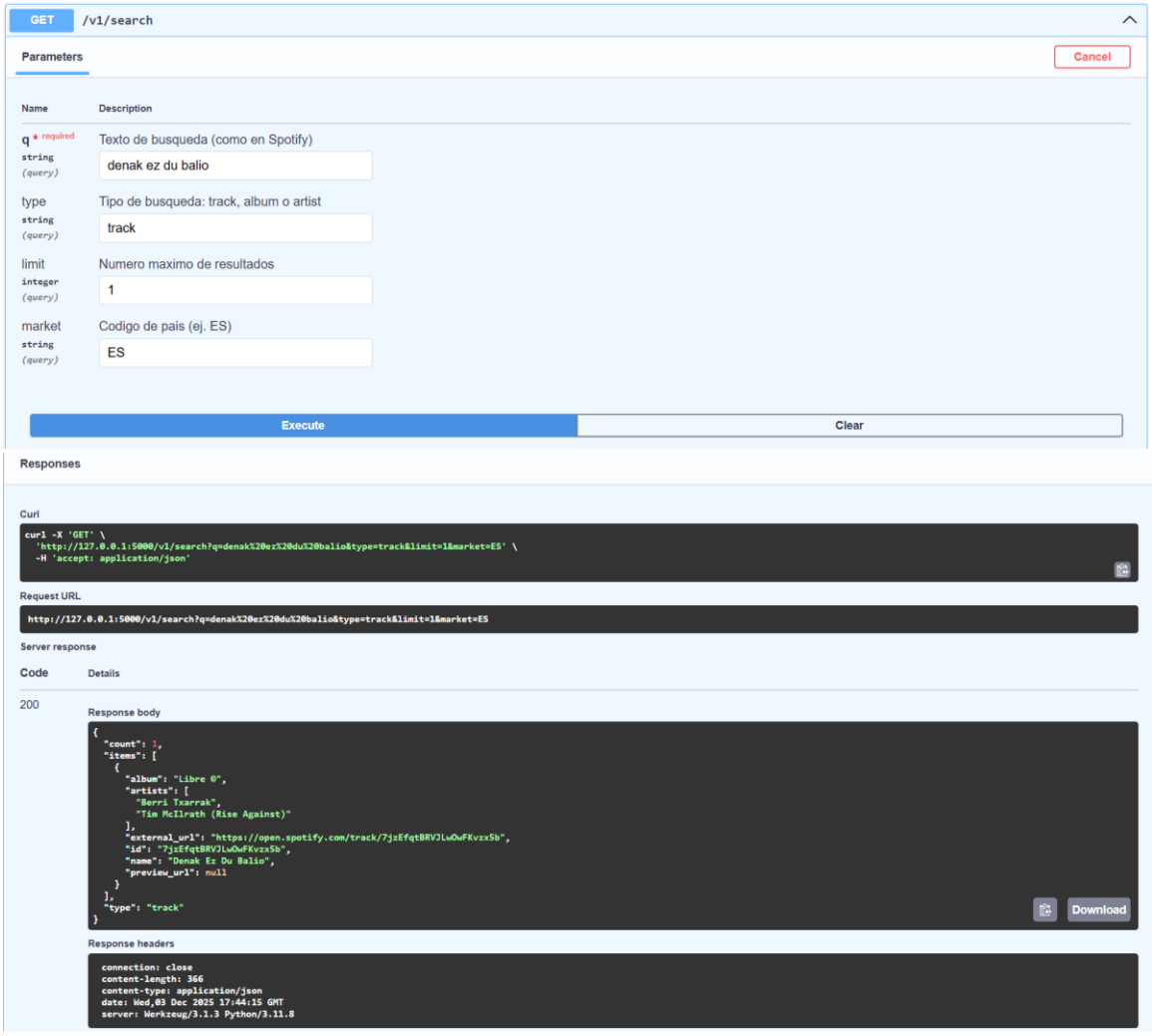
Si no se establecen estas variables, la aplicación puede utilizar valores de demostración, aunque no se garantiza que permitan acceder correctamente a la API de Spotify.

4. Exposición de la información API (Flask + Flask-Smorest)

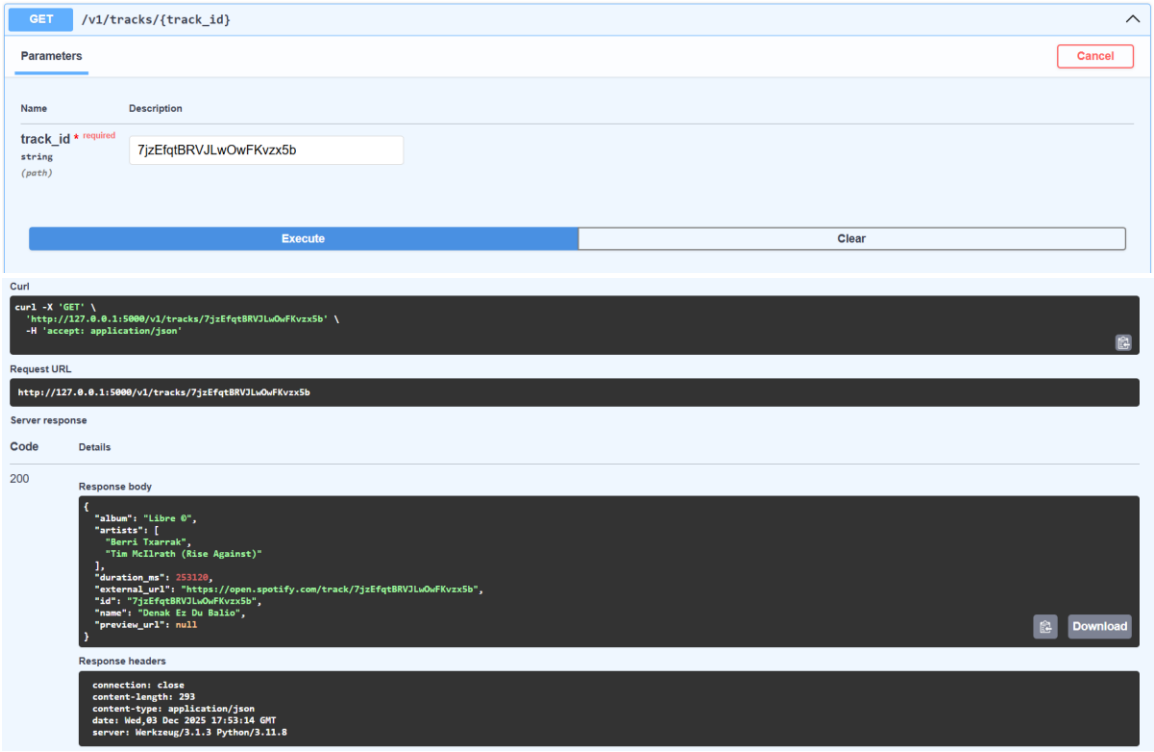
La capa de API se organiza mediante blueprints, lo que permite separar claramente las rutas relacionadas con Spotify de las rutas relacionadas con usuarios. Flask-Smorest simplifica la declaración de los endpoints, la validación y la documentación automática.

4.1 Blueprint de Spotify (app/api/spotify_api.py)

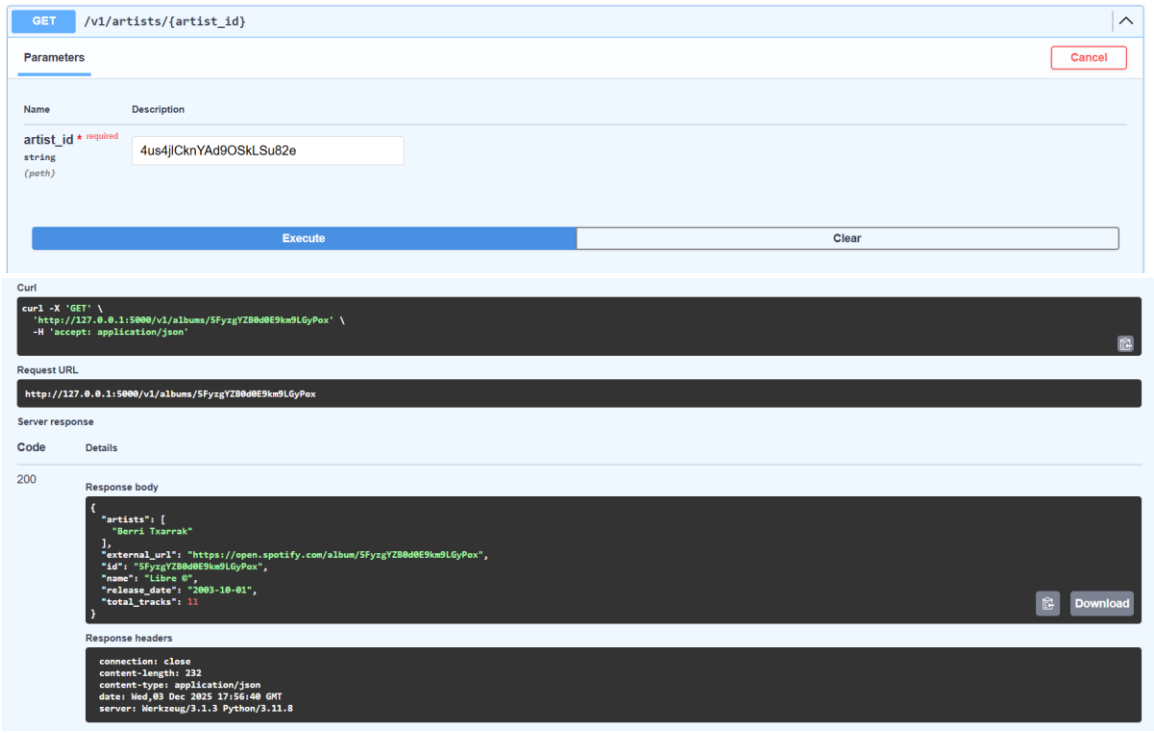
- Endpoints principales:
- GET /v1/search: recibe los parámetros de búsqueda, llama al servicio de Spotify y devuelve la lista de items.



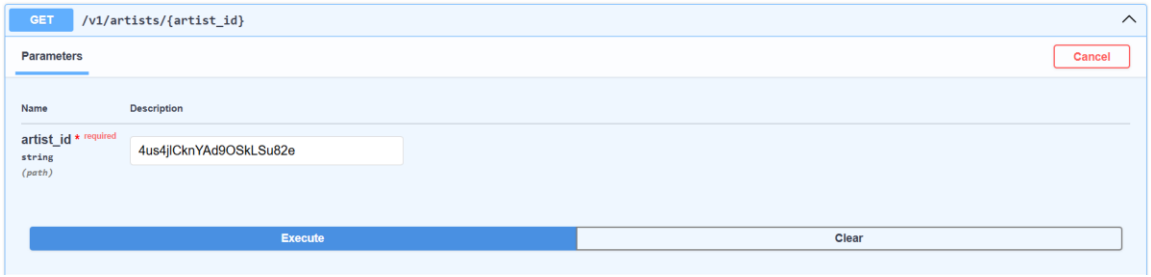
- GET /v1/tracks/<id>: devuelve el detalle de una canción concreta.

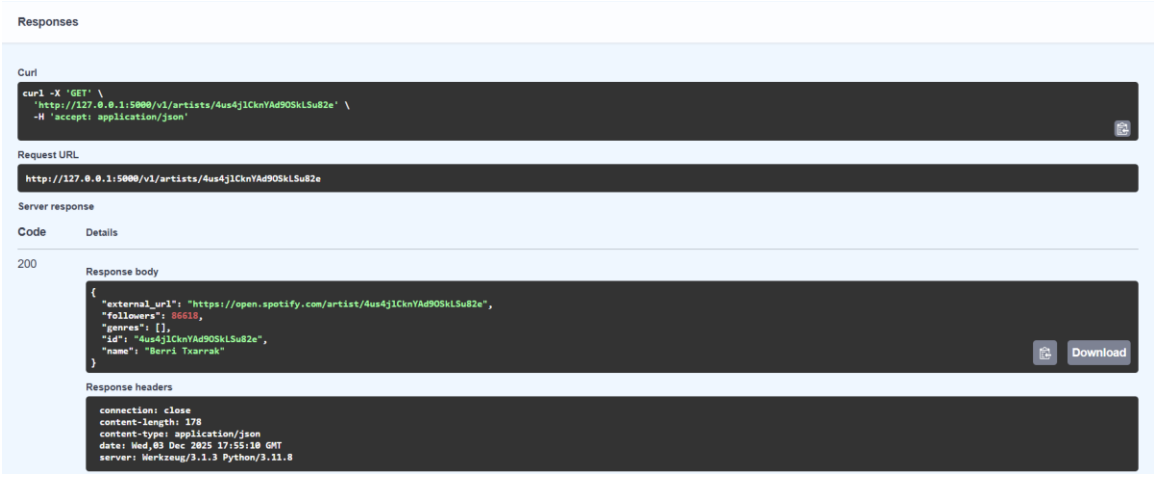


- GET /v1/albums/<id>: devuelve el detalle de un álbum concreto.



- GET /v1/artists/<id>: devuelve el detalle de un artista concreto.





4.2 Blueprint de Usuarios (app/api/users.py)

- Endpoints principales:
- GET /v1/users: devuelve la lista de usuarios obtenida del archivo JSON.
- POST /v1/users: crea un nuevo usuario con name, email y, opcionalmente, listas de favoritos.
- GET /v1/users/<id>: devuelve el detalle de un usuario concreto.
- PATCH /v1/users/<id>: permite la actualización parcial de los datos del usuario.
- DELETE /v1/users/<id>: elimina un usuario por su identificador.
- GET /v1/users/<id>/favorites/details: toma los IDs guardados en el usuario y consulta a Spotify para devolver metadatos de canciones y artistas favoritos.

4.3 Validación de usuarios

En los endpoints se sigue un flujo común:

- Se validan los datos de entrada usando esquemas Marshmallow (por ejemplo, formato de email y tipos de campos).
- Se delega la obtención de datos externos al servicio SpotifyService.
- Se delega la gestión de usuarios y favoritos al user_repository.
- Se devuelven las respuestas ya estructuradas con los esquemas definidos, lo que facilita también la documentación y las pruebas.

5. Gestión de datos de usuarios y almacenamiento de preferencias

En lugar de una base de datos real, el proyecto utiliza un archivo JSON sencillo para almacenar la información de los usuarios. Esta aproximación es suficiente para una demo o para un entorno de aprendizaje.

5.1 Archivo JSON como "base de datos"

- Características principales de data/users.json:
- No existe una base de datos relacional; todos los datos se almacenan en un único archivo JSON.
- Si el archivo no existe, el repositorio puede devolver una lista vacía y crear el archivo cuando se guarden cambios.
- Se incluyen ya dos usuarios de ejemplo para facilitar las primeras pruebas.

5.2 Repositorio de usuarios (user_repository.py)

- El repositorio se encarga de:
- Leer el archivo JSON y convertirlo a estructuras de datos de Python.
- Guardar los cambios en el archivo con indentación para que sea legible.
- Generar identificadores únicos para nuevos usuarios usando UUID.
- Trabajar con los campos soportados: name, email, favorite_tracks (IDs de canciones) y favorite_artists (IDs de artistas).

5.3 Favoritos y consulta de detalles

El endpoint de favoritos detallados toma las listas de IDs almacenadas en el usuario y las utiliza para hacer llamadas a Spotify. De esta forma, los datos persistidos localmente son mínimos (solo los identificadores), y la información actualizada se obtiene siempre desde la API externa.

6. Documentación y pruebas con Swagger / OpenAPI

La integración con Flask-Smorest permite generar automáticamente un esquema OpenAPI y una interfaz web de documentación tipo Swagger. Esto es muy útil para probar los endpoints sin necesidad de usar herramientas externas. Los puntos clave son los siguientes:

- El esquema OpenAPI se publica en /openapi.json.
- La interfaz de Swagger UI está disponible en /docs.
- Los blueprints y los esquemas Marshmallow alimentan automáticamente la documentación (paths, parámetros, cuerpos de petición y respuestas).
- Existen rutas útiles adicionales, como "/": landing con enlaces a /docs, /openapi.json, /health y /v1/users, y "/health" para verificar el estado básico de la aplicación.

7. Pasos para ejecutar el proyecto en local

Para poner en marcha la API en un entorno de desarrollo se recomienda seguir estos pasos:

- Exportar las credenciales de Spotify: set SPOTIFY_CLIENT_ID=... y set SPOTIFY_CLIENT_SECRET=... (o usar los valores de demostración).
- Instalar las dependencias del proyecto: pip install -r requirements.txt.
- Lanzar la aplicación con: python main.py.
- Probar la API desde el navegador en: http://localhost:5000/docs (Swagger) y http://localhost:5000/v1/users.

8. Limitaciones, riesgos y posibles mejoras

Aunque la API es totalmente válida como ejercicio académico y como demostración, presenta algunas limitaciones importantes que conviene tener en cuenta.

- Limitaciones y riesgos actuales:
- El uso de un archivo JSON como "base de datos" simplifica el arranque, pero no escala y puede tener problemas de concurrencia.
- La gestión de credenciales y tokens puede resultar confusa para principiantes (variables de entorno, expiración del token, etc.).
- Dependencia fuerte de la API externa de Spotify: si cambian los endpoints, los scopes o las condiciones de uso, el cliente puede dejar de funcionar.
- La validación de datos es básica; por ejemplo, se podría evitar la creación de usuarios con emails duplicados o listas de favoritos incoherentes.
- Posibles mejoras futuras:
- Sustituir el archivo JSON por una base de datos real (por ejemplo, SQLite o PostgreSQL) y añadir un sistema de migraciones.
- Añadir pruebas automáticas (tests unitarios y de integración) para asegurar el comportamiento de los endpoints.
- Implementar caché para ciertas respuestas de Spotify y así reducir el número de llamadas externas.
- Mejorar el manejo de errores y los logs, para entender mejor qué ocurre cuando algo falla.
- Incorporar autenticación y autorización para los propios usuarios de la API (por ejemplo, JWT u otro sistema sencillo).

9. Conclusión

Esta API demuestra cómo un proyecto relativamente pequeño puede integrar varias piezas habituales en el desarrollo web moderno: un framework ligero como Flask, una API externa potente como Spotify, validación de datos con Marshmallow, documentación automática con OpenAPI y un almacenamiento mínimo para usuarios y favoritos.

Aunque está pensada como ejercicio de aprendizaje para un desarrollador aficionado, la estructura adoptada (servicios, repositorios, blueprints y esquemas) sienta una base

ordenada sobre la que sería posible construir una versión más robusta y cercana a un entorno profesional.