

Introducción

En esta tarea se ha rediseñado el código desarrollado en la primera entrega con FastAPI, utilizando ahora Django como framework principal. El objetivo es construir una API RESTful que gestione la comunicación entre cliente y servidor mediante JSON, integrando además la API externa de Spotify para obtener información sobre canciones y artistas.

Se busca garantizar la calidad y confiabilidad de la API mediante la implementación de validación de datos y manejo de errores, asegurando respuestas consistentes frente a entradas incorrectas o excepciones. Esta entrega permite, además, explorar las ventajas de Django en cuanto a organización del código, manejo de base de datos y escalabilidad, ofreciendo un backend más estructurado y mantenable.

Estructura de la aplicación

```
backend/
├── manage.py
├── .env #Variables de entorno. Credenciales API Spotify
├── DataBase #Base de datos
├── api_server/
│   ├── settings.py #Notificamos al servidor las aplicaciones disponibles
│   └── urls.py #Definición de las urls a mano
└── api_users/
    ├── models.py #Definición de la clase Users
    ├── serializer.py #Serializer para la validacion de datos de User
    ├── views.py #Definición del CRUD con ViewSet + filtrado por nombre de usuario
    └── urls.py #Definición de las URLs con viewSet
└── api_spotify/
    ├── spotify_service.py #Funciones para integrar la api de Spotify recuperadas y adaptadas de la Entrega1
    ├── views.py #Definición del las funciones contra la API de spotify (generación de token, obtencion de informacion de artistas y nuevas releases)
    └── urls.py #Definición de las URLs con viewSet
```

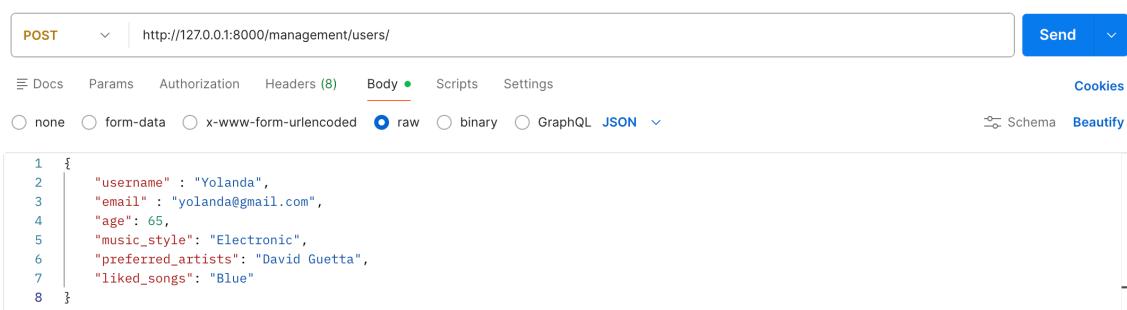
Endpoints de la API:

Gestión de Usuarios

POST Create User

El objetivo de este endpoint es poder añadir usuarios mediante un POST, para ello se deberá proporcionar los siguientes datos: nombre, email, edad*, estilos musicales, artistas y canciones favoritas.

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'pk'
```



The screenshot shows a POST request to `http://127.0.0.1:8000/management/users/`. The request body is a JSON object with the following content:

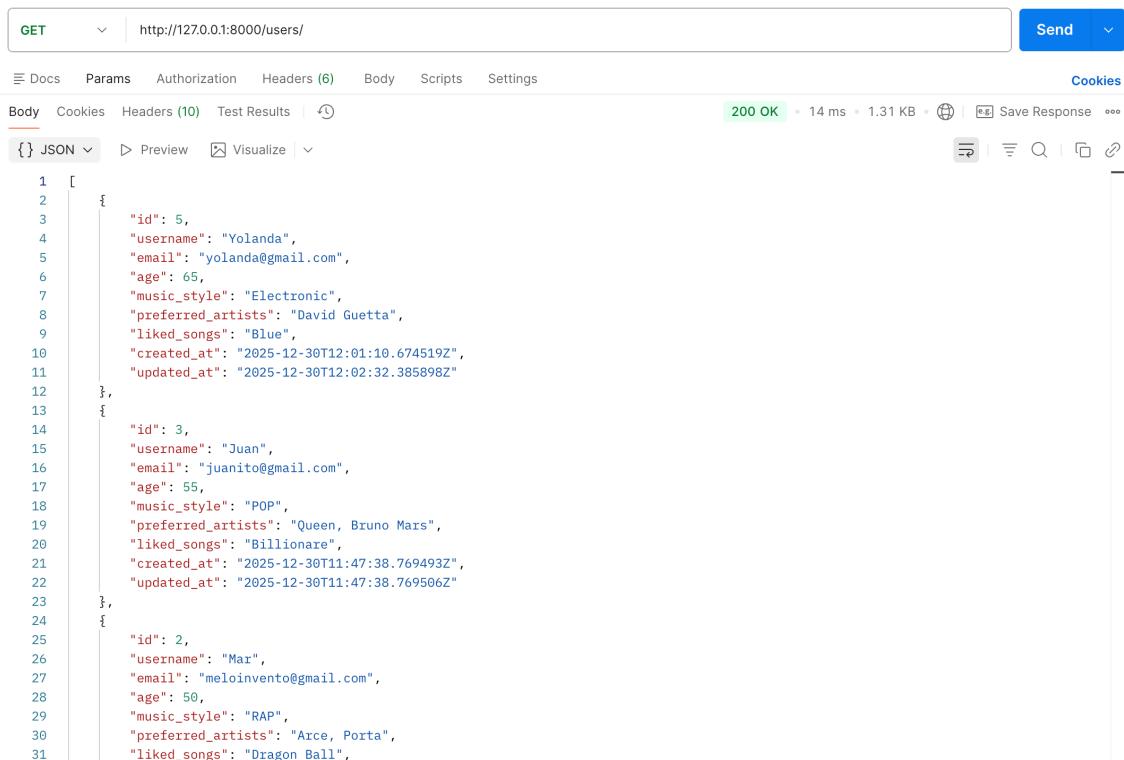
```
1  {
2      "username" : "Yolanda",
3      "email" : "yolanda@gmail.com",
4      "age": 65,
5      "music_style": "Electronic",
6      "preferred_artists": "David Guetta",
7      "liked_songs": "Blue"
8  }
```

*La edad es un dato opcional, puede no proporcionarse.

GET User List

Este endpoint está diseñado para consultar en la base de datos la tabla de usuarios. En esta tabla se mostrará el ID, nombre, email, edad, estilos musicales, artistas y canciones favoritas del usuario. Además, se han añadido dos campos de creación y actualización para poder tener un tracking de los datos.

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'pk'
```



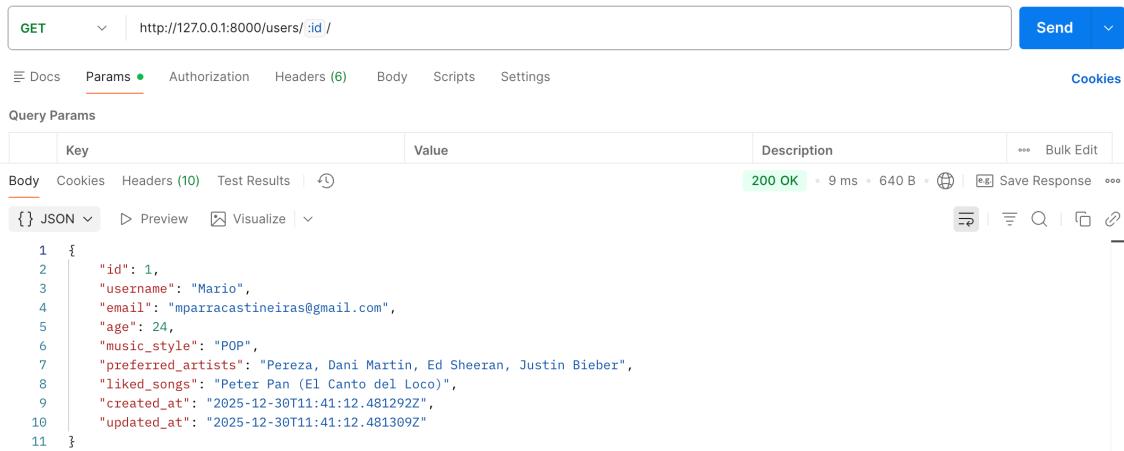
The screenshot shows a Postman request for the endpoint `http://127.0.0.1:8000/users/`. The response is a 200 OK status with 14 ms latency and 1.31 KB size. The response body is a JSON array of three user objects, each with fields: id, username, email, age, music_style, preferred_artists, liked_songs, created_at, and updated_at. The users are: Yolanda (id: 5), Juan (id: 3), and Mar (id: 2).

User	id	username	email	age	music_style	preferred_artists	liked_songs	created_at	updated_at
Yolanda	5	Yolanda	yolanda@gmail.com	65	Electronic	David Guetta	Blue	2025-12-30T12:01:10.674519Z	2025-12-30T12:02:32.385898Z
Juan	3	Juan	juanito@gmail.com	55	POP	Queen, Bruno Mars	Billionare	2025-12-30T11:47:38.769493Z	2025-12-30T11:47:38.769506Z
Mar	2	Mar	meloinvento@gmail.com	50	RAP	Arce, Porta	Dragon Ball		

GET User By ID

Este endpoint está diseñado para consultar en la base de datos la información de un único usuario, filtrando por el ID. En esta tabla se mostrará el ID, nombre, email, edad, estilos musicales, artistas y canciones favoritas del usuario. Además, se han añadido dos campos de creación y actualización para poder tener un tracking de los datos.

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'pk'
```



GET <http://127.0.0.1:8000/users/:id/> Send

Docs Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description	... Bulk Edit
Body	{} JSON	200 OK 9 ms 640 B	Save Response

{} JSON ▶ Preview □ Visualize

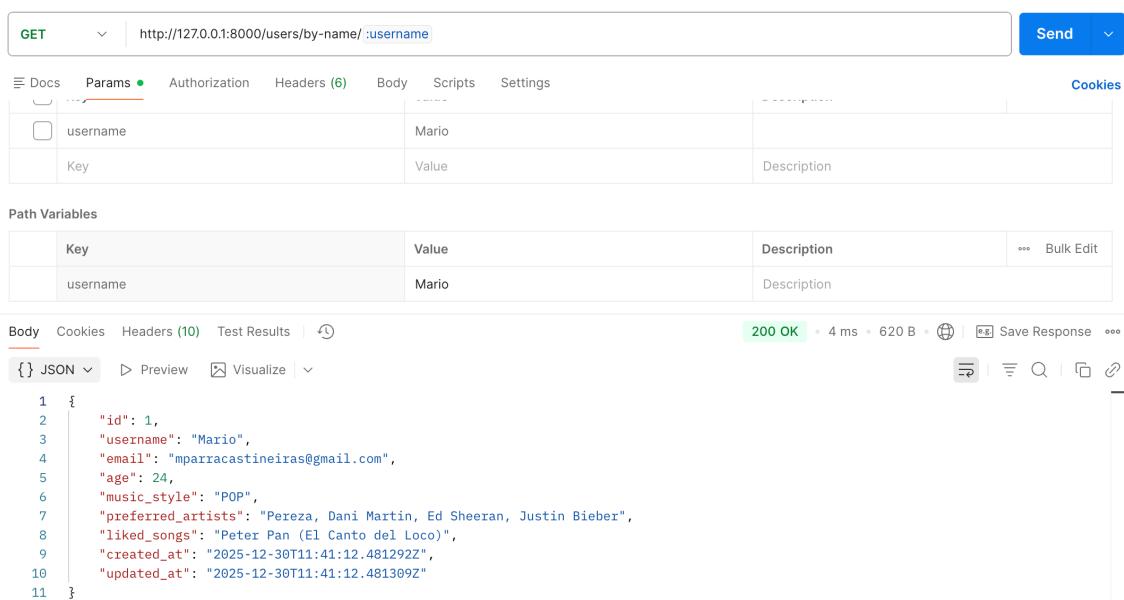
```
1 {
2   "id": 1,
3   "username": "Mario",
4   "email": "mparracastineiras@gmail.com",
5   "age": 24,
6   "music_style": "POP",
7   "preferred_artists": "Pereza, Dani Martin, Ed Sheeran, Justin Bieber",
8   "liked_songs": "Peter Pan (El Canto del Loco)",
9   "created_at": "2025-12-30T11:41:12.481292Z",
10  "updated_at": "2025-12-30T11:41:12.481309Z"
11 }
```

GET User By Username

Este endpoint está diseñado para consultar en la base de datos la información de un único usuario filtrando por Username. En esta tabla se mostrará el ID, nombre, email, edad, estilos musicales, artistas y canciones favoritas del usuario. Además, se han añadido dos campos de creación y actualización para poder tener un tracking de los datos.

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()  # You, 18 hours ago • first commit
    serializer_class = UserSerializer
    lookup_field = 'pk'

    @action(detail=False, methods=['get'], url_path='by-name/(?P<username>[^/.]+)')
    def get_by_username(self, request, username=None):
        try:
            user = User.objects.get(username=username)
            serializer = self.get_serializer(user)
            return Response(serializer.data, status=status.HTTP_200_OK)
        except User.DoesNotExist:
            return Response({'detail': "User not found"}, status=status.HTTP_404_NOT_FOUND)
```



The screenshot shows a REST API testing interface. At the top, a GET request is made to `http://127.0.0.1:8000/users/by-name/username`. The 'Params' tab is selected, showing a parameter `username` with the value `Mario`. The 'Body' tab shows the JSON response:

```
1 {  
2   "id": 1,  
3   "username": "Mario",  
4   "email": "mparracastineiras@gmail.com",  
5   "age": 24,  
6   "music_style": "POP",  
7   "preferred_artists": "Pereza, Dani Martin, Ed Sheeran, Justin Bieber",  
8   "liked_songs": "Peter Pan (El Canto del Loco)",  
9   "created_at": "2025-12-30T11:41:12.481292Z",  
10  "updated_at": "2025-12-30T11:41:12.481309Z"  
11 }
```

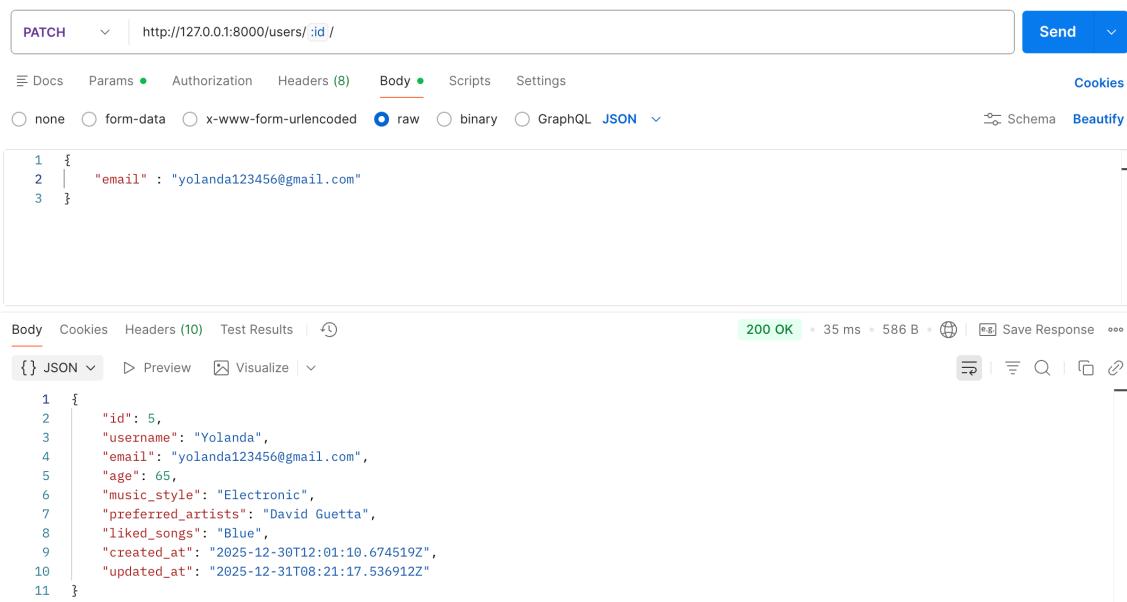
The response status is `200 OK` with a response time of 4 ms and a body size of 620 B. The response body is a JSON object with the following structure:

```
{  
  "id": 1,  
  "username": "Mario",  
  "email": "mparracastineiras@gmail.com",  
  "age": 24,  
  "music_style": "POP",  
  "preferred_artists": "Pereza, Dani Martin, Ed Sheeran, Justin Bieber",  
  "liked_songs": "Peter Pan (El Canto del Loco)",  
  "created_at": "2025-12-30T11:41:12.481292Z",  
  "updated_at": "2025-12-30T11:41:12.481309Z"  
}
```

PATCH Update Partial User Fields

Para este endpoint se busca poder ser capaces de modificar cualquiera de los parámetros de la base de datos de manera independiente (menos aquellos que se han asignado como parámetros de read-only, en este caso ID, created_at y updated_at).

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'pk'
```



The screenshot shows a Postman request for a PATCH update to a user's email. The URL is `http://127.0.0.1:8000/users/:id/`. The request body is a JSON object with a single key-value pair: `"email" : "yolanda123456@gmail.com"`. The response is a 200 OK status with a response time of 35 ms and a response size of 586 B. The response body is a JSON object containing the user's details, including the updated email address.

Body	JSON	Preview	Visualize
1	{		
2	"email" : "yolanda123456@gmail.com"		
3	}		

200 OK	35 ms	586 B	Save Response

PUT Update All User Fields

A diferencia del patch, para poder realizar esta petición se deberán de enviar si o si todos los campos del objeto (teniendo en cuenta que los campos de read-only se siguen manteniendo por lo que no se pueden modificar)

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'pk'
```

The screenshot shows the POSTMAN API client interface. The request method is set to **PUT**, and the URL is `http://127.0.0.1:8000/users/:id/`. The **Body** tab is selected, showing a JSON payload with fields: `"username": "Yolanda", "email": "yolandainvent@gmail.com", "age": 65, "music_style": "Electronic", "preferred_artists": "David Guetta", "liked_songs": "Blue"`. The response status is **200 OK** with a response time of 9 ms and a size of 586 B. The response body is identical to the request body, indicating successful update.

```
1  {
2    "username": "Yolanda",
3    "email": "yolandainvent@gmail.com",
4    "age": 65,
5    "music_style": "Electronic",
6    "preferred_artists": "David Guetta",
7    "liked_songs": "Blue"
8  }
```

```
1  {
2    "id": 5,
3    "username": "Yolanda",
4    "email": "yolandainvent@gmail.com",
5    "age": 65,
6    "music_style": "Electronic",
7    "preferred_artists": "David Guetta",
8    "liked_songs": "Blue",
9    "created_at": "2025-12-30T12:01:10.674519Z",
10   "updated_at": "2025-12-31T08:21:52.981214Z"
11 }
```

DEL Delete Users by ID

Para poder eliminar un usuario deberemos de ser conocedores de su ID y proporcionarlo como un Path Param.

```
class UserViewSet(viewsets.ModelViewSet):  
    queryset = User.objects.all()  
    serializer_class = UserSerializer  
    lookup_field = 'pk'
```

The screenshot shows a REST API testing interface. At the top, a header bar includes a 'DELETE' dropdown, a URL input field containing 'http://127.0.0.1:8000/users/:id /', and a 'Send' button. Below the header are tabs for 'Docs', 'Params' (which is selected and highlighted in green), 'Authorization', 'Headers (6)', 'Body', 'Scripts', and 'Settings'. The 'Params' tab shows a table with one row: 'Key' (id) and 'Value' (5). The 'Path Variables' section also shows a table with one row: 'Key' (id) and 'Value' (5). At the bottom, there are tabs for 'Body', 'Cookies', 'Headers (9)', and 'Test Results'. The 'Test Results' tab is selected and shows a green status bar with '204 No Content', '8 ms', '319 B', and a save response button. Below the status bar are buttons for 'Raw', 'Preview', and 'Visualize'.

Integración de la API de Spotify

POST Spotify Token

Para poder acceder a la API Pública de Spotify es necesario que generarnos un Token el cual permitirá acceder a la información pública (es decir toda aquella que su endpoint no contenga un /me/), esta es la función de este endpoint.

El planteamiento ha sido el siguiente, este endpoint solo será accesible para aquellos usuarios que consten en la base de datos, en caso de que no consten se les devolverá un mensaje conforme deben registrarse previamente.

En caso de que el usuario conste en la base de datos se le devolverá el token, el cual se guardara en una base de datos global para poder utilizarlo en los siguientes endpoints.

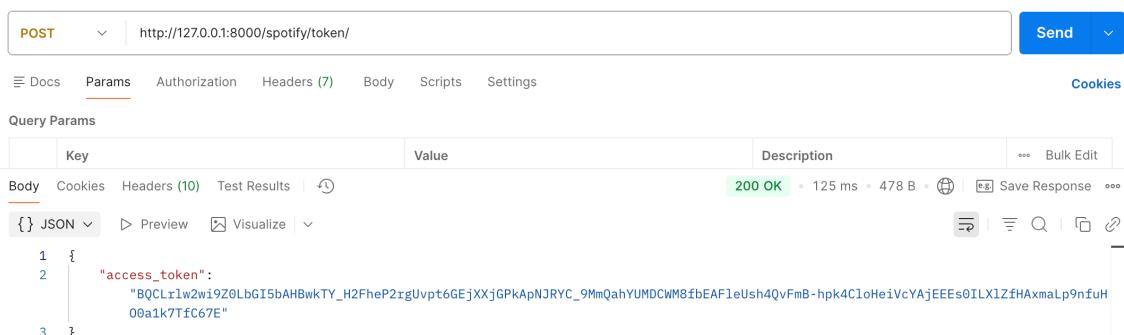
La idea de hacerlo en este endpoint es para evitar que usuarios no registrados puedan acceder a las peticiones contra la API Pública de Spotify.

```
class SpotifyViewSet(viewsets.ViewSet):
    @action(detail=False, methods=['post'], url_path='token')
    def token(self, request):
        global SPOTIFY_ACCESS_TOKEN # Usamos la variable global
        username = request.data.get("username")

        if not username:
            return Response(
                {"detail": "You must provide a username"}, status=status.HTTP_400_BAD_REQUEST)

        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            return Response({"detail": "User not registered"}, status=status.HTTP_404_NOT_FOUND)

        try:
            token = get_token(
                settings.SPOTIFY_CLIENT_ID,
                settings.SPOTIFY_CLIENT_SECRET,
            )
            # Guardamos el token globalmente
            SPOTIFY_ACCESS_TOKEN = token
        except Exception as e:
            print(f"Error obteniendo el token: {e}")
            return Response({"detail": "Token could not be generated"}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
        return Response({"access_token": token})
```



The screenshot shows a POST request to `http://127.0.0.1:8000/spotify/token/`. The response is a `200 OK` with a response time of 125 ms and a content length of 478 B. The response body is a JSON object with a single key `access_token` containing a long string of characters.

Key	Value	Description	... Bulk Edit
<code>access_token</code>	<code>"B0CLrlw2wi9Z0LbGI5bAHBwkTY_H2FheP2rgUvpt6GEjXXjGPkApNjRYC_9MmQahYUMDCWM8fbEAFlleUsh4QvFmB-hpk4C1oHeiVcYAjEEE0ILX1ZfHAxmaLp9nfuH00a1k7Tfc67E"</code>		

GET Información de artistas

Mediante este endpoint se podrá consultar la información del artista escogido, para ello se deberá de tener un token en vigor y el ID del artista a consultar. Este ID se pasará mediante un Path Param.

Si por algún motivo estuviera caducado o no existiera el token se devolverá un error al usuario para que obtenga uno nuevo.

```
@action(detail=False, methods=['get'], url_path='artist/(?P<artist_id>[^.]+)')
def artist(self, request, artist_id=None):
    global SPOTIFY_ACCESS_TOKEN # Usamos la variable global

    # Si el token no está disponible, retornamos un error
    if not SPOTIFY_ACCESS_TOKEN:
        return Response(
            {"detail": "Token is not available"},
            status=status.HTTP_400_BAD_REQUEST,
        )

    try:
        artist = get_artist(SPOTIFY_ACCESS_TOKEN, artist_id)
    except PermissionError:
        return Response(
            {"detail": "Invalid token"},
            status=status.HTTP_401_UNAUTHORIZED,
        )
    except LookupError:
        return Response(
            {"detail": "Artist not found"},
            status=status.HTTP_404_NOT_FOUND,
        )

    return Response(artist)
```

The screenshot shows a REST API testing interface. At the top, a GET request is made to `http://127.0.0.1:8000/spotify/artist/:id`. The 'Params' tab is selected, showing a table with a single row: 'Key' (id) and 'Value' (0TnOYISbd1XYRBk9myaseg). The 'Path Variables' section shows a table with a single row: 'Key' (id) and 'Value' (0TnOYISbd1XYRBk9myaseg). In the 'Body' tab, the response is displayed as JSON:

```
1 {
2     "external_urls": {
3         "spotify": "https://open.spotify.com/artist/0TnOYISbd1XYRBk9myaseg"
4     },
5     "followers": {
6         "href": null,
7         "total": 12201855
8     },
9     "genres": [],
10    "href": "https://api.spotify.com/v1/artists/0TnOYISbd1XYRBk9myaseg",
11    "id": "0TnOYISbd1XYRBk9myaseg",
12    "images": [
13        {
14            "url": "https://i.scdn.co/image/ab6761610000e5eb8d8ac7290d0fe2d12fb6e4d9",
15            "height": 640,
16            "width": 640
17        },
18    ]
```

The response status is 200 OK with 131 ms latency and 969 B size. The response body is identical to the JSON shown in the 'Body' tab.

GET Información de lanzamientos

El objetivo de este endpoint es obtener información sobre los últimos lanzamientos.

```
    @action(detail=False, methods=['get'], url_path='releases')
def releases(self, request):
    global SPOTIFY_ACCESS_TOKEN # Usamos la variable global

    # Si el token no está disponible, retornamos un error
    if not SPOTIFY_ACCESS_TOKEN:
        return Response(
            {"detail": "Token is not available"},
            status=status.HTTP_400_BAD_REQUEST,
        )

    try:
        releases = get_new_releases(SPOTIFY_ACCESS_TOKEN)
    except PermissionError:
        return Response(
            {"detail": "Invalid token"},
            status=status.HTTP_401_UNAUTHORIZED,
        )

    return Response(releases)
```

GET http://127.0.0.1:8000/spotify/releases

Send

Docs Params Authorization Headers (6) Body Scripts Settings

Body Cookies Headers (10) Test Results

Cookie

{} JSON ▾ Preview Visualize ▾

200 OK 388 ms 35.7 KB

Save Response

```
1 {  
2   "albums": {  
3     "href": "https://api.spotify.com/v1/browse/new-releases?offset=0&limit=20",  
4     "items": [  
5       {  
6         "album_type": "album",  
7         "artists": [  
8           {  
9             "external_urls": {  
10               "spotify": "https://open.spotify.com/artist/6nVcHLIgY5pE2YC18ubca1"  
11             },  
12             "href": "https://api.spotify.com/v1/artists/6nVcHLIgY5pE2YC18ubca1",  
13             "id": "6nVcHLIgY5pE2YC18ubca1",  
14             "name": "JHAYCO",  
15             "type": "artist",  
16             "uri": "spotify:artist:6nVcHLIgY5pE2YC18ubca1"  
17           },  
18         ],  
19         "available_markets": [  
20           "AR",  
21           "AU",  
22           "AT",  
23           "BE",  
24           "BO",  
25           "BR",  
26           "BG",  
27           "CA",  
28           "CL",  
29           "CO",  
30           "CR",  
31           "CY",  
32         ]  
33       }  
34     ]  
35   }  
36 }
```

Conclusión

En este proyecto se ha llevado a cabo la implementación del backend utilizando Django, tras haber trabajado previamente con FastAPI, lo que ha permitido realizar una comparación directa entre ambos enfoques.

A diferencia de FastAPI, la implementación con Django resulta inicialmente más compleja, principalmente debido a su estructura más robusta y a la cantidad de componentes que deben configurarse. Sin embargo, esta complejidad se ve compensada por una mejor organización del código, ya que la separación en modelos, vistas, serializers y rutas favorece un desarrollo más ordenado, mantenable y escalable. Esta fragmentación transmite una clara sensación de mayor control sobre el comportamiento del backend, especialmente en proyectos de mayor tamaño o con lógica de negocio más compleja.

El uso de ViewSets y del Django REST Framework supone una ventaja significativa, ya que facilita enormemente la implementación de operaciones básicas como el CRUD. Gracias a ello, el desarrollador puede centrarse en la lógica de la aplicación sin necesidad de definir manualmente todas las rutas, reduciendo el código repetitivo y el riesgo de errores.

Otro aspecto especialmente destacable es la gestión de la base de datos. Django simplifica notablemente este proceso mediante su ORM (Object-Relational Mapping), permitiendo definir los modelos de forma clara y declarativa. Con tan solo crear los modelos y especificar los campos necesarios, es posible generar y gestionar la estructura de la base de datos de manera eficiente, sin necesidad de escribir consultas SQL complejas. Además, el sistema de migraciones facilita la evolución del esquema de datos a lo largo del tiempo.

En conclusión, aunque Django presenta una curva de aprendizaje algo mayor en comparación con FastAPI, ofrece un ecosistema más completo, herramientas integradas y una arquitectura sólida que lo convierten en una excelente opción para proyectos que requieren escalabilidad, mantenibilidad y una gestión integral del backend.