

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 mantenible.

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 información de artistas y nuevas releases)
    └── urls.py #Definición de las URLs con viewSet
```

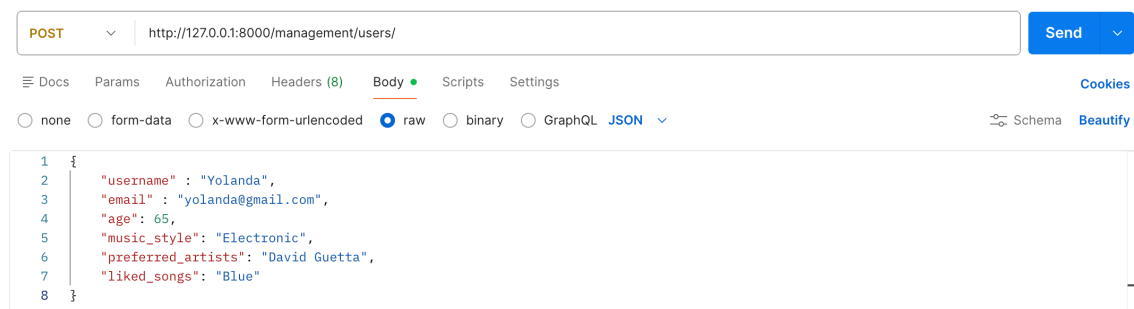
Endpoints de la API:

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'
```

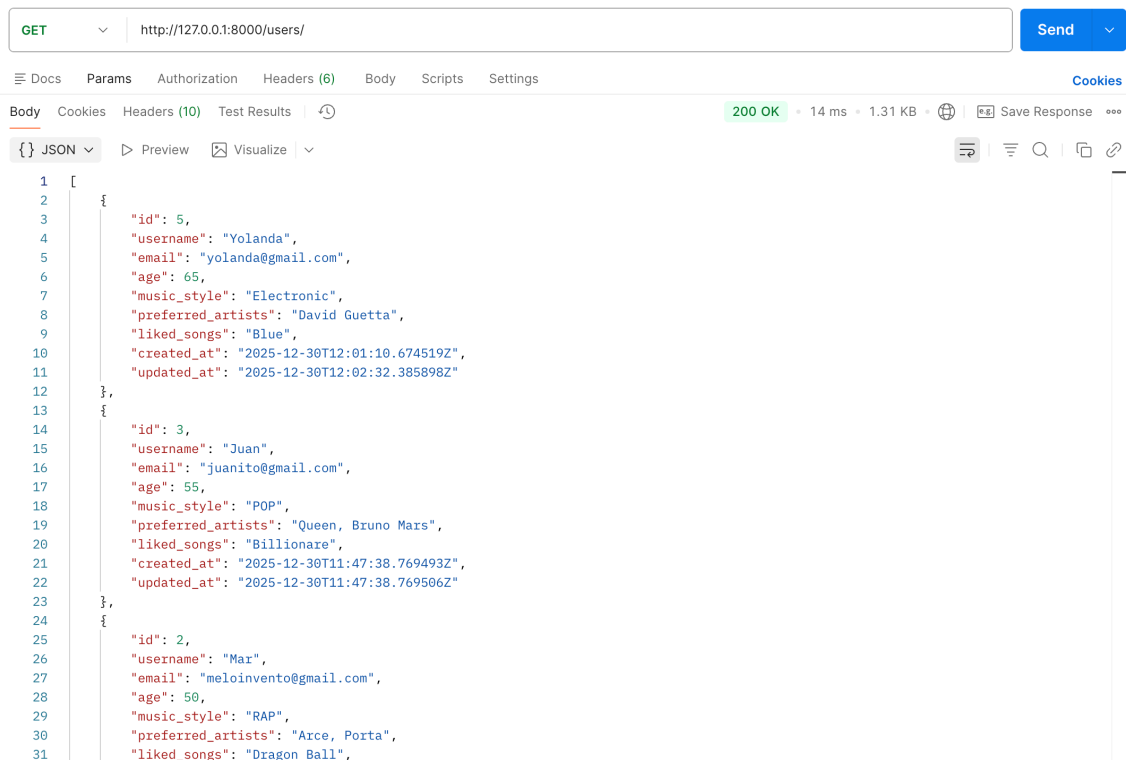


*La edad es un dato opcional, puede no proporcionarse.

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 REST client interface with the following details:

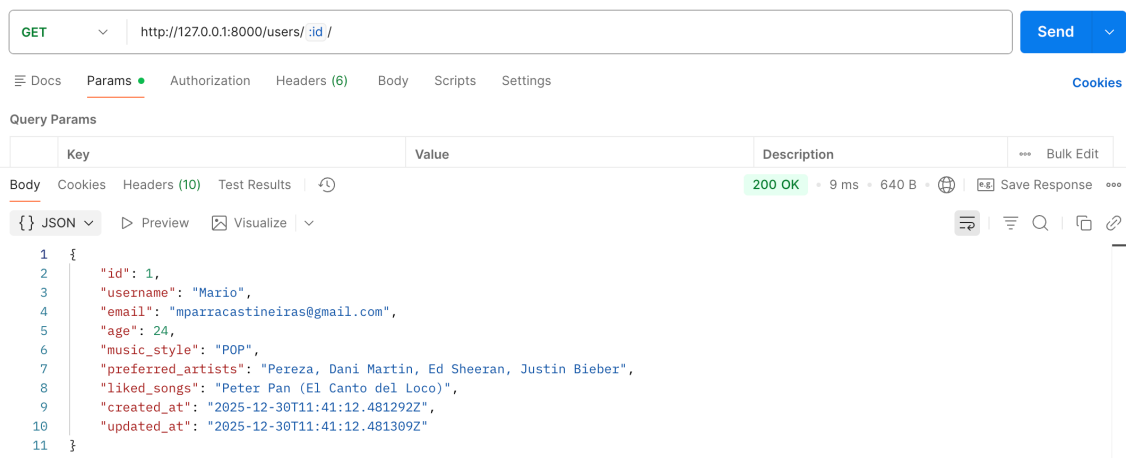
- Method:** GET
- URL:** http://127.0.0.1:8000/users/
- Status:** 200 OK
- Response Time:** 14 ms
- Response Size:** 1.31 KB
- Response Type:** JSON
- Response Body:**

```
[  
  {  
    "id": 5,  
    "username": "Yolanda",  
    "email": "yolanda@gmail.com",  
    "age": 65,  
    "music_style": "Electronic",  
    "preferred_artists": "David Guetta",  
    "liked_songs": "Blue",  
    "created_at": "2025-12-30T12:01:10.674519Z",  
    "updated_at": "2025-12-30T12:02:32.385898Z"  
  },  
  {  
    "id": 3,  
    "username": "Juan",  
    "email": "juanito@gmail.com",  
    "age": 55,  
    "music_style": "POP",  
    "preferred_artists": "Queen, Bruno Mars",  
    "liked_songs": "Billionaire",  
    "created_at": "2025-12-30T11:47:38.769493Z",  
    "updated_at": "2025-12-30T11:47:38.769506Z"  
  },  
  {  
    "id": 2,  
    "username": "Mar",  
    "email": "meloinvento@gmail.com",  
    "age": 50,  
    "music_style": "RAP",  
    "preferred_artists": "Arce, Porta",  
    "liked_songs": "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 Cookies Headers (10) Test Results 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()
    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 fund"}, status=status.HTTP_404_NOT_FOUND)
```

GET Send

Docs Params Authorization Headers (6) Body Scripts Settings Cookies

Key	Value	Description
username	Mario	

Path Variables

Key	Value	Description
username	Mario	

Body Cookies Headers (10) Test Results 200 OK 4 ms 620 B Save Response

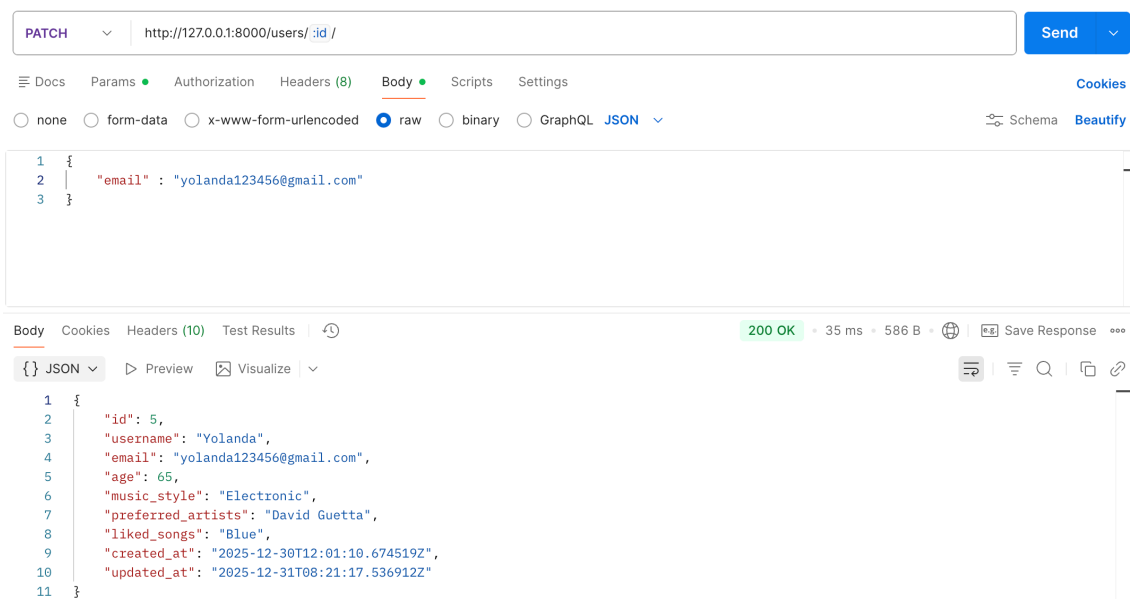
☐ JSON ☐ Preview ☐ Visualize

```
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 }
```

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 REST client interface with a PATCH request to `http://127.0.0.1:8000/users/:id/`. The request body is a JSON object: `{ "email": "yolanda123456@gmail.com" }`. The response is a 200 OK status with a response time of 35 ms and a body size of 586 B. The response body is a JSON object containing user details: `{ "id": 5, "username": "Yolanda", "email": "yolanda123456@gmail.com", "age": 65, "music_style": "Electronic", "preferred_artists": "David Guetta", "liked_songs": "Blue", "created_at": "2025-12-30T12:01:10.674519Z", "updated_at": "2025-12-31T08:21:17.536912Z" }`.

Request:

```
PATCH http://127.0.0.1:8000/users/:id/
```

Body:

```
{  
  "email": "yolanda123456@gmail.com"  
}
```

Response: 200 OK (35 ms, 586 B)

```
{  
  "id": 5,  
  "username": "Yolanda",  
  "email": "yolanda123456@gmail.com",  
  "age": 65,  
  "music_style": "Electronic",  
  "preferred_artists": "David Guetta",  
  "liked_songs": "Blue",  
  "created_at": "2025-12-30T12:01:10.674519Z",  
  "updated_at": "2025-12-31T08:21:17.536912Z"  
}
```

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 a REST client interface with a PUT request to `http://127.0.0.1:8000/users/5/`. The request body is a JSON object with the following fields: `username`, `email`, `age`, `music_style`, `preferred_artists`, and `liked_songs`. The response is a 200 OK status with a JSON object containing the same fields plus `id`, `created_at`, and `updated_at`.

PUT `http://127.0.0.1:8000/users/5/` Send

Docs Params Authorization Headers (8) Body Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Schema Beautify

```
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 }
```

Body Cookies Headers (10) Test Results 200 OK 9 ms 586 B Save Response

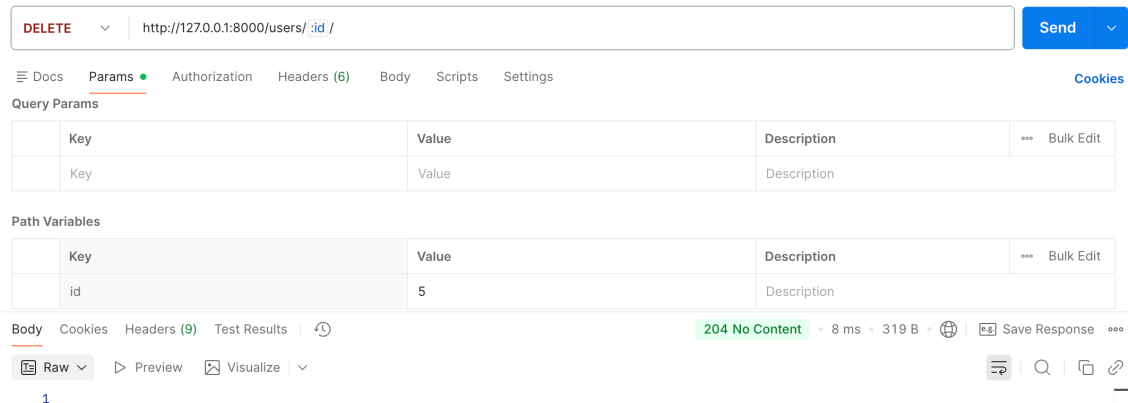
{ } JSON Preview Visualize

```
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'
```



DELETE 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
Key	Value	Description	

Path Variables

Key	Value	Description	Bulk Edit
id	5	Description	

Body Cookies Headers (9) Test Results 204 No Content 8 ms 319 B Save Response

Raw Preview Visualize

1

Integración de la API de Spotify

POST Spotify Token

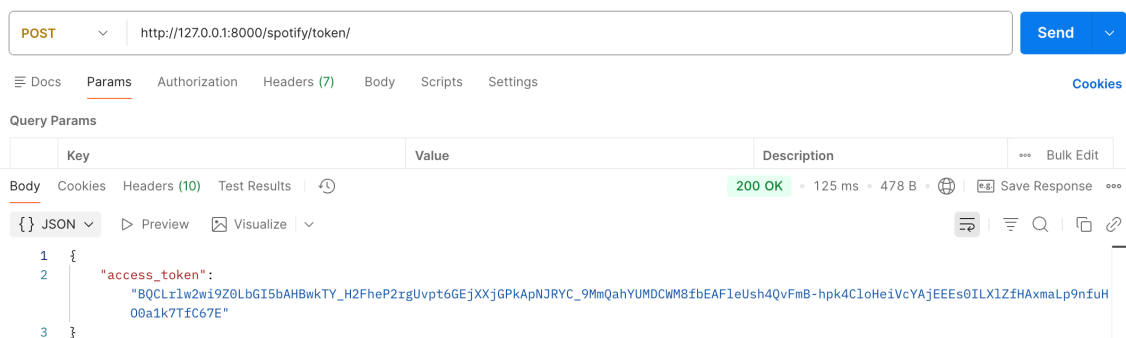
Para poder acceder a la API Publica de Spotify es necesario que generarnos un Token el cual permitirá acceder a la información pública (es decir toda aquella que su endpoint no contenga un /me/), esta es la función de este endpoint.

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 Publica 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})
```



POST http://127.0.0.1:8000/spotify/token/ Send

Docs Params Authorization Headers (7) Body Scripts Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
-----	-------	-------------	-----------

Body Cookies Headers (10) Test Results 200 OK 125 ms 478 B Save Response

{ } JSON Preview Visualize

```
1 {  
2   "access_token":  
3     "BQCli1w2wi9Z0LbGI5bAHBwKTY_H2FheP2rgUvpt6GEjXXjGPKApNJRYC_9MmQahYUMDCW8fbEAF1eUsh4QvFmB-hpk4C1oHeiVcYAjEEEs0ILXIZfHAXmaLp9nFuH00a1k7TfC67E"
```

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)
```

GET Send

Docs Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Path Variables

Key	Value	Description	Bulk Edit
id	0TnOYISbd1XYRBk9myaseg	Description	

Body Cookies Headers (10) Test Results 200 OK 131 ms 969 B Save Response

{} JSON Preview Visualize

```
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  ]
19 }
```

GET Información de lanzamientos

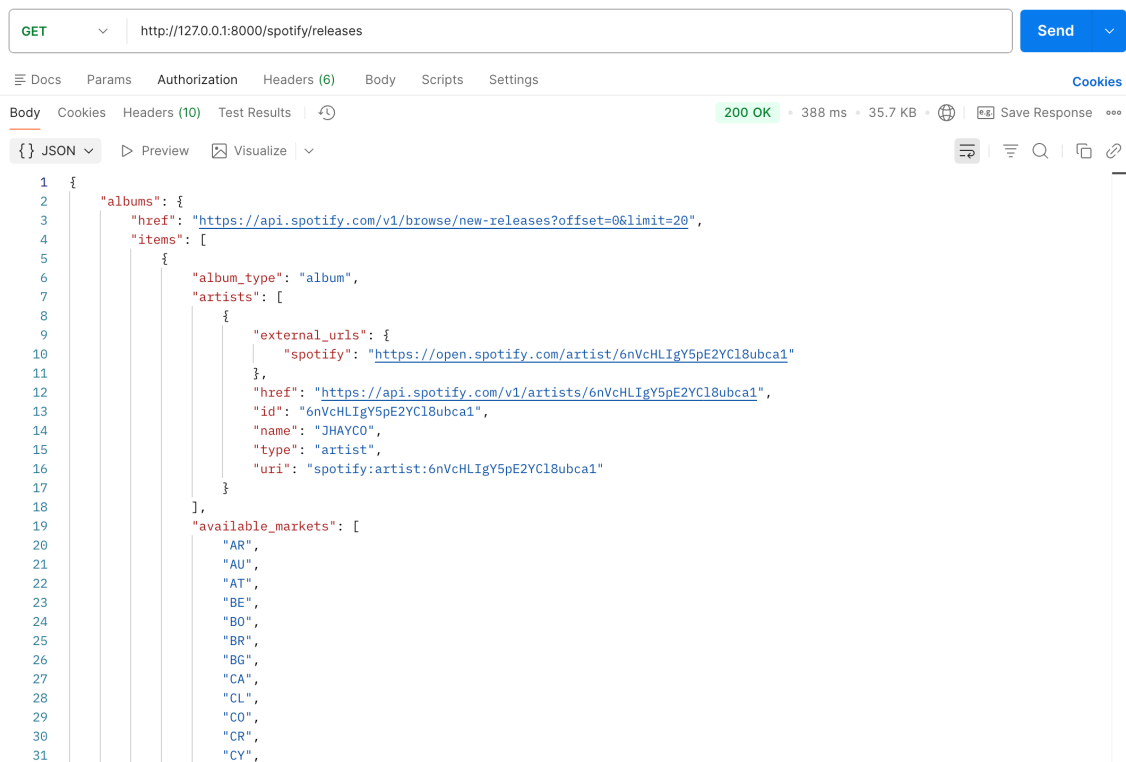
El objetivo de este endpoint es obtener información sobre los últimos lanzamientos.

```
@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)
```



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://127.0.0.1:8000/spotify/releases
- Status:** 200 OK
- Time:** 388 ms
- Size:** 35.7 KB
- Body:** JSON

The JSON response is as follows:

```
{
  "albums": {
    "href": "https://api.spotify.com/v1/browse/new-releases?offset=0&limit=20",
    "items": [
      {
        "album_type": "album",
        "artists": [
          {
            "external_urls": {
              "spotify": "https://open.spotify.com/artist/6nVcHLIgY5pE2YCl8ubca1"
            },
            "href": "https://api.spotify.com/v1/artists/6nVcHLIgY5pE2YCl8ubca1",
            "id": "6nVcHLIgY5pE2YCl8ubca1",
            "name": "JHAYCO",
            "type": "artist",
            "uri": "spotify:artist:6nVcHLIgY5pE2YCl8ubca1"
          }
        ],
        "available_markets": [
          "AR",
          "AU",
          "AT",
          "BE",
          "BO",
          "BR",
          "BG",
          "CA",
          "CL",
          "CO",
          "CR",
          "CY",

```

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, mantenible 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.