📚 Guide Complet du Projet MusicAPI

Explication pour Débutants

🎯 PARTIE 1 : C'EST QUOI CE PROJET ?

Le But Simple

Imagine que tu veux créer un **Spotify simplifié**. Tu as besoin de :

1. **Une bibliothèque musicale** : artistes, albums, chansons
2. **Des playlists personnalisées** : chaque utilisateur peut créer ses playlists
3. **Des moyens d'accéder aux données** : des APIs pour que d'autres applications puissent utiliser ta musique

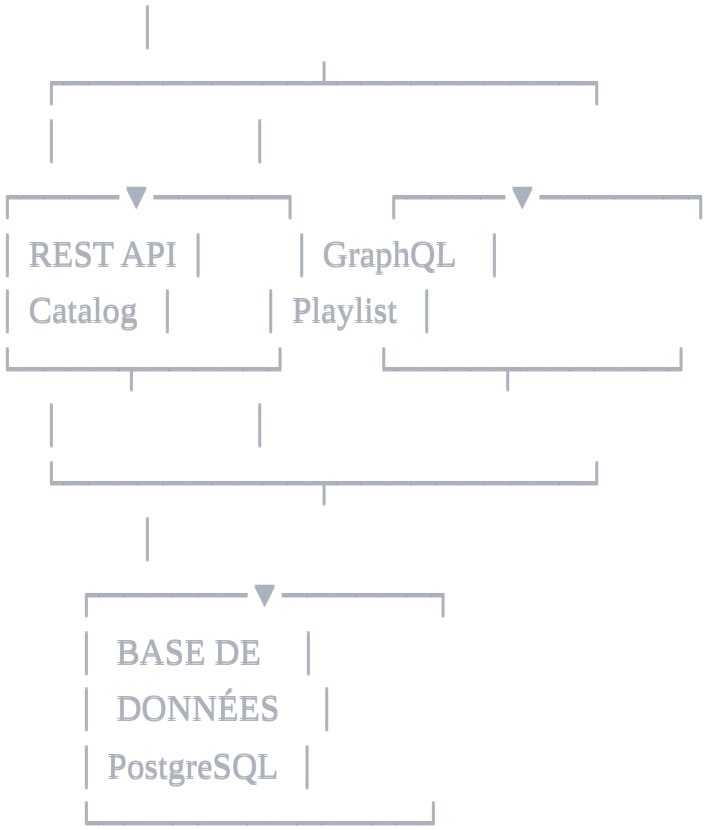
Ce que fait ce projet

C'est une **application web** qui te permet de :

- ✅ Gérer une bibliothèque musicale (catalogue)
- ✅ Créer et gérer des playlists
- ✅ Exposer ces données via 2 types d'API différentes
- ✅ Avoir une interface web pour tout manipuler

🏗️ PARTIE 2 : L'ARCHITECTURE (Comment c'est organisé)

Vue d'ensemble



Les 3 Applications Django

1️⃣ catalog_api (API REST)

- **Rôle** : Gérer les artistes, albums, et chansons
- **Type** : API REST (classique, simple)
- **Exemple** : `GET /api/catalog/artists/` → Liste tous les artistes

2️⃣ playlist_api (API GraphQL)

- **Rôle** : Gérer les utilisateurs, playlists, et leur contenu
- **Type** : API GraphQL (moderne, flexible)
- **Exemple** : Tu demandes exactement ce que tu veux

3️⃣ web_interface

- **Rôle** : Interface web pour utiliser les APIs
- **Type** : Pages HTML avec JavaScript
- **Exemple** : Boutons pour afficher les artistes, créer des playlists

---

🧱 PARTIE 3 : LES CONCEPTS DE BASE

Qu'est-ce qu'une API ?

API = Application Programming Interface (Interface de Programmation)

Analogie du Restaurant :

- Tu es le **client** (application)
- Le **serveur** (API) prend ta commande
- La **cuisine** (base de données) prépare
- Le **serveur** te ramène le plat (données)

REST vs GraphQL

REST (catalog_api)

Principe : Chaque ressource a son URL

```
GET /api/catalog/artists/          → Tous les artistes
GET /api/catalog/artists/5/        → L'artiste #5
GET /api/catalog/artists/5/albums/ → Albums de l'artiste #5
```

Avantage : Simple, standard, facile à comprendre **Inconvénient** : Plusieurs requêtes pour avoir toutes les infos

GraphQL (playlist_api)

Principe : Tu demandes exactement ce que tu veux dans UNE requête

```graphql

```

```
query {
  playlist(id: 5) {
    name
    user { username }
    playlistSongs {
      song { title }
    }
  }
}
```

Avantage : Une seule requête, données précises **Inconvénient** : Plus complexe à mettre en place

---

📦 PARTIE 4 : LA BASE DE DONNÉES

Structure des Données

```
CATALOG (Bibliothèque Musicale)
├── Artist (Artiste)
│   ├── id
│   ├── name (nom)
│   ├── country (pays)
│   └── formed_year (année de création)
│
├── Album
│   ├── id
│   ├── title (titre)
│   ├── artist_id → lien vers Artist
│   ├── release_year (année de sortie)
│   └── genre
│
└── Song (Chanson)
    ├── id
    ├── title
    ├── album_id → lien vers Album
```

```
    ├── artist_id → lien vers Artist
    ├── duration_seconds (durée en secondes)
    └── track_number (numéro de piste)

PLAYLIST (Listes de lecture)
├── User (Utilisateur)
│   ├── id
│   ├── username
│   ├── email
│   └── created_at
│
├── Playlist
│   ├── id
│   ├── name
│   ├── user_id → lien vers User
│   ├── created_at
│   └── is_public (public ou privé)
│
└── PlaylistSong (Chanson dans une playlist)
    ├── id
    ├── playlist_id → lien vers Playlist
    ├── song_id → lien vers Song
    ├── position (ordre dans la playlist)
    └── added_at (date d'ajout)
```

Relations

Exemple concret :

```
Artist "Pink Floyd"
  └── Album "The Wall" (1979)
      ├── Song "Another Brick in the Wall" (track 3)
      └── Song "Comfortably Numb" (track 6)
```

Types de relations :

- **1:n** (un-à-plusieurs) : Un artiste a PLUSIEURS albums

- **1:1** (un-à-un) : Une chanson a UN artiste, UN album

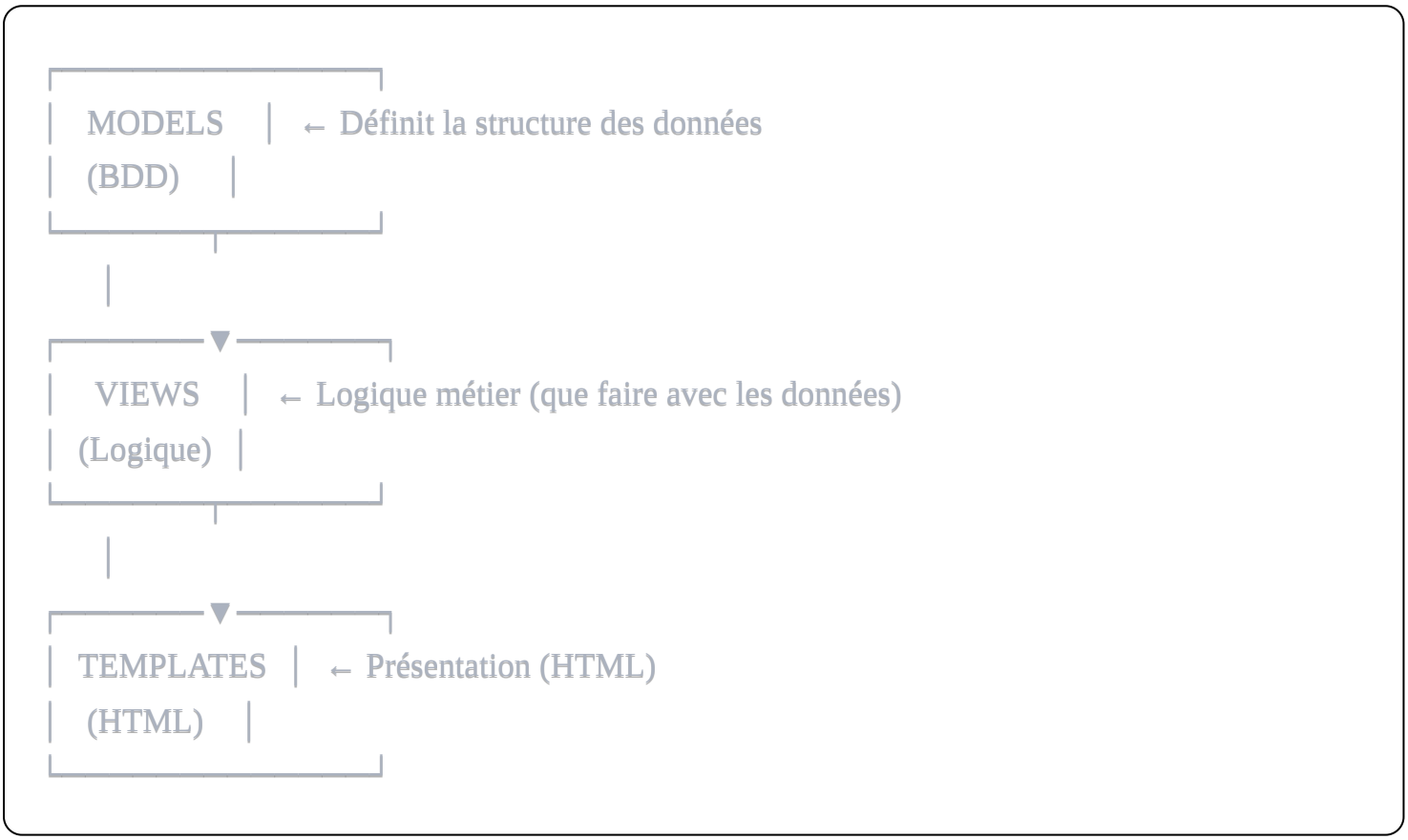
🔧 PARTIE 5 : DJANGO C'EST QUOI ?

Django = Framework Web Python

Un **framework** = une boîte à outils pour créer des sites web facilement

Pattern MVT (Model-View-Template)

Django organise ton code en 3 parties :



📂 PARTIE 6 : STRUCTURE DU PROJET

```
musicapi/
│
├── manage.py                ← Script principal Django
├── requirements.txt         ← Liste des bibliothèques nécessaires
│
├── musicapi_project/        ← Configuration du projet
```

```
│   ├── settings.py          ← Paramètres (BDD, apps installées)
│   ├── urls.py              ← Routes principales
│   └── schema.py            ← Schéma GraphQL principal
│
├── catalog_api/             ← App 1 : API REST
│   ├── models.py            ← Définition Artist, Album, Song
│   ├── serializers.py       ← Conversion objet ↔ JSON
│   ├── views.py             ← Logique des endpoints
│   └── urls.py              ← Routes de l'API
│
├── playlist_api/            ← App 2 : API GraphQL
│   ├── models.py            ← Définition User, Playlist, PlaylistSong
│   ├── types.py             ← Types GraphQL
│   ├── queries.py           ← Requêtes (lecture)
│   ├── mutations.py         ← Mutations (écriture)
│   └── schema.py            ← Schéma GraphQL
│
└── web_interface/           ← App 3 : Interface Web
    ├── views.py             ← Rendu des pages HTML
    ├── urls.py              ← Routes des pages
    └── templates/           ← Fichiers HTML
        ├── base.html
        ├── index.html
        ├── catalog.html
        └── playlists.html
```

💡 PARTIE 7 : EXPLICATION FICHIER PAR FICHIER

🔹 catalog_api/models.py

C'est quoi ? La définition de la structure des données

Exemple simple :

```
python
```

```python
class Artist(models.Model):
    name = models.CharField(max_length=200)  # Texte, max 200 caractères
    country = models.CharField(max_length=100)
    formed_year = models.IntegerField()  # Nombre entier
```

Traduction : "Un Artiste, c'est un objet qui a un nom, un pays, et une année"

Propriétés calculées :

```python
@property
def album_count(self):
    return self.albums.count()  # Compte combien d'albums cet artiste a
```

Pourquoi `managed = False` ?

- La base de données existe DÉJÀ (fournie par le prof)
- Django ne doit PAS créer/modifier les tables
- On se connecte juste à ce qui existe

Pourquoi `db_table = 'artist'` ?

- Dit à Django : "La table s'appelle 'artist' dans la base"

Pourquoi `related_name='albums'` ?

- Permet de faire `artist.albums.all()` pour avoir ses albums
- C'est un raccourci automatique

🔹 **catalog_api/serializers.py**

C'est quoi ? Des traducteurs : Objet Python ↔ JSON

Pourquoi faire ?

- Base de données stocke des objets Python
- APIs envoient du JSON
- Les serializers font la traduction

Exemple :

Objet Python :

```python
artist = Artist(id=1, name="Pink Floyd", country="UK", formed_year=1965)
```

JSON (via serializer) :

```json
{
  "id": 1,
  "name": "Pink Floyd",
  "country": "UK",
  "formed_year": 1965,
  "album_count": 15,
  "song_count": 152
}
```

2 versions pour chaque modèle :

1. **ListSerializer** : Version simple pour les listes

```python
  # Moins de détails, plus rapide
  fields = ['id', 'name', 'country']
```

2. **DetailSerializer** : Version complète pour un seul élément

```python
```

```
    # Tous les détails + propriétés calculées
    fields = ['id', 'name', 'country', 'formed_year', 'album_count', 'song_count']
```

read_only vs write_only :

- `read_only` : On peut lire mais pas modifier (ex: compteurs)
- `write_only` : On peut écrire mais pas lire (ex: IDs pour créer)

Exemple Song :

```python
artist = ArtistListSerializer(read_only=True)  # Affiche l'objet artiste complet
artist_id = serializers.PrimaryKeyRelatedField(  # Pour créer : on donne juste l'ID
    queryset=Artist.objects.all(),
    source='artist',
    write_only=True
)
```

🔹 catalog_api/views.py

C'est quoi ? La logique métier : que faire quand on reçoit une requête

ViewSet = Ensemble de vues

```python
class ArtistViewSet(viewsets.ModelViewSet):
    queryset = Artist.objects.all()  # Tous les artistes
    serializer_class = ArtistSerializer  # Comment les sérialiser
```

ModelViewSet donne automatiquement :

- `GET /artists/` → Liste (list)
- `POST /artists/` → Créer (create)

- `GET /artists/5/` → Détail (retrieve)
- `PUT /artists/5/` → Modifier complet (update)
- `PATCH /artists/5/` → Modifier partiel (partial_update)
- `DELETE /artists/5/` → Supprimer (destroy)

Actions personnalisées :

```python
@action(detail=True, methods=['get'])
def albums(self, request, pk=None):
    """Endpoint : GET /artists/5/albums/"""
    artist = self.get_object()  # Récupère l'artiste #5
    albums = artist.albums.all()  # Tous ses albums
    serializer = AlbumListSerializer(albums, many=True)  # Sérialise
    return Response(serializer.data)  # Retourne JSON
```

Filtres et recherche :

```python
search_fields = ['name', 'country']
# Permet : /artists/?search=pink
# → Cherche "pink" dans name OU country

ordering_fields = ['name', 'formed_year']
# Permet : /artists/?ordering=-formed_year
# → Trie par année décroissante (- = décroissant)
```

Optimisation avec select_related :

```python
queryset = Album.objects.select_related('artist')
```

Sans : 1 requête pour albums + 1 requête par album pour avoir l'artiste (N+1 problème)
Avec : 1 seule requête qui charge albums + artistes en même temps

---

🔹 catalog_api/urls.py

C'est quoi ? Le routeur : associe URL → Vue

```python
router = DefaultRouter()
router.register('artists', ArtistViewSet, basename='artist')
```

Génère automatiquement :

```
/artists/          → ArtistViewSet.list()
/artists/5/        → ArtistViewSet.retrieve(pk=5)
/artists/5/albums/ → ArtistViewSet.albums(pk=5)
```

🔹 playlist_api/models.py

Même principe que catalog_api mais pour les playlists

Particularité : PlaylistSong

```python
class PlaylistSong(models.Model):
    playlist = models.ForeignKey(Playlist)
    song = models.ForeignKey(Song)
    position = models.PositiveIntegerField()
```

C'est une table de liaison :

- Lie une playlist à plusieurs chansons
- Ajoute une info : `position` (ordre dans la playlist)

unique_together :

```python
unique_together = [['playlist', 'song']]
```

→ Empêche d'avoir 2 fois la même chanson dans une playlist

🔹 playlist_api/types.py

C'est quoi ? Définit comment GraphQL voit les objets

```python
class UserType(DjangoObjectType):
    playlist_count = graphene.Int()  # Champ supplémentaire

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'created_at')

    def resolve_playlist_count(self, info):
        return self.playlists.count()
```

Traduction :

- `UserType` = "Voilà comment un User se présente en GraphQL"
- `playlist_count` = Champ calculé (pas dans la base)
- `resolve_playlist_count` = Comment calculer ce champ

🔹 playlist_api/queries.py

C'est quoi ? Les requêtes GraphQL (LECTURE)

```python
```

```python
class Query(graphene.ObjectType):
    all_playlists = graphene.List(PlaylistType)  # Retourne une liste
    playlist = graphene.Field(PlaylistType, id=graphene.Int(required=True))  # Un seul

    def resolve_all_playlists(root, info):
        return Playlist.objects.all()

    def resolve_playlist(root, info, id):
        return Playlist.objects.get(id=id)
```

Usage :

```graphql
query {
  allPlaylists {      ← Appelle resolve_all_playlists()
    id
    name
  }
}
```

Optimisation :

```python
Playlist.objects.select_related('user').prefetch_related('playlist_songs__song')
```

- `select_related` : Pour relations 1:1 (playlist → user)
- `prefetch_related` : Pour relations 1:n (playlist → plusieurs songs)

🔹 **playlist_api/mutations.py**

C'est quoi ? Les mutations GraphQL (ÉCRITURE)

Structure d'une mutation :

python

```python
class CreatePlaylist(graphene.Mutation):
    # 1. Arguments (entrées)
    class Arguments:
        name = graphene.String(required=True)
        user_id = graphene.Int(required=True)

    # 2. Sorties
    playlist = graphene.Field(PlaylistType)
    success = graphene.Boolean()
    errors = graphene.List(graphene.String)

    # 3. Logique
    def mutate(self, info, name, user_id):
        try:
            user = User.objects.get(pk=user_id)
            playlist = Playlist.objects.create(name=name, user=user)
            return CreatePlaylist(playlist=playlist, success=True, errors=[])
        except User.DoesNotExist:
            return CreatePlaylist(playlist=None, success=False, errors=["User not found"])
```

Usage :

graphql

```graphql
mutation {
  createPlaylist(name: "Ma Playlist", userId: 1) {
    success
    errors
    playlist {
      id
      name
    }
  }
}
```

Gestion d'erreurs complète :

```python
try:
    # Vérifier que tout existe
    playlist = Playlist.objects.get(pk=playlist_id)

    try:
        song = Song.objects.get(pk=song_id)
    except Song.DoesNotExist:
        return AddSongToPlaylist(success=False, errors=["Song not found"])

    # Faire l'action
    # ...

except Playlist.DoesNotExist:
    return AddSongToPlaylist(success=False, errors=["Playlist not found"])
except Exception as e:
    return AddSongToPlaylist(success=False, errors=[str(e)])
```

transaction.atomic() :

```python
with transaction.atomic():
    # Toutes ces opérations réussissent ensemble
    # ou échouent ensemble (pas de moitié)
    ps1.position = 1
    ps1.save()
    ps2.position = 2
    ps2.save()
```

- **musicapi_project/settings.py**

C'est quoi ? Configuration du projet

Sections importantes :

1. Applications installées :

```python
INSTALLED_APPS = [
    'django.contrib.admin',      # Interface admin
    'rest_framework',            # Pour l'API REST
    'graphene_django',           # Pour GraphQL
    'catalog_api',               # Nos apps
    'playlist_api',
    'web_interface',
]
```

2. Base de données :

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'musicdb',          # Nom de la base
        'USER': 'musicapi',         # Utilisateur
        'PASSWORD': 'musicpass123',  # Mot de passe
        'HOST': 'ns3532542.ip-193-70-34.eu',  # Serveur distant
        'PORT': '54321',            # Port
    }
}
```

3. Configuration REST :

```python
```

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',  # Pas d'authentification
    ],
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20,  # 20 résultats par page
}
```

4. Configuration GraphQL :

python

```
GRAPHENE = {
    'SCHEMA': 'musicapi_project.schema.schema'  # Où trouver le schéma
}
```

🔄 PARTIE 8 : COMMENT ÇA MARCHE (Flux de données)

Exemple REST : Lister les artistes

```
1. UTILISATEUR
   GET /api/catalog/artists/

   ↓

2. urls.py
   router reconnaît 'artists' → ArtistViewSet

   ↓

3. views.py (ArtistViewSet)
   - Récupère Artist.objects.all()
   - Utilise ArtistListSerializer

   ↓
```

```
4. serializers.py (ArtistListSerializer)
   - Convertit chaque artiste en JSON
   - Calcule album_count, song_count

   ↓

5. RETOUR JSON
   {
    "results": [
      {"id": 1, "name": "Pink Floyd", "country": "UK", ...},
      {"id": 2, "name": "The Beatles", "country": "UK", ...}
    ]
   }
```

Exemple GraphQL : Créer une playlist

```
1. UTILISATEUR
   mutation {
     createPlaylist(name: "Rock", userId: 1) {
       success
       playlist { id name }
     }
   }

   ↓

2. musicapi_project/schema.py
   Trouve la mutation createPlaylist

   ↓

3. mutations.py (CreatePlaylist)
   - Vérifie que l'utilisateur existe
   - Crée la playlist
   - Retourne success=True et la playlist
```

```
  ↓

4. RETOUR JSON
  {
   "data": {
    "createPlaylist": {
     "success": true,
     "playlist": {
      "id": 10,
      "name": "Rock"
     }
    }
   }
  }
```

🎓 PARTIE 9 : CONCEPTS AVANCÉS EXPLIQUÉS

1. Propriétés vs Champs

Champ (dans la base) :

```python
name = models.CharField(max_length=200)
```

→ Stocké dans la base de données

Propriété (calculée) :

```python
@property
def album_count(self):
    return self.albums.count()
```

→ PAS dans la base, calculé à chaque fois

2. ForeignKey et Relations

ForeignKey = Clé étrangère = Lien vers un autre objet

```python
artist = models.ForeignKey(Artist, on_delete=models.CASCADE, related_name='albums')
```

Décryptage :

- `ForeignKey(Artist)` : Chaque album a UN artiste
- `on_delete=models.CASCADE` : Si on supprime l'artiste, supprimer ses albums aussi
- `related_name='albums'` : Permet `artist.albums.all()`
- `db_column='artist_id'` : Dans la base, la colonne s'appelle 'artist_id'

3. Sérialisation Imbriquée

Problème : Un album a un artiste, comment afficher les deux ?

Solution 1 : ID seulement :

```json
{
  "id": 5,
  "title": "The Wall",
  "artist_id": 1
}
```

Solution 2 : Objet complet :

```json
```

```
{
  "id": 5,
  "title": "The Wall",
  "artist": {
    "id": 1,
    "name": "Pink Floyd",
    "country": "UK"
  }
}
```

Comment faire la solution 2 :

```python
class AlbumSerializer(serializers.ModelSerializer):
    artist = ArtistListSerializer(read_only=True)  # Objet complet en lecture
    artist_id = serializers.PrimaryKeyRelatedField(  # ID en écriture
        queryset=Artist.objects.all(),
        source='artist',
        write_only=True
    )
```

4. Actions Personnalisées

Par défaut : list, create, retrieve, update, destroy

Ajouter la sienne :

```python
@action(detail=True, methods=['get'])  # detail=True → nécessite un ID
def albums(self, request, pk=None):
    """GET /artists/5/albums/"""
    artist = self.get_object()
    albums = artist.albums.all()
    serializer = AlbumListSerializer(albums, many=True)
    return Response(serializer.data)
```

5. Optimisation des Requêtes

Problème N+1 :

```python
albums = Album.objects.all()  # 1 requête
for album in albums:
    print(album.artist.name)  # N requêtes (1 par album)
```

Solution :

```python
albums = Album.objects.select_related('artist').all()  # 1 seule requête
for album in albums:
    print(album.artist.name)  # Pas de requête, déjà chargé
```

🚀 PARTIE 10 : COMMENT TESTER TON PROJET

1. Lancer le serveur

```bash
python manage.py runserver
```

2. Tester l'API REST dans le navigateur

```
http://localhost:8000/api/catalog/artists/
```

→ Tu verras une interface "Browsable API" avec tous les artistes

3. Tester GraphQL avec GraphiQL

```
http://localhost:8000/graphql/
```

→ Interface graphique pour tester tes queries

Exemple de query :

```graphql
query {
  allPlaylists {
    name
    songCount
    user {
      username
    }
  }
}
```

Clique sur "Play" → Tu verras le résultat

4. Tester avec curl (ligne de commande)

```bash
# REST
curl http://localhost:8000/api/catalog/artists/

# GraphQL
curl http://localhost:8000/graphql/ \
  -H "Content-Type: application/json" \
  -d '{"query": "{ allPlaylists { name } }"}'
```

❓ PARTIE 11 : FAQ - Questions Fréquentes

Q: Pourquoi 2 types d'API (REST et GraphQL) ?

R: Pour apprendre les deux ! Dans la vraie vie, tu choisirais l'un ou l'autre selon le projet.

Q: C'est quoi "managed = False" ?

R: Dit à Django : "Ne touche pas à cette table, elle existe déjà". Sinon Django voudrait la créer.

Q: Pourquoi related_name='albums' ?

R: Pour pouvoir faire `artist.albums.all()` au lieu de `Album.objects.filter(artist=artist)`.

Q: À quoi sert select_related ?

R: Optimisation : charge plusieurs objets liés en une seule requête au lieu de N+1 requêtes.

Q: GraphQL c'est mieux que REST ?

R: Ni mieux ni moins bien, juste différent :

- **REST** : Simple, standard, facile
- **GraphQL** : Flexible, une requête pour tout, mais plus complexe

Q: Pourquoi 2 serializers (List et Detail) ?

R: List = rapide, peu de détails. **Detail** = complet, plus lent. Optimisation.

Q: C'est quoi un ViewSet ?

R: Un ensemble de vues (list, create, retrieve, update, delete) regroupées en une seule classe.

🎯 PARTIE 12 : EN RÉSUMÉ

Ce que tu DOIS retenir

1. **Le projet** : Bibliothèque musicale + Playlists avec 2 types d'API
2. **REST (catalog_api)** :
   - URLs = ressources (`/artists/`, `/albums/`)
   - CRUD automatique avec ModelViewSet
   - Serializers pour convertir objets ↔ JSON

3. **GraphQL (playlist_api)** :
    - Queries = lecture
    - Mutations = écriture
    - On demande exactement ce qu'on veut
4. **Django MVC** :
    - **Models** = structure des données
    - **Views** = logique métier
    - **Serializers/Types** = présentation
5. **Optimisation** :
    - `select_related` pour relations 1:1
    - `prefetch_related` pour relations 1:n
6. **Gestion d'erreurs** :
    - Toujours des try/except
    - Messages clairs
    - Retourner success + errors

📖 Pour aller plus loin

Documentation officielle

- Django : https://docs.djangoproject.com/
- Django REST Framework : https://www.django-rest-framework.org/
- Graphene Django : https://docs.graphene-python.org/projects/django/

Prochaines étapes

1. ✅ Comprendre la structure
2. ✅ Tester tous les endpoints

3. 📝 Ajouter des fonctionnalités
4. 🔒 Ajouter de la sécurité (authentification)
5. 📚 Ajouter de la documentation
6. 🚀 Déployer en production

TU AS DES QUESTIONS ? Demande-moi d'expliquer n'importe quelle partie plus en détail ! 🚀