🔍 Explication Ligne par Ligne du Code

📄 FICHIER 1 : catalog_api/models.py

Imports

```python
from django.db import models
```

Explication : Importe les outils Django pour créer des modèles (= tables de base de données)

Classe Artist (Artiste)

```python
class Artist(models.Model):
```

Explication : Crée une classe Artist qui hérite de `Model` (= c'est une table en base)

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

Explication :

- `name` = nom de la colonne
- `CharField` = champ texte
- `max_length=200` = maximum 200 caractères

```python
```

```
    country = models.CharField(max_length=100)
    formed_year = models.IntegerField()
```

Explication :

- `country` = pays (texte)
- `formed_year` = année de création (nombre entier)

python

```
    class Meta:
        db_table = 'artist'
        managed = False
```

Explication :

- `db_table = 'artist'` : La table dans PostgreSQL s'appelle 'artist'
- `managed = False` : Django ne gère PAS cette table (elle existe déjà, on ne la crée pas)

python

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

Explication :

- `@property` : Transforme cette fonction en "fausse variable"
- On peut faire `artist.album_count` au lieu de `artist.album_count()`
- `self.albums` : Tous les albums de cet artiste (grâce au `related_name`)
- `.count()` : Compte combien il y en a

Exemple concret :

```python
artist = Artist.objects.get(id=1)  # Pink Floyd
print(artist.album_count)  # → 15 (Pink Floyd a 15 albums)
```

```python
    @property
    def song_count(self):
        return self.songs.count()  # ✅ FONCTIONNE grâce au related_name='songs'
```

Explication :

- Compte le nombre de chansons de cet artiste
- `self.songs` fonctionne maintenant car on a ajouté `related_name='songs'` dans le modèle Song

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

Explication : Quand on fait `print(artist)`, ça affiche son nom au lieu de `<Artist object>`

Classe Album

```python
class Album(models.Model):
    title = models.CharField(max_length=200)
```

Explication : Titre de l'album (texte)

```python
```

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

Explication DÉTAILLÉE :

- `ForeignKey` : C'est une relation (lien vers un autre objet)
- `Artist` : Chaque album est lié à UN artiste
- `on_delete=models.CASCADE` : Si on supprime l'artiste, supprimer aussi ses albums
- `related_name='albums'` : Permet de faire `artist.albums.all()` pour avoir ses albums
- `db_column='artist_id'` : Dans PostgreSQL, la colonne s'appelle 'artist_id' (pas 'artist')

Exemple concret :

python

```
album = Album.objects.get(id=5)  # The Wall
print(album.artist.name)  # → "Pink Floyd"

artist = Artist.objects.get(id=1)  # Pink Floyd
print(artist.albums.all())  # → [The Wall, Dark Side of the Moon, ...]
```

python

```
    @property
    def total_duration(self):
        return sum(song.duration_seconds for song in self.songs.all())
```

Explication :

- `self.songs.all()` : Toutes les chansons de cet album

- `for song in ...` : Pour chaque chanson
- `song.duration_seconds` : Sa durée
- `sum(...)` : Addition de toutes les durées

Exemple :

```
Album "The Wall"
  ├── Song 1 : 180 secondes
  ├── Song 2 : 240 secondes
  └── Song 3 : 300 secondes
  → total_duration = 720 secondes = 12 minutes
```

Classe Song

python

```
    artist = models.ForeignKey(
        Artist,
        on_delete=models.CASCADE,
        related_name='songs',  # ✅ C'ÉTAIT ÇA LE PROBLÈME !
        db_column='artist_id'
    )
```

AVANT : Pas de `related_name='songs'` → `artist.songs.all()` ne fonctionnait PAS **APRÈS** : Ajout de `related_name='songs'` → `artist.songs.all()` fonctionne !

Pourquoi c'était important ?

- Dans `views.py`, on faisait `artist.songs.all()` pour l'endpoint `/artists/5/songs/`
- Sans `related_name`, Django ne savait pas comment accéder aux chansons d'un artiste
- Maintenant ça marche !

python

```python
    @property
    def duration_formatted(self):
        minutes = self.duration_seconds // 60
        seconds = self.duration_seconds % 60
        return f"{minutes}:{seconds:02d}"
```

Explication :

- `//` : Division entière (ex: 185 // 60 = 3)
- `%` : Reste de la division (ex: 185 % 60 = 5)
- `f"{minutes}:{seconds:02d}"` : Format "3:05" (02d = 2 chiffres minimum)

Exemple :

```python
song.duration_seconds = 185
song.duration_formatted  # → "3:05"
```

📄 FICHIER 2 : catalog_api/serializers.py

Imports

```python
from rest_framework import serializers
from .models import Artist, Album, Song
```

Explication : Importe les outils REST et nos modèles

ArtistListSerializer

```python
```

```python
class ArtistListSerializer(serializers.ModelSerializer):
```

Explication : Serializer basé sur le modèle Artist

```python
    album_count = serializers.ReadOnlyField()
    song_count = serializers.ReadOnlyField()
```

Explication :

- `ReadOnlyField()` : Champ en lecture seule
- Va chercher automatiquement la propriété `album_count` du modèle
- On NE PEUT PAS modifier ce champ (c'est calculé)

```python
    class Meta:
        model = Artist
        fields = ['id', 'name', 'country', 'formed_year', 'album_count', 'song_count']
```

Explication :

- `model = Artist` : Basé sur le modèle Artist
- `fields = [...]` : Liste des champs à inclure dans le JSON

AVANT (trop vide) :

```python
fields = ['id', 'name', 'country']  # ❌ Manque des infos
```

APRÈS (plus généreux) :

```python
```

```
fields = ['id', 'name', 'country', 'formed_year', 'album_count', 'song_count']  # ✅ Plus d'infos
```

ArtistSerializer (Détail)

```python
class ArtistSerializer(serializers.ModelSerializer):
    album_count = serializers.ReadOnlyField()
    song_count = serializers.ReadOnlyField()
    # ✅ RETIRÉ: albums = AlbumListSerializer(many=True, read_only=True)
```

POURQUOI RETIRÉ ?

Problème : Relation 1:n (un artiste a PLUSIEURS albums)

```json
{
  "id": 1,
  "name": "Pink Floyd",
  "albums": [
    {"id": 5, "title": "The Wall"},
    {"id": 6, "title": "Dark Side"},
    {"id": 7, "title": "Wish You Were Here"},
    ...  // Peut être ÉNORME
  ]
}
```

→ Trop de données, trop lent, pas pratique

Solution : Utiliser un endpoint dédié

```
GET /artists/1/        → Détails de l'artiste (sans albums)
GET /artists/1/albums/  → Uniquement les albums
```

SongListSerializer

```python
class SongListSerializer(serializers.ModelSerializer):
    artist = ArtistListSerializer(read_only=True)  # ✅ CHANGÉ
    album = AlbumListSerializer(read_only=True)    # ✅ CHANGÉ
```

AVANT (juste le nom) :

```python
artist = serializers.CharField(source='artist.name', read_only=True)
# → {"artist": "Pink Floyd"}
```

APRÈS (objet complet) :

```python
artist = ArtistListSerializer(read_only=True)
# → {"artist": {"id": 1, "name": "Pink Floyd", "country": "UK", ...}}
```

POURQUOI ?

Règle du prof : Relation 1:1 → Afficher l'objet complet

- Une chanson a UN artiste (pas 50)
- Une chanson a UN album (pas 100)
- Donc on peut afficher les objets complets sans problème

SongSerializer (avec write_only)

```python
```

```python
class SongSerializer(serializers.ModelSerializer):
    # Lecture : objets complets
    artist = ArtistListSerializer(read_only=True)
    album = AlbumListSerializer(read_only=True)

    # Écriture : juste les IDs
    artist_id = serializers.PrimaryKeyRelatedField(
        queryset=Artist.objects.all(),
        source='artist',
        write_only=True
    )
    album_id = serializers.PrimaryKeyRelatedField(
        queryset=Album.objects.all(),
        source='album',
        write_only=True
    )
```

POURQUOI 2 CHAMPS POUR LA MÊME CHOSE ?

EN LECTURE (GET) : On veut les détails

json

```json
{
  "artist": {
    "id": 1,
    "name": "Pink Floyd",
    "country": "UK"
  }
}
```

EN ÉCRITURE (POST/PUT) : On donne juste l'ID

json

```
{
  "title": "New Song",
  "artist_id": 1,
  "album_id": 5
}
```

Sinon il faudrait envoyer tout l'objet artiste pour créer une chanson (lourd !)

📄 FICHIER 3 : catalog_api/views.py

Import des filtres

```python
from rest_framework import viewsets, filters
```

Explication : `filters` permet la recherche et le tri

ArtistViewSet

```python
class ArtistViewSet(viewsets.ModelViewSet):
    queryset = Artist.objects.all()
    serializer_class = ArtistSerializer
```

Explication :

- `ModelViewSet` : Génère automatiquement list, create, retrieve, update, delete
- `queryset` : Tous les artistes
- `serializer_class` : Comment les sérialiser par défaut

```python
```

```
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]
    search_fields = ['name', 'country']
    ordering_fields = ['name', 'formed_year']
    ordering = ['-formed_year']
```

Explication :

- `SearchFilter` : Permet `/artists/?search=pink`
- `search_fields` : Cherche dans `name` OU `country`
- `OrderingFilter` : Permet `/artists/?ordering=-formed_year`
- `ordering` : Tri par défaut (par année décroissante)

Exemple d'utilisation :

bash

```
# Chercher "pink" dans le nom ou le pays
GET /artists/?search=pink

# Trier par année croissante
GET /artists/?ordering=formed_year

# Trier par année décroissante (défaut)
GET /artists/?ordering=-formed_year
```

python

```
    def get_serializer_class(self):
        if self.action == 'list':
            return ArtistListSerializer
        return ArtistSerializer
```

Explication :

- Pour `/artists/` (liste) → Utilise `ArtistListSerializer` (version simple)

- Pour `/artists/5/` (détail) → Utilise `ArtistSerializer` (version complète)

Pourquoi ?

- Liste = Beaucoup d'objets → version légère
- Détail = Un seul objet → version complète

```python
@action(detail=True, methods=['get'])
def albums(self, request, pk=None):
    artist = self.get_object()
    albums = artist.albums.all()
    serializer = AlbumListSerializer(albums, many=True)
    return Response(serializer.data)
```

Explication ligne par ligne :

1. `@action(detail=True, methods=['get'])` : Crée un endpoint GET qui nécessite un ID
2. `def albums(...)` : Nom de l'action → URL sera `/artists/5/albums/`
3. `artist = self.get_object()` : Récupère l'artiste avec l'ID=pk (5)
4. `albums = artist.albums.all()` : Tous les albums de cet artiste
5. `serializer = AlbumListSerializer(albums, many=True)` : Sérialise en JSON (`many=True` car liste)
6. `return Response(serializer.data)` : Retourne le JSON

Exemple de réponse :

```
GET /artists/1/albums/

{
  "results": [
    {"id": 5, "title": "The Wall", "artist": {...}, ...},
    {"id": 6, "title": "Dark Side of the Moon", "artist": {...}, ...}
  ]
}
```

AlbumViewSet

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

Explication :

- Sans `select_related` : 1 requête pour albums + 1 requête par album pour l'artiste
- Avec `select_related` : 1 seule requête pour tout

Exemple concret :

SANS select_related (50 albums) :

```
Requête 1 : SELECT * FROM album            (50 albums)
Requête 2 : SELECT * FROM artist WHERE id=1  (pour album 1)
Requête 3 : SELECT * FROM artist WHERE id=2  (pour album 2)
...
Requête 51: SELECT * FROM artist WHERE id=50 (pour album 50)
→ 51 requêtes !!! LENT !!!
```

AVEC select_related :

```
Requête unique :
  SELECT album.*, artist.*
  FROM album
  JOIN artist ON album.artist_id = artist.id
→ 1 seule requête ! RAPIDE !
```

python

```python
    @action(detail=True, methods=['get'])
    def songs(self, request, pk=None):  # ✅ C'ÉTAIT ÇA QUI MANQUAIT
        album = self.get_object()
        songs = album.songs.all().order_by('track_number')
        serializer = SongListSerializer(songs, many=True)
        return Response(serializer.data)
```

Explication :

- Endpoint : `GET /albums/5/songs/`
- Retourne toutes les chansons de l'album #5
- `.order_by('track_number')` : Triées par numéro de piste

📄 FICHIER 4 : playlist_api/models.py

Classe Playlist

python

```python
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name='playlists',
        db_column='user_id'  # ✅ AJOUTÉ
    )
```

POURQUOI db_column ?

Dans Python :

```
python

playlist.user  # On accède comme ça
```

Dans PostgreSQL (la vraie colonne) :

```
sql

SELECT * FROM playlist WHERE user_id = 1;  -- La colonne s'appelle user_id
```

Sans `db_column='user_id'`, Django chercherait une colonne appelée `user` → ERREUR !

Classe PlaylistSong

```
python

    class Meta:
        db_table = 'playlist_song'  # ✅ CORRIGÉ (avant : playlist_songs)
```

LE GROS PROBLÈME :

Avant :

```
python

db_table = 'playlist_songs'  # Avec 's'
```

→ Django cherche la table `playlist_songs` dans PostgreSQL → Mais la table s'appelle `playlist_song` (sans 's') → **ERREUR : Table n'existe pas !**

Après :

```
python
```

```
db_table = 'playlist_song'  # Sans 's'
```

→ Django trouve la bonne table → **Tout fonctionne !**

python

```
class Meta:
    unique_together = [['playlist', 'song']]
```

Explication :

- Empêche d'avoir 2 fois la même chanson dans une playlist
- Playlist #5 + Song #10 = OK une fois
- Playlist #5 + Song #10 = ERREUR la 2ème fois

📄 FICHIER 5 : playlist_api/mutations.py

CreatePlaylist

python

```
class CreatePlaylist(graphene.Mutation):
    class Arguments:
        name = graphene.String(required=True)
        user_id = graphene.Int(required=True)
        is_public = graphene.Boolean(default_value=False)
```

Explication :

- Arguments = Ce qu'on doit envoyer
- `required=True` : Obligatoire
- `default_value=False` : Valeur par défaut si non fourni

python

```
    playlist = graphene.Field(PlaylistType)
    success = graphene.Boolean()
    errors = graphene.List(graphene.String)
```

Explication :

- Ce qu'on retourne
- `playlist` : La playlist créée (ou None si erreur)
- `success` : True/False
- `errors` : Liste de messages d'erreur

python

```
    def mutate(self, info, name, user_id, is_public):
        try:
            user = User.objects.get(pk=user_id)
            playlist = Playlist.objects.create(
                name=name,
                user=user,
                is_public=is_public
            )
            return CreatePlaylist(playlist=playlist, success=True, errors=[])
        except User.DoesNotExist:
            return CreatePlaylist(
                playlist=None,
                success=False,
                errors=["User not found"]
            )
        except Exception as e:
            return CreatePlaylist(
                playlist=None,
                success=False,
                errors=[f"Error creating playlist: {str(e)}"]
            )
```

FLUX COMPLET :

1. Requête :

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

2. Traitement :

```python
# Étape 1 : Récupérer l'utilisateur
user = User.objects.get(pk=1)  # user_id=1

# Étape 2 : Créer la playlist
playlist = Playlist.objects.create(
    name="Rock",
    user=user,
    is_public=False  # valeur par défaut
)

# Étape 3 : Retourner le résultat
return CreatePlaylist(
    playlist=playlist,  # L'objet créé
    success=True,
    errors=[]
)
```

3. Réponse :

```json
```

```
{
  "data": {
    "createPlaylist": {
      "success": true,
      "errors": [],
      "playlist": {
        "id": 10,
        "name": "Rock"
      }
    }
  }
}
```

AddSongToPlaylist (Gestion d'erreurs complète)

python

```python
def mutate(self, info, playlist_id, song_id):
    try:
        # ✅ Étape 1 : Vérifier la playlist
        playlist = Playlist.objects.get(pk=playlist_id)

        # ✅ Étape 2 : Vérifier la chanson (imbriqué)
        try:
            song = Song.objects.get(pk=song_id)
        except Song.DoesNotExist:
            return AddSongToPlaylist(
                playlist=None,
                success=False,
                errors=["Song not found"]
            )

        # ✅ Étape 3 : Vérifier si déjà présente
        if PlaylistSong.objects.filter(playlist=playlist, song=song).exists():
            return AddSongToPlaylist(
                playlist=None,
                success=False,
                errors=["Song already in playlist"]
            )

        # ✅ Étape 4 : Calculer la position
        agg = PlaylistSong.objects.filter(playlist=playlist).aggregate(
            max_position=Max('position')
        )
        last_position = agg['max_position'] or 0
        position = last_position + 1

        # ✅ Étape 5 : Créer l'entrée
        PlaylistSong.objects.create(
            playlist=playlist,
            song=song,
            position=position
        )
```

```
            return AddSongToPlaylist(playlist=playlist, success=True, errors=[])

        except Playlist.DoesNotExist:
            return AddSongToPlaylist(
                playlist=None,
                success=False,
                errors=["Playlist not found"]
            )
        except Exception as e:
            return AddSongToPlaylist(
                playlist=None,
                success=False,
                errors=[f"Error: {str(e)}"]
            )
```

POURQUOI TANT DE TRY/EXCEPT ?

Scénarios possibles :

1. ✅ Tout va bien → success=True
2. ❌ Playlist n'existe pas → "Playlist not found"
3. ❌ Chanson n'existe pas → "Song not found"
4. ❌ Chanson déjà dans la playlist → "Song already in playlist"
5. ❌ Erreur imprévu → Message d'erreur générique

Sans gestion d'erreurs : L'application plante, l'utilisateur ne sait pas pourquoi **Avec gestion d'erreurs** : Message clair, l'utilisateur comprend le problème

---

RemoveSongFromPlaylist (Réorganisation)

```
python
```

```python
# Supprimer la chanson
playlist_song.delete()

# ✅ Réorganiser les positions
for idx, ps in enumerate(
    PlaylistSong.objects.filter(playlist_id=playlist_id).order_by('position'),
    start=1
):
    ps.position = idx
    ps.save()
```

POURQUOI RÉORGANISER ?

Avant suppression :

```
Position 1 : Song A
Position 2 : Song B  ← On supprime celle-ci
Position 3 : Song C
```

Sans réorganisation :

```
Position 1 : Song A
Position 3 : Song C  ← Trou dans les positions !
```

Avec réorganisation :

```
Position 1 : Song A
Position 2 : Song C  ← Pas de trou, c'est propre
```

ReorderPlaylistSongs (Transaction)

python

```
with transaction.atomic():
    for sp in song_positions:
        ps = PlaylistSong.objects.get(playlist=playlist, song_id=sp.song_id)
        ps.position = sp.position
        ps.save()
```

POURQUOI transaction.atomic() ?

Sans transaction :

1. Song A → position 1 ✅
2. Song B → position 2 ✅
3. Song C → position 3 ❌ ERREUR !

→ On a Song A et B modifiés, mais pas C → INCOHÉRENT

Avec transaction :

1. Song A → position 1
2. Song B → position 2
3. Song C → position 3 ❌ ERREUR !

→ TOUT est annulé, on revient à l'état initial → COHÉRENT

Règle : Tout réussit ou rien ne change

🎓 RÉCAPITULATIF : Les Concepts Clés

1. related_name

```
python

artist = ForeignKey(Artist, related_name='songs')
```

→ Permet `artist.songs.all()`

2. db_column

```python
artist = ForeignKey(Artist, db_column='artist_id')
```

→ Indique le vrai nom de la colonne dans PostgreSQL

3. db_table

```python
class Meta:
    db_table = 'playlist_song'
```

→ Indique le vrai nom de la table dans PostgreSQL

4. select_related (optimisation 1:1)

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

→ Charge les objets liés en une seule requête

5. @property (calcul à la volée)

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

→ Pas stocké en base, calculé à chaque fois

6. read_only vs write_only

```python
```

```
artist = ArtistSerializer(read_only=True)      # Lecture : objet complet
artist_id = PrimaryKeyRelatedField(write_only=True)  # Écriture : juste l'ID
```

→ 2 champs différents pour lecture et écriture

7. Gestion d'erreurs

```
python

try:
    # Code principal
except SpecificError:
    # Erreur spécifique
except Exception as e:
    # Erreur générique
```

→ Toujours gérer TOUS les cas d'erreur possibles

8. transaction.atomic()

```
python

with transaction.atomic():
    # Tout réussit ou rien
```

→ Garantit la cohérence des données

VOILÀ ! Tu as maintenant TOUTES les explications détaillées ! 🎉

Des questions sur une partie spécifique ?