PAO: Développement d'une bibliothèque musicale

Alexis Durieux

ASI4 - 2017

Table des matières

Introduction

L'objectif de ce PAO est la mise en application des concepts de bases de données par la mise en place d'une base de données et de son utilisation pour réaliser un ensemble de fonctionnalités. Dans notre cas, l'objectif est la création d'une bibliothèque musicale permettant à un utilisateur l'enregistrement de sa discographie. Pour concevoir cette base de données au mieux nous allons procéder par étapes. Dans un premier temps, nous allons analyser les besoins de l'application, puis nous allons réaliser le modèle entité-attribut avant de passer au modèle relationnel. Avec la base de données créée nous allons pouvoir réaliser une application java simple basée sur cette dernière. Nous implémenterons ensuite le pattern **DAO** afin de passer du *modèle relationnel* à un *modèle objet*. En terme de technologie, nous utiliserons une base de données *Postgres* et *JDBC* pour faire le liant avec l'application JAVA afin d'implémenter le pattern **DAO**

Chapitre 1

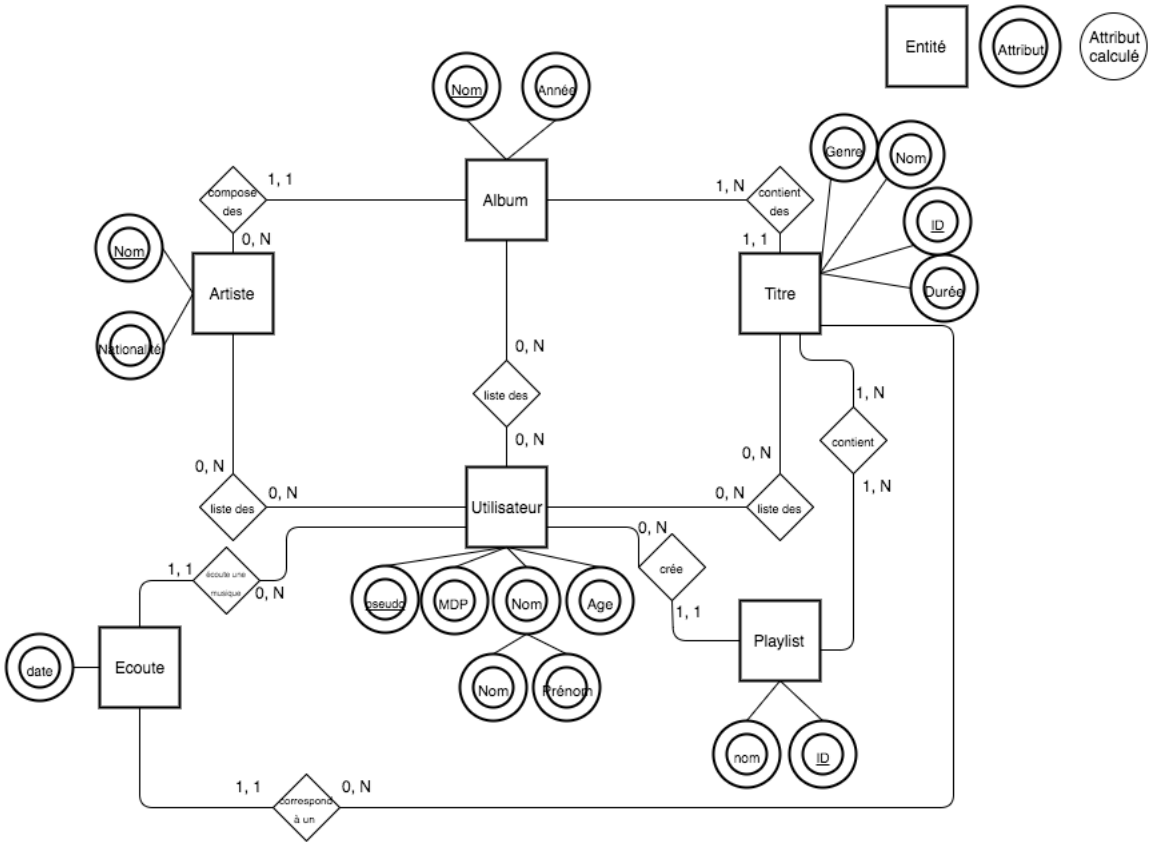
Analyse des besoins de l'application

Nous voulons donc créer une application permettant à l'utilisateur d'enregistrer ses musiques, artistes et albums afin de pouvoir les consulter. L'utilisateur utilise la bibliothèque de l'application pour ajouter ces derniers à sa bibliothèque personnelle. Néanmoins l'utilisateur peut ajouter directement des musiques, albums et artistes à sa bibliothèque personnelle qui sont donc ajoutés à la bibliothèque de l'application. C'est donc une bibliothèque musicale collaborative que nous cherchons à créer où les données ajoutées par un utilisateur servent à tous les utilisateurs. On suppose également les relations suivantes. Un titre est contenu dans un album et un album est composé par un artiste. Ces relations sont primordiales à la bonne création de nos tables. En plus de vouloir stocker des titres, artistes et albums, on veut également stocker des informations propres à l'utilisateur. Enfin ce dernier doit être en mesure de stocker des écoutes et de créer et modifier des playlists. On suppose enfin qu'une fois créé un titre ne peut être modifié ou supprimé de la bibliothèque de l'application. En effet une modification par un utilisateur engendrera sinon une modification potentielle dans la bibliothèque d'autres utilisateurs.

Chapitre 2

Modèle entité-attribut et relationnel

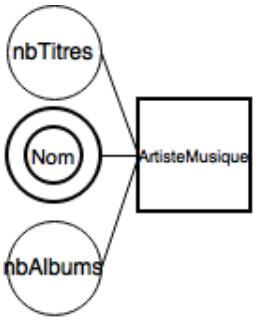
2.1 Modélisation entité-attribut



L'explication des relations du diagramme ci-dessus est faite dans les sous-parties

suivantes.

2.2 Modélisation de la vue



L'explication des relations de la vue ci-dessus est faite dans les sous parties suivantes.

2.3 Du modèle entité-attribut vers le modèle relationnel

2.3.1 Tables, clés primaires et cardinalité

— Utilisateur(pseudo, motDePasse, nom, prenom, age)
La table *Utilisateur* référence les utilisateurs. On choisit comme **clé primaire** un *pseudo* en raison des homonymes possibles entre utilisateurs.
— Titre(id, nom, duree, nomAlbum, genre)
La table *Titre* référence les titres. On choisit un **id clé primaire** en raison des homonymies possibles entre **plusieurs** titres. On a également un *nomAlbum* afin de représenter la cardinalité. En effet on suppose qu'un titre n'appartient au plus qu'à un seul album. Le *genre* quand à lui est un attribut issu d'une énumération définissant une liste de genre de musique.
— Album(nom, nomArtiste, annee)
La table *Album* référence les albums. On suppose dans notre cas que deux albums ne peuvent avoir le même nom. On peut donc choisir **nom** comme **clé primaire**. La clé étrangère *nomArtiste* représente le fait qu'un album est composé par un seul artiste.
— Artiste(nom, nationalite)
La table *Artiste* référence les *artistes*. Ici on a choisit comme **clé primaire** le *nom* car on suppose que deux artistes ne peuvent pas avoir le même nom.
— ListeTitre(pseudoUser, titreId)
La table *ListeTitre* correspond aux titres enregistrés par un utilisateur dans sa bibliothèque. On associe un utilisateur à un titre. La **clé primaire** : est

(pseudoUser, titreId) car un utilisateur ne peut enregistrer deux titres identiques dans sa bibliothèque. En revanche un titre peut être enregistré dans la bibliothèque respective de **plusieurs** utilisateurs.

— ListeAlbum(pseudoUser, nomAlbum)
La table *ListeAlbum* correspond aux albums enregistrés par un utilisateur dans sa bibliothèque. On associe un utilisateur à un titre. La **clé primaire** : est **(pseudoUser, nomAlbum)** car un utilisateur ne peut enregistrer deux albums identiques dans sa bibliothèque. En revanche un album peut être enregistré dans la bibliothèque respective de **plusieurs** utilisateurs.

— ListeArtiste(pseudoUser, nomArtiste)
La table *ListeArtiste* correspond aux artistes enregistrés par un utilisateur dans sa bibliothèque. On associe un **utilisateur** à un **artiste**. La **clé primaire** : est **(pseudoUser, nomArtiste)** car un utilisateur ne peut enregistrer deux artistes identiques dans sa bibliothèque. En revanche un artiste peut être enregistré dans la bibliothèque respective de **plusieurs** utilisateurs.

— Ecoute(pseudoUser, date, idTitre)
La table *Ecoute* correspond à une liste d'écoute. À un instant t, un utilisateur écoute un titre. On les associe donc dans la table. On choisit comme **clé primaire** : **(pseudoUser, date)**. En effet, un utilisateur ne peut écouter deux musiques en même temps.

— Playlist(id, pseudoUser, nom)
La table *Playlist* référence les playlists des utilisateurs. La **clé primaire** est un *id* car un utilisateur peut avoir **plusieurs** playlists. On ne peut également choisir le *nom* car deux utilisateurs différents peuvent avoir une playlist ayant le même nom. En raison de la table **PlaylistTitre**, on utilise **id** comme clé primaire car clé étrangère de **PlaylistTitre**.

— PlaylistTitre(idTitre, idPlaylist)
La table *PlaylistTitre* référence les titres des playlists. La **clé primaire** est **(idTitre, idPlaylist)** car un titre peut être présent une seule fois dans une même *playlist* mais être présent dans **plusieurs** playlists différentes.

2.3.2 Vue

— La **vue** *ArtisteMusique* agrège un *artiste* avec son *nombre de titres* et *nombre d'albums* calculés respectivement à partir des tables *Titre* et *Artiste*. Pour créer cette vue, on effectue deux jointures successives. Une première entre les tables **Artiste** et **Album** puis entre le résultat et la table **Titre**. Enfin on groupe les résultats par *artistes* et on il reste à compter.

Listing 2.1 – Création de la vue

```
CREATE VIEW ArtisteVue AS SELECT nomArtiste,
    COUNT(DISTINCT(nomAlbum)) AS nbAlbums, COUNT
    (DISTINCT(nomTitre)) AS nbTitres FROM (
    SELECT art.nom AS nomArtiste, alb.nom AS
    nomAlbum, tit.nom AS nomTitre FROM ARTISTE
    art INNER JOIN ALBUM alb ON alb.nomartiste =
     art.nom INNER JOIN TITRE tit on tit.
    nomAlbum=alb.nom) v group by v.nomArtiste;
```

2.4 Contraintes d'intégrité et types

2.4.1 Définitions des types et tables

- GENRE_MUSIQUE - ÉNUMÉRATION
  - **pop, rock, jazz, blues, electronique, variété, rap, reggae**
- Utilisateur
  - pseudo : **VARCHAR(15) PRIMARY KEY**
  - motDePasse : **VARCHAR(64) NOT NULL**
  - nom : **VARCHAR(15)**
  - prenom : **VARCHAR(15)**
  - age : **INTEGER** $\in]0, 99[$
- Titre
  - id : **INTEGER PRIMARY KEY**
  - nom : **VARCHAR(50) NOT NULL**
  - duree : **INTEGER NOT NULL**
  - nomArtiste : **VARCHAR(25) FOREIGN KEY REFERENCES ARTISTE NOT NULL**
  - nomAlbum : **VARCHAR(25) FOREIGN KEY REFERENCES ALBUM**
  - genre : **GENRE_MUSIQUE** r la valeur de **id**
- Album
  - nom : **VARCHAR(25) PRIMARY KEY**
  - nomArtiste : **VARCHAR(25) REFERENCES ARTISTE NOT NULL**
  - annee : **INTEGER** $\in]0, 3000[$
- Artiste
  - nom : **VARCHAR(25) PRIMARY KEY**
  - nationalite : **VARCHAR(50)**
- ListeTitre

  - pseudoUser : **VARCHAR(15) FOREIGN KEY REFERENCES UTILISATEUR**
  - titreId : **INTEGER FOREIGN KEY REFERENCES TITRE**
  - **PRIMARY KEY : (pseudoUser, titreId)**
- ListeAlbum
  - pseudoUser : **VARCHAR(15) FOREIGN KEY REFERENCES UTILISATEUR**
  - nomAlbum : **VARCHAR(25) FOREIGN KEY REFERENCES ALBUM**
  - **PRIMARY KEY : (pseudoUser, nomAlbum)**
- ListeArtiste
  - pseudoUser : **VARCHAR(15) FOREIGN KEY REFERENCES UTILISATEUR**
  - nomArtiste : **VARCHAR(25) FOREIGN KEY REFERENCES ARTISTE**
  - **PRIMARY KEY : (pseudoUser, nomArtiste)**
- Ecoute
  - pseudoUser : **VARCHAR(15) FOREIGN KEY REFERENCES UTILISATEUR**
  - dateEcoute : **TIMESTAMP DEFAULT CURRENT_TIMESTAMP**
  - titreId : **INTEGER FOREIGN KEY REFERENCES TITRE**
  - **PRIMARY KEY : (pseudoUser, dateEcoute)**
- Playlist
  - id : **VARCHAR(10) PRIMARY KEY**
  - nom : **VARCHAR(25) NOT NULL**
  - pseudoUser : **VARCHAR(15) FOREIGN KEY REFERENCES UTILISATEUR** Il faut noter qu'on utilise une **séquence** de **INTEGER** pour définir la valeur de **id**
- PlaylistTitre
  - titreId : **INTEGER FOREIGN KEY REFERENCES TITRE**
  - playlistId : **INTEGER FOREIGN KEY REFERENCES PLAYLIST**
  - **PRIMARY KEY : (titreId, playlistId)**

Listing 2.2 – Création des tables

```
CREATE TYPE GENRE_MUSIQUE AS ENUM(
'pop', 'rock', 'jazz', 'blues', 'electronique', '
   variete', 'rap', 'reggae'
);

CREATE SEQUENCE TITRE_SEQ
```

```
 start 1
 increment 1
NO MAXVALUE
CACHE 1;

CREATE SEQUENCE PLAYLIST_SEQ
 start 1
 increment 1
NO MAXVALUE
CACHE 1;

CREATE TABLE UTILISATEUR(
 pseudo VARCHAR(15) PRIMARY KEY,
 motDePasse VARCHAR(64) NOT NULL,
 nom VARCHAR(15),
 prenom VARCHAR(15),
 age INTEGER CONSTRAINT age_constraint CHECK (age
    > 0 AND age < 99)
);

CREATE TABLE ARTISTE(
 nom VARCHAR(25) PRIMARY KEY,
 nationalite VARCHAR(50)
);

CREATE TABLE ALBUM(
 nom VARCHAR(25) PRIMARY KEY,
 nomArtiste VARCHAR(25) REFERENCES ARTISTE NOT
    NULL,
 annee INTEGER CONSTRAINT year_constraint CHECK (
    annee > 0 AND annee < 3000)
);

CREATE TABLE TITRE(
 id INTEGER PRIMARY KEY DEFAULT nextval('TITRE_SEQ
    '),
 nom VARCHAR(50) NOT NULL,
 duree INTEGER NOT NULL,
 nomAlbum VARCHAR(25) REFERENCES ALBUM(nom),
 genre GENRE_MUSIQUE
```

```
);

CREATE TABLE LISTE_TITRE(
pseudoUser VARCHAR(15) REFERENCES UTILISATEUR(
    pseudo),
titreId INTEGER REFERENCES TITRE(id),
PRIMARY KEY (pseudoUser, titreId)
);

CREATE TABLE LISTE_ALBUM(
pseudoUser VARCHAR(15) REFERENCES UTILISATEUR(
    pseudo),
nomAlbum VARCHAR(25) REFERENCES ALBUM(nom),
PRIMARY KEY (pseudoUser, nomAlbum)
);

CREATE TABLE LISTE_ARTISTE(
pseudoUser VARCHAR(15) REFERENCES UTILISATEUR(
    pseudo),
nomArtiste VARCHAR(25) REFERENCES ARTISTE(nom),
PRIMARY KEY (pseudoUser, nomArtiste)
);

CREATE TABLE ECOUTE(
pseudoUser VARCHAR(15) REFERENCES UTILISATEUR(
    pseudo),
dateEcoute TIMESTAMP DEFAULT (NOW()::TIMESTAMP AT
    TIME ZONE 'UTC'),
titreId INTEGER REFERENCES TITRE(id),
PRIMARY KEY (pseudoUser, dateEcoute)
);

CREATE TABLE PLAYLIST(
id INTEGER PRIMARY KEY DEFAULT nextval('
    PLAYLIST_SEQ'),
nom VARCHAR(25) NOT NULL,
pseudoUser VARCHAR(15) REFERENCES UTILISATEUR(
    pseudo)
);
```

```
CREATE TABLE PLAYLIST_TITRE(
 titreId INTEGER REFERENCES TITRE(id),
 playlistId INTEGER REFERENCES PLAYLIST(id),
PRIMARY KEY (titreId, playlistId)
);
```

2.4.2 Définitions des séquences

Deux séquences sont nécessaires pour le développement de l'application.

— **'TITRE_SEQ'** Cette séquence part de 1 et s'incrémente de 1 à l'ajout d'un titre puisque lors de la création d'un titre dans la table **Titre**, l'id de ce dernier est la valeur suivante de la séquence.

— **'PLAYLIST_SEQ'** Cette séquence fonctionne exactement comme la séquence **'TITRE_SEQ'** à la différence qu'elle sert pour les *id* de la table **Playlist**. Elle démarre à 1 et s'incrémente de 1 à l'ajout d'un titre puisque lors de la création d'un titre, l'id de ce dernier est la valeur suivante de la séquence.

2.4.3 Définitions des triggers

— Suppression dans **PLAYLIST**
Une suppression d'une playlist dans la table **PLAYLIST**, engendre la suppression de toutes les références associées dans la table **'PLAYLIST_TITRE'**.

Listing 2.3 – Trigger sur la table Playlist

```
CREATE OR REPLACE FUNCTION
    cleanUpPlaylistTitre() RETURNS TRIGGER as
    $$
BEGIN
DELETE FROM PLAYLIST_TITRE WHERE playlistId=
    OLD.ID;
RETURN OLD;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER DELETE_IN_PLAYLIST_TITRE
BEFORE DELETE ON PLAYLIST
FOR EACH ROW
EXECUTE PROCEDURE cleanUpPlaylistTitre();
```

Le principe est simple, avant de supprimer une entrée dans la table **Playlist**, on s'assure que toutes les entrées de la table **Playlist_Titre** ayant pour clé étrangère *l'id* de l'entrée supprimée dans **Playlist** soient supprimées préalablement.

2.4.4 Importation de données

Afin d'ajouter facilement des données à la base de données, on décide d'importer des données via un fichier au format **csv** à l'aide de la fonction **COPY**.

Listing 2.4 – Copie des données

```
COPY ARTISTE FROM 'artistes.csv' DELIMITER ',' CSV;

COPY ALBUM FROM 'albums.csv' DELIMITER ',' CSV;

COPY TITRE (nom, duree, nomAlbum, genre) FROM '
    titres.csv' DELIMITER ',' CSV;
```

Chapitre 3

Développement de l'application JAVA

3.1 Cahier des charges

3.1.1 Présentation du principe de l'application

La bibliothèque musicale que l'on souhaite développer est une bibliothèque musicale inspirée des plateformes populaires existantes (Spotify, Deezer)...À Contrario de ces dernières, cette application ne permettra pas d'écouter directement des titres mais permettra néanmoins d'enregistrer les écoutes récentes. Néanmoins, cette application permettra à l'utilisateur de classifier sa musique, de la trier et potentiellement dans un dernier de se voir recommander des musiques.

3.1.2 Cas d'utilisations et fonctionnalités

Les spécifications que nous allons lister ci-après sont l'ensemble des fonctionnalités permisent à l'utilisateur de l'application :

- Un utilisateur peut créer un compte auquel il pourra se connecter à l'aide d'un pseudonyme et d'un mot de passe.
- La connexion au compte est nécessaire pour accéder à l'application.
  Nous supposerons maintenant que l'utilisateur est connecté pour les fonctionnalités suivantes
- L'utilisateur peut ajouter manuellement des titres à la bibliothèque de l'application.
- L'utilisateur peut ajouter manuellement des artistes à la bibliothèque de l'application.
- L'utilisateur peut ajouter manuellement des albums à la bibliothèque de l'application.

— L'utilisateur peut ajouter des titres à sa bibliothèque personnelle [1].
— L'utilisateur peut ajouter des artistes à sa bibliothèque personnelle.
— L'utilisateur peut ajouter des albums à sa bibliothèque personnelle.
— L'utilisateur peut créer des playlists à partir des titres de la bibliothèque de l'application.
— L'utilisateur peut enregistrer ses écoutes récentes.
— L'utilisateur peut retirer des écoutes récentes.
— L'utilisateur peut voir des recommandations basées sur ses écoutes récentes
— L'utilisateur peut rechercher des titres, artistes, albums.

3.1.3 Modèle DAO et Diagramme de classe

Dans le cadre de ce projet nous utiliserons le **Design Pattern DAO (Data Access Object)**. Ce **design pattern** permet de séparer l'accès aux bases de données à l'utilisation même de ces données au sein de l'application. Ainsi cela va consister dans notre cas à créer des classes correspondant aux entités que l'on compte manipuler dans notre application. Seulement au lieu de faire ces classes manipuler directement la base de données, nous allons utiliser des classes **DAO** s'occupant de l'accès à la base de données pour l'entité correspondante. Par exemple **TitreDAO** va contenir les opérations relative à la classe **Titre** en base de données. Ce pattern **DAO** nous permet de passer d'un **modèle relationnel à une architecture objet**

3.1.4 JDBC

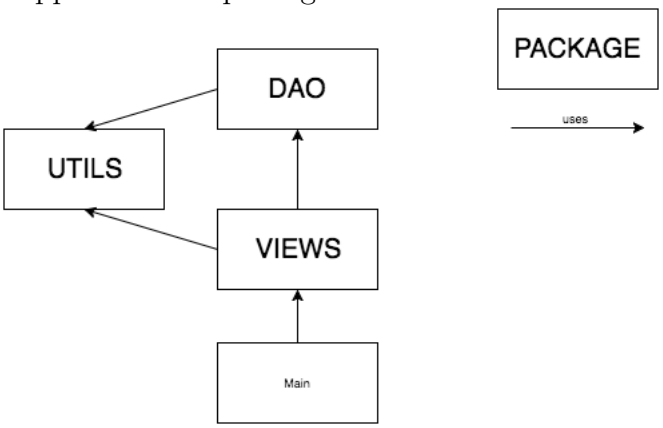
Afin d'implémenter le design pattern **DAO**, nous allons utiliser **JDBC (Java Database Connectivity)**. JDBC est une *API* pour JAVA permettant d'accéder à une base de donnée. Cette API fournit des méthodes pour se connecter à une base de données et effectuer des requêtes sur cette dernière depuis une application JAVA. L'utilisation de cette *API* conjointement avec le design pattern **DAO** va se faire par l'intermédiaire d'une classe JAVA **DatabaseConnection** dont le rôle va être de créer une connexion à une base de données et d'implémenter les requêtes standard **select, update, insert, delete**. Cette classe va donc être une classe parente des implémentations des objets DAO.

3.2 Structure de l'application

Une application *Java* se divise en packages. On décide donc de diviser l'application en trois packages : **dao, views, et utils**. On obtient donc le diagramme de

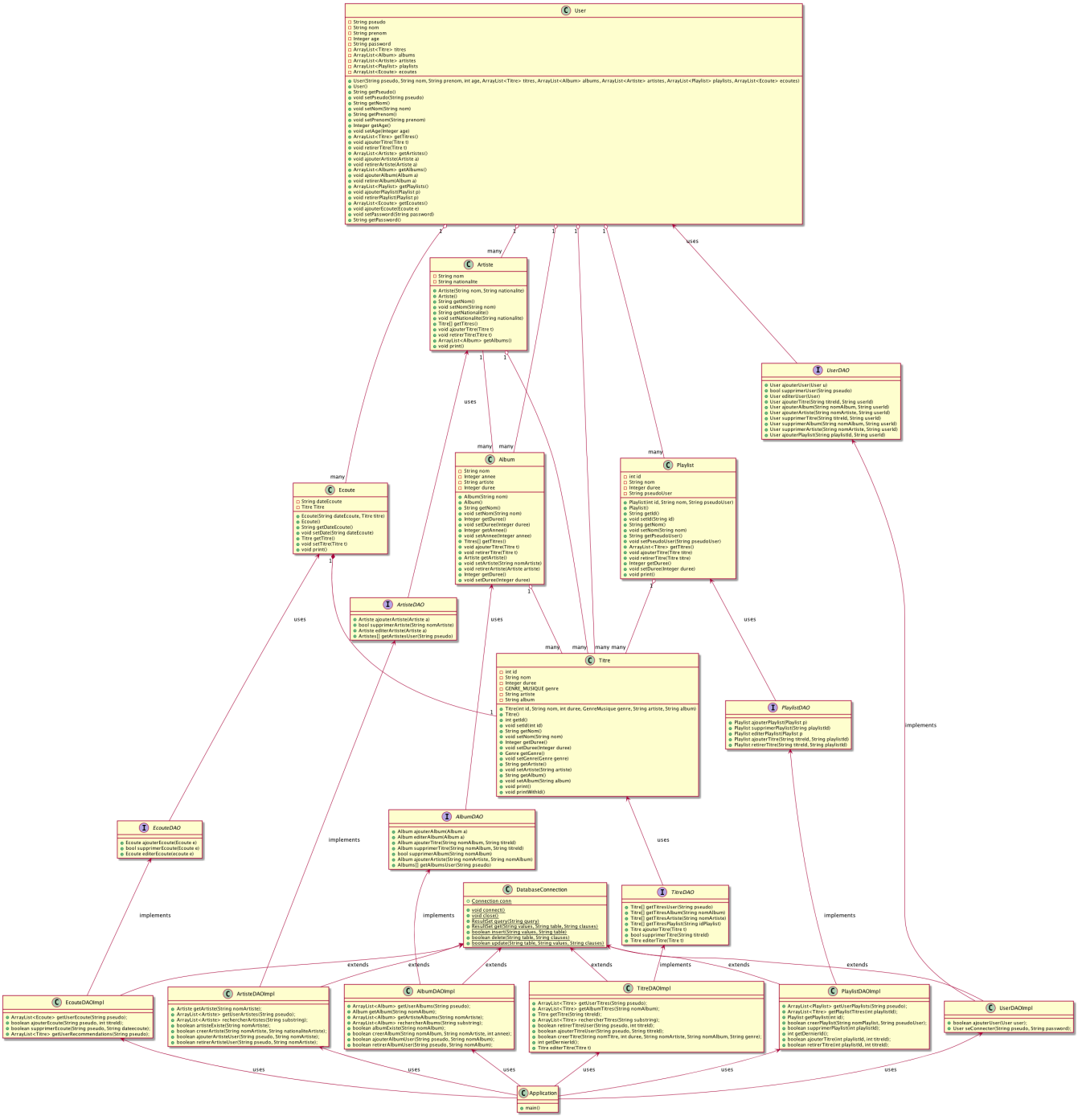
1. Contenu visible seulement par l'utilisateur

l'application en packages ci-dessous :



3.2.1 Package DAO

Comme expliqué ci-dessus le package **DAO**va permettre l'implémentation de ce pattern. Dans ce package on va créer les interfaces **DAO** et on va créer deux sous packages **models et impl**. Le package **impl** contient les implémentations des interfaces DAO. Le package **models** contient lui les *entités* de notre modèle relationnel nécessaires à l'architecture orientée objet et à l'implémentation du pattern. On utilise un **singleton** *DatabaseConnection* afin d'implémenter la liaison entre l'application *JAVA* et la base de données *PostgreSQL*.


User
Artiste
Album
Playlist
Ecoute
Titre
UserDAO
ArtisteDAO
AlbumDAO
TitreDAO
PlaylistDAO
EcouteDAO
DatabaseConnection
UserDAOImpl
ArtisteDAOImpl
AlbumDAOImpl
TitreDAOImpl
PlaylistDAOImpl
EcouteDAOImpl
Application
uses
many
implements
extends

3.2.2 Package Views

Le package **views** contient toutes les vues de l'application. Il consiste en une agrégation de classes permettant d'afficher les vues de l'application et d'interagir avec l'utilisateur. On utilise un **singleton** *MainFrameController*. Cette classe est la classe nous permettant de naviguer entre les différentes vues de l'application. Elle possède **deux attributs statiques** qui sont une instance de *Scanner* nous permettant de récupérer les entrées utilisateur et une instance de *User* qui est l'utilisateur connecté à l'application.

3.2.3 Package Utils

Le package **utils** contient l'énumération *GenreMusique* correspondant à la colonne *genre* dans la table **Titre**. Ce package comporte également la classe **Utils** contenant quelques fonctions utiles à la réalisation de l'application.

3.2.4 Main

Le **Main** a pour rôle de créer une instance de **MainFrameController** afin de lancer l'application. Le constructeur de **MainFrameController** appelle la vue de bienvenue de l'application.

3.3 Explication du fonctionnement des fonctionnalités avancées

3.3.1 Recherche de titres, artistes, albums

Pour rechercher un titre, artiste, ou un album, on demande à l'utilisateur d'entrer une chaîne de caractères. On va ensuite mettre cette chaîne de caractères en minuscule et enlever les espaces de la chaîne de caractères On effectue la même opération sur la colonne **nom** de la table dans laquelle on recherche. Enfin on retourne toutes les entrées dont la chaîne de caractères recherchée est contenue dans la colonne **nom** après "transformation". Par exemple si la musique recherchée par l'utilisateur est définie par la chaîne de caractères : *giVe LifE* , la requête sera la suivante.

Listing 3.1 – Exemple de requête de recherche

```
SELECT * FROM TITRE WHERE strpos(lower(
    replace(nom, ' ', '')), 'givelife') > 0;
```

3.3.2 Suggestion de titres

Pour la suggestion de titre, nous allons utiliser les écoutes saisies par l'utilisateur afin de lui proposer du contenu adapté à ces dernières. Nous allons supposer qu'un utilisateur sera susceptible d'aimer une musique si ce dernier a déjà écouté une musique de cet artiste ou encore s'il a écouté une musique dans le genre de cette musique. L'idée va donc être de récupérer les *titres* en base de données que l'utilisateur n'a pas encore écoutés et dont l'artiste ou le genre est présent dans les écoutes. On va donc dans un premier temps récupérer les écoutes de l'utilisateur à partir de son *pseudo*, puis extraire de ces dernières les genres et les artistes écoutés à l'aide d'une jointure intérieure avec la table titre. On obtient ainsi une matrice contenant les genres et artistes écoutés par l'utilisateur. En effectuant à nouveau une jointure intérieure avec le résultat obtenu et la table **Titre** sans les titres écoutés, on obtient **Titre**. L'idée est ensuite d'ordonner les résultats au hasard afin de proposer des suggestions différentes à chaque fois et de récupérer les dix premiers résultats.

Listing 3.2 – Exemple de requête de suggestion

```
SELECT * FROM (
        SELECT * FROM TITRE WHERE id NOT IN
            (
                SELECT titreid as id FROM
                    ECOUTE WHERE pseudoUser=
                    'alexis'
            )
) as titre
INNER JOIN (
        SELECT genre, nomartiste from titre
            INNER JOIN (
                SELECT titreId as id FROM
                    ECOUTE WHERE pseudoUser=
                    'alexis'
            ) as titresecoutes on titre.id =
                titresecoutes.id
) as res on titre.genre=res.genre OR titre.
    nomartiste=res.nomartiste ORDER BY
    RANDOM() limit 10;
```

Conclusion

Dans ce projet nous avons dans un premier temps effectué une analyse de l'objectif de l'application afin de déterminer les besoins de cette dernière. À partir de ces besoins, nous avons pu mettre en place un modèle Entité-Attribut représentant les relations, et cardinalités entre les entités et attributs de la bibliothèque musicale.Grâce à ce modèle entité-attribut, nous avons pu passer au modèle relationnel avant de concevoir puis réaliser la base de données en **PostgreSQL**. À partir de cette base de données nouvellement créée nous avons pu réaliser une application *Java* de bibliothèque musicale à l'aide du driver **JDBC** nous permettant de faire le liant entre une base de données et une application. De plus cette implémentation nous a permis la découverte du pattern **DAO** permettant le passage d'un modèle relationnel à un *modèle objet*. Néanmoins,il y a certaines problématiques qui n'ont pas été prises en compte dans le projet, comme le volume des données, ou encore une gestion de la concurrence intensive.