Programmation Orientée Objet
IG2I, LE3 – 2024-2025

TP1 – Requêtes sur une base de données - API JDBC

Maxime Ogier

Contexte et objectif du TP

Ce TP propose de mettre en place un système d'information pour la gestion d'une bibliothèque municipale. Il s'agit donc de gérer les informations associées aux livres de la bibliothèques, aux usagers, ainsi qu'aux emprunts de livre réalisés par les usagers. En particulier, nous étudions comment réaliser des requêtes sur une base de données depuis une application en POO.

Ce TP permet ainsi d'illustrer les notions suivantes :

- les diagrammes de classe ;
- les modèles logiques de données ;
- quelques requêtes SQL ;
- l'utilisation d'un serveur de données SQL dans une application Java ;
- l'API Java DataBase Connectivity (JDBC) ;
- les requêtes statiques et dynamiques ;
- l'exploitation du résultat des requêtes.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez, en particulier celles de l'API JDBC.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes).

1 Présentation de l'application

1.1 Description du contexte

La bibliothèque municipale de Trifouilly-les-Oies souhaite se doter d'une application permettant de gérer les emprunts de livres effectués par les usagers. Cette application sera utilisée par les bibliothécaires.

Pour réaliser cette application, la municipalité de Trifouilly-les-Oies a fait appel à des étudiants en informatique d'une école d'ingénieur.

1.2 Diagramme de classes UML

Les étudiants chargés de réaliser l'application commencent par réaliser un diagramme de classes UML. Pour ce faire, ils commencent par interroger les bibliothécaires de leur décrire les données à manipuler. Voici les explications qui leur sont fournies.

- La bibliothèque est contient de nombreux livres. Chaque livre est défini par un numéro ISBN qui l'identifie de manière unique. Ce numéro comporte 13 chiffres. Pour chaque livre, on souhaite également connaître son titre, son auteur, son éditeur, le nombre de pages qu'il contient ainsi que l'année de publication. La bibliothèque municipale de Trifouilly-les-Oies étant une petite structure, aucun livre n'est présent en plusieurs exemplaires, et donc on ne gérera pas cet aspect là.
- La bibliothèque est au service de ses usagers qui payent une cotisation annuelle. Chaque usager est défini par son nom, son prénom son adresse et sa date de naissance. On peut supposer que le triplet nom, prénom et adresse est unique. Aussi, on peut associer un identifiant unique à chaque usager.
- Les usagers de la bibliothèque peuvent venir consulter les livres sur place, ou bien les emprunter pour les lire chez eux. On souhaite ici gérer les emprunts.

Un emprunt concerne un livre et un usager, et il est également défini par une date d'emprunt et une date de rendu. On peut également associer un identifiant unique à chaque emprunt. Notez que la date d'emprunt correspond à la date à laquelle le livre est emprunté, et la date de retour correspond à la date à laquelle l'usager rapporte le livre. Ainsi, lors de l'emprunt la date de retour n'est pas renseignée. Ainsi, on peut connaître les livres qui sont actuellement empruntés en listant tous les emprunts pour lesquels la date de retour est vide.

À partir de cette description des données, un diagramme de classes est proposé dans la Figure 1, avec le formalisme UML.

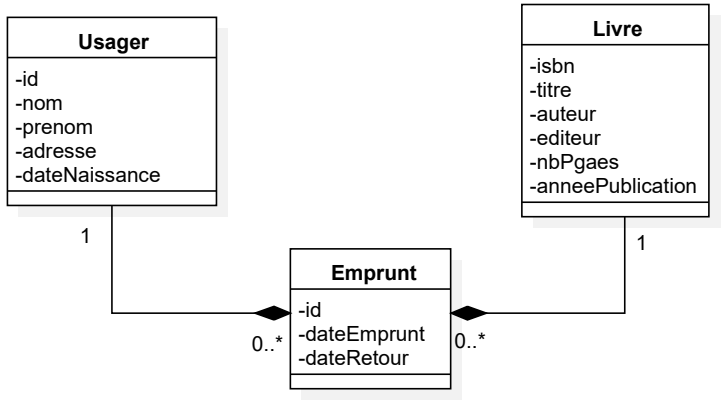

Figure 1: Diagramme de classes.

1.3 Modèle logique de données

À partir du diagramme de classes, un modèle logique de données relationnel est présenté dans la Figure 2. Regardez bien quelles sont les différence entre le diagramme de classes et ce modèle logique de données.

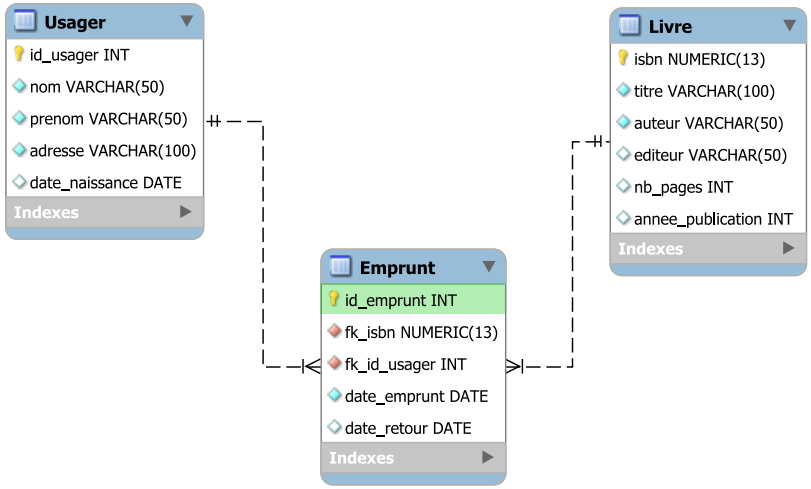

Figure 2: Modèle logique de données.

2 Base de données et connexion avec une application Java

À partir du modèle logique de données, le fichier `myBiblio.sql` propose un script de création de la base de données.

2.1 Mise en place de la base de données

Question 1. En utilisant le script SQL (pour un serveur de base de données MySQL) `myBiblio.sql` disponible sur Moodle, et en suivant les instructions proposées en Section 5.1, mettez en place la base de données pour l'application de gestion des clients en utilisant le logiciel DataGrip. Le script SQL propose déjà de remplir les tables avec quelques informations.

Question 2. Avec le logiciel DataGrip, exécutez à présent quelques requêtes SQL sur la base de données afin de fournir les informations suivantes :

- la liste des noms des usagers ;
- la liste des livres triée par ordre alphabétique du titre ;
- la liste des livres empruntés pas encore rendus.

Vous pouvez consultez la Section 5.1.2 pour plus d'informations sur l'exécution des requêtes sur la base de données.

2.2 Théorie : l'API JDBC et connexion à une base de données

2.2.1 L'API JDBC

L'API JDBC (Java DataBase Connectivity) permet de programmer en Java l'accès aux bases de données locales ou distantes. Pour qu'une application Java puisse accéder à une base de données, il suffit que le système de gestion de bases de données (SGBD) fournisse ce que l'on nomme un pilote JDBC, i.e. un ensemble de classes Java qui vont permettre l'utilisation du SGBD par le biais de requêtes SQL. Ainsi, il existe par exemple un pilote JDBC pour MySQL.

Notez qu'en général le pilote JDBC doit être importé dans les librairies de l'application.

2.2.2 Connexion à une base de données

Pour qu'une application Java puisse utiliser convenablement le pilote JDBC voulu, il existe un objet dit "gestionnaire de pilotes", qui est une instance de la classe **DriverManager**. Cet objet va gérer les différents pilotes des bases de données existantes.

Pour établir une connexion avec une base de données, il faut appeler la méthode statique **getConnection()** de la classe **DriverManager**. On peut appeler cette méthode en fournissant trois paramètres : l'url de la base de données, le nom d'utilisateur et le mot de passe. Cette méthode renvoie un objet de type **Connection** contenant toutes les informations nécessaires à l'utilisation ultérieure de la base de données si l'établissement de la connexion a pu s'opérer, et renvoie **null** sinon. Enfin, notez que la méthode **getConnection()** peut lever une exception de type **SQLException** si une erreur survient lors de l'accès à la base de données.

Ainsi, un exemple de connexion à un moteur de base de données MySQL est le suivant :

```
1  String urlBDD = "jdbc:mysql://localhost:3306/nomBDD";
2  String utilisateur = "nomUtilisateur";
3  String motDePasse = "motDePasse";
4  Connection connexionBdd = DriverManager.getConnection(urlBDD,
       utilisateur, motDePasse);
```

Notez que l'objet de type **Connection** sera utilisé ultérieurement pour effectuer des requêtes sur la base de données.

2.3 Connexion à la base de données

Question 3. Créez un nouveau projet **Bibliotheque**, et importez le pilote JDBC dans les librairies du projet (voir Section 5.2).

Question 4. Créez un paquetage **requete** et ajoutez-y une classe **RequeteBiblio**. Dans cette classe, ajoutez un attribut de type **Connection** et ajoutez un constructeur par défaut qui initialisera cet attribut en se connectant à la base de données. Testez le bon établissement de la connexion à la base.

Notez que la méthode **getConnection()** peut lever une exception de type **SQLException**, et faites bien à la manière dont vous gérez cette exception : est-il préférable de traiter l'exception dans un bloc **catch** ou bien de faire remonter l'exception à la méthode appelante ?

3 Création des classes du modèle objet

Dans cette section nous allons créer les classes qui découlent du diagramme de classe UML de notre application. Ces concepts ont déjà été étudiés dans le cours de LE2-IPO. Par conséquent, les questions ne seront pas détaillées.

Question 5. Créez un nouveau paquetage **modele** et ajoutez-y les classes correspondant au diagramme de classe UML présenté dans la Figure 1. Pour chaque classe, veillez notamment à avoir un constructeur par données, ainsi que de redéfinir les méthodes de la classe **Object** (**toString()**, **equals()** et **hashCode()**).

Faites bien attention qu'il s'agit ici de faire un modèle objet, et donc par conséquence dans la classe **Emprunt**, il doit y avoir des attributs de type **Livre** et **Usager**, et non pas des attributs qui correspondraient à des clés étrangère. Il faut donc bien distinguer le modèle objet d'un modèle relationnel.

4 Interrogation de la base de données

4.1 Théorie : requêtes statiques

Lorsque la connexion avec une base de données est établie, on peut ensuite exécuter des requêtes SQL sur la base de données afin d'accéder à certains champs de tous les enregistrements ou d'une partie des enregistrements d'une table.

Nous nous intéressons ici aux requêtes statiques pour récupérer des données depuis la base de données. Ici, le terme requête statique signifie que la requête ne dépend pas de paramètres externes et peut donc être compilée une seule fois, ce qui en améliore les performances.

Pour exécuter une requête statique sur un SGBD depuis une application Java, on peut suivre les étapes suivantes.

1. Écrire la requête SQL dans une chaîne de caractères.
2. Appeler la méthode **createStatement()** sur un objet de type **Connection** (celui que l'on a initialisé pour se connecter à la base de données). Cette méthode renvoie un objet de type **Statement**.

3. Sur l'objet précédemment créé, appeler la méthode **executeQuery(String requete)** à laquelle on fournit la requête SQL sous forme de chaîne de caractères. Cette méthode renvoie un objet de type **ResultSet** contenant les informations sélectionnées, et peut provoquer une exception de type **SQLException**.

L'objet de type **ResultSet** fourni par la méthode **executeQuery()** dispose de méthodes permettant de le parcourir enregistrement par enregistrement, à l'aide d'un curseur. La méthode **next()** fait progresser le curseur d'un enregistrement au suivant, et renvoie la valeur **false** s'il n'y a plus d'enregistrements. Au départ, le curseur est positionné avant le premier enregistrement. Il faut donc effectuer un premier appel à **next()** pour que le curseur soit bien positionné sur le premier enregistrement de l'objet résultat. Pour obtenir l'information d'un champ donné de l'enregistrement "courant", la classe **ResultSet** propose des méthodes dont le nom dépend du type de l'information concernée. Par exemple, la méthode **getString()** pour une chaîne de caractères, ou la méthode **getInt()** pour un entier. Ces différentes méthodes peuvent accéder à l'information d'un champ d'un enregistrement donné par son nom ou par sa position dans le résultat. Pour une meilleure lisibilité et maintenance du code, on préférera donner le nom. On notera aussi qu'il y a une correspondance entre les types SQL et les types Java.

Par ailleurs, les objets **Statement** et **ResultSet** étant consommateurs de ressources, il est préférable de les fermer explicitement en faisant appel à leur méthode **close()**.

Ainsi, voici un exemple de requête statique pour récupérer les noms, prénoms et revenus dans une table nommée 'client' :

```
1  String requete = "SELECT nom, prenom, revenu FROM client";
2  Statement statement = connexionBdd.createStatement();
3  ResultSet result = statement.executeQuery(requete);
4  while (result.next()) {
5     String nom = results.getString("nom");
6     String prenom = result.getString("prenom");
7     int revenu = result.getInt(revenu);
8     // ...
9  }
10 result.close();
11 statement.close();
```

4.2 Quelques exemples de requêtes statiques

Question 6. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie la liste des livres triée par ordre alphabétique du titre. Testez que cette méthode fonctionne correctement.

À partir du résultat de la requête, il vous faudra donc créer des objets de type **Livre** et les ajouter dans une liste.
Pensez à gérer convenablement les exceptions.

Question 7. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie la liste des usagers, triée par ordre alphabétique du nom, puis du prénom en cas d'égalité des noms. Testez que cette méthode fonctionne correctement.

Question 8. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie l'ensemble des livres qui sont actuellement empruntés. Testez que cette méthode fonctionne correctement.

Faites bien attention de la structure de données utilisée pour renvoyer un ensemble. Ici on ne souhaite pas faire de tri particulier.

4.3 Théorie : requêtes dynamiques

Nous avons vu comment traiter les requêtes statiques, nous allons voir à présent comment faire avec des requêtes dynamiques, i.e. des requêtes dont les clauses sont fournies à l'exécution. C'est le cas des requêtes paramétrées en fonction d'une valeur entrée par l'utilisateur. Une idée naturelle serait de modifier la chaîne de caractères qui contient la requête en y insérant les paramètres nécessaires. **Ceci n'est pas une bonne idée !!** En effet, le code est alors exposé aux injections SQL (faille de sécurité qui permet l'injection d'une requête SQL non prévue par le système et qui peut compromettre sa sécurité). En effet, il n'y a pas de contrôle sur les paramètres rentrés par l'utilisateur.

Pour pallier ce problème, l'API JDBC permet l'utilisation de requêtes préparées au travers de l'interface **PreparedStatement**. Lors de l'écriture de la requête SQL dans une chaîne de caractères, les paramètres dont la valeur sera fixée par la suite sont remplacés par le symbole '?'. À la différence d'une requête usuelle, une requête préparée est transmise au SGBD à l'aide de la méthode **prepareStatement(String requete)** de la classe **Connection**. Cette méthode prend en argument la requête SQL sous forme de chaîne de caractères. Il est ensuite possible de préciser les valeurs des paramètres à l'aide de méthodes de la classe **PreparedStatement**. Il s'agit de méthodes nommées **setXXX()** où **XXX** correspond au type de donnée du paramètre. Ces méthodes prennent en argument le rang du paramètre concerné (1 pour le premier) ainsi que la valeur du paramètre.

En plus d'éviter les injections SQL, l'utilisation de requêtes préparées est plus performante en terme de temps d'exécution si l'on souhaite exécuter plusieurs fois une même requête (avec peut-être des paramètres différents). En effet, avec un objet **PreparedStatement**, la requête est envoyée au SGBD et compilée dès la création de l'objet ; et lorsque le **PreparedStatement** est exécuté, le SGBD n'a alors plus qu'à réaliser l'exécution sans avoir besoin de compiler la requête au préalable.

Ainsi, voici un exemple de requête dynamique pour récupérer les noms, prénoms et revenus des clients dont le nom est égal à un paramètre nommée **nomClient** :

```
String requete = "SELECT nom, prenom, revenu FROM client "
      + "WHERE UPPER(nom) = ?";
PreparedStatement prepStatement =
     connexionBdd.prepareStatement(requete);
prepStatement.setString(1, nomClient.toUpperCase());
ResultSet result = prepStatement.executeQuery();
while (result.next()) {
   String nom = results.getString("nom");
   String prenom = result.getString("prenom");
   int revenu = result.getInt(revenu);
   // ...
}
result.close();
prepStatement.close();
```

On remarquera que pour éviter les problèmes de casse dans les chaînes de caractères, on peut utiliser la fonction **UPPER()** du langage SQL et la méthode **toUpperCase()** de la classe **String**.

4.4 Quelques exemple de requêtes dynamiques

Question 9. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie la liste des livres contenant dans leur titre, leur auteur ou leur éditeur une chaîne de caractères qui est donnée en paramètre de la méthode. Testez que cette méthode fonctionne correctement.

Question 10. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie la liste des usagers contenant dans leur nom ou leur prénom une chaîne de caractères qui est donnée en paramètre de la méthode. Testez que cette méthode fonctionne correctement.

Question 11. Dans la classe **RequeteBiblio**, ajoutez une méthode qui renvoie la liste de tous les emprunts qui sont en retard. On supposera qu'un livre est en retard s'il a été emprunté il y a (strictement) plus de 20 jours et qu'il n'est pas encore été retourné. Testez que cette méthode fonctionne correctement.

Faites en sorte que votre code soit évolutif, et que le nombre de jours provoquant un retard puisse facilement être modifié.
Par ailleurs, observez bien ici la manière dont on doit écrire la requête avec un langage relationnel pour pouvoir récupérer les informations dans un modèle objet.

4.5 Théorie : requêtes de mise à jour

L'API JBC permet également de réaliser des requêtes de mises à jour d'une table (insertion, suppression, modification d'enregistrements). Si l'on souhaite réaliser des requêtes de mise à jour sur la base de données (**INSERT**, **UPDATE**, **DELETE**), la méthode **executeQuery()** de l'interface **Statement** ne peut plus être utilisée, et il faut alors utiliser la méthode **executeUpdate()** qui renvoie un entier qui indique le nombre d'enregistrements affectés par la requête (ou 0 en cas d'utilisation d'une instruction LDD (langage de définition de données)).

Ainsi, voici un exemple de requête de mise à jour (pour une requête statique) :

```
String requete = "INSERT INTO client(nom, prenom) VALUES
     ('dupont', 'marta')";
Statement statement = connexionBdd.createStatement();
int nbRows = statement.executeUpdate(requete);
System.out.println(nbRows + " enregistrement(s) ajouté(s)");
statement.close();
```

4.6 Quelques exemples de requêtes de mise à jour

Question 12. Dans la classe **RequeteBiblio**, ajoutez une méthode qui permet de rendre un livre passé en paramètre de la méthode. Testez que cette méthode fonctionne correctement.

Question 13. Dans la classe **RequeteBiblio**, ajoutez une méthode qui permet d'emprunter un livre par un usager, tous deux étant passés en paramètre de la méthode. Testez que cette méthode fonctionne correctement.

5 Quelques notions utiles

5.1 Envoyer des requêtes sur un serveur de base de données avec DataGrip

DataGrip est une application cliente qui permet d'envoyer des requêtes sur un serveur de bases de données. Nous supposons ici que nous utilisons un serveur de base de données MySQL. Assurez-vous d'avoir bien démarré votre serveur de base de données.

5.1.1 Création d'une base de données

Afin de créez une nouvelle base de données sur DataGrip, voici les étapes à suivre :

1. cliquez sur 'File', puis 'New', puis 'Data Source', puis 'MySQL' ;
2. dans la fenêtre qui vient de s'ouvrir, si vous voyez en bas un message d'avertissement avec écrit 'Download missing data files', cliquez sur le texte 'Download' afin de procéder au téléchargement du driver de connexion à un serveur MySQL ;
3. toujours dans cette même fenêtre, remplissez les informations pour vous connecter au serveur MySQL : le port par défaut 3306 est à modifier si besoin, et il faut rentrer les informations de l'utilisateur et du mot de passe ; puis cliquez sur 'Test connection' pour vérifier qu'il n'y a pas de soucis, puis sur 'OK' ;
4. vous devez à présent voir dans l'onglet sur la gauche la connexion avec votre serveur MySQL ;
5. faites un clic droit sur le nom de la connexion, puis sélectionnez 'New', puis 'Query console' : vous pouvez alors écrire des requêtes en SQL et les envoyer sur le serveur ;
6. pour créer une nouvelle base nommée 'nom_bdd', écrivez le code suivant dans la console puis exécutez-le :

```
1 CREATE DATABASE nom_bdd;
```

5.1.2 Requêtes SQL sur le serveur de base de données

Une fois que vous créez une nouvelle base de données, toujours sur DataGrip, vous pouvez effectuer des requêtes SQL sur cette base de données. Pour ce faire, suivre les instructions suivantes :

1. dans la fenêtre 'Database Explorer' (en principe en haut à gauche), faites un clic droit sur le nom de la base de données, puis cliquez sur 'New', puis sur 'Query Console' ;
2. vous pouvez maintenant écrire vos requêtes dans la console, puis les exécuter en cliquant sur la flèche verte (ou raccourci *Ctrl + Entrée*.

5.2 Importer un pilote JDBC dans un projet sous IntelliJ IDEA

En principe, le pilote JDBC pour MySQL a été téléchargé au travers de DataGrip lorsque vous avez fait la connexion avec le serveur de base de données (voir Section 5.1.1). Ainsi, vous pouvez récupérer le chemin d'accès au fichier jar du pilote MySQL en suivant les étapes suivantes :

1. dans le logiciel DataGrip, allez dans l'onglet 'Database Explorer' (en principe sur la gauche), puis faites un clic droit sur le nom de la connexion à votre base de données, et cliquez sur 'Properties' ;
2. dans la fenêtre qui vient de s'afficher, vous devez voir dans l'onglet général 'Driver: MySQL', cliquez alors sur le lien 'MySQL', puis sur 'Go to Driver' ;
3. dans la fenêtre qui vient de s'ouvrir, dans la partie 'Driver Files', cliquez sur le signe '+', puis sur 'Custom JARs...' ;
4. dans la fenêtre qui vient de s'ouvrir, développez le contenu du répertoire 'jdbc-drivers', puis du répertoire 'MySQL ConnectorJ', puis sélectionnez le fichier jar qui correspond au connecteur ;
5. dans la barre en haut de la fenêtre, copiez le chemin vers le fichier jar.

À présent, il reste à importer le pilote JDBC dans votre projet Java. Pour ce faire, voici les étapes à suivre :

1. cliquez sur 'File', puis sur 'Project Structure...' ;
2. dans la fenêtre qui vient de s'ouvrir, dans l'onglet sur la gauche, cliquez sur 'Libraries' ;
3. cliquez en haut à gauche sur le signe '+', puis sur 'Java' ;
4. dans la fenêtre qui vient de s'ouvrir, coller le lien vers le fichier jar du connecteur dans la barre d'adresse en haut, puis cliquez sur 'OK' ;
5. cliquez sur 'OK' pour valider les modifications.