Atelier de gestion de projet

Rapport de projet : BDA

~

Système intelligent de création d'offres de voyage
(Les Petites Antilles)

Auteurs

Valentin BELYN
Vincent ARCHAMBAULT
Arnaud SERY
Benoit CONS
Thomas RE
Loïc TRAMIS

Université de Cergy-Pontoise
Master 1 IISC 2018-2019

Remise du rapport le 18/01/2019
UE : AGP

SOMMAIRE

TABLE DES ILLUSTRATIONS

1. Introduction

Dans le cadre de l'atelier de gestion de projet organisé lors de la semaine du 14 au 18 janvier 2019, nous devions concevoir et implémenter un système permettant de générer des offres de voyages en fonction de paramètres fournis par l'utilisateur.

Pour y parvenir, nous nous sommes basés à la fois sur une base textuelle et sur une base de données relationnelle. Une API a été conçue à cette occasion pour extraire et lier les données issues des deux sources. Le présent rapport décrit cette partie de la conception.

2. Structure de la base de données

2.1 Modèle conceptuel des données (MCD)

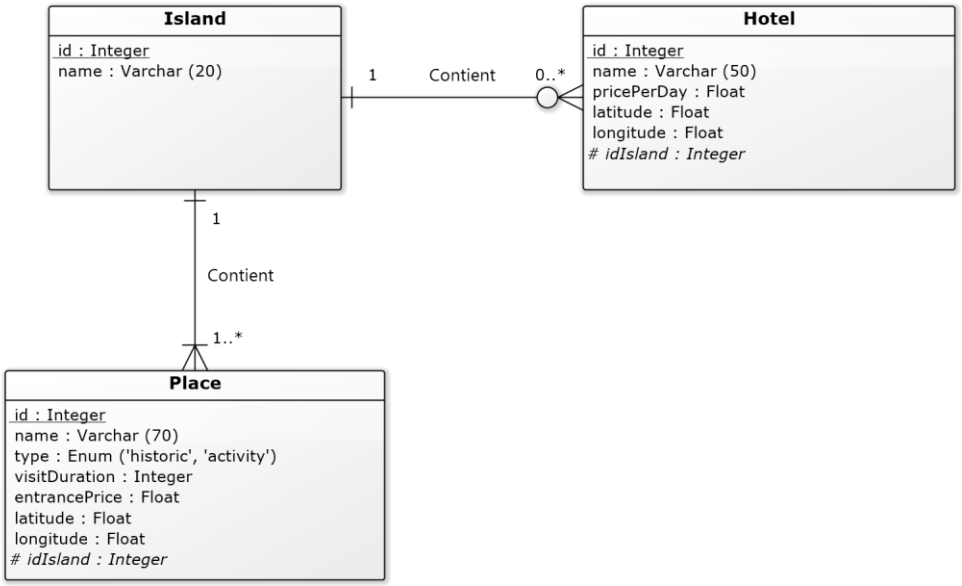

Figure 1 Modèle conceptuel des données

Nous nous sommes basés sur trois tables pour répondre à la problématique : Island, Hotel et Place.

Chaque lieu (Hotel ou Place) a une clé étrangère avec l'identifiant de l'île sur laquelle le lieu est situé. Les coordonnées de chaque lieu sont également stockées afin de calculer dynamiquement les distances entre les sites dans la couche métier.

La table Island est quant à elle essentielle pour déterminer quel transport utiliser pour aller d'un site à un autre. Par exemple, si l'île du lieu de destination est différente de celle du lieu source, le moyen de transport sera le bateau et non le bus.

2.2 Modèle logique des données (MLD)

Island (id, name)
Place (id, name, type, visitDuration, entrancePrice, latitude, longitude, *#idIsland*)
Hotel (id, name, pricePerDay, latitude, longitude, *#idIsland*)

3. Conception de l'API

3.1 La couche persistance

Notre API est constituée d'un paquetage « persistence », ainsi que de plusieurs sous-paquetages :

- **persistence** : à ce niveau, nous disposons de l'ensemble des classes permettant de faire le lien entre la couche métier et notre API de bases de données étendue.

- **persistence.extendeddb** : ce niveau contient les classes permettant d'exploiter les deux sous-systèmes JDBC et Lucene. Il s'agit d'une façade et permet de simplifier les recherches (gestion en un endroit unique). Cette façade transmet ensuite les requêtes aux sous-paquetages appropriés.

- **persistence.extendeddb.lucene** : ce package contient les classes relatives aux bases de données textuelles. Il est notamment possible d'effectuer une recherche textuelle et de créer un index avec ajout de documents.

- **persistence.extendeddb.jdbc** : ce package contient les classes permettant de manipuler notre base de données relationnelle.

Nous avons fait le choix de JDBC par rapport à Hibernate car il nous était demandé que notre API supporte des requêtes SQL classiques, avec éventuellement une clause « WITH » de précisée. Hibernate utilise une syntaxe très puissante mais qui lui est propre : la syntaxe HQL. Nous avons donc construit l'API autour des bibliothèques JDBC et Lucene.

Voici une vue globale de ce système :

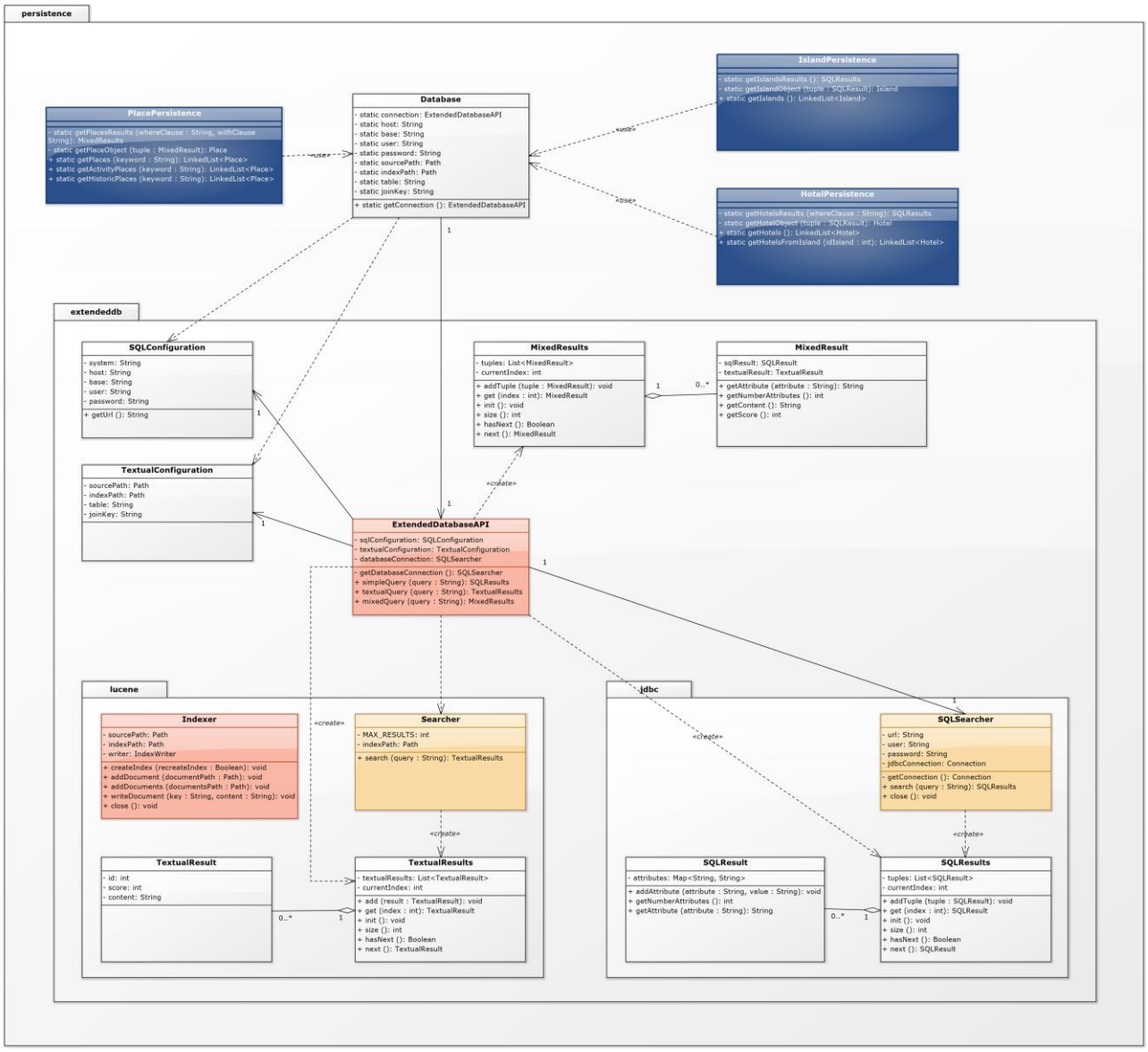

Figure 2 Vue globale du système (diagramme de classes)

Nous détaillerons chaque partie de l'API dans les prochains chapitres.

3.2 La partie Lucene

Lucene est une bibliothèque open source écrite en Java qui permet d'indexer et de chercher du texte. Il est utilisé dans certains moteurs de recherche.

Voici le sous-package dédié à Lucene :

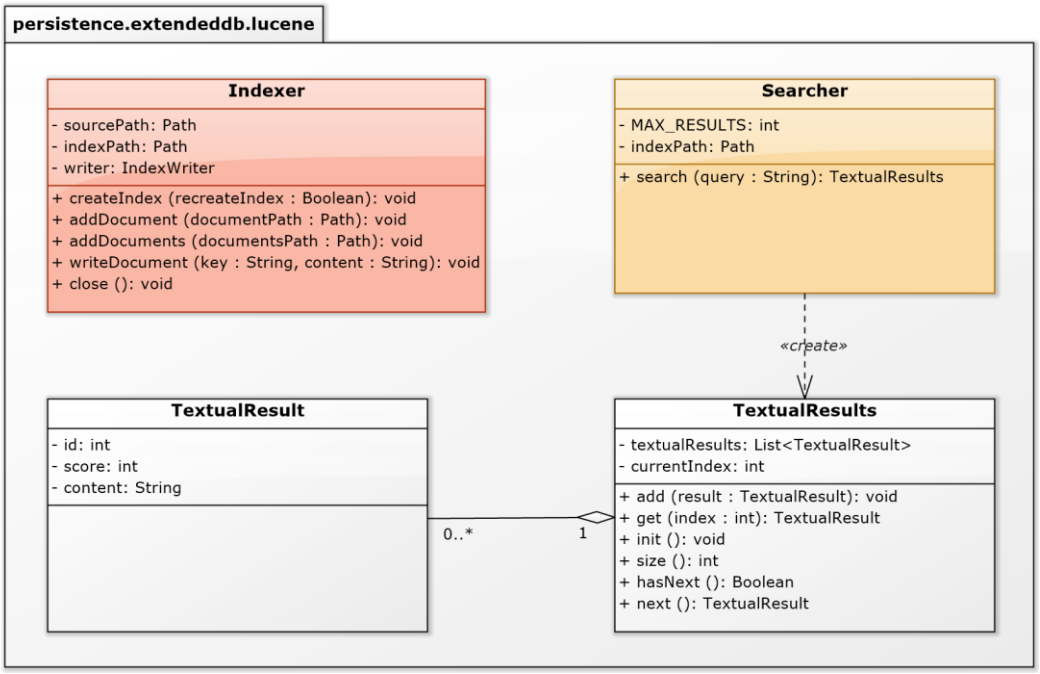

Figure 3 Sous-package Lucene (digramme de classes)

Nous avons implémenté quatre classes dans cette partie.

Nous y trouvons notamment la classe Indexer qui utilise également le design pattern Façade (masque l'utilisation des mécanismes d'indexation de Lucene) et permet de créer des documents dans un répertoire que l'on aura spécifié dans sa configuration, de créer un index qui les référencera et de les indexer.

Nous disposons d'une classe Searcher pour réaliser des recherches textuelles sur la base. Les résultats d'une requête sont contenus dans une instance de TextualResults qui implémente l'interface Interable de Java, ainsi que les méthodes init(), next() et hasNext() pour parcourir les résultats. Chaque résultat extrait se présente dans une structure TextualResult qui contient trois éléments : la clé utilisée pour la jointure (qui est le nom du fichier), le score du résultat et le contenu du fichier (la description d'un lieu à visiter, donc). Les résultats sont triés sur le score, en ordre décroissant.

3.3 La partie JDBC

JDBC est une bibliothèque également écrite en Java et permettant d'exploiter des bases de données relationnelles.

La même structure que pour la précédente partie a été utilisée afin de conserver une certaine cohérence.

Voici le sous-package de la partie JDBC :

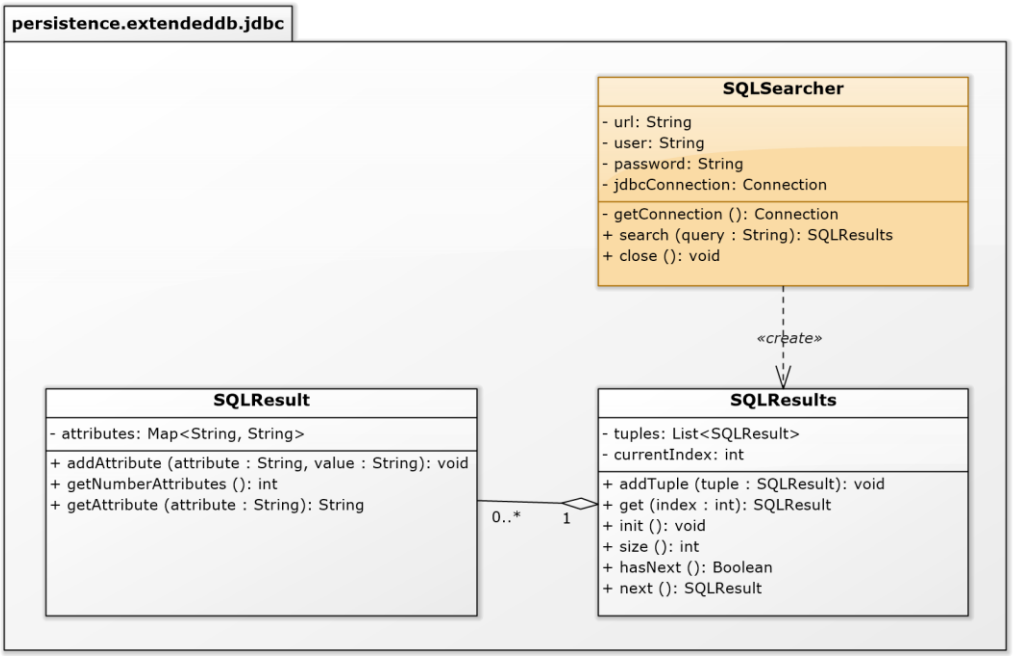

Figure 4 Sous-package JDBC (digramme de classes)

SQLSearcher permet d'effectuer des requêtes sur la base de données relationnelle. Celle-ci utilise les design patterns Façade et Lazy Singleton (ce dernier pattern permet de sauvegarder la connexion JDBC à la base en vue de la réutiliser pour plusieurs requêtes, économisant ainsi un temps important). Cette classe retourne pour chaque requête un objet SQLResults qui implémente les itérateurs Java.

3.4 Le cœur du système

L'utilisation des deux sous-packages précédents est simplifiée par la classe ExtendedDatabaseAPI. Cette dernière délègue les requêtes SQL et textuelles aux sous-packages. En plus de méthodes permettant de réaliser les requêtes SQL et textuelles, elle dispose aussi d'une méthode permettant de réaliser des requêtes mixtes. Cette méthode effectue alors les deux requêtes séparément et effectue une jointure sur la base de la clé de jointure spécifiée lors de l'instanciation de la classe. Les résultats sont toujours triés par ordre décroissant sur le score.

La partie du système concernée :

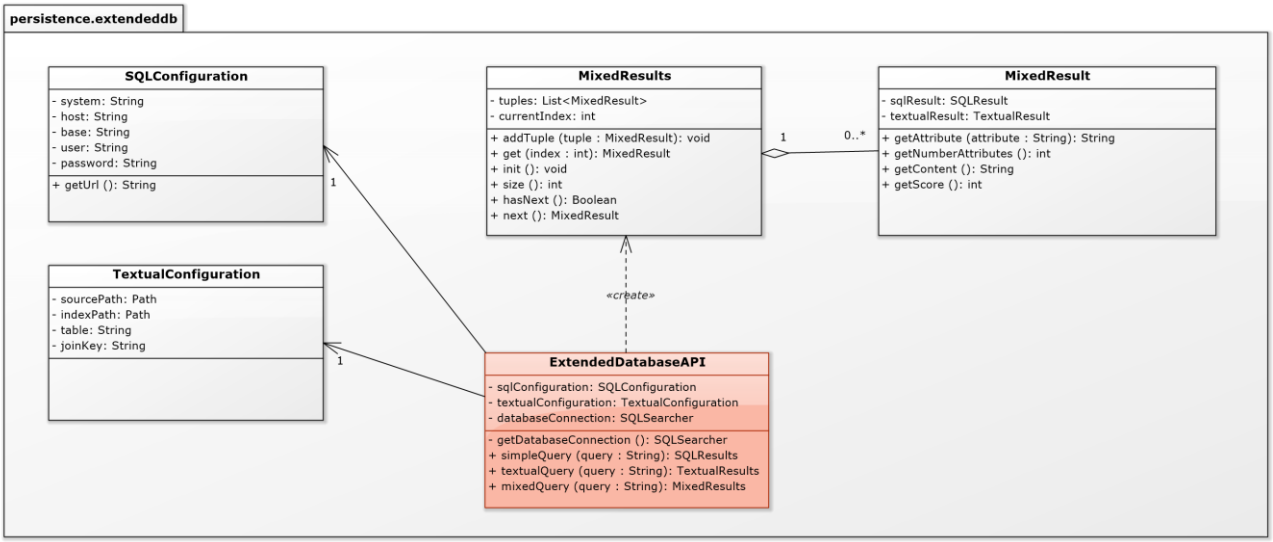

Figure 5 Le coeur du système (diagramme de classes)

Le constructeur de la classe ExtendedDatabaseAPI prend en paramètre deux objets : SQLConfiguration et TextualConfiguration. Ces objets peuvent être configurés de telle manière que l'on puisse préciser :

- Les moyens de connexion à la base de données relationnelle (utilisateur, mot de passe, base de données et type de base).
- Les paramètres pour la recherche textuelle (le chemin d'accès aux données, à l'index, la table concernée par la jointure et le nom de l'attribut qui sera utilisé pour la jointure).

Les résultats des requêtes mixtes retournés par la méthode mixedQuery(query) sont stockés dans une classe MixedResults qui arbore une structure similaire à TextualResults et SQLResults.

4. Comparaison des deux types de plan

4.1 Premier plan

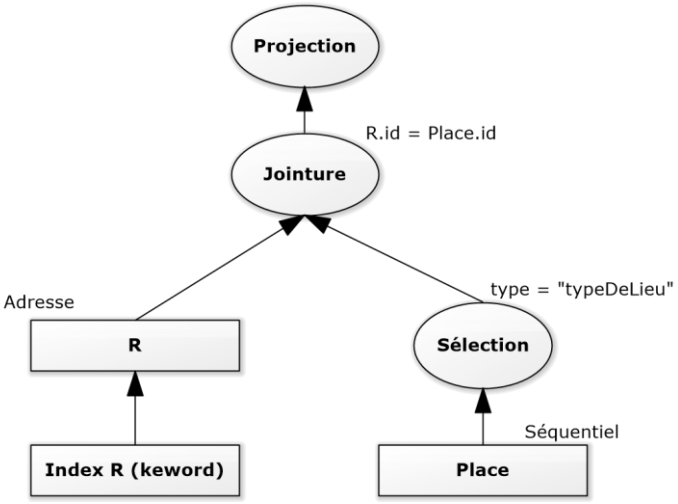

Figure 6 Comparatif : plan d'éxécution 1

Pour ce premier plan, nous effectuons les requêtes SQL et textuelles simultanément et nous fusionnons les résultats ayant la même valeur au niveau de l'attribut de jointure. Afin d'éviter de trier les données après la jointure (ce qui représente des calculs supplémentaires), nous parcourons d'abord les résultats textuels et pour chaque valeur récupérée, nous recherchons dans les résultats SQL s'il y a la même valeur pour la clé. Dans ce cas, la jointure est réalisée, sinon le résultat n'est pas pris en compte.

Une requête textuelle prend en moyenne 5 ms et une requête SQL 7 ms. Le temps minimum est donc de 12 ms (5 ms + 7 ms).

4.2 Second plan

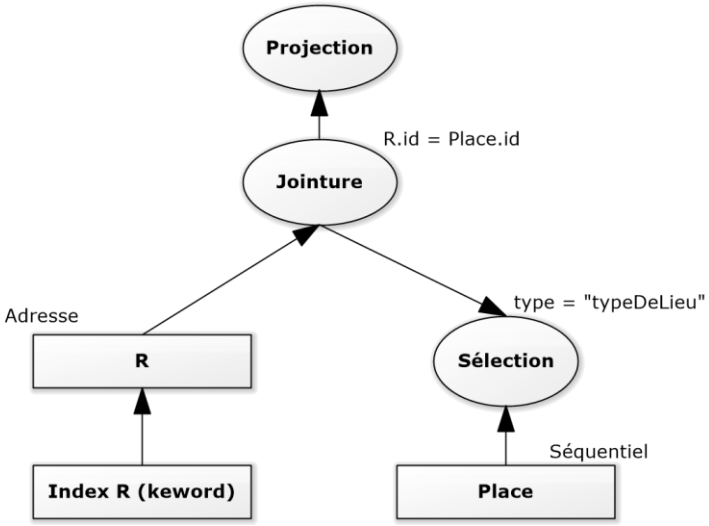

Figure 7 Comparatif : plan d'éxécution 2

Le second plan effectue une requête textuelle et récupère tous les identifiants. Pour chaque identifiant récupéré, une requête SQL est exécutée afin de récupérer la donnée correspondante dans la base de données relationnelle. Cela est donc plus coûteux.

L'implémentation n'étant pas demandée, nous avons estimé le temps d'exécution :

- Une requête textuelle prend en moyenne 5 ms et une requête SQL 7 ms.
- Avec la requête mixte demandant tous les lieux avec le mot clé musée (SELECT name FROM Place WITH musée), nous obtenons 6 résultats.

Le temps d'exécution est donc : 5 ms (requête mixte) + 6 (résultats/identifiants récupérés) * 7 ms = 47 ms.

Ce plan est donc ici quatre fois moins performant et avec un nombre de données plus conséquent, la différence de temps de calcul aurait été exponentielle.

4.3 Améliorations possibles

Les résultats des requêtes SQL étant volumineux, nous aurions pu utiliser le second plan en remplaçant chaque accès à la base de données par un seul accès. En effet, pour chaque résultat de la requête textuelle, nous aurions pu récupérer les identifiants et demander à la base de données de récupérer toutes les tuples contenant cette liste d'identifiants. Cela pourrait se traduire par la requête :

SELECT * FROM Place WHERE id IN (identifiants)

Où « identifiants » est les identifiants récupérés par la requête textuelle.

Une autre solution (pour réduire le temps de recherche seulement) aurait été d'utiliser du multithreading pour récupérer les résultats dans les deux bases en parallèle.