



|  |  |  |  |
| --- | --- | --- | --- |
| DIALLO Alpha Abdourahamane |  | ETOU GALLOU |  |
|  |  |  |  |
|  |  |  |  |  |

RAPPORT MINI PROJET

23/03/2017

Mise en pratique de ORM Hibernate et reponses aux questions posées dans le sujet du projet

INTRODUCTION

Dans la première partie du TP nous avons donc eu un premier aperçu de JDBC. Dans ce projet nous allons mettre à profit le Framework Hibernate.

Hibernate dans la même catégorie qu’OJB, [Apache Torque](https://fr.wikipedia.org/w/index.php?title=Apache_Torque&action=edit&redlink=1), [SimpleORM](https://fr.wikipedia.org/w/index.php?title=SimpleORM&action=edit&redlink=1) est une solution open source de type **ORM** (Object Relational Mapping) qui permet de faciliter le développement de la couche persistance d'une application. ORM est une solution qui convertit automatiquement, *à la demande*, la base de données sous forme d’un *graphe d’objet*. L’ORM s’appuie pour cela sur une *configuration* associant les *classes* du modèle fonctionnel et le schéma de la base de données. L’ORM génère des requêtes SQL qui permettent de matérialiser ce graphe ou une partie de ce graphe en fonction des besoins. Hibernate permet par conséquent de représenter une base de données en objets Java et vice versa.

Hibernate, comme tous les autres outils de mapping objet/relationnel, nécessite des métadonnées qui régissent la transformation des données d'une représentation vers l'autre (et réciproquement). Dans Hibernate 2.x, les métadonnées de mapping sont pour la plupart du temps déclarées dans des fichiers XML. Les mêmes annotations et plus sont maintenant disponible avec le JDK standard, quoique plus puissant et mieux pris en charge par les outils qui suivent : IntelliJ IDEA et Eclipse, par exemple, prennent en charge la complétion automatique et la coloration syntaxique des annotations du JDK 5.0. Les annotations sont compilées en bytecode et lues au moment de l'exécution (dans le cas d'Hibernate, au démarrage) en utilisant la réflexion, ce qui rend non nécessaires les fichiers XML externes.

|  |  |
| --- | --- |
| ****Monde objet**** | **Monde relationnel** |
| Classe | Table |
| Attribut | Colonne |
| Instance | Enregistrement |
| Association | Clé étrangère |

MÉTADONNÉES

Hibernate, comme tous les autres outils d’ORM objet / relationnel, nécessite des métadonnées qui régissent la transformation des données d'une représentation à l'autre. Hibernate Annotations fournit des métadonnées de mappage basées sur des annotations depuis JDK 5 au lieu de fichier XML.

Les annotations JPA sont dans le **javax.persistence.\* .** Chaque classe persistante est une entité et est déclarée à l'aide de l'annotation **@Entity** (au niveau de la classe). **@Table** est définie au niveau de la classe; Il permet de définir les noms de table, et ses propriétés pour le mappage de table de la base de données.

Si aucune **@Table** n'est définie, les valeurs par défaut sont utilisées (full qualifie Name de la classe). La classe d'entité doit avoir **un constructeur public ou protégé sans argument**. Il peut également définir des constructeurs supplémentaires. Il est recommandé d’implémenter les méthodes **equals** () et **hashCode** () pour ne pas avoir plusieurs instance d’un même objet.

Chaque entité doit définir un identifiant, il peut être simple (valeur unique, un seul attribut de base) et sera annoté par **@Id** ou composite (plusieurs valeurs) représenté par une «classe de clé primaire» annoté **@EmbeddedId** comme pour la clé métier de bureau de vote identifiée par l’id de bureau et une ville, ou définie à l'aide de l'annotation **@IdClass**.

La classe de clé primaire doit être publique et doit avoir un constructeur public sans arguments et sérialisable. Des valeurs pour des identificateurs simples peuvent être générées. Pour indiquer qu'un attribut d'identificateur est généré, il est annoté avec **@GeneratedValue** et son paramètre le plus important est ***GenerationType*** qui permet d’indiquer comment les valeurs sont générées.

Pour les attributs de type de base, la règle de nommage implicite est que le nom de colonne est le même que le nom d'attribut. Si cette règle de implicite ne répond pas aux exigences, Il est possible explicitement indiquer à Hibernate le nom de la colonne à utiliser avec **@Colonne** pour associer explicitement l'attribut à la colonne.

**@Embeddable** définit une composition de valeurs. Par exemple, nous avons une classe d'**Adresse** qui est une composition de rue, code postal, etc. Son de type de valeur, son cycle de vie est lié à un type d'entité parent bureau de vote, héritant ainsi de l'accès d'attribut de son parent. Hibernate définit deux termes pour travailler avec un type incorporable: **@Embeddable** et **@Embedded**.

**@Embeddable** est utilisé pour décrire le type de mappage lui-même (classe Adresse). **@Embedded** sert à référencer un type incorporable dans la classe fille ou utilisatrice (classe Bureau Vote).

Hibernate met en place des associations qui décrivent comment deux ou plusieurs entités forment une relation basée sur une sémantique de jointure de base de données (**@OneToOne, @OneToMany, @ManyToOne, @ManyToMany)**.

L'association **@OneToOne** peut être unidirectionnelle ou bidirectionnelle. Une association unidirectionnelle suit la sémantique de la clé étrangère de la base de données relationnelle, c'est-à-dire que le client est propriétaire de la relation. Une association bidirectionnelle comporte un côté parent ***mappedBy*** qui doit référencer le champ qui porte la relation côté du parent. Lors de l'utilisation d'une association **@OneToOne**, Hibernate applique la contrainte unique lors de la récupération de l'enfant. S'il y a plus d'un enfant associé au même parent, Hibernate lancera un ***ConstraintViolationException***.

Les associations possèdent le paramètre ***FetchType*** (par défaut ***EAGER***) qui définit si l’attribut doit être récupéré directement avec la requête ou à la demande. ***EAGER*** impose que la valeur doit être récupérée lorsque le propriétaire de cet attribut est récupéré, tandis que ***LAZY*** suppose simplement que la valeur est extraite lorsque cet attribut est appelé directement.

Outre les annotations ***FetchType.LAZY*** ou ***FetchType.EAGER,*** Il est également possible d’utiliser l'annotation **@Fetch** spécifique à Hibernate et pas JPA qui accepte les stratégies suivantes:  
**SELECT** : charge toujours toutes les collections et entités avec un select secondaire pour chaque entité, collection ou de jointure individuelle.  
**JOIN** : utilisez une jointure externe pour charger les entités associées, les collections ou les jointures.  
**SUBSELECT** : uniquement pour les collections, charge la collection dans une instruction de sous-sélection.

L'annotation **@JoinColumns** est utilisée pour regrouper plusieurs annotations **@JoinColumn**, qui sont utilisées lors de l'association de mapping ou d'une collection **Embedble** à l'aide d'un identificateur composite comme dans le cas du **Bureau de vote** dans la classe **Electeur**.

L'annotation **@LazyCollection** est utilisée pour spécifier le comportement d'extraction paresseuse d'une collection donnée. Les valeurs possibles sont données par l'énumération LazyCollectionOption:  
**TRUE** : charger à la demande correspond en JPA à ***FetchType.LAZY***.  
**FALSE** : charge la collection entièrement la collection correspond en JPA ***FetchType.EAGER***.  
**EXTRA** : préférer des requêtes supplémentaires sur le chargement de la collection complète. La valeur EXTRA n'a pas d'équivalent dans la spécification JPA et elle est utilisée pour éviter de charger la collection entière même si la collection est consultée pour la première fois (**LAZY** ou **EAGER**). Chaque élément est extrait individuellement à l'aide d'une requête secondaire et fonctionne uniquement sur des collections ordonnées (annotées avec **@OrderColumn**)

Les deux premières valeurs sont maintenant obsolètes car il est essentiel d’utiliser l'attribut ***FetchType*** JPA de la collection **@ElementCollection**, **@OneToMany** ou **@ManyToMany**.  
  
La valeur EXTRA n'a pas d'équivalent dans la spécification JPA et elle est utilisée pour éviter de charger la collection entière même si la collection est consultée pour la première fois. Chaque élément est extrait individuellement à l'aide d'une requête secondaire.

SESSION DAO

## DAO

Les opérations de base pour la persistance des données sont presque toutes identiques quelques soit l’objet à sauvegarder, les méthodes en question sont celles d’un CRUD : enregistrement (Create), lecture (Read), mise à jour (Update) et suppression (Delete).



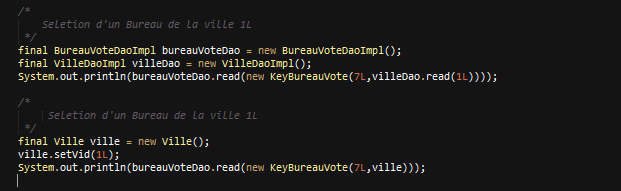
La plupart du temps ces méthodes sont répétées et redéfinies dans chacun des DAO de notre application, la seule différence notable est le type des objets que ces méthodes prennent en paramètre, cependant avec Java 5 et les types génériques pourquoi ne pas les écrire une fois pour toutes dans un DAO générique (bonne pratique).

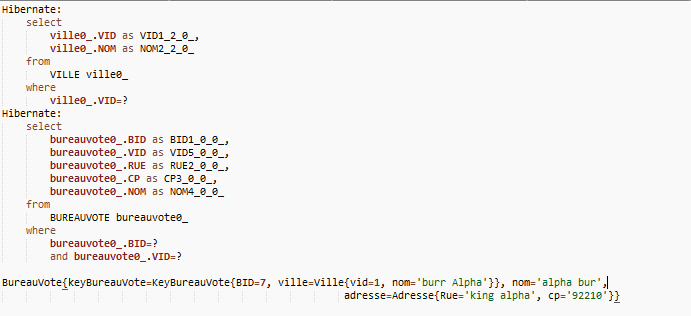
## SESSION

L’interface est la même que celle utilisée en JDBC mais le contenu des méthodes générique changent notamment avec l’arrivée d’un nouvel objet la SessionFactory. La sessionFactory est le concept que l’on peut voir comme une fabrique de session de données unique construit au démarrage et thread-safe dont les propriétés définissent l’environnement. La sessionFactory crée les sessions qui résument la notion de service et permet d'offrir des opérations de création, de lecture et de suppression etc., et plus complexes (une centaine) pour les instances de classes d'entités mappées. Hibernate génère le SQL qui correspond aux opérations (session.get (**persistentClass**, ID) et le dit SQL select converti la ligne en objet et le retourne). Concrètement au lieu d’écrire la requête en SQL la création d’un objet et récupérer le résultat avant de le transformer en objet java correspondant à la classe (35 lignes de code), la session nous met à disposition une méthode **create** qui le fait a notre place en une ligne.

REQUÊTE CRUD

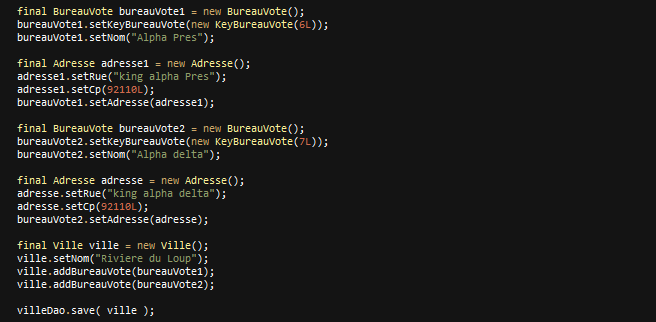
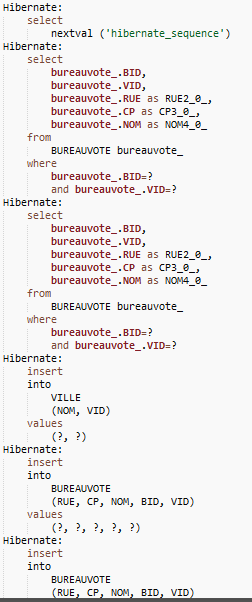
## READ Bureau de vote



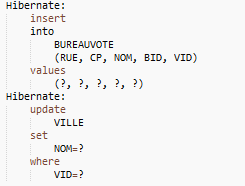


La première requête (R1) est générée pour récupérer l’ID de la ville 1 au travers du son DAO (villeDao) puis avec le second select (R2) permet de récupérer le bureau de vote qui correspond à l’ID ville. Dans la seconde lecture (R1) n’est pas exécuté de ce fait, l’ID de la vill seul sera accessible sans les autres informations.

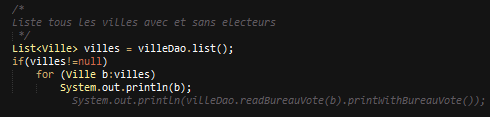
## SAVE VILLE WITH 2 BUREAU DE VOTE

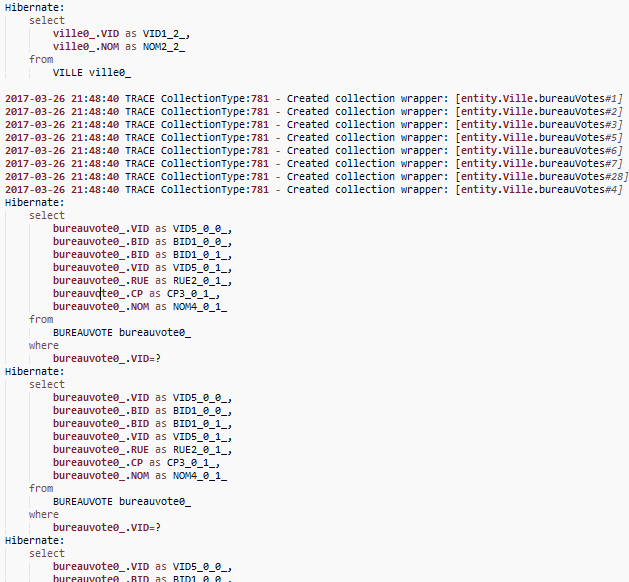


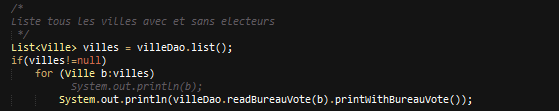
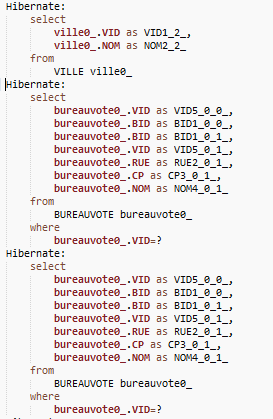
Hibernate commence par créer une séquence pour la ville VID (R1) , nous verifions ensuite on verifie l’existence des bureaux de votes (R2, R3) : si elles n’existent pas, nous les créons (R5, R6) après avoir créé une ville (R4), si elles existent les requêtes (R5, R6) ne sont pas générées. Il faut noter que dans ce cas (R2, R3) retourne **null** mais si nous fournissons une ville déjà existant (R1) ne sera pas générer et si à la place de **save** on utilise **saveOrUpadate** nous aurons plus de création de la ville mais uniquement la vérification et la création de l’association entre un nouveau Bureau de vote et la ville.



## GET LAZY COLLECTION



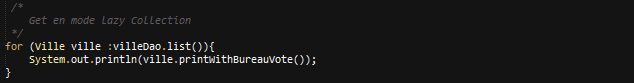
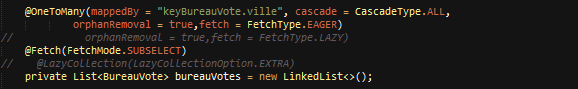




Dans le premier cas le **toString** de Ville n’affiche pas les bureaux de vote de sorte qu’une seule requête est générer pour afficher les villes (R1).

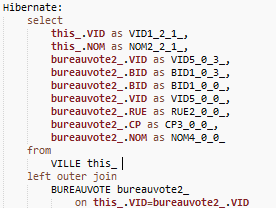
Dans le second cas nous affichons les villes et ses bureaux de votes associés, pour chacune des huit (8) villes récupérées par (R1) un select (R2) est générée pour récupérer les bureaux de vote qui lui sont associés étant donné que la collection Bureau de vote est à chargement **Lazy** et **Fetch.Select** ce qui fait qu’on a une requête pour tous les bureaux de vote.

## GET EAGER SOUS SELECT, SELECT, JOIN COLLECTION



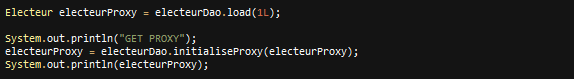


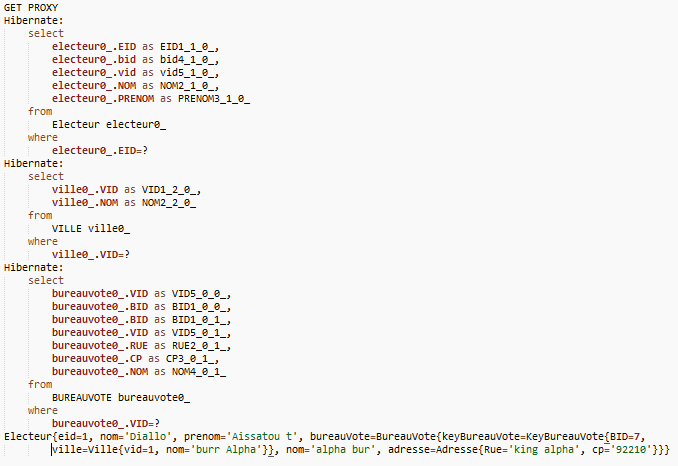
Le première requête permet de récupère la liste des villes (VID). Hibernate va éviter le problème de requête N + 1 en générant une seule instruction SQL pour initialiser toutes les collections des Bureaux de vote pour toutes les entités de Ville qui ont été extraites auparavant puis que la collection de bureau de vote est **Lazy**. Pour le **Join**, il n’y a aucune requête secondaire parce que la collection bureau de vote a été chargée avec l’entité ville.

**JOIN**

## SELECT

## INITIALISE PROXY



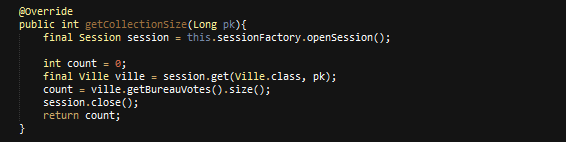


Nous récupérons l’électeur (R1) à la demande sans requête, l’initialise (**initialiseProxy**) et récupère la ville(R2) Select défaut, puis le bureau de vote (R3) car bureau de vote est **Lazy**.

GET COLLECTION BUREAU DE VOTE SIZE PAR VILLE

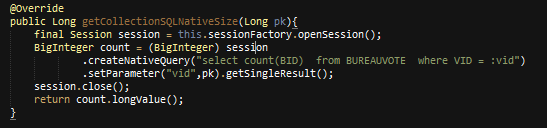
## FULL JAVA

Nous commençons par récupérer la ville puis nous faisons une jointure pour récupérer le nombre de bureaux de vote qui lui est associé, enfin de déterminer la taille de la collection. deux requêtes sont générer : la première pour la sélection de la ville et seconde qui ne fait pas la jointure et le comptage, mais une requête de compactage pour retourner la taille sans initiales la collection(LAZY) au moment du .size() ***select count(BID) from BUREAUVOTE where VID =? (\*)***



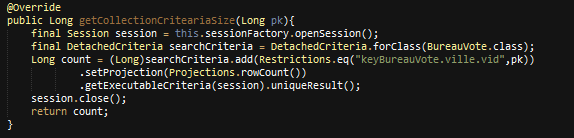
## SQL NATIF :

La deuxième méthode consiste a écrirai la requête en SQL (\*) natif qui compte directement les bureaux de votes de la ville et retourne le résultat en une seule requête.



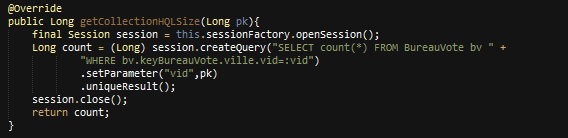
## CRITERIA NATIF :

Cette méthode génère la même SQL que le précèdent (\*)



## HQL NATIF :

Cette méthode génère la même SQL que le précèdent aussi (\*)



## STRATÉGIE MEILLEURE

Il est démontré que la seconde méthode SQL pur, HQL ou CRITERIA (génère le même SQL , trois façon d’écrire du SQL) est la plus efficace : nous avons le résultat en une requête au lieu de deux.

SQL natif n'est pas nécessairement plus rapide que Hibernate. Hibernate traduit également le java en SQL (voir l'instruction générée lors de l'exécution de l'application avec la propriété **show\_sql** définie sur **true**). Dans certains cas, il peut arriver que Hibernate ne génère pas les instructions les plus efficaces, alors le SQL natif (ou HQL ou CRITERIA) peut être plus rapide, comme dans ce cas, le résultat est obtenu par une requête et non deux. De l'autre côté avec le SQL natif, le cache Hibernate peut-être manquant- non visible, en conséquence, dans certains cas, le SQL natif peut être plus lent que Hibernate.  
  
Lorsque vous utilisez **session.load (class, ID)** et que la ligne n'est pas encore dans le cache, Hibernate génère également un ***select \* from classTable***, lors de son accès. La vitesse est donc la même à partir de Hibernate, toutefois quand l'objet est déjà dans le cache, alors le chargement est probablement plus rapide.

En utilisant Hibernate, les mêmes requêtes SQL ne s'exécutent pas plus rapidement. Cependant, lorsque vous utilisez Hibernate, vous pouvez profiter de choses comme le chargement paresseux, mise en cache de second niveau, cache de requêtes et ces fonctionnalités peuvent aider à améliorer les performances.

CONCLUSION

Comme tout ORM Hibernate imposer certaines restrictions. La fermeture de la session pour les collections en chargement **Lazy**. Le chargement **Lazy** est l'un des grands problèmes qu’il est possible de rencontrer, surtout si un modèle DAO standard est suivi. Les collections seront chargées paresseusement, mais en sortant de votre couche DAO, la session se retrouve fermée. Dans notre cas, lorsque l'appel à «**Read** de ville » est terminé, la session est fermée. En conséquence, tous les appels à la collection bureau de vote chargé en mode **Lazy** lanceront une exception **LazyInitializationException**.

Ou encore sur les collections, si la méthode "update" est utilisée encore une fois, il faut faire attention à ne pas remplacer les instances de collection, mais plutôt à manipuler les objets de la collection qui est déjà chargée. Dans le cas contraire, Hibernate aura du mal à comprendre ce qui doit être faire : ajouté, supprimé ou mettre à jour ?

Le problème du N+1 select qui permet de charger uniquement les objets **Lazy** demandé, en absence de maîtrise du chargement est le suivant: chaque appel peut potentiellement provoquer une requête en base de données, et ce, de manière complètement transparente et incontrôlable.

Le mode **Lazy** one-2-one bidirectionnel est impossible et le mode Eager one-2-one crée une **StackOverFlowException**.

Un autre problème que nous avons observé est celui de la modification simultanée des données. Lorsque nous téléchargeons un même objet sur deux threads différents, Hibernate ne peut pas toujours fusionner les deux changements (comportement pas toujours souhaité), donc il rejette donc la seconde mise à jour en lançant **StaleObjectStateException** *(Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect)).* Hibernate vous donne beaucoup d'options pour résoudre ce problème. Le plus simple, peut-être, est d'ajouter un champ de version en **@Version** sur les objets de données. Hibernate maintiendra automatiquement la "version" des données. Chaque fois qu'une mise à jour a lieu, la version sera automatiquement modifiée par Hibernate.

A l'issue de ce tutoriel, nous avons maintenant suffisamment de connaissances pour réaliser nos propres projets Hibernate.

––

Temps de travail une semaine : développement 7 jours et 3 jours pour le rapport.