Persistance en Java avec JPA

Travaux Pratiques

**Java Persistence API**

Ce TP utilise l'implémentation JPA de référence [EclipseLink](http://www.eclipse.org/eclipselink/jpa.php).

**Modèle des données**

Vous allez écrire les classes Java qui correspondent à la base de données des employés de l’entreprise Bankonet, utilisée depuis le début du cours.

Classes :

* Departement qui représente un département de l'entreprise Bankonet.
* Employe qui représente un employé de l'entreprise Bankonet.
* Projet qui représente un projet dans lequel des employés peuvent participer.
* Participation qui représente la participation d'un employé à un projet.

Dans le deuxième exercice, pour illustrer les problèmes de mapping liés à l'héritage, vous ajouterez une classe Personne (avec un champ nom et un identificateur non significatif) dont hérite la classe Employe, et une classe Client (avec seulement une adresse pour simplifier) qui hérite aussi de Personne. La classe Adresse ne sera pas une entité mais une classe insérée (Embedded).

Associations :

* Un employé appartient à un département et peut participer à des projets. Il a une fonction (*et une seule*) dans chaque projet auquel il participe (par exemple, chef de projet ou trésorier ou simple membre).
* Plusieurs employés peuvent participer à un même projet.
* Un employé a au plus un supérieur.
* Un employé appartient à au plus un département.
* Les associations entre les employés et les départements et entre les employés et les projets sont bidirectionnelles.
* L'association entre un employé et son supérieur est unidirectionnelle (d'un employé vers son supérieur, s'il existe).

Pour vous faire gagner du temps, des squelettes de classes vous sont fournis. Il vous faudra placer les classes au bon endroit et y ajouter des annotations ou même éventuellement d'autres informations (l'association entre les employés et les projets n'a pas été implémentée par exemple).

vous trouverez aussi un fichier persistence.xml que vous devrez placer au bon endroit (dans un répertoire META-INF, dans le *classpath*) et modifier pour indiquer les bonnes informations pour la connexion à la base de données et pour ajouter des noms de classes entités. Des propriétés propres à EclipseLink ont été positionnées pour faire afficher des logs sur le déroulement de l'exécution de JPA, en particulier les ordres SQL générés (lisez la console pendant l'exécution du programme) et pour recréer les tables à chaque exécution (pratique pour tester mais évidemment à enlever en production) ; vous pourrez enlever des propriétés à partir d'un certain moment si vous voulez conserver les données déjà dans la base de données.

**TP0 - Pour se chauffer...**

* Créer un nouveau projet Java dans Eclipse appelé « JPA2\_TP »
* Importer les squelettes de classes fournis

Juste pour tester, vous allez travailler avec uniquement les départements, sans les autres classes. **Dans la classe Departement, mettez en commentaires tout ce qui est lié à l'association avec Employe.**

* Organisez les fichiers correctement pour les mettre dans les bons packages (com.bankonet.model et com.bankonet.test)
* Mettre le squelette du fichier persistence.xml au bon endroit
* Compléter le fichier persistence.xml avec les informations de connexions MySQL :

Driver: com.mysql.jdbc.Driver

URL : jdbc:mysql:///banque

Identifiant : root

MotDePasse :

* Importer les librairies nécessaires à JPA (fournies) dans le build path du projet (créer un nouveau repertoire « lib » dans le projet)
* Complétez la classe Departement pour la transformer en entité JPA. Les numéros de département devront être générés automatiquement.
* La méthode main de la classe Test1 fournie crée 3 départements dont seulement 2 seront rangés dans la base de données. Le lieu de l'emplacement d'un des départements est modifié après sa création et après l'appel de persist et pourtant cette modification est bien enregistrée dans la base (vérifiez-le à l’aide du Datasource Explorer d’Eclipse).
* A la fin de l'exécution de cette méthode main, la base de données devra contenir les données associées aux départements ajoutés dans la méthode main. Vérifiez-le. Remarquez aussi que le département de Nantes n'a pas été enregistré dans la base de données.
* Étudiez ce qui est affiché lors de l'exécution de la classe, en particulier les définitions des tables créées par JPA. **Vous étudierez ces définitions à chaque fois que vous modifierez votre modèle objet dans les prochains exercices pour constater comment JPA effectue le mapping objet-relationnel.**
* Régénérer le schéma de BD en vous arrangeant pour que le nom du département soit limité à la longueur 25 dans la table DEPARTEMENT générée automatiquement.

**Ex 1 : associations 1:N**

* Enlevez les commentaires que vous avez mis dans la question précédente dans la classe Departement pour tout ce qui est lié à l'association avec la classe Employe. Complétez les classes Employe, Client et Adresse pour en faire des Entités.
* Implémenter les relations entre ces classe dans les entités (@OneToMany, etc.)
* Utilisez la classe de test « Test2 » fournie : A la fin de l'exécution de cette méthode main, la base de données devra contenir les bonnes informations. Vérifiez-le.
* Quel est le problème avec les tables CLIENT et EMPLOYE ?

**Ex 2 : Héritage**

* Compléter la classe Personne pour en faire une Entity
* Utilisez la stratégie "SINGLE TABLE" pour implémenter l’héritage : modifier/compléter les classes Personne, Employe, Client pour cela
* Utilisez la classe de test « Test2 » fournie : A la fin de l'exécution de cette méthode main, la base de données devra contenir les bonnes informations avec la bonne structure liée à la stratégie d’héritage. Vérifiez-le.

**Ex 3 - Langage d'interrogation de JPA : JPQL**

Commencez par modifier le fichier persistence.xml pour ne pas écraser les tables existantes : remplacez drop-and-create par create.

1. Cette fois-ci vous allez écrire une classe Test3 qui va récupérer tous les employés du département "DIRECTION" (supposez que le nom du département est toujours donné en majuscule ; attention, le nom dans la base n'est pas toujours en majuscule) et qui affiche **ensuite** leur nom. Le nom du département devra être un paramètre de la requête. La réponse ne devra pas tenir compte de la casse dans le nom du département. Dans cette question et la suivante vous utiliserez des requêtes dynamiques.
2. Ecrivez une 2ème version du programme qui ne récupère pas les employés mais seulement leur nom et leur salaire dans la base de données et qui les affiche.
3. Ecrivez une 3ème version qui affiche encore les noms des employés, en utilisant une requête nommée.
4. Utilisez la requête de la première version pour augmenter de 5 % le salaire des employés récupérés et pour enregistrer les modifications dans la base.
5. Mettez tous les salaires des employés à 2200 euros par une modification "en volume" (*bulk update*). Vérifiez que cette opération n'a pas modifié les entités en mémoire. A cause de ce fait, il ne faut pas lancer une modification en volume si le contexte de persistance contient déjà des entités. Il est tout de même possible de synchroniser une entité en mémoire avec les valeurs de la base de données. Comment faire ? Testez.

**Ex 4 - Association M:N**

Complétez la classe Projet écrivez une classe Test4 dont la méthode main crée 3 employés, 2 projets et répartit les 3 employés dans ces 2 projets. Attention, c'est pour cet exercice qu'il faut penser à ne pas écraser vos tables PROJET et PARTICIPATION.

Voici la méthode de la classe Projet qui servira à ajouter un employé dans un projet :  
void ajouterParticipant(Employe employe, String fonction)  
Le concepteur de l'application souhaite que l'entité Participation ne soit pas visible pour l'utilisateur de la classe Projet. Que devrez-vous faire pour rendre persistantes les participations sans que ces participations ne soient visibles ?

Que faudrait-il modifier si un employé pouvait avoir plusieurs fonctions dans un même projet ?

**Ex 5 - Requêtes polymorphe (ou non). Navigation dans les requêtes**

Ecrivez une classe Test5 dont la méthode main

* affiche la liste des noms de toutes les personnes ;
* affiche la liste des noms de tous les clients (faite-le en ajoutant une clause where à la requête précédente) ;
* affiche ensuite la liste de noms des employés avec, pour chaque employé et sur la même ligne d'écran, les noms des projets auxquels l'employé participe. Les noms des employés qui ne participent à aucun projet devront apparaître seuls sur une ligne.

**Ex 6 - Pour ceux qui ont déjà fini...**

**N + 1 select**

Récupérez tous les employés en dehors d'une transaction avec la requête "select e from Employe as e".

Faites afficher les noms de tous les employés.

Sans relancer une requête, en utilisant seulement les associations des objets Employe récupérés, récupérez tous les projets auxquels participent les employés. Faites afficher la liste des noms des employés ; un nom par ligne, suivi de tous les projets auxquels l'employé participe. Les noms des employés qui ne participent à aucun projet devront être affichés.

En utilisant le logging de EclipseLink, regardez quelles sont les requêtes SQL lancées par EclipseLink et à quel moment. Modifiez le mode de récupération de l'association entre les employés et les participations pour voir l'impact sur les requêtes SQL.

Comment éviter le problème des N + 1 selects ?

**Ex 7 - Transactions**

Il est très important de comprendre les relations qui lient les contextes de persistance et les transactions. Dans cet exercice vous allez tester quelques situations particulières.

1. Créez un département et ajoutez-le à un contexte de persistance. Ne commencez pas de transaction et modifiez ce département (changez son lieu par exemple). Est-ce les modifications ont été effectuées dans la base de données ?
2. A la suite du code qui crée le département et le modifie, ouvrez une transaction et refermez-la tout de suite. Est-ce que les modifications effectuées sur le département sont enregistrées dans la base de données ?
3. Après avoir fermé la transaction, modifiez à nouveau le département et lancez un *flush* (sans ouvrir de transaction). Que se passe-t-il ?

Avertissement pour ceux qui vont travailler avec un serveur d'applications (avec Java EE) : par défaut les contextes de persistance seront alors limités à une transaction et le comportement ne sera pas le même que dans cet exercice où le contexte de persistance et les transactions ne sont pas gérés par le container mais par l'application.

**Ex 8 - Entité versionnée**

Ajoutez un attribut de version (@Version) dans l'entité Departement.

1. Modifiez le nom d'un département et vérifiez si le numéro de version a été incrémenté.
2. Ajoutez un employé dans un département et vérifiez si le numéro de version a été incrémenté.
3. Testez un lock (READ) et écrivez du code Java pour obtenir une lecture répétable sur ce département. Essayez de modifier ce département dans une autre transaction en parallèle pour voir ce qui se passe (ne passez pas le département en paramètre, récupérez-le dans la 2ème transaction lancée en parallèle comme dans la première transaction). Quelle est la différence de comportement avec du code qui n'aurait pas exécuté un lock en mode READ ? Vérifiez.
4. Utilisez un lock (WRITE) et écrivez du code Java pour faire incrémenter le numéro de version, sans modifier le département. Vérifiez en faisant afficher le numéro de version.