Programmation Orientée Objet
IG2I, LE3 – 2024-2025

TP 4 – JPA : Hôpital

Maxime Ogier

Contexte et objectif du TP

Ce TP propose d'utiliser le concept de *mapping* objet-relationnel afin de mettre en place le système d'information des médecins et des malades d'un hôpital. Il s'agit d'utiliser l'API JPA (*Java Persistence API*), grâce à laquelle nous serons capables de créer la base de données (le modèle relationnel) à partir du modèle objet développé en Java.

Ce TP est la suite du TP 4 et permet d'illustrer les notions suivantes :

- le *mapping* des relations d'héritage ;
- les classes insérées ;
- le langage d'interrogation de JPA.

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.
- 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 Héritage : personnes, médecins et malades

Les entités sont des objets, et devraient donc être capables d'hériter de l'état et du comportement d'autres entités. Ceci est même essentiel pour le développement d'applications orientées-objet.

1.1 Théorie : modèles d'héritage avec JPA

JPA propose trois manières différentes de représenter les relations d'héritage dans la base de données. Nous parlons ici de hiérarchie d'héritage où la classe mère est une entité. La classe mère doit indiquer la hiérarchie d'héritage en utilisant l'annotation **@Inheritance** au-dessus de la définition de la classe. Cette annotation permet d'indiquer la stratégie qui est utilisée pour le *mapping*, et qui doit être une des trois stratégies définies par la suite.

1.1.1 Stratégie "une seule table"

Une manière performante et répandue pour stocker les états de plusieurs entités qui héritent d'une même entité mère est de définir une seule table dans la base de données. Cette table contient alors l'ensemble de tous les états possibles de chaque entité fille, i.e. que chaque attribut des entités filles aura une colonne correspondante dans la table. Il s'agit de la stratégie "une seule table". En conséquence, pour une ligne donnée de la table, qui représente une instance d'une classe concrète, il peut y avoir un certain nombre de colonnes qui ne sont pas renseignées car elles appartiennent à d'autres classes filles. Par ailleurs, il est nécessaire que toutes les entités dans la hiérarchie d'héritage utilisent des identifiants (pour la clé primaire) de même type.

En général, l'approche "une seule table" est plus coûteuse en terme d'espace de stockage pour les données, mais elle permet d'offrir de meilleures performances pour les requêtes et l'écriture en base de données. En effet, il n'est pas nécessaire de faire de jointures car tout est dans une seule table.

Pour indiquer le choix de cette stratégie, il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.SINGLE_TABLE**. Notez que cette stratégie est celle par défaut si rien n'est précisé.

En outre, dans la table générée dans la base de données, il y aura une colonne supplémentaire qui ne correspond à aucun attribut d'aucunes des classes mère

ou filles. Il s'agit d'une colonne discriminante qui est nécessaire lorsqu'on utilise la stratégie "une seule table" pour faire de l'héritage. Grâce à cette colonne, il est possible de savoir à quelle entité fille est liée chaque enregistrement dans la base. L'annotation **@DiscriminatorColumn** permet de paramétrer cette colonne, en particulier l'attribut **name** permet de spécifier son nom dans la table, et l'attribut **discriminatorType** permet de spécifier le type de données. Ce dernier attribut peut prendre les valeurs **DiscriminatorType.STRING** (par défaut), **DiscriminatorType.INTEGER** ou **DiscriminatorType.CHAR**.

Chaque ligne dans la table aura une valeur dans la colonne discriminante ; cette valeur est appelée valeur discriminante, et elle indique le type de l'entité qui est mémorisée dans la ligne. Chaque entité concrète dans la hiérarchie d'héritage a donc besoin d'une valeur discriminante spécifique par rapport à son type qui permette au fournisseur de persistance d'utiliser le bon type lors de l'enregistrement d'une entité, ou d'une requête. Dans chaque entité fille, l'annotation **@DiscriminatorValue** permet de spécifier la valeur discriminante pour l'entité en question. Cette valeur est passée sous forme d'une chaîne de caractères dans l'annotation. Cette valeur doit être du même type que celui spécifié par l'annotation **@DiscriminatorColumn** dans l'entité mère. Si l'annotation **@DiscriminatorValue** n'est pas spécifiée, alors c'est le fournisseur de persistance qui attribuera automatiquement une valeur.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="EMP_TYPE",
                     discriminatorType = DiscriminatorType.STRING)
public abstract class Employee {
   // ...
}
```

```
@Entity
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends Employee {
   // ...
}
```

```
@Entity
@DiscriminatorValue("PTEmp")
public class PartTimeEmployee extends Employee {
   // ...
}
```

1.1.2 Stratégie "jointure"

Du point de vue d'un développeur Java, un modèle de données qui *mappe* chaque entité avec sa propre table fait sens. Ainsi, il y aura une table différente pour chaque entité, qu'elle soit abstraite ou concrète. On parle de stratégie "jointure". Ceci est la manière la plus efficace de stocker les données partagées par plusieurs entités filles dans une hiérarchie. Ce qui est moins performant, c'est lorsqu'il faut ré-assembler une instance d'une entité fille : il faut utiliser des jointures entre la table d'une entité fille et la table de l'entité mère. L'insertion est aussi plus coûteuse car elle se fait dans plusieurs tables.

Les identifiants (clés primaires) doivent être du même type dans toutes les entités de la hiérarchie. Au niveau de la base de données, la clé primaire dans la table d'une entité fille sera aussi une clé étrangère qui référence la clé primaire de la table de l'entité mère.

Pour utiliser cette stratégie "jointure", il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.JOINED**. Les annotations **@DiscrinatorColumn** et **@DiscrimatorValue** peuvent aussi être utilisées ici. Cette fois la colonne discriminante sera présente dans la table correspondant à l'entité mère de la hiérarchie d'héritage.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE",
           discriminatorType = DiscriminatorType.INTEGER)
public abstract class Employee {
   // ...
}
```

```
@Entity
@DiscriminatorValue("1")
public class FullTimeEmployee extends Employee {
   // ...
}
```

```
@Entity
@DiscriminatorValue("2")
public class PartTimeEmployee extends Employee {
   // ...
}
```

1.1.3 Stratégie "une table par classe concrète"

Une troisième approche pour le *mapping* d'une hiérarchie d'héritage est d'utiliser une stratégie où une table est définie pour chaque classe concrète. Ainsi, tous les attributs partagés dans l'entité mère sont redéfinis séparément dans chaque table associée à une entité fille qui hérite de l'entité mère.

L'aspect négatif de cette stratégie est que les requêtes polymorphes sont alors plus coûteuses qu'avec les autres stratégies : il faut utiliser la commande **UNION** afin de faire la requête sur toutes les tables correspondant aux entités filles, ce qui est coûteux s'il y a beaucoup de données. En revanche, cette stratégie offre de meilleures performances que la stratégie "jointure" si l'on souhaite faire des requêtes sur une classe fille. Par ailleurs, avec cette stratégie, la colonne discriminante n'a plus d'utilité.

Pour utiliser cette stratégie, il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.TABLE_PER_CLASS**.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
   // ...
}
```

```
@Entity
public class FullTimeEmployee extends Employee {
   // ...
}
```

1.2 Les médecins et les malades sont des personnes

Nous allons à présent modifier le modèle objet, et faire apparaître l'entité **Personne** définie par son identifiant, son nom et son prénom. Comme précédemment, les couples nom/prénom sont uniques. Cette entité pourra être déclarée abstraite car les personnes seront soit des médecins (pour lesquels on définit un salaire), soit des malades. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 1.

Question 1. Ajoutez dans le paquetage **modele** une entité abstraite **Personne**, une entité **Malade** qui hérite de **Personne**, et modifiez l'entité **Medecin** qui hérite aussi de **Personne**. Référez-vous à la Figure 1, et pensez bien à la génération des clés primaires, l'unicité du couple nom/prénom, et l'écriture en majuscule.

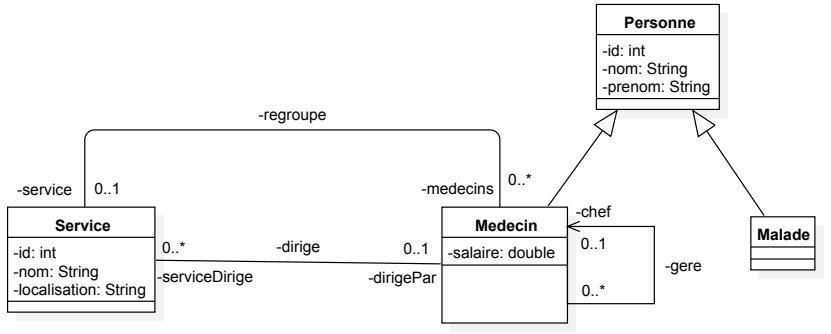

Figure 1: Diagramme de classe pour les entités **Service**, **Personne**, **Medecin** et **Malade**.

Question 2. Complétez le test précédent de la classe **Test3**, en y ajoutant quelques malades qui seront rendus persistants. Vous pouvez vous baser sur le code suivant :

```
Malade mal1 = new Malade("Proviste", "Alain");
Malade mal2 = new Malade("Trahuil", "Phil");
Malade mal3 = new Malade("Ancieux", "Cecile");
Malade mal4 = new Malade("Conda", "Anna");
// ...
et.begin();\
// ...
em.persist(mal1);
em.persist(mal2);
em.persist(mal3);
em.persist(mal4);
et.commit();
```

Étudiez ce qui se passe au niveau de la base de données selon la stratégie d'héritage utilisée.

2 Classe insérée : la classe adresse

2.1 Théorie : définition d'une classe insérée

Une classe insérée est une classe qui dépend d'une autre entité pour son identifiant. Elle n'a pas d'identifiant et fait donc partie de l'état d'une entité distincte. En Java, les classes insérées sont similaires à des associations dans lesquelles la

classe (insérée) est la cible d'une association et une entité (celle dans laquelle elle est insérée) est la source. Au niveau de la base de données, la classe insérée est stockée avec le reste de l'état de l'entité dans une ou plusieurs colonnes (une colonne par attribut de la classe insérée). On voit là un exemple du décalage entre les modèles objets et relationnels. Dans le modèle objet, il est tout à fait naturel de considérer une classe distincte pour mettre en évidence l'aspect conceptuel. Dans le modèle relationnel, il est plus simple au niveau du stockage des données de tout stocker dans une seule table.

2.2 Théorie : *mapping* d'une classe insérée

Un exemple très classique de classe insérée est l'adresse. Un numéro de rue, un nom de rue, un code postal et une ville forment logiquement le concept d'adresse. Au niveau du modèle objet il est tout à fait naturel d'utiliser une classe spécifique pour représenter une adresse au lieu de dupliquer ses attributs dans tous les objets qui ont une adresse. Par exemple, comme présenté dans la Figure 2, entre un employé et une adresse, on a une relation de composition, i.e. que l'employé est propriétaire de l'adresse et qu'une instance d'une adresse ne peut pas être partagée par un autre objet que l'employé qui la possède.

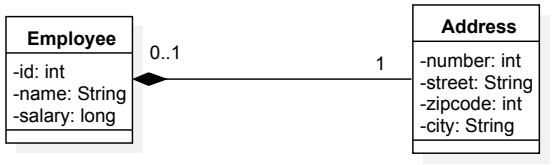

Figure 2: Diagramme de classe : association de composition entre **Employee** et **Address**.

Avec cette représentation, les informations qui composent une adresse sont donc parfaitement encapsulées dans un objet. De plus, si une autre entité, une entreprise par exemple possède aussi comme information une adresse, alors cette entité peut posséder un attribut qui réfère sur sa propre classe insérée **Address**.

Une classe insérée est marquée comme telle en utilisant l'annotation **@Embeddable** au-dessus de la définition de la classe. Lorsqu'une classe est définie comme pouvant être insérée, alors ses attributs seront rendus persistants en tant qu'une partie d'une autre entité. Les annotations concernant les colonnes (**@Basic**, **@Column**, **@Temporal**, **@Enumerated**) peuvent être ajoutés aux attributs de la classe insérée. Mais la classe insérée ne doit pas contenir d'associations. Par exemple, la classe insérée **Address** est définie de la manière suivante :

```
1  @Embeddable
2  public class Address {
3     private int number;
4     private String street;
5     @Column(nullable=false)
6     private String city;
7     @Column(name="ZIP_CODE")
8     private String zipcode;
9     // ...
10 }
```

Pour utiliser cette classe insérée dans une entité, l'entité doit posséder un attribut du même type annoté par **@Embedded**. Par exemple, l'entité **Employee** est définie de la manière suivante :

```
1  @Entity
2  public class Employee {
3     @Id
4     private int id;
5     private String name;
6     private long salary;
7     @Embedded
8     private Address address;
9     // ...
10 }
```

La décision d'utiliser des classes insérées ou des entités est fonction de si l'on aura besoin de créer des associations avec ces objets. Les classes insérées ne sont pas destinées à être des entités, et si l'on commence à les concevoir comme tel, alors il est préférable d'utiliser une entité au lieu d'une classe insérée.

2.3 Insertion de la classe Adresse

Nous allons à présent modifier le modèle objet, et faire apparaître la classe insérée **Adresse** définie par un numéro, une rue et une ville. L'entité **Malade** possédera une référence vers cette classe insérée. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 3.

Question 3. Ajoutez dans le paquetage **modele** une classe insérée **Adresse**, et modifiez l'entité **Malade** afin qu'elle possède un attribut de type **Adresse**, comme présenté dans la Figure 3. Les champs rue et ville sont non nuls. N'oubliez pas de modifier le constructeur de la classe **Malade**. Modifiez en conséquence le jeu de test précédent, vérifiez que tout fonctionne correctement et observez la représentation des données dans la base.

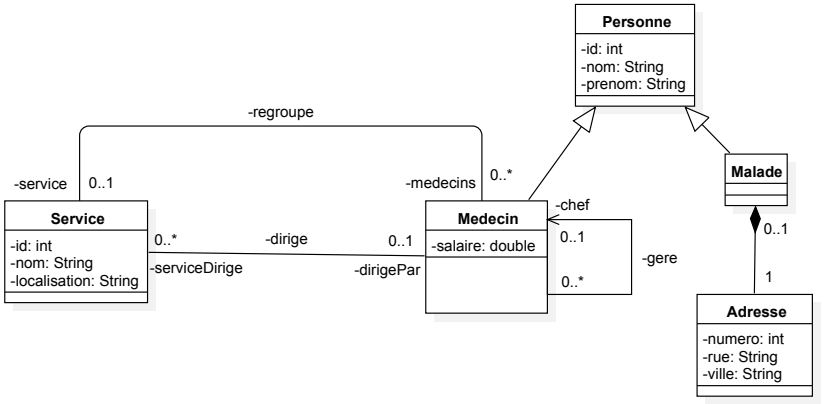

Figure 3: Diagramme de classe avec la classe insérée **Adresse**.

3 Langage d'interrogation de JPA

JPA supporte deux types de langages pour rechercher des entités et autres données persistantes depuis la base de données. Le premier langage est *Java Persistence Query Language* (JPQL) qui opère sur le modèle logique des entités et non pas sur le modèle physique de données. Les requêtes peuvent aussi être exprimées en SQL afin de tirer avantage de la base de données sous-jacente.

3.1 Mise en place d'une classe pour effectuer des requêtes

Question 4. Lancez une dernière fois votre denier test afin de populer la base de données. Ensuite, modifiez l'unité de persistance (fichier **persistance.xml**) afin de ne pas écraser les tables existantes et leur contenu. Pour cela, choisissez **none** pour l'action à réaliser lors du lancement de l'application.

Question 5. Ajoutez un nouveau paquetage **requete** dans lequel vous ajouterez une classe **RequeteHopital**. Vous enrichirez cette classe au fur et à mesure cette classe afin de disposer de méthode pour faire des requêtes sur la base de données. Ajoutez un attribut de type **EntityManagerFactory**, ainsi qu'un constructeur par défaut pour initialiser cet attribut.

3.2 Théorie : première découverte de JPQL

Un exemple de requête simple en JPQL pour sélectionner toutes les instances d'une classe d'entités est donné par :

```
1  SELECT e FROM Employee e
```

Cela ressemble fortement à SQL, et c'est normal car JPQL utilise la syntaxe SQL, ce qui permet aux habitués de SQL de facilement s'adapter à JPQL. La différence majeure entre SQL et JPQL pour la requête ci-dessus est qu'au lieu de sélectionner les données dans une table de la base de données, on spécifie une entité du modèle objet. Une autre différence est l'utilisation d'alias (ici **e** pour **Employee**). Cela permet d'indiquer que le résultat de la requête est de type **Employee**, et ainsi l'exécution de cette requête renverra une liste d'instances de **Employee**. Par ailleurs, avec l'alias il est possible de spécifier un attribut persistant ou même de naviguer à travers les relations entre les entités en utilisant l'opérateur point ('.'). Par exemple, si on souhaite récupérer seulement les noms des employés, on peut utiliser la requête suivante :

```
1  SELECT e.name FROM Employee e
```

Il est aussi possible de sélectionner une entité qui n'est pas listée dans la clause *FROM*. Par exemple, dans le cas où il y a une association *Many-to-One* entre **Employee** et **Department** (l'attribut est nommé **department**), on peut écrire la requête :

```
1  SELECT e.department FROM Employee e
```

3.3 Théorie : exécuter une requête sur la base de données

Pour exécuter une requête avec JPA, on utilise des objets qui sont du type de l'interface **Query**. Il est possible de récupérer un tel objet en appelant la méthode **createQuery()** de l'interface **EntityManager**.

Pour exécuter dynamiquement une requête, il suffit d'écrire la requête sous forme d'une chaîne de caractères, puis de la passer en paramètre de la méthode **createQuery()** appelée sur l'instance de **EntityManager**. On obtient alors une instance de type **Query**.

Si l'on souhaite exécuter la requête et récupérer un seul résultat, on fait appel à la méthode **getSingleResult()** de l'interface **Query**. Cette méthode renvoie un objet de type **Object** qu'on pourra *caster* au besoin.

Si l'on souhaite exécuter la requête et récupérer une liste de résultats, on fait appel à la méthode **getResultList()** de l'interface **Query**. Cette méthode renvoie un objet de type **List**.

Par exemple, pour récupérer l'ensemble des employés, on peut utiliser le code suivant :

```
EntityManager em = ... ; // récupérer une instance
String strQuery = "SELECT e FROM Employee e";
Query query = em.createQuery(strQuery);
List<Emplyee> employees = query.getResultList();
em.close();
```

3.4 Premières requêtes de sélection

Question 6. Ajoutez dans la classe **RequeteHopital** une méthode pour récupérer l'ensemble des services. Testez que cette méthode fonctionne correctement.

Question 7. Ajoutez dans la classe **RequeteHopital** une méthode pour récupérer l'ensemble des médecins qui sont des chefs. Testez que cette méthode fonctionne correctement.

3.5 Théorie : compléments sur les requêtes de sélection avec JPQL

Comme SQL, JPQL supporte la clause *WHERE* pour mettre des conditions sur les données recherchées. De nombreux opérateurs SQL sont aussi disponibles : *IN*, *LIKE*, *BETWEEN*, et des fonctions comme *SUBSTRING* et *LENGTH*. La différence réside surtout dans le fait qu'avec JPQL on utilise des expressions avec les entités et non pas les colonnes des tables. Par exemple, on peut avoir la requête suivante :

```
SELECT e FROM Employee e
   WHERE e.department.name = 'SECRETARIAT'
   AND e.address.city IN ('Lille','Lens')
```

Si on ne souhaite pas récupérer toutes les données contenues dans les instances d'une classe d'entité, alors il est possible de définir les noms des attributs que l'on souhaite récupérer, comme dans l'exemple suivant :

```
SELECT e.name, e.salary FROM Employee e
```

Le type de résultat d'une requête de sélection ne peut pas être une collection, il faut que ce soit une instance d'entité ou un type d'attribut persistant. Ainsi, si on souhaite naviguer dans une association de type *One-to-Many* par exemple, on ne peut pas utiliser directement la notation pointée avec l'attribut qui est une collection d'objets. Il faudra alors faire une jointure entre les entités.

Il existe deux manières de faire des jointures, comme présenté dans les exemples ci-dessous :

```
SELECT p.number FROM Employee e, Phone p
   WHERE e = p.employee
   AND e.department.name = 'SECRETARIAT'
   AND p.type = 'PORTABLE'
```

Ou encore, on peut faire apparaitre explicitement l'opérateur *JOIN* :

```
SELECT p.number FROM Employee e
   JOIN e.phones p
   WHERE e.department.name = 'SECRETARIAT'
   AND p.type = 'PORTABLE'
```

La syntaxe pour les requêtes d'agrégation est très similaire à SQL. On retrouve les fonctions d'agrégat *AVG*, *COUNT*, *MIN*, *MAX* et *SUM*, les résultats peuvent être groupés dans la clause *GROUP BY* et filtrés en utilisant la clause *HAVING*. Par exemple, on peut écrire la requête suivante en JPQL :

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
   FROM Department d
   JOIN d.employees e
   GROUP BY d
   HAVING COUNT(e) >= 5
```

3.6 Requêtes de sélection avancées

Question 8. Ajoutez dans la classe **RequeteHopital** une méthode pour récupérer l'ensemble des médecins dont le salaire est supérieur ou égal à 2800 euros. Testez que cette méthode fonctionne correctement.

Question 9. Ajoutez dans la classe **RequeteHopital** une méthode pour récupérer l'ensemble des salaires moyens par service. Testez que cette méthode fonctionne correctement.

3.7 Théorie : requêtes paramétrées

JPQL permet de faire des requêtes paramétrées, et supporte deux types de syntaxe pour la liaison des paramètres. Le premier est la liaison par position : les paramètres sont indiqués dans la requête par un point d'interrogation suivi du numéro de paramètre. Lors de l'exécution de la requête, il faut spécifier le numéro de paramètre qui doit être remplacé. Cela donne par exemple la requête paramétrée suivante :

```
SELECT e FROM Employee e
   WHERE e.department = ?1
   AND e.salary > ?2
```

Avec le second type de syntaxe, des noms sont donnés aux paramètres, et on indique dans la requête qu'il s'agit de paramètre en faisant précéder le nom d'un symbole deux-points (':'). Lors de l'exécution de la requête il faut spécifier le nom du paramètre qui doit être remplacé. Cela donne par exemple la requête paramétrée suivante :

```
SELECT e FROM Employee e
   WHERE e.department = :dept
   AND e.salary > :base
```

Pour exécuter une requête paramétrée avec JPA, après avoir appelé la méthode **createQuery()** sur une instance de **EntityManager**, on récupère une instance de type **Query** sur laquelle on peut appeler les méthodes **setParameter(int position, Object value)** ou **setParameter(String name, Object value)** afin de spécifier la valeur des paramètres. Le choix dépend de la manière dont ont été définis les paramètres. On appelle ensuite la méthode **getSingleResult()** ou la méthode **getResultList()** de l'interface **Query**.

Par exemple, pour récupérer tous les employés du département 'Comptabilité' qui gagnent plus de 2000 euros, on peut utiliser le code suivant :

```
String strQuery = "SELECT e FROM Employee e "
      + "WHERE e.department = :dept "
      + "AND e.salary > :base";
Query query = em.createQuery(strQuery);
query.setParameter("dept", "Comptabilite");
query.setParameter("base", 2000);
List<Employee> employees = query.getResultList();
```

3.8 Requêtes paramétrées

Question 10. Ajoutez dans la classe **RequeteHopital** une méthode pour récupérer l'ensemble des médecins d'un service dont on donne le nom en paramètre. Testez que cette méthode fonctionne correctement.

3.9 Théorie : requêtes nommées

Il y a deux approches pour définir une requête : une requête peut être spécifiée dynamiquement à l'exécution (c'est ce que nous avons vu jusqu'ici), ou bien elle peut être configurée dans les métadonnées de l'unité de persistance et référencée par son nom. Les requêtes dynamiques qui ont été présentées précédemment ne sont rien d'autre que des chaînes de caractères et peuvent être définies à la volée. Les requêtes nommées sont statiques et non modifiables mais elles sont plus efficaces à l'exécution car le fournisseur de persistance fait une seule fois la traduction de la requête de JPQL vers SQL au démarrage de l'application.

Les requêtes nommées sont un outil puissant pour organiser la définition des requêtes et améliorer les performances d'une application. Une requête nommée est définie en utilisant l'annotation **@NamedQuery** qui doit être placée dans la définition de la classe d'une entité. Les requêtes nommées sont en général définies dans la classe d'entité qui correspond le plus directement au résultat de la requête. L'annotation définie le nom de la requête et le texte de la requête.

Par exemple, si on souhaite définir une requête nommée pour trouver le salaire d'un employé dont on donne en paramètres le nom et le nom de département, on pourra écrire dans l'entité **Employee** :

```
@NamedQuery(name="findSalaryForNameAndDepartment",
        query="SELECT e.salary FROM Employee e "
              + "WHERE e.department.name = :deptName "
              + "AND e.name = :empName")
@Entity
public class Employee {
   // ...
}
```

Le nom de la requête nommée est connu au niveau de l'unité de persistance et doit donc être unique. Ceci est important à garder à l'esprit, car si on utilise des noms génériques comme "findAll" dans plusieurs entités cela ne fonctionnera pas correctement. Une bonne pratique consiste donc à préfixer le nom de la requête nommée par l'entité dans laquelle elle est définie. Par exemple, la requête "findAll" dans l'entité **Employee** devrait être nommée "Employee.findAll".

Si l'on souhaite définir plusieurs requêtes nommées concernant l'entité **Employee** on pourra écrire :

```java
@NamedQueries({
        @NamedQuery(name="Employee.findAll",
              query="SELECT e FROM Employee e"),
        @NamedQuery(name="Employee.findByPrimaryKey",
              query="SELECT e FROM Employee e "
                    + "WHERE e.id = :id"),
        @NamedQuery(name="Employee.findByName",
              query="SELECT e FROM Employee e "
                    + "WHERE e.name = :name")
})
@Entity
public class Employee {
   // ...
}
```

Pour exécuter une requête nommée, on appellera la méthode **createNamedQuery()** de l'interface **EntityManager**. Par exemple, pour exécuter la requête "Employee.findByName", on écrira :

```java
Query query = em.createNamedQuery("Employee.findByName");
query.setParameter("name", name);
Employee emp = (Employee) query.getSingleResult();
```

3.10 Requêtes nommées

Question 11. Ajoutez dans l'entité **Medecin** une requête nommée qui sélectionne tous les médecins dont le nom et le prénom du chef sont passés en paramètres. Ajoutez une méthode dans la classe **RequeteHopital** qui renvoie l'ensemble des médecins dont le nom et le prénom du chef sont passés en paramètres. Testez le bon fonctionnement de cette méthode.

3.11 Théorie : mise à jour des entités

3.11.1 Modification des entités dans une transaction

Il est possible de mettre à jour des entités (en modifiant leurs attributs) dans une transaction. Pour ce faire, on peut d'abord récupérer les entités que l'on souhaite modifier avec une requête de sélection. Puis, dans une transaction, on fait appel aux méthodes de la classe pour mettre à jour certains attributs. Lors de la validation de la transaction, les données seront alors mises à jour dans la base.

Par exemple, si on souhaite ajouter 10 euros au salaire de l'employé dont le nom est "DUPONT", on peut écrire le code suivant :

```java
Query query = em.createNamedQuery("Employee.findByName");
query.setParameter("name", "DUPONT");
Employee emp = (Employee) query.getSingleResult();
final EntityTransaction et = em.getTransaction();
try {
   et.begin();
   emp.setSalary(emp.getSalary() + 10);
   et.commit();
} catch (Exception ex) {
   et.rollback();
}
```

3.11.2 Mise à jour en volume

Il est aussi possible de faire des mises à jour en volume (*bulk update*). Ce type de mise à jour est réalisé avec une requête de type *UPDATE*. La requête doit concerner un seul type d'entité et on peut modifier un ou plusieurs attributs des entités (un attribut simple ou une association *One-to-One* ou *Many-to-One*). Il est possible d'utiliser la clause *WHERE* pour spécifier les entités concernées par la mise à jour. La syntaxe est semblable à celle de SQL mais on utilise les expressions sur les entités au lieu des tables et des colonnes. Par ailleurs, pour effectuer cette mise à jour en volume, il faut appeler la méthode **executeUpdate()** de l'interface **Query**. Cette méthode doit être appelée à l'intérieur d'une transaction, et lors de la validation les données sont mises à jour dans la base.

Par exemple, si l'on souhaite augmenter de 10 euros le salaire des employés du département "SECRETARIAT", on peut écrire le code suivant :

```java
final String strQuery = "UPDATE Employee e "
      + "SET e.salary = e.salary + 10 "
      + "WHERE e.department.name = :deptName";
Query query = em.createQuery(strQuery);
query.setParameter("deptName", "SECRETARIAT");
final EntityTransaction et = em.getTransaction();
try {
   et.begin();
   query.executeUpdate();
   et.commit();
} catch (Exception ex) {
   et.rollback();
}
```

Il faut cependant faire attention avec ces mises à jour en volume car le contexte de persistance n'est pas mis à jour afin de refléter le résultat de la mise à jour. Les opérations en volume sont exécutées comme des requêtes SQL directement sur la base de données, contournant l'état des entités dans le contexte de persistance. Ainsi, dans l'exemple précédent, si au début de la transaction des entités on exécute une requête de sélection, les entités récupérées par la requêtes seront dans le contexte de persistance, et on ne verra donc pas la mise à jour du salaire pour ces entités (même si la mise à jour à bien été effectuée en base de données). Si par ailleurs on opère des modifications sur les entités dans le contexte de persistance, alors il est possible de se retrouver dans un état inconsistant.

Ainsi, soit il faut faire la mise à jour en volume au tout début de la transaction, soit il faut faire appel à la méthode **refresh()** de l'interface **EntityManager**. Cette méthode prend en paramètre une entité qui est dans le contexte de persistance. Si l'on suspecte que des changements ont eu lieu dans la base de données mais ne sont pas répercutées sur une entité gérée dans le contexte de persistance, la méthode **refresh()** permet de mettre à jour l'état de l'entité avec les données présentes dans la base. Voici un exemple d'utilisation de **refresh()** pour mettre à jour le contexte de persistance :

```
et.begin();
// Requête de sélection et récupération des entités
// Ces entités font alors partie du contexte de persistance
List<Employee> employes = querySelect.getResultList();
// Mise à jour en volume qui concerne certaines entités
// du contexte de persistance (qui ne sont donc plus à jour)
queryUpdate.executeUpdate();
// On met à jour le contexte de persistance
for(Employee emp : employes) {
   em.refresh(emp);
}
et.commit();
```

3.12 Mise à jour des salaires des médecins

Question 12. Ajoutez dans la classe **RequeteHopital** une méthode qui prend en paramètres un nom de service, et qui augmente le salaire des médecins de ce service de 5% (on ne fera pas de mise à jour en volume). Testez que l'augmentation fonctionne bien et qu'elle a bien été prise en compte dans la base de données.

Question 13. Ajoutez dans la classe **RequeteHopital** une méthode qui fixe les salaires de tous les médecins à 2000 euros avec une mise à jour en volume. Testez que tout fonctionne correctement.

References

[1] Keith, M. and Schincariol, M., *Pro JPA 2: Second Edition - A definitive guide to mastering the Java Persistence API, APress, 2013*