Programmation Orientée Objet
IG2I, LE3 – 2024-2025

TP 3 – 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 permet d'illustrer les notions suivantes :

- le *mapping* objet-relationnel ;
- l'API JPA (*Java Persistence API*) ;
- le *mapping* des relations d'association.

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 Description de l'application

Vous êtes chargé de mettre en place le système d'informations pour gérer les médecins et les malades d'un hôpital.

Les médecins et les malades sont tous des personnes caractérisées par un identifiant, un nom et un prénom. Chaque malade possède une adresse (rue, numéro de rue et ville). Les médecins sont caractérisés par leur salaire.

Par ailleurs, il existe une hiérarchie interne qui fait qu'un médecin peut gérer un groupe de médecins subalternes, et il est stipulé qu'un médecin a au maximum un seul chef.

L'hôpital est organisé en services. Chaque service est caractérisé par un identifiant, un nom et une localisation. Un service regroupe un ou plusieurs médecins et est dirigé par au plus un médecin. Chaque médecin appartient à exactement un service, mais il est possible qu'un médecin dirige plusieurs services (si il a de très bonnes compétences de management).

Par ailleurs, au niveau médical, les médecins forment des équipes chirurgicales lors des opérations. Chaque équipe est caractérisée par un identifiant et un nom. Un médecin peut participer à plusieurs équipes, avec à chaque participation une seule fonction spécifique (par exemple : responsable, adjoint, interne ou élève).

Le responsable du service informatique de l'hôpital propose le diagramme de classe de la Figure 1 qui reprend l'ensemble de ces éléments.

Notre objectif ici est de développer la base de données pour la gestion de l'hôpital ainsi que le modèle objet correspondant.

2 Théorie : intérêt du *mapping* objet-relationnel

Nous avons étudié dans les séances précédentes l'API **JDBC** qui permet d'accéder à une base de données depuis une application Java. Cependant, il vous faudrait concevoir d'une part le modèle relationnel pour la base de données, et d'autre part le modèle objet pour l'application. Les objets dans le modèle relationnel sont représentés sous une forme tabulaire, alors que dans le modèle objet ils sont représentés par un graphe interconnecté d'objets. Lorsque l'on souhaite stocker ou récupérer un objet dans une base de données relationnelle, certaines incompatibilités peuvent survenir pour les raisons suivantes :

- les modèles objets ont plus de granularité que les modèles relationnels ;

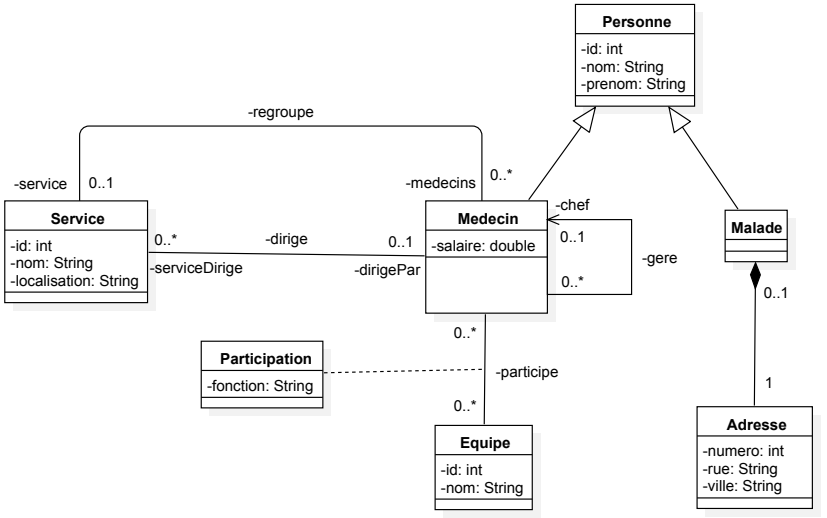

Figure 1: Diagramme de classe pour la gestion du système d'informations de l'hôpital.

- l'héritage n'est pas supporté de base dans les modèles relationnels ;
- les objets ne possèdent pas d'identifiant unique, contrairement au modèle relationnel ;
- la relation entre les entités se traduit de manière différente dans les deux modèles ;
- les modèles relationnels ne sont pas capables de gérer directement les associations à cardinalités multiples.

Par ailleurs, l'utilisation de **JDBC** pour faire des requêtes sur la base oblige aussi à constamment faire la traduction entre modèle objet et modèle relationnel, ce qui est fastidieux !

Ainsi, il est avantageux dans ces cas là d'utiliser ce qu'on appelle le *mapping* objet-relationnel. Il s'agit d'une technique de programmation qui consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objet de la façon la plus facile possible. Il s'agit donc d'une couche qui va interagir entre l'application et la base de données. Ainsi, lors du développement de l'application, il n'est pas nécessaire de connaître l'ensemble des tables et des champs de la base de données, et il n'est plus obligatoire d'utiliser *SQL* pour faire des requêtes sur la base de données.

Dans ce TP, nous étudions l'utilisation de JPA (*Java Persistance API*) pour faire le *mapping* objet-relationnel dans une application Java. Ce *mapping* permet d'assurer la transformation d'objets vers la base de données et vice versa ; que cela soit pour des lectures ou des mises à jour (création, modification ou suppression). L'utilisation de JPA ne requiert aucune ligne de code mettant en œuvre l'API JDBC : JPA propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données.

JPA repose sur des entités qui sont de simples POJOs (*Plain Old Java Object*) annotés et sur un gestionnaire de ces entités (*entity manager*) qui propose des fonctionnalités pour les manipuler (ajout, modification, suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

3 Création de la première entité : les services

3.1 Mise en place du projet

Question 1. Avec le logiciel DataGrip, créez une nouvelle base de données, que vous pouvez nommer **hopital**. Vous pouvez vous reporter à la Section 5.1 si vous ne vous rappelez plus comment faire (normalement il n'est pas nécessaire de refaire les premières étapes si vous avez gardé votre connexion sur le serveur MySQL). Pour le moment, on n'ajoute rien dans cette base de données, on utilisera le code Java pour cela.

Question 2. Créez un nouveau projet JPA (de type *Jakarta EE*) avec IntelliJ IDEA en suivant les indications de la Section 5.2. Ajoutez dans le paquetage source **src/main/java** un paquetage **modele** qui contiendra le modèle de données (les entités).

Question 3. Ajoutez la dépendance pour la connexion avec le serveur MySQL dans le fichier 'pom.xml' comme indiqué dans la Section 5.3.

Question 4. Mettez en place la configuration de l'unité de persistance dans le fichier 'persistence.xml' comme indiqué dans la Section 5.4. En particulier, renommez l'unité de persistance 'HopitalPU', et spécifiez le driver JDBC, l'url de la base de données, ainsi que le nom d'utilisateur et le mot de passe pour se connecter à la base. De plus, assurez-vous d'être en mode *drop and create* lorsque l'application est lancée (ce qui permet pour les premiers tests de toujours partir d'une base de données vide).

3.2 Théorie : généralités sur la persistance des objets avec JPA

3.2.1 Notion d'entité

Une entité est un objet qui peut être rendu persistant (l'état de l'objet peut être stocké dans une base de données). Typiquement, une entité représente une

table dans le modèle relationnel, et chaque instance d'entité correspond à une ligne dans cette table. L'état persistant d'une entité est représenté à travers des champs ou propriétés persistants. Ces champs ou propriétés utilisent des annotations afin de réaliser le *mapping* objet-relationnel.

En fait, une entité est une classe qui doit respecter quelques spécifications :

- elle doit être annotée avec les annotations du paquetage **jakarta.persistence.Entity** ;
- elle doit posséder un constructeur par défaut (avec une visibilité **public** ou **protected**) ;
- elle ne doit pas être déclarée **final**, et aucune de ses méthodes ou attributs persistants ne doivent être déclarés **final** ;
- elle doit posséder un attribut qui représente la clé primaire dans la base de données.

De manière simple, il suffit de rajouter l'annotation **@Entity** au-dessus de la définition d'une classe, et l'annotation **@Id** au-dessus de l'attribut qui représente la clé primaire dans la base pour qu'elle devienne une entité. Il faut également s'assurer d'avoir un constructeur par défaut. Par exemple :

```
1  @Entity
2  public class Employee {
3     @Id
4     private int id;
5     ...
6
7     public Employee() {
8        ...
9     }
10 }
```

3.2.2 Gestionnaire d'entités

Il s'agit d'objets de type **EntityManager** définis dans le paquetage **jakarta.persistence**. Il fournit les méthodes pour gérer les entités : les rendre persistantes, supprimer leur valeur de la base de données, les retrouver dans la base de données.

La méthode **persist(objet)** de la classe **EntityManager** permet de rendre persistant un objet (en fait une entité). L'objet est alors géré par le gestionnaire d'entités : toute modification apportée à l'objet sera enregistrée dans la base de données. L'ensemble des entités gérées par un gestionnaire d'entités s'appelle un contexte de persistance.

3.2.3 Unité de persistance

Il s'agit d'un fichier de configuration qui contient les informations nécessaires pour persister des entités dans une base de données. Elle est définie dans un fichier xml appelé **persistence.xml**. Ce fichier comporte en particulier le nom de l'unité de persistance, un ensemble de propriétés qui permettent de spécifier la connexion à la base de données, ainsi que les noms des classes entités qui sont gérées par cette unité de persistance. C'est le fichier que vous avez configuré en suivant les éléments de la Section 5.4.

3.3 Théorie : quelques précisions sur les entités

Cette section présente quelques détails sur les entités, et en particulier sur les annotations (du paquetage **jakarta.persistence**) qui peuvent être utilisées.

3.3.1 Accès par les attributs

L'état d'une entité doit être accessible au fournisseur de persistance afin de pouvoir stocker les informations dans la base de données. De même, lorsque l'état d'une entité est chargé depuis la base de données, le fournisseur doit pouvoir insérer les informations dans une nouvelle instance de l'entité.

L'accès par les attributs signifie que le fournisseur de persistance passera directement par les attributs de la classe entité pour récupérer ou modifier l'état d'une entité. Dans ce cas là, les annotations se font au-dessus des attributs de la classe entité. Les attributs doivent alors être déclarés avec une visibilité **private** (de préférence), **protected** ou **package**. La visibilité **public** n'est pas autorisée car l'état de l'entité pourrait alors être accédé et modifié par n'importe quelle classe de la machine virtuelle.

3.3.2 *Mapping* vers une table

Pour définir une entité, il faut utiliser, au-dessus de la définition de la classe, l'annotation **@Entity**. Cette annotation peut avoir un attribut **name** qui donne le nom de l'entité. Par défaut, le nom de l'entité est le nom terminal (sans le nom du paquetage) de la classe.

Par ailleurs, lors du *mapping*, dans les cas simples, une table correspond à une classe, et le nom de la table est le nom de la classe. Pour donner à la table un autre nom que le nom de la classe, il faut ajouter une annotation **@Table** en dessous de l'annotation **@Entity**. Cette annotation a un attribut **name** qui donne le nom de la table qui correspond à l'entité. Il est aussi possible à ce niveau de définir des contraintes d'unicité sur plusieurs colonnes de la table, comme présenté dans l'exemple ci-après.

Ainsi, on peut avoir l'exemple suivant :

```
1  @Entity(name="emplEntity")
```

```java
@Table(name="EMP",
   uniqueConstraints={
      @UniqueConstraint(
         columnNames ={"FIRSTNAME", "LASTNAME"})
   }
)
public class Employee {
   @Id
   private int id;
   private String firstName;
   private String lastName;
   ...
}
```

3.3.3 *Mapping* pour les types basiques

Tous les attributs d'une entité qui sont d'un type "basique" de Java peuvent être rendus persistants, et le sont par défaut, sans annotations. Derrière ces types "basiques", on trouve les types primitifs de Java, les **String**, les types temporels de Java et de JDBC, les types énumérés et les types d'objets sérialisables. Notez que le fournisseur de persistance se chargera de faire la conversion entre le type Java et le type JDBC approprié. Il est possible qu'une exception soit levée si la conversion ne peut pas être effectuée de manière automatique, ou si le type n'est pas sérialisable.

En revanche, si on souhaite qu'un attribut ne soit pas persistant, on peut utiliser l'annotation **@Transient** au-dessus de l'attribut. Il est aussi possible d'utiliser le mot clé **transient** lors de la définition : **private transient int var;**

Si l'on souhaite définir de manière plus précise les caractéristiques de la colonne (dans la base de données) associée à un attribut, alors il faut utiliser l'annotation **@Column** au-dessus de la définition de l'attribut. Cette annotation comporte les attributs suivants :

- **name** : précise le nom de la colonne liée (le nom de l'attribut est utilisé par défaut) ;
- **unique** : précise s'il y a contrainte d'unicité sur la colonne (**true**) ou non **false**) ;
- **nullable** : précise si la colonne accepte les valeurs nulles ou non ;
- **insertable** : précise si la valeur doit être incluse lors de l'exécution de la requête SQL INSERT (**true** par défaut) ;
- **updatable** : précise si la valeur doit être mise à jour lors de l'exécution de la requête SQL UPDATE (**true** par défaut) ;
- **columnDefinition** : précise le code SQL pour la définition de la colonne dans la base de données (utile en cas de problème de conversion automatique) ;
- **length** : précise la longueur maximale pour un champ texte (255 par défaut) ;
- **precision** : précise le nombre maximum de chiffres que la colonne peut contenir pour un champ numérique.

Ainsi, on peut avoir l'exemple suivant :

```java
@Entity
public class Employee {
   @Id
   private int id;
   @Column(name="FNAME",
      length = 45,
      unique = true,
      nullable = false
   )
   private String firstName;
   @Column(columnDefinition = "NUMERIC(5,2)")
   private double height;
   ...
}
```

Pour des attributs dont le type est une énumération (qui peut avoir été définie par le programmeur), par défaut la colonne associée sera de type entier, et contiendra la valeur ordinale assignée à l'élément de l'énumération (0 pour le premier élément, 1 pour le deuxième, $n-1$ pour le $n^{\text{ième}}$). Ceci n'est pas toujours très pertinent, notamment par rapport aux potentielles évolutions du code, et on peut souhaiter stoker dans la base les chaînes de caractères qui définissent les constantes de l'énumération.

Pour ce faire, il existe une annotation **@Enumerated** qui se place au-dessus de la définition de l'attribut dont le type est un type énuméré. On peut ainsi définir le type de données stockées : **@Enumerated(EnumType.ORDINAL)** stocke les valeurs ordinales avec un type entier, et **@Enumerated(EnumType.STRING)** stocke les valeurs des constantes avec un type chaîne de caractères.

Les types temporels qui sont considérés comme des types basiques sont : **Date**, **Time** et **Timestamp** du paquetage **java.sql**, ainsi que **Date** et **Calendar** du paquetage **java.util**. Les types de **java.sql** ne posent pas de soucis.

Mais pour les types de **java.util**, il faut des méta-données supplémentaires pour indiquer quel type de données utiliser lors de la conversion vers un type SQL. Pour ce faire, on utilise l'annotation **@Temporal** au-dessus de la définition de l'attribut de type temporel. Et on spécifie la conversion à utiliser : **TemporalType.DATE**, **TemporalType.TIME** and **TemporalType.TIMESTAMP**.

3.3.4 *Mapping* pour les clés primaires

Chaque entité qui est mappée avec une table dans la base de données relationnelle doit posséder un mappage vers la clé primaire dans la table. Ceci est réalisé par l'annotation **@Id** qui indique l'attribut qui identifie de manière unique l'entité. Les règles vues précédemment pour le *mapping* des types "basiques" s'appliquent aussi pour la colonne qui représente l'identifiant de l'entité. Par contre, les clés primaires sont supposées être insérables, non nulles et sans possibilité de mise à jour. Ainsi, si on utilise l'annotation **@Column** sur la clé primaire, il faut veiller à ne pas modifier les attributs **insertable**, **nullable** et **updatable**.

Plusieurs types de données sont autorisés pour les clés primaires, dont les entiers, les flottants, les caractères, **String**, **Date**. On évitera tout de même les types flottants à cause des problèmes de précision.

Si la clé est de type entier, l'annotation **@GeneratedValue** peut être utilisée pour indiquer que la clé sera générée automatiquement par le système de gestion de bases de données. Cette annotation possède un attribut **strategy** qui indique comment la clé sera générée. Cet attribut peut prendre les valeurs :

- **GenerationType.AUTO** : le type de génération est choisi par le fournisseur de persistance, selon le SGBD (c'est la valeur par défaut) ; **si on connait le SGBD utilisé et que l'on sait quelle valeur choisir, il est préférable de ne pas utiliser AUTO** ;
- **GenerationType.SEQUENCE** : la génération de la clé est réalisée en utilisant une séquence (c'est le cas pour une base Oracle ou PostGreSQL par exemple) ; l'annotation **@SequenceGenerator** peut être utilisée pour préciser comment est réalisée la génération de la séquence ;
- **GenerationType.IDENTITY** : une colonne de type **IDENTITY** est utilisée (c'est le cas pour une base MySQL ou Derby par exemple) ;
- **GenerationType.TABLE** : une table qui contient la prochaine valeur de l'identifiant est utilisée ; il est possible de spécifier la table utilisée avec l'annotation **@TableGenerator**.

En résumé, faites votre choix en fonction des possibilités offertes par le SGBD utilisé.

Ainsi, si l'on travaille avec une base MySQL, on peut avoir l'exemple suivant :

```
1  @Entity
2  public class Employee {
3     @Id
4     @GeneratedValue(strategy = GenerationType.IDENTITY)
5     @Column(name = "NEMP")
6     private int id;
7     ...
8  }
```

3.4 Création de l'entité Service

Question 5. En suivant les indications de la Section 5.5, ajoutez dans le paquetage **modele** une entité **Service** qui représente un service de l'hôpital avec un identifiant, un nom et une localisation (voir la Figure 1). La clé primaire sera générée automatiquement (donc avec une stratégie **IDENTIY** pour un serveur MySQL), et dans la base de donnée la colonne devra s'appeler **SERVNO**. Les noms doivent être uniques, et le nom et la localisation doivent avoir une taille maximale et être non nuls. Ils doivent aussi être écrits en majuscules pour éviter les problèmes liés à la casse. Par ailleurs, ajoutez un constructeur par défaut et un constructeur par données du nom et de la localisation. N'oubliez pas de redéfinir les méthodes **equals** et **hashCode** (il faut éviter de se baser sur l'identifiant généré automatiquement pour l'égalité car l'identifiant n'est généré que lorsque l'entité est rendue persistante dans la base). Redéfinissez aussi la méthode **toString**.

3.5 Théorie : utilisation du gestionnaire d'entités

3.5.1 Présentation du concept de gestionnaire d'entités

Pour qu'une entité puisse être rendue persistante dans la base de données, et que l'on puisse manipuler les entités persistantes, il est nécessaire de faire appel aux méthodes d'un gestionnaire d'entités. Ce gestionnaire d'entités est défini en Java par l'interface **EntityManager**. C'est ce gestionnaire d'entités auquel on délègue le réel travail de persistance. Jusqu'à ce qu'un gestionnaire d'entités soit utilisé pour effectivement créer, lire ou écrire une entité, cette entité n'est rien d'autre qu'un objet Java classique (non persistant).

Lorsqu'un gestionnaire d'entités obtient une référence sur une entité, qu'on lui a passé ou qu'il a lue dans la base, cet objet (l'entité) est dit géré par le gestionnaire d'entités. L'ensemble des instances d'entités gérées par un gestionnaire d'entités à un instant donné est appelé un contexte de persistance. Un contexte de persistance vérifie la propriété suivante : à chaque instant, il ne peut pas exister deux entités différentes qui représentent des données identiques dans la base. Par exemple, s'il existe une entité **Employee** avec un identifiant (clé

primaire dans la base) de 158 dans le contexte de persistance, alors aucun autre objet de type **Employee** avec un identifiant valant 158 ne peut exister dans ce même contexte de persistance.

Les gestionnaires d'entités sont configurés pour être capables de rendre persistant et de gérer certains types d'objets, de lire et d'écrire dans une base de données. Les gestionnaires d'entités sont implémentés par un fournisseur de persistance (*EclipseLink* ou *Hibernate* par exemple).

3.5.2 Obtenir un gestionnaire d'entités

Tous les gestionnaires d'entités sont créés par un objet de type **EntityManagerFactory** (on parle de "fabrique"). L'unité de persistance indique les paramètres et les classes d'entités utilisés par tous les gestionnaires d'entités (**EntityManager**) obtenus depuis une instance unique de **EntityManagerFactory** qui est liée directement à l'unité de persistance. Il y a donc une relation $1-1$ entre une unité de persistance et son **EntityManagerFactory**. Pour récupérer l'instance de **EntityManagerFactory** liée à notre unité de persistance, nous pouvons utiliser la classe appelée **Persistence**, dont la méthode statique **createEntityManagerFactory(String persistenceUnitName)** retourne l'instance unique de **EntityManagerFactory** liée à l'unité de persistance nommée **persistenceUnitName**. L'exemple suivant montre comment créer un **EntityManagerFactory** pour l'unité de persistance nommée **EmployeePU** :

```
1  EntityManagerFactory emf =
       Persistence.createEntityManagerFactory("EmployeePU");
```

À présent que nous avons une fabrique (**EntityManagerFactory**), nous pouvons facilement en obtenir un gestionnaire d'entités. L'exemple suivant montre comment obtenir un gestionnaire d'entités à partir de la fabrique obtenue dans l'exemple précédent :

```
1  EntityManager em = emf.createEntityManager();
```

Avec ce gestionnaire d'entités, nous pouvons maintenant commencer à travailler avec des entités persistantes.

3.5.3 Rendre une entité persistante

Rendre une entité persistante est l'opération qui consiste à prendre une entité qui n'a pas de représentation persistante dans la base de données (ou qui a été modifiée hors de la base), et à sauvegarder son état afin qu'il puisse être récupéré plus tard. Pour ce faire, l'interface **EntityManager** dispose de la méthode **persist()**. Le code suivant présente un exemple pour rendre persistant une instance de la classe **Employee** :

```
1  Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
2  em.persist(emp);
```

La première ligne de cet exemple est simplement la création d'une instance **emp** de **Employee**. La seconde ligne utilise le gestionnaire d'entités créé précédemment afin de rendre persistante l'instance **emp**. L'employé **emp** sera alors sauvegardé dans la base de données. Et à la fin de l'appel à **persist()**, **emp** sera une entité gérée dans le contexte de persistance du gestionnaire d'entité.

3.5.4 Rechercher une entité

Lorsqu'une entité est dans la base de données, alors ce que l'on souhaite souvent faire c'est de la récupérer à nouveau (pour la lire ou la modifier). Ceci s'effectue avec la méthode **find()** du gestionnaire d'entité. Par exemple, pour récupérer l'instance de **Employee** dont la clé primaire (identifiant) est 158, on écrira :

```
1  Employee emp = em.find(Employee.class, 158);
```

On passe en paramètres de la méthode **find()** la classe de l'entité recherchée et l'identifiant ou la clé primaire qui identifie l'entité particulière recherchée. Lorsque l'appel à la méthode **find()** est terminé, l'objet renvoyé par la méthode sera une entité gérée par le gestionnaire d'entités, i.e. qu'elle fera partie du contexte de persistance associé au gestionnaire d'entité.

Par ailleurs, vous pouvez remarquer qu'il n'est pas nécessaire de caster l'objet renvoyé par la méthode **find()**. En effet, le type de retour de la méthode **find** est le même que la classe qui est passée en paramètre, i.e. que formellement la signature de la méthode est : **<T> T find(class<T> classeEntite, Object clePrimaire)**.

Si l'objet recherché a été supprimé de la base de données, ou que la clé primaire n'existe pas, alors la méthode retourne **null**. Lors de l'utilisation de la variable qui doit contenir l'entité recherchée, il faut donc vérifier que sa valeur n'est pas **null**.

3.5.5 Supprimer une entité

La suppression d'une entité revient à la supprimer de la base de données, comme si l'on effectuait une requête de type *DELETE*. Afin de supprimer une entité,

cette entité doit être gérée, i.e. qu'elle est présente dans le contexte de persistance. L'application doit donc charger l'entité (la rendre persistance ou la rechercher dans la base) avant de la supprimer. La méthode pour supprimer une entité est la méthode **remove()**. Par exemple, pour supprimer une entité que l'on vient de rechercher dans la base, on écrira :

```
1  Employee emp = em.find(Employee.class, 158);
2  if (emp != null) {
3     em.remove(emp);
4  }
```

Dans cet exemple, on fait d'abord le chargement de l'entité depuis la base de données, puis on la supprime par un appel à la méthode **remove()** sur le gestionnaire d'entités. On vérifie avant de supprimer une entité que cette dernière n'est pas **null** afin que la méthode **remove()** ne lève pas une exception de type **IllegalArgumentException**.

3.5.6 Mise à jour d'une entité

Une entité peut être mise à jour de différentes manières. Pour le moment, nous présentons le cas le plus fréquent et le plus simple. C'est le cas où une entité est gérée et nous souhaitons y apporter des modifications. Il est toujours possible de faire appel à la méthode **persist()** ou **find()** sur le gestionnaire d'entités afin de récupérer une référence sur une entité gérée. L'exemple ci-dessous propose d'augmenter de 1000 le salaire de l'employé dont l'identifiant est 158 :

```
1  Employee emp = em.find(Employee.class, 158);
2  emp.setSalary(emp.getSalary() + 1000);
```

Notez bien que dans ce cas là nous ne faisons pas appel au gestionnaire d'entités pour modifier l'objet, on modifie directement l'objet. Pour cette raison il est important que l'entité soit gérée, sinon le fournisseur de persistance n'a aucun moyen de détecter le changement, et aucun changement ne sera alors effectué sur la représentation persistante de l'employé.

3.5.7 Transactions

Pour le moment, nous n'avons pas évoqué la notion de transaction lorsque l'on manipule des entités. Normalement, les entités sont créées, mises à jour, et supprimées dans une transaction, et une transaction est nécessaire lorsque des changements sont réalisés sur la base de données. Les changements réalisés sur la base de données réussissent ou échouent de manière atomique, c'est pourquoi la vue persistante d'une entité doit être transactionnelle. On évite ainsi des états inconsistants.

En fait, dans tous les exemples précédents, sauf pour l'appel de la méthode **find()**, nous aurions dû entourer les appels aux méthodes par une transaction. La méthode **find()** n'est pas une opération mutante, ainsi elle ne nécessite pas une transaction.

Les transactions sont gérées à l'aide d'objets de type **EntityTransaction**. Il est nécessaire de commencer et de valider (*commit*) une transaction avant et après l'appel aux méthodes de manipulation des entités. Pour récupérer un **EntityTransaction**, il faut appeler la méthode **getTransaction()** sur le gestionnaire d'entités (**EntityManager**). Ensuite, la méthode **begin()** de **EntityTransaction** permet de commencer une transaction, et la méthode **commit()** permet de valider la transaction. Si on ne souhaite pas valider la transaction (si une exception a été levée par exemple), alors on peut appeler la méthode **rollback()** de **EntityTransaction** afin d'annuler la transaction.

Par exemple, la persistance d'une nouvelle entité peut se faire de la manière suivante :

```
1  EntityTransaction et = em.getTransaction();
2  et.begin();
3  Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
4  em.persist(emp);
5  et.commit();
```

3.5.8 Synthèse

Voici un exemple complet d'utilisation du gestionnaire d'entités :

```
1  public static void main(String[] args) {
2     final EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("EmployeePU");
3     final EntityManager em = emf.createEntityManager();
4     try{
5        final EntityTransaction et = em.getTransaction();
6        try{
7           et.begin();
8           // creation d'une entite persistante
9           Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
10          em.persist(emp);
11          et.commit();
12       } catch (Exception ex) {
13          System.out.println("exception: " + ex);
14          System.out.println("rollback");
15          et.rollback();
```

```
16          }
17      } finally {
18          if(em != null && em.isOpen()){
19              em.close();
20          }
21          if(emf != null && emf.isOpen()){
22              emf.close();
23          }
24      }
25  }
```

Les objets de type **EntityManagerFactory**, **EntityManager** et **EntityTransaction** peuvent être déclarés **final** car ils ne doivent pas être modifiés. Par ailleurs, en cas de levée d'une exception, il faut bien veiller à faire un **rollback()** sur la transaction pour l'annuler. Et dans tous les cas (bloc **finally**), on ferme le gestionnaire d'entités et la fabrique de gestionnaire d'entités afin de libérer les ressources qui leur sont associées, avec la méthode **close()**.

3.6 Test persistance d'entités Service

Question 6. Ajoutez un paquetage pour faire des tests, et ajoutez-y une classe **Test1** dans laquelle vous ferez un test de persistances d'entités **Service**. Vous pouvez vous baser sur le code suivant :

```
1  et.begin();
2  Service serv1 = new Service("Cardiologie", "Bat A, 1er étage");
3  Service serv2 = new Service("Pneumologie", "Bat B, 1er étage");
4  Service serv3 = new Service("Urgence", "Bat C, 1er étage");
5
6  em.persist(serv1);
7  em.persist(serv2);
8  serv1.setLocalisation("Bat D, 2ème étage");
9  et.commit();
```

Quel est le contenu de la base de données après l'exécution de la méthode de test ? Vérifiez que vous savez expliquer pourquoi la base contient ces données.

Question 7. Afin de bien comprendre la notion de transaction, et l'intérêt du *rollback*, modifiez le test précédent de la manière suivante : lors de la création du service **serv2**, (ligne 3 du code de la question précédente), donnez le nom 'Cardiologie' comme pour le service **serv1**. Les noms devant être uniques, l'entité **serv2** ne devrait pas être persistée. Faites le test, et regardez ce qui apparait sur la console (en détail), ainsi que ce qui se passe dans la base de données.

Refaites le même test en commentant l'appel à la méthode **rollback** de la classe **EntityTransaction**, et vérifiez à nouveau ce qu'il se passe dans la base de données.

Assurez-vous de bien comprendre ce qui se passe. Ensuite, revenez au test initial de la question 6.

Question 8. Dans le fichier *persistance.xml* qui décrit l'unité de persistance, modifiez l'action à réaliser lorsque l'application est lancée : choisissez *none* à la place de *drop and create*. Que se passe-t-il quand vous ré-exécutez le test ?

Revenez ensuite en mode *drop and create*. Modifiez le fichier *persistance.xml*, afin de générer les scripts SQL de création et suppression des tables. Vous pouvez vous référer à la Section 5.4 pour plus de détails. Exécutez à nouveau le test, et regarder que deux scripts ont bien été créés dans le répertoire de votre projet. Regardez ces scripts en mode texte et assurez-vous de comprendre ce qui est écrit.

4 Associations : médecins et services

4.1 Théorie : *mapping* des associations avec JPA

Si les entités étaient seulement composées d'attributs simples, alors le *mapping* objet-relationnel serait trivial. Mais la plupart des entités doivent être capables de référencer ou d'être en relation avec d'autres entités.

Pour rappel, les associations bidirectionnelles peuvent être vues comme une paire d'associations unidirectionnelles. Ainsi, en fonction des cardinalités de la source et de la cible de la relation, on définit 4 types de relations unidirectionnelles :

- association *Many-to-One* (N − 1) ;
- association *One-to-One* (1 − 1) ;
- association *One-to-Many* (1 − N) ;
- association *Many-to-Many* (M − N).

Ces noms de type de relation sont aussi ceux utilisés dans les annotations qui sont utilisées pour indiquer le type de relation d'un attribut pour le *mapping*. Les associations sont donc déclarées dans les entités à l'aide d'annotations.

4.1.1 *Mapping* Many-to-One

En UML, la classe source possède un attribut implicite du type de la classe cible si l'association est navigable. Considérons l'exemple de la Figure 2 avec une association *Many-to-One* entre **Employee** et **Department**.

Dans le modèle objet, la classe **Employee** possédera un attribut de type **Department** qui permettra de référencer un seul département.

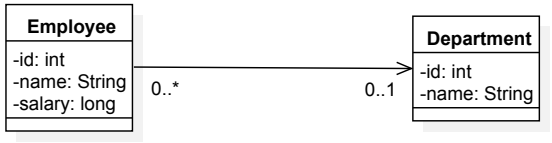

Figure 2: Exemple d'association *Many-to-One* en UML.

Un *mapping Many-to-One* est défini en annotant l'attribut dans l'entité source (l'attribut qui réfère sur l'entité cible) avec l'annotation **@ManyToOne**. En poursuivant notre exemple, cela donne :

```
1  @Entity
2  public class Employee {
3  // ...
4     @ManyToOne
5     private Department department;
6  // ...
7  }
```

Dans la base de données, le *mapping* d'une association signifie que l'on aura une clé étrangère : une colonne d'une table réfère la clé (en principe la clé primaire) d'une autre table. Avec JPA, on parle de colonne de jointure (*join column*), et l'annotation **@JoinColumn** est utilisée pour configurer ce type de colonne.

Dans presque toutes les associations, indépendamment de la source et de la cible, un des deux côtés de l'association aura la colonne de jointure dans sa table. Ce côté est appelé le propriétaire de l'association. Ceci est important pour le *mapping* car les annotations qui définissent le *mapping* sur les colonnes dans la base (par exemple **@JoinColumn**) sont toujours définies du côté du propriétaire de l'association.

Dans le cas d'une association **Many-to-One**, le côté propriétaire de l'association est la source, qui contient aussi la référence vers la cible. C'est donc là aussi que devra se trouver l'annotation **@JoinColumn**.

Par défaut une colonne de jointure sera nommée dans la base de données avec le nom de l'attribut qui référence la cible dans l'entité source, un *underscore* et le nom de la clé primaire dans l'entité cible. Avec l'annotation **@JoinColumn** il est possible de définir le nom de la colonne de jointure avec l'attribut **name**, comme présenté dans l'exemple suivant :

```
1  @Entity
2  public class Employee {
3  // ...
4     @ManyToOne
5     @JoinColumn(name="DEPT_ID")
6     private Department department;
7  // ...
8  }
```

4.1.2 *Mapping* One-to-One

Dans le cas d'un *mapping One-to-One*, on utilisera l'annotation **@OneToOne**. Et on pourra naturellement utiliser l'annotation **@JoinColumn** afin de nommer la colonne de jointure.

Dans le cas d'un *mapping* **One-to-One** bidirectionnel, il faudra utiliser deux associations unidirectionnelles. Mais au niveau de la base de données, une seule entité pourra être propriétaire de l'association, et dans le cas de l'association *One-to-One*, cela peut être n'importe lequel des côtés de l'association. L'annotation **@JoinColumn** sera donc utilisée uniquement du côté du propriétaire. En revanche, de l'autre côté, il faut spécifier quel est le côté propriétaire en définissant l'attribut **mappedBy** de l'annotation **@OneToOne**. La valeur affectée à **mappedBy** doit être le nom de l'attribut dans l'entité propriétaire qui réfère sur l'entité non propriétaire de l'association.

Voici un exemple d'association *One-to-One* bidirectionnelle entre un employé et une place de parking :

```
1  @Entity
2  public class Employee {
3  // ...
4     @OneToOne
5     @JoinColumn(name="PSPACE_ID")
6     private ParkingSpace parkingSpace;
7  // ...
8  }
```

```
1  @Entity
2  public class ParkingSpace {
3     // ...
4     @OneToOne(mappedBy="parkingSpace")
5     private Employee employee;
6     // ...
7  }
```

4.1.3 *Mapping* One-to-Many

Lorsque l'entité source référence une ou plusieurs instances d'entités cibles, une collection d'associations est utilisée. Lorsqu'une entité est associée avec une collection d'autres entités, c'est le plus souvent sous la forme d'un *mapping One-to-Many*.

Par exemple, si l'on prend le cas d'un département constitué de plusieurs employés (un employé ne pouvant faire partie que d'un seul département), alors il s'agit cette fois d'une association *Many-to-One*. Mais cette fois, l'association est a priori bidirectionnelle par nature. En effet, lorsque l'entité source possède un nombre arbitraire d'entités cibles dans une collection, il n'est pas possible au niveau de la base de données de stocker un nombre indéfini de clés étrangères dans la table de l'entité source. Ainsi une solution souvent mise en place est de laisser la table de l'entité cible stocker une clé étrangère qui réfère l'entité source de l'association. C'est pourquoi les associations *One-to-Many* sont bidirectionnelles et que le propriétaire de l'association est la cible de l'association.

Ainsi, pour l'association entre les départements et les employés, le propriétaire de l'association étant l'entité **Employee**, l'entité sera celle définie précédemment dans le *mapping Many-to-One*. Du côté non propriétaire de l'association (l'entité **Department**), il faut faire le *mapping* de la collection de **Employee** comme une association *One-to-Many* en utilisant l'annotation **@OneToMany**. Comme il ne s'agit pas du côté propriétaire de l'association, il faut alors définir l'attribut **mappedBy** de l'annotation **@OneToMany** comme présenté pour le *mapping One-to-One* bidirectionnel. Ainsi, pour la définition de l'entité **Department**, nous obtenons :

```
1  @Entity
2  public class Department {
3  // ...
4     @OneToMany(mappedBy="department")
5     private Collection<Employee> employees;
6  // ...
7  }
```

Plusieurs types de collections peuvent être utilisés pour sauvegarder les entités multiples d'une association. Selon les besoins, il est possible d'utiliser les interfaces génériques **Collection<E>**, **Set<E>**, **List<E>** et **Map<K,E>**. Il y a juste quelques règles à suivre pour leur utilisation. La première règle est de définir la collection avec l'une des interfaces mentionnées : les classes qui implémentent ces interfaces ne doivent pas être utilisées dans la définition des attributs mais seront utilisées pour l'initialisation des attributs. La seconde règle est que la classe d'implémentation de la collection peut être invoquée tant que l'entité n'est pas gérée par le gestionnaire d'entités. Une fois que l'entité est gérée, il faut toujours utiliser l'interface de la collection pour manipuler cette dernière. Ceci est dû au fait que lorsque l'entité est gérée, le fournisseur de persistance peut remplacer le type concret de la collection par une autre implémentation de son choix.

Si on utilise une collection de type **List**, alors il est possible d'utiliser l'annotation **@OrderBy** au-dessus de la définition de la liste afin de spécifier l'ordre dans lequel on veux récupérer les éléments dans la base de données. Cependant, l'ordre n'est pas préservé lorsque les données sont écrites dans la base. Par défaut, l'ordre est donné par la clé primaire, mais on peut aussi spécifier en paramètre de l'annotation les attributs (séparés par des virgules) qui définissent l'ordre, et si l'ordre est croissant (**ASC**) ou décroissant (**DESC**).

Ainsi, si l'entité **Employee** possède un attribut **status** et un attribut **name**, alors on peut écrire dans l'entité **Department** :

```
1  @OneToMany(mappedBy="department")
2  @OrderBy("status DESC, name ASC")
3  private List<Employee> employees;
```

Si on utilise une collection de type **Map**, et que la clé est un attribut de l'entité cible de l'association, alors il faut utiliser l'annotation **@MapKey** au-dessus de la définition de la table associative, et spécifier avec l'attribut **name** quel est le nom de l'attribut qui va servir de clé.

Ainsi, si dans **Department**, on souhaite utiliser une table associative pour stocker les employés, et dont la clé est le nom de l'employé, alors on a :

```
1  @OneToMany(mappedBy="department")
2  @MapKey(name="name")
3  private Map<String, Employee> employees;
```

4.2 Théorie : opérations en cascade

Par défaut, chaque méthode du gestionnaire d'entités s'applique uniquement à l'entité passée en argument de la méthode. La méthode ne va pas être appliquée en cascade sur les autres entités qui sont liées par une association à l'entité sur laquelle s'opère la méthode. Pour des méthodes comme **remove()**, c'est en général le comportement souhaité. Mais pour **persist()**, il est fortement probable que si une entité est en relation avec une autre entité, on souhaite que les deux entités soient rendues persistantes en même temps.

Heureusement, JPA fournit un mécanisme pour définir quand est-ce que les méthodes comme **persist()** doivent être appliquées en cascade sur les relations. Pour cela, on utilise l'attribut **cascade** des annotations d'association (**@OneToOne**, **@OneToMany**, **@ManyToOne**). On y définit la liste des opérations du gestionnaire d'entités à appliquer en cascade pour l'association

considérée. Les valeurs peuvent être : **CascadeType.PERSIST**, **CascadeType.REFRESH**, **CascadeType.REMOVE**, **CascadeType.MERGE** et **CascadeType.ALL** qui correspond à toutes ces opérations.

L'attribut **cascade** est unidirectionnel. Si on souhaite avoir le même comportement des deux côtés d'une association, il faut donc l'écrire de manière explicite des deux côtés de l'association.

Par exemple, si l'on souhaite que lorsqu'un employé est rendu persistant ou supprimé, sa place de parking le soit aussi, on peut écrire :

```
1  @Entity
2  public class Employee {
3     // ...
4     @OneToOne(cascade={
5        CascadeType.PERSIST,
6        CascadeType.REMOVE })
7     @JoinColumn(name="PSPACE_ID")
8     private ParkingSpace parkingSpace;
9     // ...
10 }
```

4.3 Associations entre les médecins et les services

On souhaite à présent ajouter l'entité **Medecin** ainsi que les relations entre les entités **Service** et **Medecin** et la relation réflexive sur **Medecin**. Pour le moment, on ne s'occupe pas de la relation d'héritage : **Medecin** hérite de **Personne**, et tous les attributs de l'entité **Personne** sont déportés dans l'entité **Medecin**. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 3.

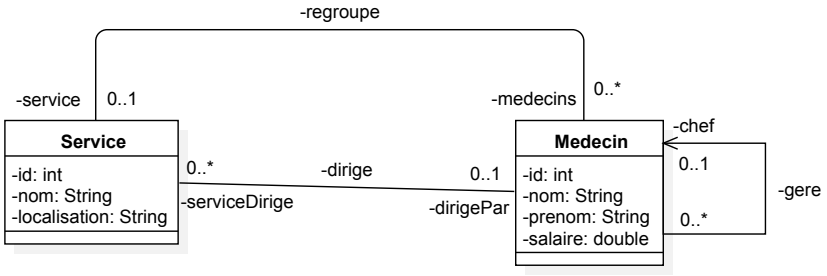

Figure 3: Diagramme de classe pour les entités **Medecin** et **Service**.

Question 9. Ajoutez dans le paquetage **modele** une entité **Medecin** qui représente un médecin de l'hôpital avec un identifiant, un nom, un prénom et un salaire (voir la Figure 3). La clé primaire sera générée automatiquement. Le couple nom et prénom doit être unique. Les noms et prénoms doivent être non nuls. Ils doivent aussi être écrits en majuscules pour éviter les problèmes liés à la casse. Le salaire contient deux chiffres après la virgule. Par ailleurs, ajoutez un constructeur par défaut et un constructeur par données du nom, du prénom et du salaire. Redéfinissez aussi les méthodes **equals**, **hashCode** et **toString**.

Une remarque sur la précision numérique de la colonne liée à l'attribut **salaire** : on peut utiliser dans l'annotation **@Column** l'attribut **columnDefinition** est lui donner une valeur de type **NUMERIC(p,s)** où **p** représente la précision et **s** le nombre de chiffres après la virgule.

Question 10. Dans le paquetage **tests**, ajoutez une classe **Test2** dans laquelle vous ferez un test de persistance d'entités **Medecin**. Vous pouvez vous baser sur le code suivant :

```
1  et.begin();
2  Medecin med1 = new Medecin("Trancen", "Jean", 2135.23);
3  Medecin med2 = new Medecin("Gator", "Coralie", 3156.00);
4  Medecin med3 = new Medecin("Gator", "Magalie", 2545.3723);
5  em.persist(med1);
6  em.persist(med2);
7  em.persist(med3);
8  et.commit();
```

Vérifiez que les médecins sont bien enregistrés en base, et que le salaire contient au maximum 2 chiffres de précision. Vous pouvez aussi tester ce qu'il se passe si vous essayez de rendre persistants deux médecins avec le même nom et prénom.

Question 11. Modifiez les entités **Service** et **Medecin** afin d'ajouter les associations bidirectionnelles **regroupe** et **dirige** entre ces entités (voir la Figure 3). Ajoutez dans la classe **Service** une méthode **public boolean addMedecin(Medecin m)** qui ajoute le médecin **m** au service, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non. Si l'ajout est réalisé, il faut aussi mettre à jour le service du médecin. Si le médecin était déjà dans un autre service, il faut l'en retirer, pour garder les données consistantes. De la même manière, ajoutez dans la classe **Medecin** une méthode **public boolean addServiceDirige(Service s)** qui ajoute le service **s** aux services dirigés par le médecin, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non. Si l'ajout est réalisé, il faut aussi mettre à jour le médecin qui dirige le service. Si le service était dirigé par un autre médecin, il faut retirer ce service des services dirigés par ce médecin, pour garder les données consistantes.

Il faut donc faire ici deux associations bidirectionnelles entre les classes **Service** et **Medecin**. Vous pouvez donc ajouter toutes les méthodes que vous jugez nécessaires afin de réaliser proprement ces associations bidirectionnelles. Il faut vous assurer que les données restent bien cohérentes.
Par ailleurs, réfléchissez bien au choix des structures de données que vous utilisez.

Question 12. Dans le paquetage **tests**, ajoutez une classe **Test3** dans laquelle vous ferez un test de persistance d'entités **Medecin** et **Service**. Vous pouvez vous baser sur le code suivant :

```
1  Service serv1 = new Service("Cardiologie", "Bat A, 1er étage");
2  Service serv2 = new Service("Pneumologie", "Bat B, 1er étage");
3  Service serv3 = new Service("Urgence", "Bat C, 1er étage");
4  Medecin med1 = new Medecin("Trancen", "Jean", 2135.23);
5  Medecin med2 = new Medecin("Gator", "Coralie", 3156.00);
6  Medecin med3 = new Medecin("Gator", "Magalie", 2545.37);
7  Medecin med4 = new Medecin("Hitmieu", "Helmer", 1873.30);
8  Medecin med5 = new Medecin("Cambronne", "Maude", 3765.20);
9  Medecin med6 = new Medecin("Haybon", "Sylvain", 2980.00);
10 serv1.addMedecin(med1);
11 serv1.addMedecin(med2);
12 serv1.addMedecin(med3);
13 serv2.addMedecin(med4);
14 serv3.addMedecin(med5);
15 serv3.addMedecin(med6);
16 med4.addServiceDirige(serv2);
17 med5.addServiceDirige(serv1);
18 med5.addServiceDirige(serv3);
```

Pour la persistance, vous pourrez par exemple utiliser la persistance en cascade, et utiliser le code suivant :

```
1  et.begin();
2  em.persist(serv1);
3  em.persist(serv2);
4  em.persist(serv3);
5  et.commit();
```

Vérifiez que les enregistrements en base sont effectués correctement, et regardez aussi la structure des tables avec les clés étrangères. Par ailleurs, n'hésitez pas à faire des tests différents afin de bien comprendre la notion de persistance en cascade.

Question 13. Modifiez l'entité **Medecin** afin d'ajouter l'association réflexive **gere** (voir la Figure 3). Ajoutez dans la classe **Medecin** une méthode **public boolean setChef (Medecin m)** qui défini le chef du médecin, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non (typiquement un médecin ne peut pas être le chef de lui-même ...). Modifiez le jeu de test précédent (**Test3**) en ajoutant une hiérarchie dans les médecins, et vérifiez que tout fonctionne correctement.

Ici, nous proposons une association unidirectionnelle, donc c'est plus simple à gérer.

5 Quelques notions utiles

5.1 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 'hopital', écrivez le code suivant dans la console puis exécutez-le :

```
1 CREATE DATABASE hopital;
```

5.2 Création d'une application JPA avec IntelliJ IDEA

IntelliJ IDEA permet de créer des applications Jakarta EE. Jakarta EE est une extension de la plateforme standard Java SE que nous utilisons d'habitude. Jakarta EE fournit entre autre une API de *mapping* objet-relationnel (JPA), ce qui nous intéresse ici.

Voici les étapes à suivre afin de créer une application JPA avec IntelliJ IDEA.

1. Ouvre IntelliJ IDEA.
2. Cliquez sur 'File', puis 'New', puis 'Project'.
3. Dans la fenêtre qui s'est ouverte, dans la colonne à gauche, cliquez sur 'Jakarta EE'.
4. Renseignez le nom du projet, le répertoire dans lequel vous souhaitez enregistrer le projet, et choisissez pour le 'Template' la valeur 'Library'.
5. Cliquez sur 'Next'.
6. Dans la fenêtre qui vient de s'ouvrir, sélectionnez dans la colonne de gauche, 'Persistence (JPA)' parmi les spécifications, ainsi que 'Hibernate' parmi les implémentations. Notez que comme pour JDBC, l'utilisation de JPA nécessite un fournisseur de persistance qui implémente les classes et les méthodes de l'API. EclipseLink et Hibernate sont les fournisseurs de persistance les plus connus. Nous proposons ici de choisir Hibernate.
7. Cliquez sur 'Create'.
8. Attendez un petit peu, et vous devriez voir apparaître en bas à droite un message 'Maven 'ProjectName' build scripts found'. Cliquez alors sur le bouton 'Load Maven Project'. Pour votre information, notez que Maven est un outil permettant d'automatiser la gestion de projets Java. Il permet notamment de compiler et de déployer des applications Java, ainsi que de gérer les librairies requises par l'application.

Vous devez maintenant voir dans l'onglet 'Project' un paquetage 'src' dans lequel se trouve un paquetage 'main' dans lequel il y a un paquetage 'java' qui contiendra vos fichiers sources. Dans le paquetage 'main' il y a également un paquetage 'resources' dans lequel vous allez trouver un fichier 'persitence.xml' qui contient la configuration pour réaliser le *mapping* objet-relationnel. Enfin, vous devez voir dans le projet un fichier 'pom.xml' qui contient les informations à propos du projet et de sa configuration, en particulier les informations nécessaires à la compilation, ainsi que les dépendances à d'autres librairies.

5.3 Dépendance pour la connexion avec le serveur MySQL

Après avoir créé une application JPA (voir Section 5.2), vous pouvez suivre les étapes suivantes pour ajouter les dépendances pour la connexion avec le serveur MySQL. Ceci se fait en modifiant le fichier 'pom.xml' de la manière suivante :

1. dans votre projet, cliquez sur le fichier 'pom.xml' ;
2. ajoutez une ligne blanche avant la fin de la section '<dependencies>' ;
3. faites un clic-droit puis 'Generate...', et sélectionnez 'Dependency' (ou 'Add dependency...') ;
4. dans l'onglet qui vient de s'ouvrir, il y a une barre de recherche dans laquelle vous tapez 'mysql', puis vous lancez la recherche ;
5. sélectionnez la ligne avec le driver MySQL Connector/J 'com.mysql:mysql-connector-j' ;

6. cliquez sur 'Add', et vous devez voir apparaître la dépendance vers le connecteur MySQL dans le fichier 'pom.xml' ;

7. il reste alors à charger les modifications dans Maven : pour se faire, vous pouvez cliquer sur le bouton qui est apparu en haut à droite dans le fichier 'pom.xml', ou bien utiliser le raccourci 'Ctrl+Maj+O'.

5.4 Configuration du *mapping* objet-relationnel dans le fichier de persistance

Après avoir créé une application JPA (voir Section 5.2), vous pouvez suivre les étapes suivantes pour configurer le fichier de configuration du *mapping* objet-relationnel :

1. dans le projet, dans le paquetage 'src', puis 'main', puis 'resources' puis 'META-INF', ouvrez le fichier 'persistence.xml' ;

2. dans ce fichier, dans la balise 'persistence-unit', vous trouvez l'attribut 'name' dans lequel vous pouvez modifier le nom de l'unité de persistance ;

3. dans le contenu de l'élément 'persistence-unit', vous pouvez ajouter un élément 'properties' qui contiendra les propriétés de l'unité de persistance ;

4. dans les propriétés, vous pouvez en particulier ajouter les éléments suivants (une balise 'property' pour chaque propriété, avec un attribut 'name' et un attribut 'value') :

   - 'jakarta.persistence.jdbc.driver' pour donner la classe du driver JDBC ('com.mysql.cj.jdbc.Driver' pour le driver MySQL) ;
   - 'jakarta.persistence.jdbc.url' pour donner l'url de la base de données (devrait être 'jdbc:mysql://localhost:3306/nombdd' pour une base nommée 'nombdd' avec un serveur MySQL) ;
   - 'jakarta.persistence.jdbc.user' pour donner le nom d'utilisateur de la base de données ;
   - 'jakarta.persistence.jdbc.password' pour donner le mot de passe pour l'accès à la base de données ;
   - 'jakarta.persistence.schema-generation.database.action' permet de spécifier l'action à réaliser sur la base de donnée lorsque l'application est exécutée ; la valeur 'drop-and-create' permet d'effacer toutes les tables et de les recréer, ce qui est pratqiue lors des premiers tests ; la valeur 'none' permet de ne rien modifier ce qui est pratique une fois que les tests ont été effectués ;
   - 'jakarta.persistence.schema-generation.scripts.action' permet de spécifier pour quelle action on souhaite générer un script SQL ; les valeurs sont 'drop', 'create', 'drop-and-create' et 'none' ;
   - 'jakarta.persistence.schema-generation.scripts.create-target' permet de spécifier le nom du fichier dans lequel on souhaite écrire le script de création des tables (le fichier sera à la racine du projet) ;
   - 'jakarta.persistence.schema-generation.scripts.drop-target' permet de spécifier le nom du fichier dans lequel on souhaite écrire le script de suppression des tables (le fichier sera à la racine du projet).

5.5 Ajouter une entité dans un projet JPA

Voici les étapes à suivre pour créer une nouvelle entité dans un projet JPA avec IntelliJ IDEA :

1. dans un premier temps, il faut ouvrir l'onglet avec la vue de persistance ; pour ce faire, cliquez sur 'View', puis 'Tool Windows', puis 'Persistence' ;

2. dans l'onglet 'Persistence', faites un clic-droit sur le nom de l'unité de persistance, puis cliquez sur 'New', puis sélectionnez 'Entity' ;

3. dans la fenêtre qui vient de s'ouvrir, rentrez le nom de la classe (i.e. de l'entité) et le nom du paquetage, puis cliquez sur 'OK' ;

4. vous devez alors voir dans l'onglet 'Persistence' que la nouvelle entité est apparue sous le nom de l'unité de persistance ; et qu'une nouvelle classe a été crée ;

5. il vous reste alors à modifier cette classe avec les attributs, constructeurs, méthodes et annotations nécessaires.

Par ailleurs, notez que dans l'unité de persistance, i.e. le fichier 'persistence.xml', vous devez voir que le nom de la nouvelle classe est apparu, comme étant une entité gérée par votre unité de persistance.

References

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