La persistance des données et le mapping relationnel objet

Dans le langage Java, l'interaction avec les bases de données relationnelles se fait généralement via l'interface (API) JDBC (Java DataBase Connectivity). Les classes de cette interface sont dans les packages `java.sql` et `javax.sql`.
Comme il existe différents fournisseurs de SGBD, il faut disposer d'un drivers qui permet de communiquer avec le SGBD que l'on a choisi.

JDBC fourni les outils pour :

- se connecter à un SGBD (interface `Connection`),
- envoyer des requêtes SQL ou appeler des procédures stockées,
- représenter les résultats sous forme de `ResultSet` et les parcourir

Vous pouvez trouver plus d'information sur :
https://en.wikipedia.org/wiki/Java_Database_Connectivity

Au dessus de JDBC, il existe des interfaces de programmation pour faire le lien entre un modèle objet java et une modèle relationnel. C'est ce que l'on appelle le mapping relationnel objet (ORM).
Ces interfaces permettent de manipuler une base de données relationnelle comme si c'était un stockage d'objets. Le standard utilisé dans JakartaEE est **Jakarta Persistence API (JPA)**.
Il existe plusieurs implémentation de JPA dont les plus connues sont Hibernate, EclipseLink et OpenJPA. Cette année nous utiliseront EclipseLink (livré par défaut dans Payara)

Avec JPA, la définition du mapping relationnel-objet est fait grâce aux annotations, l'interrogation de la base est réalisée via le langage JPQL (+ EntityGraphs) ou la Criteria API.

Pour pouvoir utiliser JPA dans un projet de web service REST, il faut
- ajouter la dépendance vers `jakarta.jakartaee-web-api`
- créer le fichier `persistence.xml` dans `META-INF`
- configurer une base de données.

1- La base

Dans le projet fourni, tout cela est préparé.
La base de donnée est configurée dans la classe `Cours3RestApplication` avec l'annotation `@DataSourceDefinition`.
Le fichier `persistence.xml` est dans le dossier `resources/META-INF` sous Idea. Vous pourrez constater que l'élément `<jta-data-source>`pointe vers le nom de la source de donnée définie via l'annotation `@DataSourceDefinition`.

Ensuite il faut annoter les classes du modèle objets qui seront mappés dans la base de données.
Pour qu'une classe soit gérée par JPA, il faut au minimum :

- qu'elle soit un Bean (au moins un constructeur sans paramètre + attributs privés avec getters et setters)

- ajouter l'annotation `@Entity` devant la classe (avec `import javax.persistence.*`).
- ajouter l'annotation `@Id` devant l'attribut clé primaire de la classe (si la clé primaire est composée de plusieurs attributs, c'est autre chose…)

Annotez la classe `Person` pour qu'elle soit une Entité JPA. Ajouter l'annotation @GeneratedValue en plus de @Id pour que l'identifiant soit généré automatiquement.

Dans la classe PersonEndpoint :

Annotez la classe avec @Transactional. Cela permet d'indiquer à l'environnement d'exécution que chaque méthode de cette classe s'exécutera dans une transaction. Si une exception est levée durant l'exécution d'une méthode, alors la transaction en cours (et donc les modifs à la bd) sera annulée (rollback). On aurait aussi pu annoter seulement les méthodes concernée au lieu d'annoter la classe entièrement.

Ajoutez l'attribut :

```
@PersistenceContext(unitName="cours3PU")
private EntityManager em;
```

L'entity manager est l'objet à partir duquel la base de donnée sera manipulée.
L'annotation au dessus permet d'indiquer, à l'environnement d'exécution, d'injecter une instance à partir du contexte de persistance « `cours3PU` » que nous avons déclaré dans le fichier `persistence.xml`.

A partir d'une instance d'`EntityManager`, on peut ajouter (`persist`), lire (`find`), mettre à jour (`merge`), supprimer (`remove`).
En vous aidant de la documentation, https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html, réalisez les différentes méthodes (create, read, updateFull et delete) du service web PersonEndpoint.

Testez vos différentes méthodes. Vous pouvez visualiser le contenu de la BD en allant sur l'URL: http://localhost:8080/cours3/dbconsole
Les paramètres d'accès sont les suivants :

```
Driver class : org.h2.Driver
JDBC URL : jdbc:h2:./test
User Name : cours3Adm
Password : cours3Pass
```

2- Les relations entre classes.

JPA permet de définir des relations entre classes via les annotations `@ManyToMany`, `@ManyToOne`, `@OneToMany`.

Par exemple, nous voulons ajouter des contacts (téléphoniques) aux personnes.
Pour cela, nous ajoutons la classe `Contact` (elle est fournie). Une personne peut avoir plusieurs contacts et un contact appartient qu'à une seule personne.

Dans cette classes la relation vers Personne est annotée via `@ManyToOne`.

A partir de cette classe (décommentez le `@Entity` pour l'utiliser), proposez une méthode dans le service web pour ajouter un contact à une personne.
Pour cela, on enverra une requête POST vers `/data/persons/1/contacts` par exemple.

Essayez et vérifier que la table `Contact` dans la base de données est correctement alimentée.

Maintenant, si on veut voir les contact du coté de la classe `Person`, il faut ajouter la relation de ce coté. Pour cela, il faut déclarer un attribut du type `Set<Contacts>` et l'annoter avec `@OneToMany(mappedBy = "owner")` . Le `mappedBy` est important car il permet d'indiquer que la relation déclaré ici dans la classe personne est le symétrique de celle déclarée par l'attribut `Person owner` dans la classe `Contact`. Sans cela, deux relations indépendantes seraient créées (vous pouvez tester).

Ajouter un getter sur cet attribut et vérifiez que vous obtenez la liste des contacts lorsque vous récupérez les informations sur une personne.

Vous remarquerez que la méthode `public Person getOwner()` de `Contact` a été annotée avec `@JsonbTransient`. Ceci permet d'éviter le rendu Json de boucler à l'infini…

Voici un petit tuto qui explique le mapping onetomany de manière détaillée :
https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/

3- Le lazy loading

Combien de requêtes SQL sont effectuées lors de la lecture d'un contacts ?
Ceci illustre le fait que les relations @OneToMany (et @ManyToMany) sont par défaut chargées à la demande. En effet, le chargement de toutes les relations par défaut (avec l'annotation `@ManyToOne(fetch = FetchType.EAGER))` pourrait provoquer un chargement de beaucoup de données inutiles.
Ce mode de chargement est assez flexible mais pas toujours très efficace. Par exemple, combien de requêtes seront effectuées dans notre cas pour afficher une liste de n personnes ?

Les moyens pour éviter les « n+1 » select sont :
- d'utiliser des requêtes JPQL avec JOIN FETCH (https://www.baeldung.com/jpa-join-types)
- d'utiliser les EntityGraph (voir https://www.baeldung.com/jpa-entity-graph)

Pour chacun de ces moyens, proposez une solution pour éviter les n+1 selects lors du chargement d'une personne et de ses contacts.