# Manipulating and Queying Data

**AMT**

5 - Manipulating And Querying Data

**Résumé du document**

Definition

## Table des matières

# 1. Manipulating data

Pour manipuler de la donnée dans une application Java nous pouvons travailler avec ce que l'on appelle des entitées. Ces entitées représente (de manière générale) des tables stockées dans une base de données.

## 1.1. Entités

En se basant sur l'exemple suivant d'un modèle de base de données

```
Person(id, first_name, last_name)
Order(id, person_id, date)
OrderLine(id, order_id, item_id, quantity)
Item(id, name, price)
```

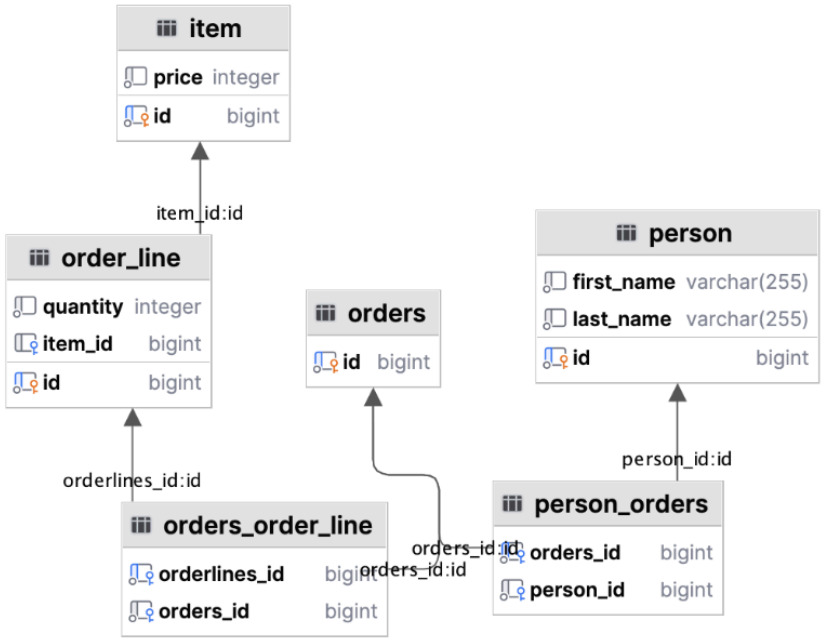
Nous pourrions définir ce schéma en créant des entités directement dans une classe Java. Pour cela, si nous souhaitons créer l'entitées pour la classe `Person` nous pourrions écrire:

```java
@Entity(name = "persons")
public class Person {

    @Id
    @Column(name = "id", nullable = false)
    private Long id;

    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Column(name = "last_name", nullable = false)
    private String lastName;

    @OneToMany
    private List<Order> orders;

    // ...
}
```

Grâce aux différentes entités Hibernate pourra générer le schéma de base de données automatiquement!

### 1.1.1. Génération d'un schéma

Voici le schéma construit grâce aux différentes entitées:



Cependant nous remarquons que des tables intermédiraires ont été crée au lieu de faire des relations directes lors ce que nous avons utilisé l'annotation `@OneToMany` ou `@ManyToOne`.

**C'est pourquoi il est primordial de dire précisement à l'ORM ce que l'on souhaite comme comportement!**

Pour corriger l'erreur de création d'une table `person_orders` nous devons préciser sur quelle colonne faire la jointure! Pour cela il faut ajouter l'annotation:

```
@OneToMany
@JoinColumn(name = "person_id", referencedColumnName = "id")
private List<Order> orders;
```

En regénérant le schéma nous aurons:


### 1.1.2. Comportement par défaut

Lorsque seule l'annotation `@OneToMany` est utilisée, sans plus de détails, JPA applique un comportement par défaut pour gérer cette relation. Si on ne spécifie pas d'informations supplémentaires, une table de jointure est automatiquement créée. C'est cette table de jointure qui permet de lier les deux entités sans que l'entité de destination (côté non-possédant) contienne une clé étrangère. Cette table est générée avec le nom `table1_table2`.

## 1.2. Persistance des données

Il existe plusieurs type de persistance au niveau des entités et celle-ci sont gérées grâce à l'`EntityManager` qui dans cet exemple est définit de la manière suivante:

```
EntityManager em = ...;
```

1. **Nouvelle instance**
   - Une entité créée n'a pas encore d'identité persistante ni de lien avec le contexte de persistance.

     ```
     var p = new Person("John", "Doe");
     ```

2. **Entité gérée (Managed)**
   - Une entité devient gérée lorsqu'on l'enregistre avec `persist` ou qu'on la récupère avec `find`. Elle est alors suivie par le contexte de persistance, et ses modifications seront synchronisées avec la base de données lors du commit.

     ```
     var p = new Person("John", "Doe");
     em.persist(p);  // Enregistre dans le contexte
     var p = em.find(Person.class, 1L);  // Trouve et gère l'entité
     ```

3. **Entité détachée (Detached)**
   - Une entité peut être détachée manuellement avec `detach`, ou automatiquement après un commit de transaction. Elle garde son identifiant mais n'est plus suivie.

   ```
   var p = em.find(Person.class, 1L);
   em.detach(p);  // Détache l'entité du contexte de persistance
   ```

4. **Entité supprimée (Removed)**
   - Une entité peut être marquée pour suppression avec `remove`. Elle sera retirée de la base de données lors du prochain commit de transaction.

   ```
   var p = em.find(Person.class, 1L);
   em.remove(p);  // Marque l'entité pour suppression lors du commit
   ```

5. **Synchronisation**
   - Les modifications des entités gérées sont écrites dans la base de données au moment du commit de la transaction, mais peuvent aussi être forcées avec `flush`.

   ```
   em.flush();  // Force la synchronisation immédiate avec la base de données
   ```

## 1.3. Opérations batch

Lors-ce que nous souhaitons insérer ou mettre à jour un grand nombre de ligne, il est plus efficace d'utiliser des opérations batchs. Le comportement par défaut dans une base de donnée est de commit chaque opérations individuellement.

Avec JDBC nous pouvons effectuer le genre de batch suivant:

```
PreparedStatement statement = connection.prepareStatement("INSERT INTO person (first_name,
last_name) VALUES (?, ?)");
statement.setString(1, "John");
statement.setString(2, "Doe");
statement.addBatch();
statement.setString(1, "Jane");
statement.setString(2, "Doe");
statement.addBatch();
statement.executeBatch();
```

Il est possible d'activer le comportement batch globallement ou par session en ajoutant les paramètres suivants:

```
hibernate.jdbc.batch_size // Zero or a negative number disables this feature.
quarkus.hibernate-orm.jdbc.statement-batch-size // For Quarkus
```

## 1.4. Méchanismes bas niveau

Lors-ce que nous souhaitons insérer des millions de lignes certaines base de données offre des méchanismes bas niveau pour optimiser le chargement. Par exemple PostgreSQL offre la commande `COPY`.

```
COPY person (first_name, last_name) FROM '/tmp/person.csv' DELIMITER ',' CSV HEADER;
```

Cette commande est beaucoup plus rapide qu'utiliser JDBC ou JPA.

# 2. Querying databses

# 3. Object relational impedance mismatch