Optimisation des lectures

adriencaubel.fr

Table des matières

Introduction

Rappels

- Nous avons étudié les concepts suivants
  - La notion `@Entity`
  - Les associations `@OneToMany` ... et `mappedBy` (bidirectionnelle)
  - L'héritage `@Inheritance`

Nous allons maintenant nous intéresser à **l'optimisation des lectures**

Pourquoi ?

Lorsqu'on travaille avec JPA, il est essentiel de comprendre comment sont chargées les relations entre entités pour optimiser les performances de l'application.

Exemple

- Un `Etudiant` a une liste de `Note`
- Lorsqu'on charge un `etudiant` doit-on également charger les `notes` ?

Stratégies Fetching

Les deux stratégies de fetching

Deux stratégies principales existent :

- **Lazy Fetching** (chargement paresseux)
- **Eager Fetching** (chargement immédiat).

Fetching par défaut

- Par défaut, `@OneToMany` et `@ManyToMany` adopte une approche **Lazy**
- Par défaut, `@OneToOne` et `@ManyToOne` adapte une approche **Eager**

```
class Etudiant {
    ...
    @OneToMany(mappedBy = "etudiant") // Lazy par défaut
    private List<Note> notes
}
```

Par défaut la liste de `notes` ne sera pas chargée en même temps de l'étudiant

Fetch LAZY

```java
@Entity
public class Etudiant {
    @OneToMany(mappedBy = "etudiant") private List<Livre> livresLus;
}
```

```java
Etudiant etudiant = entityManager.find(Etudiant.class, etudiantId); // 1
List<Livre> livres = etudiant.getLivresLus(); // 2
```

Comme LAZY deux requêtes SQL vont être nécessaires

1. `SELECT * FROM etudiant WHERE id = ?;`
2. puis `SELECT * FROM livre WHERE etudiant_id = ?;`

Fetch EAGER

```
    @OneToMany(mappedBy = "etudiant", fetch = FetchType.EAGER)
    private List<Livre> livresLus;
```

```
Etudiant etudiant = entityManager.find(Etudiant.class, etudiantId);
List<Livre> livres = etudiant.getLivresLus();
```

Une seule requête, tout est récupéré en même temps

```
SELECT e.*, l.*
FROM etudiant e
LEFT JOIN livre l ON e.id = l.etudiant_id
WHERE e.id = ?;
```

Toujours choisir EAGER ?

Et bien ça dépend.

- Dans le cas où l'on sait que la relation *livres* sera explorée systématiquement après la lecture d'un étudiant, il serait plus malin de n'émettre qu'un seul SELECT avec une jointure, de manière à peupler la relation *livres* à l'avance.
  - Cela ne ferait qu'un seul aller-retour avec la base de données.
- En revanche, dans le cas d'une relation qui, pour des raisons applicatives, ne serait pas explorée, ou rarement, alors l'exécution de la jointure lors du SELECT serait un surcoût inutile.

Le problème des N+1 Queries

Définition et exemple

Définition
The N+1 query problem happens when the data access framework executed N additional SQL statements to fetch the same data that could have been retrieved when executing the primary SQL query (JOIN)

Exemple

```
@Entity
class User {
    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;
}
```

```
User users = userRepository.findById(1); // 1 requêtes

for (Order order : users.getOrders()) {
    System.out.println("Total : " + order.getTotal()); // N requêtes
}
```

Pourquoi N requêtes ?

```
User users = userRepository.findById(1); // 1 requêtes

for (Order order : users.getOrders()) {
    System.out.println("Total : " + order.getTotal()); // N requêtes
}
```

Etape 1 : charger tous les utilisateurs

```
SELECT * FROM user;
```

Etape 2 : Pour l'utilisateur charger ses commandes (LAZY)

```
SELECT * FROM orders WHERE user_id = 1;
```

Solutions

Il existe plusieurs solutions

- Mettre en `EAGER`
- Faire un Fetch Join (à préférer)
- Faire un EntityGraph

Mettre en `EAGER`

```
@Entity
class User {
    @OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
    private List<Order> orders;
}
```

Mais une mauvaise idée

- Avec EAGER, Hibernate charge toujours `orders`, même si on en a pas besoin

```
User user = entityManager.find(User.class, id);
// Même si on n'appelle jamais user.getOrders(), les orders sont chargées
```

=> On peut donc charger énormément de données : 1 utilisateurs et 1000 commandes

=> Trouver des solutions pour avoir le choix

- soit charger uniquement l'utilisateur
- soit charger à la fois l'utilisateur et ses commandes

Faire un Fetch Join (à préférer)

Rajouter une méthode complémentaire `findByIdWithOrders(Integer id)` qui retournera un utilisateur avec ses commandes.

```
public User findByIdWithOrders(Integer id) {

    return entityManager.createQuery(
        "SELECT u FROM User u " +
        "JOIN FETCH u.orders " +
        "WHERE u.id = :id",
        User.class
    )
    .setParameter("id", id)
    .getSingleResult();
}
```

SQL généré

```
SELECT u.*, o.*
FROM user u
JOIN orders o ON o.user_id = u.id
WHERE u.id = ?;
```

Faire un EntityGraph

```java
@Entity
@NamedEntityGraph(
    name = "User.orders",
    attributeNodes = @NamedAttributeNode("orders")
)
public class User {

    @Id
    private Integer id;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;
}
```

Utilisation

```java
EntityGraph<?> graph =
    entityManager.getEntityGraph("User.orders");

Map<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.fetchgraph", graph);

User user = entityManager.find(User.class, id, hints);
```

SQL généré

```sql
SELECT u.*, o.*
FROM user u
LEFT JOIN orders o ON o.user_id = u.id
WHERE u.id = ?;
```

LazyInitializationException

Définition et exemple

> **Définition**
> La LazyInitializationException se produit quand Hibernate essaye de charger une relation `LAZY` en dehors d'une session active

Exemple 1 : Session fermée trop tôt

```
@OneToMany(fetch = FetchType.LAZY)
private List<Order> orders;
```

Hibernate ne peut charger une relation lazy que si :

- la session est encore ouverte
- la transaction est active

```
User user = entityManager.find(User.class, 1);

entityManager.close(); // session fermée

user.getOrders().size(); // ❌ crash car Hibernate voulait exécuter SELECT * FROM orders WHERE user_id = 1;
```

Exemple 2 — Retour d’une entité hors transaction

```java
public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public User getUser(Integer id) {
        return entityManager.find(User.class, id);
        // Orders ne sont pas chargés ici (LAZY)
    }
}
```

```java
public class OrderService {

    private UserService userService;

    public void afficherCommandes(Integer userId) {
        User user = userService.getUser(userId);

        // ❌ La session est déjà fermée ici
        for (Order order : user.getOrders()) {
            System.out.println(order.getTotal());
        }
    }
}
```

Exemple 3 — API REST (sérialisation JSON)

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Integer id) {
    return userService.getUser(id); //
}
```

Quand Jackson transforme l'objet en JSON :

```
{
  "id": 1,
  "orders": [...]
}
```

Il appelle automatiquement `user.getOrders()`

Mais la session est déjà fermée → exception.

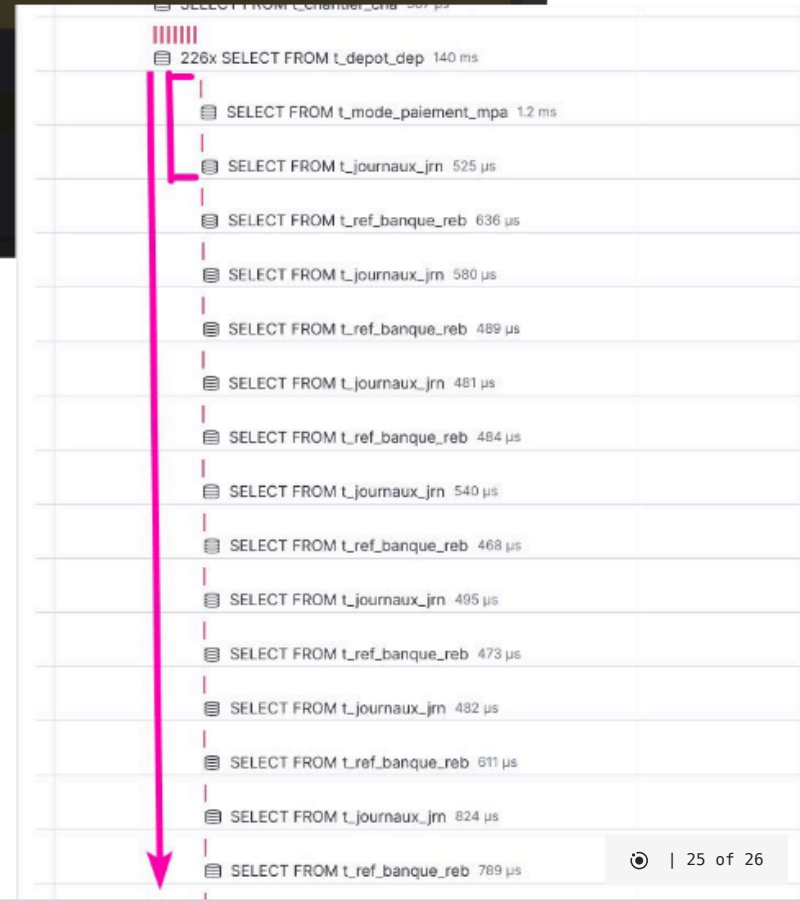
Comment corriger la LazyInitializationException

On retombe sur la section précédente :

- Mettre en `EAGER`
- Faire un Fetch Join (à préférer)
- Faire un EntityGraph

Exemple concrets

```java
@Override
public List<ModePaiement> trouverModesPaiementActifs() {
    List<ModePaiement> result = sessionFactory.getCurrentSession().createCriteria( aClass: ModePaiement.class)
            .add(Restrictions.eq( propertyName: "horsListe",  value: false))
            .add(Restrictions.eq( propertyName: "modePaiementPerteEtProfit",  value: Boolean.FALSE))
            .addOrder( order: Order.asc( propertyName: "libelle"))
            .list();
    for (ModePaiement mpa : result) {
        Hibernate.initialize( proxy: mpa.getJournal());
    }
    return result;
}
```



Solution

```java
@Override
public List<ModePaiement> trouverModesPaiementActifs() {
    List<ModePaiement> result = sessionFactory.getCurrentSession() Session
            .createCriteria( aClass: ModePaiement.class) Criteria
            .setFetchMode( s: "journal",  fetchMode: FetchMode.JOIN)  // Eagerly fetch the journal assoc
            .add(Restrictions.eq( propertyName: "horsListe",  value: false))
            .add(Restrictions.eq( propertyName: "modePaiementPerteEtProfit",  value: Boolean.FALSE))
            .addOrder( order: Order.asc( propertyName: "libelle"))
            .list();
    return result;
}
```