2 - JPA Avancé et Optimisation

Séance 2 (4h)

Contexte: fil rouge e-commerce

Objectifs pédagogiques

- Maîtriser les associations bidirectionnelles et leurs pièges
- Comprendre et résoudre les problèmes N+1
- Utiliser l'héritage JPA à bon escient
- Optimiser les requêtes avec fetch strategies et projections

Partie 1 : Associations JPA (1h30)

1.1 Implémentation des associations manquantes

A

À réaliser :

- Compléter Order ↔ OrderItem (bidirectionnel)
- Implémenter Order → User (unidirectionnel)
- Gérer User ↔ Category (preferences, Many-to-Many)
- Ajouter @JsonIgnore / @JsonManagedReference pour éviter les boucles

Points d'attention:

- Choix du côté propriétaire (mappedBy)
- Cascade types appropriés
- Orphan removal
- · Lazy vs Eager loading

1.2 Exercice pratique : Orders & OrderItems

// Contraintes métier à implémenter

- Un Order doit toujours avoir au moins 1 OrderItem
- Suppression d'un Order → suppression des OrderItems
- totalAmount calculé automatiquement
- Gestion du stock produit lors de la création

Tests attendus:

- Création d'une commande avec items
- Calcul automatique du total
- Mise à jour du stock
- Suppression en cascade

Last update: 2025/10/08 00:12

Partie 2 : Problèmes de performance (1h30)

2.1 Diagnostic du problème N+1



Scénario:

GET /users/{id}/orders
// Retourne les commandes avec leurs items et produits

Mission:

- Activer les logs SQL (spring.jpa.show-sql=true)
- 2. Identifier le problème N+1
- 3. Compter le nombre de requêtes générées

2.2 Solutions d'optimisation

À implémenter et comparer :

Solution	Cas d'usage	Avantages	Inconvénients
@EntityGraph	Requêtes standards	Simple	Moins flexible
JOIN FETCH	Requêtes complexes	Contrôle total	Code JPQL
@BatchSize	Lazy loading	Transparent	Moins optimal
DTO Projection	Lecture seule	Performances max	Plus de code

Exercices:

- 1. Optimiser /users/{id}/orders avec JOIN FETCH
- 2. Créer une projection pour / products (liste)
- 3. Comparer les performances avant/après

Partie 3 : Héritage JPA (1h)

3.1 Cas d'usage : Typologie de produits

Nouveau besoin métier :

Différencier 3 types de produits :

PhysicalProduct : poids, dimensions, frais de port
 DigitalProduct : taille fichier, URL download, format

• ServiceProduct : durée, date prestation

Tous partagent: id, name, price, stock, category

3.2 Implémentation avec stratégies d'héritage

À explorer (au choix ou comparaison) :

```
// Option 1 : SINGLE_TABLE (par défaut)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "product_type")

// Option 2 : JOINED
@Inheritance(strategy = InheritanceType.JOINED)

// Option 3 : TABLE_PER_CLASS
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
```

Exercice comparatif:

- Schéma base de données généré
- Requêtes SQL produites
- Avantages/inconvénients de chaque stratégie

3.3 Requêtes polymorphiques

```
// Repository
List<Product> findAll(); // Tous types confondus
List<PhysicalProduct> findPhysicalProducts();

// Nouveaux endpoints
GET /products?type=PHYSICAL
GET /products?type=DIGITAL
```

Partie 4: Hypersistence Utils - Outils avancés (30min-1h)

4.1 Introduction à Hypersistence Utils

Hypersistence Utils est une bibliothèque créée par Vlad Mihalcea qui apporte :



- Des types personnalisés (JSON, Array, etc.)
- Des utilitaires de diagnostic de performance
- Des listeners pour optimiser les opérations
- Des identifiants optimisés (Tsid)

Dépendance Maven

```
<dependency>
     <groupId>io.hypersistence</groupId>
          <artifactId>hypersistence-utils-hibernate-63</artifactId>
          <version>3.7.0</version>
</dependency>
```

4.2 Détection automatique des problèmes N+1



Objectif : Détecter automatiquement les problèmes de performance sans analyse manuelle des logs

Configuration

```
# application.properties - Ajout pour Hypersistence

# Détection des problèmes N+1
logging.level.io.hypersistence.utils=DEBUG

# Limites d'alerte (optionnel)
hypersistence.query.fail.on.pagination.over.collection.fetch=false
```

Utilisation du QueryStackTraceLogger

```
// Configuration globale (classe @Configuration)
@Configuration
public class HypersistenceConfiguration {
    @Bean
    public QueryStackTraceLogger queryStackTraceLogger() {
        return new QueryStackTraceLogger();
    }
    @EventListener
    public void onApplicationEvent(ApplicationReadyEvent event) {
        // Active la détection des problèmes N+1
        QueryStackTraceLogger.INSTANCE.setThreshold(10); // Alerte si > 10 requêtes
    }
}
```

Exercice:

- Activer le logger sur l'endpoint /users/{id}/orders
- Observer les alertes automatiques
- Corriger les problèmes détectés

4.3 Types JSON natifs



Cas d'usage : Stocker des métadonnées flexibles sur les produits

Exemple: Attributs dynamiques produit

```
@Entity
@Table(name = "products")
public class Product {
    // ... attributs existants
    @Type(JsonType.class)
    @Column(columnDefinition = "json")
    private Map<String, Object> attributes;
    // Pour PhysicalProduct : {"weight": 2.5, "dimensions": "30x20x10"}
    // Pour DigitalProduct : {"fileSize": "1.2GB", "format": "PDF"}
}
```

Données exemple

```
{
  "id": "550e8400-e29b-41d4-a716-446655440020",
  "name": "iPhone 15 Pro",
  "price": 1199.99,
  "stock": 25,
  "categoryId": "550e8400-e29b-41d4-a716-446655440010",
  "attributes": {
     "color": "Titanium Blue",
     "storage": "256GB",
     "warranty": "2 years"
  }
}
```

Exercice:

- Ajouter le champ attributes à Product
- Créer un endpoint GET /products/{id}/attributes
- Filtrer les produits par attribut : GET /products?attr.color=Blue

4.4 Optimisation des identifiants avec Tsid



Tsid (Time-Sorted Identifiers):

- Alternative performante aux UUID
- Triables chronologiquement

Last update: 2025/10/08 00:12



- Plus compacts (Long au lieu de UUID)
- Meilleure performance en base

Comparaison UUID vs Tsid

```
// Avant (UUID)
@Id
@GeneratedValue(strategy = GenerationType.UUID)
private UUID id;

// Après (Tsid) - Pour nouvelles entités
@Id
@TsidGenerator
private Long id;
```

Exercice optionnel:

- Créer une nouvelle entité Review avec Tsid
- Comparer les performances d'insertion (benchmark)

```
@startuml Review Entity
!define ENTITY class
!define PK
!define FK
ENTITY Review {
PK id: Long @TsidGenerator
FK productId: UUID
FK userId: UUID
rating: Integer (1-5)
title: String
comment: String
verified: Boolean
helpfulCount : Integer
createdAt: LocalDateTime
updatedAt : LocalDateTime
}
ENTITY Product {
PK id: UUID
name: String
}
ENTITY User {
```

PK id: UUID

```
username: String
...
}

Review "0..*" --> "1" Product: reviews
Review "0..*" --> "1" User: author

note right of Review

**Contraintes métier:**

• rating entre 1 et 5

• verified = true si achat confirmé

• helpfulCount initialisé à 0

• Utilisateur = 1 review max par produit end note

note top of Review::id

**Tsid** pour performance
et tri chronologique
end note
```

@enduml

4.5 Monitoring des requêtes en temps réel

${\bf Data Source Proxy Bean Post Processor}$

```
@Configuration
public class DataSourceProxyConfiguration {
    @Bean
    public DataSourceProxyBeanPostProcessor dataSourceProxyBeanPostProcessor() {
        return new DataSourceProxyBeanPostProcessor() {
            @Override
            protected DataSourceProxy createDataSourceProxy(DataSource dataSource)
{
            return new DataSourceProxy(dataSource, new QueryCountHolder());
            }
        };
    }
}
```

Exercice:

- Mettre en place le monitoring
- Créer un test d'intégration qui vérifie le nombre exact de requêtes
- Exemple : assertQueryCount(3) après un appel API

4.6 Exercice intégratif



Mission: Améliorer l'endpoint recommendations



GET /users/{id}/recommendations

Avec Hypersistence:

- Détecter automatiquement les problèmes N+1
- Limiter à 5 requêtes maximum (assertion en test)
- Stocker les préférences utilisateur en JSON
- Logger les performances de la recommandation

Structure JSON recommandée:

```
// User.preferences (JSON)
{
   "priceRange": {"min": 50, "max": 500},
   "brands": ["Apple", "Samsung"],
   "excludeCategories": ["550e8400-..."]
}
```

Configuration complète

```
# application.properties - Configuration complète Séance 2
# H2 Database
spring.datasource.url=jdbc:h2:file:./data/ecommerce
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
# H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format sql=true
spring.jpa.properties.hibernate.use sql comments=true
spring.jpa.properties.hibernate.generate statistics=true
# Logging SQL et statistiques
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.hibernate.stat=DEBUG
logging.level.org.hibernate.orm.jdbc.bind=TRACE
# Hypersistence Utils
logging.level.io.hypersistence.utils=DEBUG
```

Livrables attendus

Priorités (4h)

Must have:

- 🛘 Associations Order/OrderItem/User complètes avec tests
- 🛘 Résolution problème N+1 sur au moins 2 endpoints
- | Implémentation héritage produits (1 stratégie au choix)
- ☐ Hypersistence : détection automatique N+1 activée
- 🛘 Tests d'intégration validant les performances

Nice to have:

- Comparaison des 3 stratégies d'héritage
- Type JSON pour attributs dynamiques produits
- Tsid sur une nouvelle entité (Review, Wishlist...)
- Benchmark avant/après optimisations avec query count assertions
- Documentation des choix architecturaux

Ressources

- Hibernate Performance Best Practices
- Hypersistence Utils GitHub
- Documentation officielle Hypersistence
- Spring Data JPA Query Methods

Conseils:



- Commencer par les associations avant l'optimisation
- Toujours mesurer avant d'optimiser (logs SQL)
- L'héritage n'est pas toujours la meilleure solution (composition > héritage)
- Privilégier @ManyToOne LAZY par défaut

From:

http://slamwiki2.kobject.net/ - SlamWiki 2.1

Permanent link:

http://slamwiki2.kobject.net/eadl/bloc3/dev_av/td2?rev=1759875126

Last update: 2025/10/08 00:12

