

Séance 2 - JPA Avancé et Optimisation (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

Contenu de la séance

Partie 1 : Associations JPA (1h30)

1.1 Implémentation des associations manquantes



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

``java *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 — **### Partie 2 : Problèmes de performance (1h30) ####**
2.1 Diagnostic du problème N+1

Scénario : ``java GET /users/{id}/orders *Retourne les commandes avec leurs items et produits* `` **Mission :** 1. Activer les logs SQL (``spring.jpa.show-sql=true``) 2. Identifier le problème N+1 3. Compter le nombre de requêtes générées </WRAP> **#### 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) :** ``java 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

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

Nouveau endpoint GET /products?type=PHYSICAL GET /products?type=DIGITAL ```` — ## ☐ Livrables attendus

#### ### Priorités (4h)



**Must have** : 1. ☐ Associations Order/OrderItem/User complètes avec tests 2. ☐ Résolution problème N+1 sur au moins 2 endpoints 3. ☐ Implémentation héritage produits (1 stratégie au choix) 4. ☐ Tests d'intégration validant les performances

**Nice to have** : - Comparaison des 3 stratégies d'héritage - DTO Projections avec MapStruct - Benchmark avant/après optimisations - Documentation des choix architecturaux



— ## Critères d'évaluation | Critère | Points | |—|—| | Associations correctement mappées | 25% | | Résolution problèmes N+1 | 30% | | Implémentation héritage | 25% | | Tests et qualité code | 20% | — ## ☐ Configuration supplémentaire ````yaml # application.yml - pour la séance spring: jpa: show-sql: true properties: hibernate: format\_sql: true use\_sql\_comments: true generate\_statistics: true # Pour mesurer les perfs logging: level: org.hibernate.stat: DEBUG # Statistiques Hibernate ```` — ## Ressources - [Hibernate Performance Best Practices](https://vladmihalcea.com/tutorials/hibernate/) - [Spring Data JPA Query Methods](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#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=1759872801](http://slamwiki2.kobject.net/eadl/bloc3/dev_av/td2?rev=1759872801)

Last update: 2025/10/07 23:33

