Tu as absolument raison, pardon pour l'erreur ! Voici la correction en syntaxe DokuWiki :

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

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

@Entity
@Table(name = "orders")
class Order(
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
```

```
    val user: User,

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    var status: OrderStatus = OrderStatus.PENDING,

    @Column(nullable = false)
    var totalAmount: BigDecimal = BigDecimal.ZERO,

    @Column(nullable = false)
    val createdAt: Instant = Instant.now()
) {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null

    @OneToMany(
        mappedBy = "order",
        cascade = [CascadeType.ALL],
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    @JsonManagedReference
    private val _items: MutableList<OrderItem> = mutableListOf()

    val items: List<OrderItem>
        get() = _items.toList()

    fun addItem(item: OrderItem) {
        require(_items.isEmpty() || _items.size < 100) {
            "Cannot add more than 100 items to an order"
        }
        _items.add(item)
        item.order = this
        recalculateTotal()
    }

    fun removeItem(item: OrderItem) {
        _items.remove(item)
        item.order = null
        recalculateTotal()
    }

    private fun recalculateTotal() {
        totalAmount = _items.sumOf { it.unitPrice * it.quantity.toBigDecimal() }
    }

    init {
        require(user.id != null) { "User must be persisted before creating an
order" }
    }
}

@Entity
@Table(name = "order_items")
class OrderItem(
```

```
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "product_id", nullable = false)
    val product: Product,

    @Column(nullable = false)
    val quantity: Int,

    @Column(nullable = false, precision = 10, scale = 2)
    val unitPrice: BigDecimal
) {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id", nullable = false)
    @JsonBackReference
    var order: Order? = null

    init {
        require(quantity > 0) { "Quantity must be positive" }
        require(unitPrice > BigDecimal.ZERO) { "Unit price must be positive" }
    }
}

enum class OrderStatus {
    PENDING,
    CONFIRMED,
    SHIPPED,
    DELIVERED,
    CANCELLED
}
```

Tests attendus :

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

```
@SpringBootTest
@Transactional
class OrderServiceTest {

    @Autowired
    private lateinit var orderService: OrderService

    @Autowired
    private lateinit var userRepository: UserRepository

    @Autowired
    private lateinit var productRepository: ProductRepository

    @Test
```

```
    fun `should create order with items and calculate total`() {
        // Given
        val user = userRepository.save(User("John Doe", "john@example.com"))
        val product1 = productRepository.save(
            Product("iPhone", BigDecimal("999.99"), 10, category)
        )
        val product2 = productRepository.save(
            Product("MacBook", BigDecimal("1999.99"), 5, category)
        )

        val dto = CreateOrderDto(
            userId = user.id!!,
            items = listOf(
                OrderItemDto(product1.id!!, 2),
                OrderItemDto(product2.id!!, 1)
            )
        )

        // When
        val order = orderService.createOrder(dto)

        // Then
        assertThat(order.items).hasSize(2)
        assertThat(order.totalAmount).isEqualByComparingTo("3999.97") // 2*999.99 +
1999.99
        assertThat(product1.stock).isEqualTo(8) // 10 - 2
        assertThat(product2.stock).isEqualTo(4) // 5 - 1
    }

    @Test
    fun `should fail when insufficient stock`() {
        // Given
        val user = userRepository.save(User("Jane", "jane@example.com"))
        val product = productRepository.save(
            Product("Limited Item", BigDecimal("50.00"), 2, category)
        )

        val dto = CreateOrderDto(
            userId = user.id!!,
            items = listOf(OrderItemDto(product.id!!, 5))
        )

        // When & Then
        assertThatThrownBy { orderService.createOrder(dto) }
            .isInstanceOf(InsufficientStockException::class.java)
    }
}
```

Chargement minimaliste

Pour recréer à moindre coût une relation (sans charger complètement l'instance depuis le repository)

```
val user = entityManager.getReference(User::class.java, userId)
```

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 :

1. 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

```
// ❌ PROBLÈME N+1 : Sans optimisation
interface OrderRepository : JpaRepository<Order, UUID> {
    fun findByUserId(userId: UUID): List<Order>
    // 1 requête pour les orders
    // N requêtes pour charger les items de chaque order
    // M requêtes pour charger les produits de chaque item
}

// ✅ SOLUTION 1 : JOIN FETCH
interface OrderRepository : JpaRepository<Order, UUID> {
    @Query("""
        SELECT DISTINCT o FROM Order o
        JOIN FETCH o.items i
        JOIN FETCH i.product
```

```
        WHERE o.user.id = :userId
    """)
    fun findByUserIdWithItems(userId: UUID): List<Order>
}

// ✅ SOLUTION 2 : @EntityGraph
interface OrderRepository : JpaRepository<Order, UUID> {
    @EntityGraph(attributePaths = ["items", "items.product"])
    fun findByUserId(userId: UUID): List<Order>
}

// ✅ SOLUTION 3 : DTO Projection
data class OrderSummaryDto(
    val id: UUID,
    val totalAmount: BigDecimal,
    val status: OrderStatus,
    val itemCount: Long,
    val createdAt: Instant
)

interface OrderRepository : JpaRepository<Order, UUID> {
    @Query("""
        SELECT new com.ecommerce.order.dto.OrderSummaryDto(
            o.id, o.totalAmount, o.status, COUNT(i), o.createdAt
        )
        FROM Order o
        LEFT JOIN o.items i
        WHERE o.user.id = :userId
        GROUP BY o.id, o.totalAmount, o.status, o.createdAt
    """)
    fun findOrderSummariesByUserId(userId: UUID): List<OrderSummaryDto>
}
```

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)
@Entity
@Table(name = "products")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "product_type", discriminatorType =
DiscriminatorType.STRING)
abstract class Product(
    @Column(nullable = false)
    open var name: String,

    @Column(nullable = false, precision = 10, scale = 2)
    open var price: BigDecimal,

    @Column(nullable = false)
    open var stock: Int,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    open var category: Category
) {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    open var id: UUID? = null
}

@Entity
@DiscriminatorValue("PHYSICAL")
class PhysicalProduct(
    name: String,
    price: BigDecimal,
    stock: Int,
    category: Category,

    @Column(name = "weight_kg")
    var weight: Double,

    @Column(name = "dimensions")
    var dimensions: String, // "30x20x10"

    @Column(name = "shipping_cost", precision = 10, scale = 2)
    var shippingCost: BigDecimal
) : Product(name, price, stock, category)

@Entity
@DiscriminatorValue("DIGITAL")
class DigitalProduct(
    name: String,
    price: BigDecimal,
    stock: Int,
    category: Category,

    @Column(name = "file_size_mb")
    var fileSize: Double,

    @Column(name = "download_url")
    var downloadUrl: String,
```

```
    @Column(name = "file_format")
    var format: String // PDF, MP4, ZIP...
) : Product(name, price, stock, category)

@Entity
@DiscriminatorValue("SERVICE")
class ServiceProduct(
    name: String,
    price: BigDecimal,
    stock: Int,
    category: Category,

    @Column(name = "duration_hours")
    var duration: Int,

    @Column(name = "service_date")
    var serviceDate: LocalDate?
) : Product(name, price, stock, category)

// Option 2 : JOINED (tables séparées)
@Entity
@Table(name = "products")
@Inheritance(strategy = InheritanceType.JOINED)
abstract class Product(
    // ... mêmes champs
)

@Entity
@Table(name = "physical_products")
class PhysicalProduct(
    // ... champs spécifiques
) : Product(...)

// Option 3 : TABLE_PER_CLASS (une table complète par classe concrète)
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
abstract class Product(
    // ... mêmes champs
)
```

Exercice comparatif :

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

3.3 Requêtes polymorphiques

```
// Repository
interface ProductRepository : JpaRepository<Product, UUID> {
    // Tous types confondus
    override fun findAll(): List<Product>
    // Seulement les produits physiques
```

```
    @Query("SELECT p FROM PhysicalProduct p")
    fun findPhysicalProducts(): List<PhysicalProduct>
    // Filtrage par type
    @Query("SELECT p FROM Product p WHERE TYPE(p) = :type")
    fun findByType(type: Class<out Product>): List<Product>
}

// Controller
@RestController
@RequestMapping("/products")
class ProductController(private val repository: ProductRepository) {

    @GetMapping
    fun getProducts(@RequestParam(required = false) type: String?): List<Product> {
        return when (type?.uppercase()) {
            "PHYSICAL" -> repository.findByType(PhysicalProduct::class.java)
            "DIGITAL" -> repository.findByType(DigitalProduct::class.java)
            "SERVICE" -> repository.findByType(ServiceProduct::class.java)
            else -> repository.findAll()
        }
    }
}
```

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

```
@Configuration
class HypersistenceConfiguration {
    @Bean
    fun queryStackTraceLogger() = QueryStackTraceLogger()
    @EventListener
    fun onApplicationReady(event: ApplicationReadyEvent) {
        // Active la détection des problèmes N+1
        QueryStackTraceLogger.INSTANCE.threshold = 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")
class Product(
    @Column(nullable = false)
    var name: String,

    @Column(nullable = false, precision = 10, scale = 2)
    var price: BigDecimal,

    @Column(nullable = false)
    var stock: Int,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    var category: Category,

    // □ Stockage JSON pour attributs dynamiques
    @Type(JsonType::class)
    @Column(columnDefinition = "json")
    var attributes: Map<String, Any> = emptyMap()
) {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null
}

// Utilisation
val product = Product(
    name = "iPhone 15 Pro",
    price = BigDecimal("1199.99"),
    stock = 25,
    category = electronicsCategory,
    attributes = mapOf(
        "color" to "Titanium Blue",
        "storage" to "256GB",
        "warranty" to "2 years",
        "features" to listOf("5G", "Face ID", "A17 Pro")
    )
)
```

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",
    "features": ["5G", "Face ID", "A17 Pro"]
  }
}
```

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

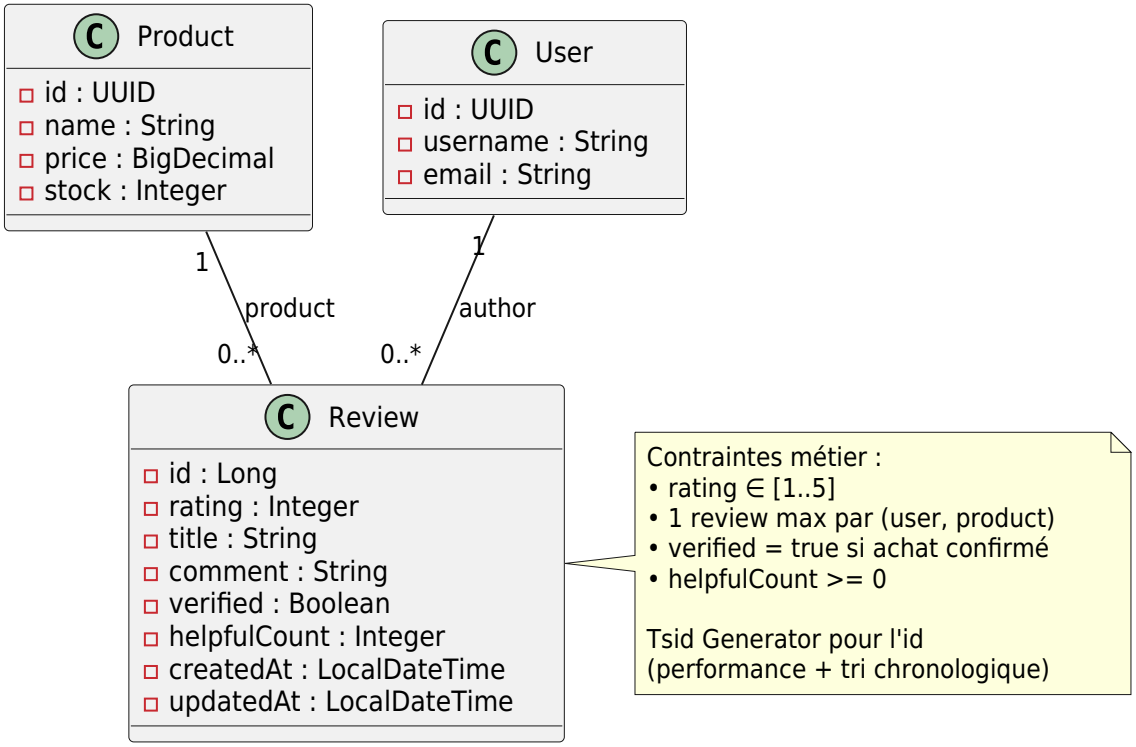
Comparaison UUID vs Tsid

```
// Avant (UUID)
@Entity
class Review(
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null,
    // ...
)

// Après (Tsid) - Pour nouvelles entités
@Entity
@Table(name = "reviews")
class Review(
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "product_id", nullable = false)
    val product: Product,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    val author: User,

    @Column(nullable = false)
    val rating: Int, // 1-5

    @Column(nullable = false, length = 200)
    val title: String,

    @Column(nullable = false, length = 2000)
    val comment: String,

    @Column(nullable = false)
    var verified: Boolean = false,

    @Column(nullable = false)
    var helpfulCount: Int = 0,

    @Column(nullable = false)
    val createdAt: Instant = Instant.now(),

    @Column(nullable = false)
    var updatedAt: Instant = Instant.now()
) {
    @Id
    @TsidGenerator
    var id: Long? = null

    init {
        require(rating in 1..5) { "Rating must be between 1 and 5" }
        require(title.isNotBlank()) { "Title cannot be blank" }
        require(comment.isNotBlank()) { "Comment cannot be blank" }
        require(helpfulCount >= 0) { "Helpful count cannot be negative" }
    }
```

```
}
```

Exercice optionnel :

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



4.5 Monitoring des requêtes en temps réel

DataSourceProxyBeanPostProcessor

```
@Configuration
class DataSourceProxyConfiguration {
    @Bean
    fun dataSourceProxyBeanPostProcessor() = object :
DataSourceProxyBeanPostProcessor() {
        override fun createDataSourceProxy(dataSource: DataSource): DataSourceProxy
{
            return DataSourceProxy(dataSource, 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

```
@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
@Transactional
class OrderPerformanceTest {

    @Autowired
    private lateinit var mockMvc: MockMvc

    @Test
    fun `should not trigger N+1 queries when fetching user orders`() {
        // Given
        val userId = createUserWithOrders()

        // When
        SQLStatementCountValidator.reset()
        mockMvc.perform(get("/users/$userId/orders"))
            .andExpect(status().isOk)

        // Then - Vérifier le nombre de requêtes SQL
        assertSelectCount(2) // 1 pour User + 1 pour Orders avec items (JOIN FETCH)
    }
}
```

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 :

```
@Entity
@Table(name = "users")
class User(
    @Column(nullable = false)
    var name: String,

    @Column(nullable = false, unique = true)
    var email: String,

    // □ Préférences stockées en JSON
```

```
    @Type(JsonType::class)
    @Column(columnDefinition = "json")
    var preferences: UserPreferences = UserPreferences()
) {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null
}

data class UserPreferences(
    val priceRange: PriceRange = PriceRange(),
    val brands: List<String> = emptyList(),
    val excludeCategories: List<UUID> = emptyList()
)

data class PriceRange(
    val min: BigDecimal = BigDecimal.ZERO,
    val max: BigDecimal = BigDecimal("10000")
)
```

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
- Kotlin JPA Plugin

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
- Utiliser des data classes pour les DTOs
- Attention aux classes ouvertes (`open`) nécessaires pour JPA en Kotlin