FORMATION JPA-HIBERNATE

Woodson JUSTE
Formateur-développeur Java/.NET
Download Eclipse : https://www.eclipse.org/downloads/
logback: https://logback.qos.ch/manual/architecture.html

cours SQL : https://sql.sh/cours/where

Modèle persistence.xml : https://gist.github.com/hurynovich/dc283e6d1aa91dff442b33353084d14f

Download StarUML : https://staruml.io/

https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

1- Florian Dray
- BAC +2 (BTS SIO) / BAC +2 Développeur web (CESI)
- PHP
- NodeJS
- Nantes

2- Pierre-Yves Castelleta :
- 28 ans
- Bac +2 Titre pro Développement web et web mobile
- PHP, JS

3- Riff Baptiste
- 23 ans
- bac +3 développement JV (c++ / web)
- poei cobol
- Strasbourg

4- Nathan Frappier:
- 23 ans
- Bordeau
- 3 ans a Epitech
- Bac +2 (Concepteur et développeur d'application) a la 3wa
- C/C++/Python/NodeJs

5- Romain Chuche
- 21 ans
- Nantes (< Angers)
- Bac +2 : DUT Informatique
- 1 ans d'hôtellerie

- Java / Python / C / JavaScript

6- Louërat Baptiste
- 26 ans
- Bac +2 BTS SIO SLAM
- 2 Licence 3 MIAGE non validées
- 5 ans de restauration, reconversion professionnelle
- Java / Vba / PHP-Symfony
- Objectifs : CDI ? Bac +5 ? Montée en compétence

7- Léandre Blanchard :
-20 ans
-Bac+2 (BTS SIO)
-Compétences :
-> C#,Javasript, Python, PHP

8- MARTELLI Arnaud ;
22 ans
-Bac +2 (BTS SIO validé en 2021 et BTS MV validé en 2023)
-PHP, Python

9- Noé VILLENEUVE :
- 21 ans
- +/- bac+2 (DUT Informatique)
- Toulouse

10 - METRIAU Titouan :
- 23 ans
- Bac + 2 (CDA)
- Compétences :
  - PHP Symfony/Laravel/Yii 2 / Javascript / Java

11- Chayanne PIRRAKU
- 22 ans
- BAC+2 SNIR
- Des notions technos un peu partout mais pas spécialisé pour le moment
- Pour les recruteurs manque de diplômes et expériences
- Centre de Nantes

12- Yannick Pitaud
- 22 ans
- BAC+2 (TP Développer Web)
- Php Symfony/ Javascript

13- Alexandre Ivanov
- 23 ans

- BTS SIO
- Typescript React Node VB NET
Ruby on Rails CSS
- 3 expériences en alternance
- gros burn-out à la 3ème
- je veux me spécialiser en Spring Boot
- et coupler ça avec Angular et / ou React

14- MEYER Max
- 21 ans
- bachelor science du numériques

Création d'un projet Maven

Maven est un outil de construction de projets (build) open-source développé par la fondation Apache.

Il permet notamment :

- D'automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet
- De gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet
- De générer des documentations concernant le projet

Un projet Maven est un projet qui possède un fichier qui s'appelle **pom.xml.** POM (Project Object Model).

Conventions JavaBeans

Un composant JavaBean est une simple classe Java qui respecte certaines conventions sur le nommage, la construction et le comportement des méthodes. Le respect de ces conventions rend possible l'utilisation, la réutilisation, le remplacement et la connexion de JavaBeans par des outils de développement.
Les conventions à respecter sont les suivantes :

- la classe doit être « Serializable » pour pouvoir sauvegarder et restaurer l'état d'instances de cette classe ;
- la classe doit posséder un constructeur sans paramètre (constructeur par défaut) ;
- les attributs privés de la classe (variables d'instances) doivent être accessibles publiquement via des méthodes accesseurs construit avec *get* ou *set* suivi du nom de l'attribut avec la première lettre capitalisée. Le couple d'accesseurs est appelé *Propriété* ;
- la classe ne doit pas être déclarée final.

LOG : Journalisation

Dépendances à ajouter dans le projet :

- <!-- Journalisation-->

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.4.14</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.9</version>
</dependency>

<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.14</version>
</dependency>
```

Configuration logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>

    <!-- Configuration de l'appender pour la console -->
    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>

    <!-- Configuration de l'appender pour le fichier -->
<!--    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>app.log</file> Spécifiez le chemin et le nom du fichier
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
```

```
            </encoder>
        </appender> -->

        <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>app2.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
      <!-- rollover daily -->
      <fileNamePattern>app2-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
       <!-- each file should be at most 10MB, keep 5 days worth of history, but at most 2GB -->
       <maxFileSize>10MB</maxFileSize>
       <maxHistory>5</maxHistory>
       <totalSizeCap>2GB</totalSizeCap>
    </rollingPolicy>
    <encoder>
      <pattern>%date{yyyy-MM-dd HH:mm:ss.SSS} %-5level %-5relative --- [%thread] %logger{35} :
%msg%n</pattern>
    </encoder>
  </appender>


        <!-- Configuration du logger racine -->
        <root level="INFO">
            <appender-ref ref="STDOUT" /> <!-- Utilisez l'appender console pour le logger racine -->
            <appender-ref ref="FILE" /> <!-- Utilisez l'appender fichier pour le logger racine -->
        </root>


</configuration>
```

JPA - HIBERNATE

Ajouter la dépendance :

- <!-- Framework ORM ((Object-Relational Mapping)-->

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.1.Final</version>
</dependency>
```

Optimisation du générateur d’identifiants

https://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/#mapping-declaration-id-enhanced-optimizers

Identifiers in Hibernate/JPA : https://www.baeldung.com/hibernate-identifiers

Un générateur d'identifiants est utilisé pour créer des valeurs uniques qui serviront d'identifiants pour les entités persistantes. Hibernate propose plusieurs stratégies (**Identity, Séquence, Table, Auto**) de génération d'identifiants, et le choix de la stratégie dépend des besoins.

On peut optimiser le générateur d’identifiant afin de minimiser le nombre d'accès à la base de données.

Pour cela vous pouvez en regrouper plusieurs en mémoire et accéder à la base de données uniquement lorsque vous avez épuisé votre groupe de valeurs en mémoire.
Cela devient particulièrement important lorsque vous stockez des valeurs dans la base de données pour les utiliser comme identifiants, car l'accès fréquent à la base de données peut avoir un impact sur les performances.

C’est le rôle des Optimiseurs :

- **none** : C'est la valeur par défaut. Avec cet optimiseur, une requête est effectuée à la base de données chaque fois qu'un nouvel identifiant est nécessaire. Cela peut entraîner un grand nombre d'accès à la base de données, ce qui peut être inefficace en termes de performances.

- **hilo** : Cet optimiseur utilise un algorithme hi/lo (high/low) pour générer des identifiants. Il regroupe plusieurs identifiants en mémoire avant de faire un seul accès à la base de données pour récupérer de nouveaux identifiants. Les valeurs de la base de données indiquent le "numéro de groupe", et le regroupement en mémoire est effectué en utilisant cet identifiant de groupe. L'option increment_size détermine combien de valeurs sont stockées en mémoire à chaque regroupement.

- **pooled** : Similaire à l'optimiseur hilo, mais au lieu de stocker des valeurs séquentielles en mémoire, il stocke la valeur de départ du "groupe suivant" dans la base de données. Encore une fois, l'option increment_size est utilisée pour déterminer le nombre d'identifiants dans chaque groupe.

Ces optimiseurs offrent des approches différentes pour minimiser les accès à la base de données, en regroupant plusieurs identifiants en mémoire avant de faire une requête à la base de données.

Les valeurs « 1, 2, 52, 102, 152, 202, 252, 302, 352, 402 » sont des identifiants générés pour vos entités par le générateur d'identifiants en utilisant un générateur d’identifiant hi-lo ou pooled avec un increment_size de 50

next_val est une valeur que Hibernate utilise pour suivre la prochaine valeur de séquence disponible

Gestion de la concurrence :

https://www.baeldung.com/jpa-optimistic-locking

- Lorsqu'il s'agit d'applications d'entreprise, il est crucial de gérer correctement les accès simultanés à une base de données. Cela signifie que nous devrions être en mesure de gérer plusieurs transactions de manière efficace et, surtout, sans erreur.

- De plus, nous devons garantir que les données restent cohérentes entre les lectures et les mises à jour simultanées.

- Pour y parvenir, nous pouvons utiliser un mécanisme de v**errouillage optimiste** (Optimistic Locking) fourni par l'API Java Persistence. De cette façon, plusieurs mises à jour effectuées simultanément sur les mêmes données n'interfèrent pas les unes avec les autres.

Comprendre le verrouillage optimiste

- Afin d'utiliser le verrouillage optimiste, nous devons disposer d'une entité incluant une propriété avec l'annotation **@Version** . Lors de son utilisation, chaque transaction qui lit des données contient la valeur de la propriété version.

- Avant que la transaction souhaite effectuer une mise à jour, elle vérifie à nouveau la propriété de version.

- Si la valeur a changé entre-temps, une *OptimisticLockException* est levée. Sinon, la transaction valide la mise à jour et incrémente une propriété de version de valeur.

- Donc le verrouillage optimiste repose sur la détection des modifications apportées aux entités en vérifiant leur attribut de version. Si une mise à jour simultanée a lieu, OptmisticLockException se produit. Après cela, nous pouvons réessayer de mettre à jour les données.

Il faut savoir que l'on peut récupérer une valeur de l'attribut version via l'entité, mais il ne faut pas la mettre à jour ou l'incrémenter. Seul le fournisseur de persistance peut le faire, afin que les données restent cohérentes.

Lors de la gestion de la concurrence, il existe différentes approches pour traiter les conflits. Voici trois

solutions courantes :

- **Ne rien faire (Last Write Wins)** : Dans cette approche, lorsque plusieurs requêtes concurrentes tentent de modifier la même ligne, la dernière requête exécutée écrase les modifications précédentes. Cela peut entraîner la perte de données ou la suppression de modifications apportées par d'autres utilisateurs.

- **Verrouillage de ligne (Lock)** : Cette solution consiste à utiliser des verrous pour empêcher les accès concurrents à une ligne tant qu'une opération n'est pas terminée. Cela garantit qu'une seule requête peut modifier la ligne à la fois, évitant ainsi les conflits de concurrence. Cependant, cela peut entraîner des problèmes de performances, en particulier si les verrous sont conservés pendant de longues périodes.

- **Verrou optimiste (Optimistic Locking)** : Le verrou optimiste est une approche basée sur la version. Chaque ligne de la base de données possède une colonne de version(@Version de JPA) qui est incrémentée à chaque modification. Lorsqu'une requête tente de modifier une ligne, la version est vérifiée. Si la version de la ligne a été modifiée par une autre requête, une exception de concurrence (par exemple, StaleObjectStateException ou OptimisticLockException) est levée. L'application peut alors gérer cette exception et décider de la meilleure façon de gérer le conflit.

Ce matin :

- Créer une interface et une classe générique : IGenericRepository et GenericRepository
    - Contient des méthodes Generic utilisées par toutes les entités
- Modifier l'interface IUtilisateurRepository et la classe UtilisateurRepository
    - Les interfaces et classes précédemment créées les renomer IUtilisateurRepositoryOld et UtilisateurRepositoryOld
    - Compléter l'interface ICompteRepository et créer la classe CompteRepository

- Créer une interface et une classe génériques : IGenericRepository et GenericRepository. Elles contiennent des méthodes génériques utilisées par toutes les entités.

- Compléter l'interface ICompteRepository et créer la classe CompteRepository

- Faire des tests
    - Créer la classe de test CompteRepositoryTest (Ajouter un compte)

- Ajouter une méthode spécifique pour l'utilisateur : Rechercher un utilisateur par son adresse mail

- Ajouter une méthode spécifique pour le compte : Recherche une liste de comptes par date (date de création)

Comprendre @Embedded et @Embeddable

L'annotation **@Embedded** est utilisée dans une entité pour incorporer un type embeddable dans cette entité. Cela permet de traiter les propriétés du type embeddable comme faisant partie de l'entité elle-même

Supposons que nous ayons une classe embeddable Adresse que nous voulons réutiliser dans plusieurs entités.

```
@Embeddable
public class Adresse {
    private String rue;
    private String ville;
    private String codePostal;
    // Getters et setters...
}
```

```
@Entity
public class Client {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String nom;

   @Embedded
   private Adresse adresse;
   // Autres attributs propres au client...
   // Getters et setters...
}

@Entity
public class Fournisseur {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;

    @Embedded
    private Adresse adresse;
    // Autres attributs propres au fournisseur...
    // Getters et setters...
}
```

CASCADE

Le mode cascade permet d'effectuer des opérations en cascade. Par exemple Cascade.persist permet en ajoutant un utilisateur (parent) dans la base de données d'ajouter tous les comptes(enfants) de l'utilisateur Avec JPA, Il existe plusieurs type de cascades qu'on peut utiliser pour définir comment les operations sur une entité parente doivent être propagées à ses entités enfants associées.

- **CascadeType.All** : Cette Cascade propage toutes les operations (PERSIST, MERGE, REMOVE...) à l'entité enfant
- **CascadeType.persist** : Cette cascade propage l'opération 'persist'. Lorsqu'on persist une entité parente, les entités enfants associées sont également persistées
- **CascadeType.remove :** Cette cascade propage l'opération 'remove'. Lorsqu'on utilise l'opération remove une entité parente, les entités enfants associées sont également supprimées.
- **CascadeType.merge :** Cette cascade propage l'opération 'merge'. Lorsqu'on utilise l'opération merge une entité parente, les entités enfants associées sont également mise à jours (fusionnées)

FETCH-TYPE (chargement)

- **Mode de chargement Lazy** : Résoudre le problème LaztInitializationException

2. Créer une méthode findUserWithComptes en utilisant Join FETCH et le tester

SELECT u FROM Utilisateur u JOIN FETCH u.comptes c WHERE...

2. Remplacer FetchType.Lazy par FetchType.EAGER, tester la methode findById

Relation ManyToMany

- **Entities** : Créer la classe Role et modifier la classe Utilisateur

- **Repository** : Créer l'interface et la classe pour l'entité Role
    - Il contient une fonctionnalité : Rechercher un (les) role(s) par mot clé. Pour cela créer une méthode List<Role> findRoleByNomContaining(String keyword)
    - La méthode findRoleByNomContaining rechercher des rôles dont le nom contient un mot-clé (keyword)

- **Tester** : Créer trois rôles (Client, Admin, Dev, Conseiller Financier)

Conventions JavaBeans

Un composant JavaBean est une simple classe Java qui respecte certaines conventions sur le nommage, la construction et le comportement des méthodes. Le respect de ces conventions rend possible l'utilisation, laréutilisation, le remplacement et la connexion de JavaBeans par des outils de développement.

Les conventions à respecter sont les suivantes :

- La classe doit être ***« Serializable »*** pour pouvoir sauvegarder et restaurer l'état d'instances de cetteclasse ;
- La classe doit posséder un constructeur sans paramètre (constructeur par défaut) ;
- Les attributs privés de la classe (variables d'instances) doivent être accessibles publiquement via des méthodes accesseurs construit avec get ou set suivi du nom de l'attribut avec la première lettre capitalisée. Le couple d'accesseurs est appelé Propriété ;
- La classe ne doit pas être déclarée final.

MAPPING DE L'HERITAGE

1. Faire l'association entre Compte et Operation
2. Mettre en place le mapping de l'héritage (SINGLE TABLE)
3. Créer l'interface IOperationReposity puis la classe OperationRepositoru
4. Créer une **couche service**
    1. Créer l'interface ICompterService et la classe CompteServiceImpl
    2. Contient les fonctionnalités :

        - L'utilisateur pourra consulter un compte
        - L'utilisateur(client) pour effectuer un versement
        - L'utilisateur(client) pourra effectuer un retrait
        - L'utilisateur (client) pourra effectuer un virement
        - ~~Les listes d'Operations seront paginées~~

1. Tester : Tester les méthodes qui sont dans la couche service