M2 IMIS – M2 MIAGE 2021-2022

Architectures Applicatives Réparties

TD n°3 : Premiers pas en JPA ...

Objectif : Découvrir le mapping objet-relationnel de la Java Persistence API (JPA)

Si vous avez quelque souvenir de Doctrine (via PHP/Symfony) vous avez déjà fait de la création et de la manipulation d'entity (à la sauce PHP, mais la syntaxe est assez proche).
Nous allons ici voir comment ça se passe en java et approfondir quelque peu…

Exemple n°1 : L'éternel retour du login password…
On reprend ici l'exemple des TDs précédents. Mais les utilisateurs sont désormais stocker en BD. La classe User devient donc un entity.

On remarquera que :

- L'entity est doté des annotations qui vont bien :
  - @Entity
  - @Id
- Pour faire du stockage en BD il faut… une BD ! Ici il s'agit d'une BD en mémoire de type H2, et même d'une BD embarquée, c'est à dire créée quand on déploie l'application (et supprimée quand l'application s'arrête...). Pour ce faire on a mis les bonnes dépendances dans le pom.xml
  - h2 pour importer la librairie contenant le soft de H2
  - spring-jdbc, spring-orm et hibernate-entitymanager pour gérer la persistence via JPA
- Le pom.xml ne fait pas tout : il faut ensuite démarrer la machinerie. Ca se passe dans ClientWebConfig.java :
  - la méthode dataSource() crée un Bean d'accès à la BD (et ici il démarre la BD en mémoire). Dans le jargon JPA, la BD est une source de données ou Data Source...
  - On n'utilise pas la dataSource directement (on peut le faire, mais alors c'est du bon vieux JDBC). Pour bénéficier du mapping objet-relationnel, on utilise un EntityManager, que l'on déclare dans la méthode entityManagerFactory (OK, ça crée plutôt une factory, mais l'utilisation de celle-ci est transparente).
  - La méthode entityManagerFactory s'appuie sur la méthode additionalProperties, qui complète le paramétrage d'Hibernate (JPA est une spécification. Hibernate est une librairie qui implémente JPA. L'Entity Manager est un concept de la JPA, et donc Hibernate fournit une implémentation de l'Entity Manager. C'est bon, vous suivez?) Quels les paramètres spécifiés dans additionalProperties ? Quel est leur rôle ?
- Quand notre application se déploie, une instance de H2 est démarrée et une table y est automatiquement créée pour stocker les instances de notre entity.
- Comment consulter la BD, pour vérifier que la table existe et que les tuples sont bien ajoutée ?
  - On ajouté quelques lignes magiques dans WebServletConfiguration.java, qui lancent une console web pour accéder à h2 (jetez-y un oeil)
  - se connecter à http://localhost:8080/exemple1_war_exploded/console/
  - modifier les paramètres par défaut pour leur donner **exactement** les valeurs ci-dessous :
    - **JDBC URL : jdbc:h2:mem:testdb**
    - **User Name : sa**
      (il n'y a pas de mot de passe)
    - Si vous ne modifiez pas avec ces valeurs, vous vous connecterez bien à H2 mais une base vide !
  - Ensuite vous devez voir votre table dans la barre de gauche. Si vous cliquez sur le nom de la base cela génère un SELECT *, il n'y a plus qu'à cliquer sur « Run »…

L’exemple est un peu plus complexe que dans les Tds précédents…

- L’utilisateur est en fait défini à deux endroits :
  - comme backing bean de spring MVC (UserDto)
  - comme entity (User)
- On aurait pu se contenter d’une seule classe et utiliser l’entity comme backing bean. Cela fonctionne très bien, est économique en déclaration de classes, mais présente quelques inconvénients (lesquels?)
- On a créé à l’avance les comptes d’Alice et Bob.
  - Il y a dans resources un fichier sql contenant les insert into (du vrai SQL)
  - Dans la méthode additionalProperties (ClientWebConfig.java), on a spécifié la propriété `"hibernate.hbm2ddl.import_files"`, en l’associant à notre fichier SQL. Ainsi le script est exécuté au déploiement.
  - Remarque : quand on utilise une « vraie » base de données, où les tables sont déjà créées, l’écriture des insert peut être assisté par IntelliJ. Mais avec une base embarquée ce n’est pas possible…

Exercice n°1 : tout tout tout vous saurez tout...
En reprenant l’exemple n°1 modifier le contrôleur, le service et la page thymeleaf pour afficher la liste des utilisateurs inscrits (vous pouvez même afficher leurs mots de passe si vous voulez, et méditer là dessus...

Exemple n°2 : Transactons !
On poursuit avec le login/password, mais on rajoute une fonctionnalité :On peut non seulement vérifier une paire login / password, mais également inscrire de nouveaux utilisateurs. Au passage on vérifie que le nouveau login est disponible et en cas de doublon on a recours à une bonne vieille exception…

- Comment créer et stocker un entity ? Regardons notre Facade de plus près…
  - L’entity manager est injecté au début de la classe. Normal, c’est un bean…
  - La méthode create est appelé par le contrôleur
  - l’entity est créé en mémoire (c’est un objet de base…)
  - puis on invoque la méthode persist de l’entity manager. Une ligne est ajoutée dans la BD. Ca y est l’entity est persisté
  - on n’est pas en Doctrine : pas la peine de faire un flush(), il se fait tout seul (la méthode existe mais on en a très rarement besoin)
  - par contre on a donc mettre une annotation @Transactional avant notre méthode. Enlevez-la. Que se passe-t-il ?
- Ne reste qu’à tester en jetant un œil au contenu de la BD
  - Vous devez voir la liste des messages s’allonger à chaque fois que vous en saisissez un nouveau…
- Pour que tout fonctionne il a fallu rajouter une méthode dans ClientWebConfig qui paramètre la gestion de transaction : transactionManager

Exercice n°2 : You say hello I say good bye...
Symétriquement à l’opérateur persist l’entity manager fournit un opérateur remove. Modifiez le code pour pouvoir supprimer l’utilisateur connecté. Attention remove s’applique à un entity se trouvant déjà dans la base de données. Vous devrez donc enchaîner un find (pour accéder à l’entity persisté) puis un remove. Prenez le temps de bien observer ce qu’il se passe dans la base et dans la mémoire (peut-on encore manipuler l’entity en mémoire après le remove ? Pourquoi?)

Exemple n°3 : On touille le tout...
Dans cet exemple on défini maintenant deux entities : User et Message. Et on relie les deux classes. On introduit donc une relation.

Pour la classe Message, la valeur de la clé est incrémentée automatiquement par le gestionnaire d'entities car on a précisé « GeneratedValue » (attention : ça ne veut pas dire que la BD gère cela avec de l'autoincrement. Ca veut juste dire que JPA s'en charge et que vous ne gérez donc pas à la main).

Ici le User possède deux listes de Message :
- sent pour les messages envoyés
- received pour les messages reçus

Symétriquement chaque Message présente deux champs :
- from qui est une référence vers l'émetteur
- to qui est une référence vers le destinataire

Déployez et consultez les tables créées. Comment se fait le mapping de la relation ?

On notera la présence d'annotations OneToMany et ManyToOne pour indiquer que les attributs sont des références vers des entities (ou des listes d'entities).
- Commentez ces annotations (les 4 + les deux OrderBy) et redéployez. Que se passe-t-il (s'il ne se passe rien, insérez des messages) ? Pourquoi ?
- Décommentez du côté User et redéployez. Que se passe-t-il ? Pourquoi ?
- Les OrderBy sont de petits plus pratiques : il trie les listes suivant un critère donné. Et ça marche aussi avec des Set !
- Enlevez les éléments « fetch », redéployez et observez ce qu'il se passe quand vous utilisez l'application. Pourquoi ?
- Retirez les « mapped by », redéployez et observez les tables. Qu'est-ce qui a changé ? Pourquoi ? Quel est donc le rôle du « mapped by » ?

On pourra enfin observer que l'on a utilisé un attribut de type date (localDateTime) dans les messages. L'annotation @Basic permet à JPA de savoir qu'il gérera cette date avec les types par défaut de SQL.

Exercice n°3 : Good bye again
Reprendre l'exercice précédent. Mais cette fois on s'assure que tous les messages associés à l'utilisateur supprimé le sont bien également. Retirer l'attribut « cascade » dans les annotations OneToMany de l'utilisateur. Observez ce qu'il se passe. Pourquoi ?

Exemple n°4 : Stratégie et requêtes
Ici nous allons directement insérer quelques messages au déploiement, via le insert.sql.

Nous ajoutons également une méthode qui récupère tous les messages contenant le mot « AAR » (plus précisément la séquence de caractères « aar » en majuscules ou minuscules). Pour cela le find ne suffit plus. Il faut forger une requête. Observez la méthode « findByTextAAR ». Elle contient un select qui ressemble à du SQL mais qui n'en est pas : Il s'agit de JPQL. On y manipule des objets java, pas des tables SQL.

Notez que vous pouvez retirer le début de la commande « Select m », elle fonctionnera toujours (même si IntelliJ râle un peu).

Si vous avez été curieux, vous avez déjà noté des commandes de ce type dans l'exemple précédent (getAllUserNameExcept). Cette méthode est même encore un peu plus complexe : elle ne renvoie pas des entitites complets mais uniquement leur champ login. Cette méthode repose également sur une spécification a posteriori de la valeur de ses paramètres (voir « setParameter("n",name) »). Et en plus elle utilise la possibilité d'empiler les invocations de méthodes pour faire plus compact.

On notera enfin que l'on a précisé une « strategy » pour l'autoincrémentation de la clé des messages. Quel est son intérêt ? (consulter les docs). Retirez-le, insérez un message. Observez. Que se passe-t-il et pourquoi ?

Exercice n°4 :
Modifiez le code pour pouvoir chercher n'importe quel séquence de caractère dans le texte des messages et aussi dans le nom de l'émetteur ou du récepteur.

Exercice n°5 :
Créer un site de TODO List :

- On peut se créer un compte
- Une fois identifié on a accès à sa propre TODO List.
- On peut ajouter des tâches avec un degré d'urgence (urgent ou pas) et d'importance (important ou pas). Les tâches ont un titre et une description.
- On peut consulter la liste des tâches coupée en 4 groupes (combinaisons des 2 critères)
- Les tâches peuvent être modifiées, marquées comme terminées et supprimées.

Seule la trame (principaux packages, classes de configuration Spring, arborescences pour les fichiers html/thymeleaf) est fournie.

Variantes (si vous vous ennuyez ou avez des insomnies) :

- Remplacer le degré d'urgence par une date butoir
- Ajouter un degré d'avancement de la tâche (en %)