Partie 2 – Persistance et gestion des relations

Important ! Nous étudions les différentes étapes du développement d'une application web dynamique avec Spring Boot et IntelliJ. Ces étapes étant dépendantes, **il est indispensable que chaque partie soit entièrement réalisée avant de passer à la suivante**. Une partie pourra occuper plusieurs séances de TP.

1. Mise en place de la persistance des données

Nous allons utiliser Spring Data pour gérer la persistance des données représentant les utilisateurs et les activités manipulés par notre application web dynamique. Puisque nous sommes en phase de développement, nous allons utiliser une base de données *in-memory* H2 plus lègère.

1. Annotez les classes du domaine « Utilisateur » et « Activite » pour qu'elles soient indentifiées commes des entités à faire persister. Associez-leur une clé primaire de type Long générée automatiquement.
2. Afin de mettre en oeuvre le mécanisme de *repository* de Spring Data au sein d'une couche « service », ajoutez :
   - une interface « friendsofmine.repositories.ActiviteRepository »,
   - un service « friendsofmine.service.ActiviteService » dans lequel est injecté un « ActiviteRepository »
   - une interface « friendsofmine.repositories.UtilisateurRepository »,
   - un service « friendsofmine.service.UtilisateurService » dans lequel est injecté un « UtilisateurRepository ».

   Dans le répertoire « test/java », créez des classes de tests « friendsofmine.ActiviteServiceTest » et « friendsofmine.UtilisateurServiceTest » dont les contenus respectifs sont exactement les suivants :
   https://gist.github.com/raclet/dd8bf908d4290633402bb9f0916ed044
   https://gist.github.com/raclet/9e5956daa3e270ebc4b38fff407df20f
   Exécutez les tests et modifiez votre code jusqu'à ce que tous les tests passent avec succès.
3. En cours, vous aviez vu l'utilisation de CrudRepository. Que permet PagingAndSortingRepository et que ne permet pas CrudRepository ? Nous utiliserons cette spécificité un peu plus loin dans le projet..

2. Gestion de relation de type to-one ou to-many entre Activite et Utilisateur

On considère que toute activité à un responsable qui est un utilisateur de l'application. Un utilisateur peut d'ailleurs être responsable de plusieurs activités.

1. Ajoutez au sein des classes Activite et Utilisateur cette relation entre entités de telle façon que les tests de la nouvelle version des classes ActiviteTest et UtilisateurTest ci-dessous passent avec succès :

https://gist.github.com/raclet/12215ffbbd33766a6c49693b99ec7a9a
https://gist.github.com/raclet/4e9e16f4cda7a730264fcdc8dd905822

2. Modifiez le code de la classe « ActiviteServiceTest » pour que son contenu soir identique à celui-ci :
   https://gist.github.com/raclet/63ea3b00c5db49d350685a559ac40b78
   Modifiez le code de « ActiviteService » de telle façon que les tests précédents passent avec succès.

3. Bootstrap exploitant la base de données et transactions

Maintenant que la persistance de données grâce à Spring Data est en place, le bootstrap peut-être réécrit pour exploiter la base de données.

1. Ajoutez dans « ActiviteService » une méthode findAllActivite() qui retourne un objet de type «ArrayList<Activite> » listant les activités triées par titre croissant.
   De manière similaire, ajoutez une méthode findAllUtilisateur() dans « UtilisateurService » qui retourne un objet de type « ArrayList<Utilisateur> » qui liste les utilisateurs triés par nom croissant.
2. Modifiez les classes « UtilisateurController » et « ActiviteController » pour qu'elles exploitent le service correspondant à chaque classe du domaine au lieu de faire référence directement au bootstrap.
3. Modifiez la classe « InitialisationService » de telle façon que le jeu de données créé soit persisté en base de données au lieu d'être sauvegardé au sein d'une paire de ArrayList.
4. Lancez les tests suivants dans une classe « friendsofmine.BootstrapTest » dans « test/java » :
   https://gist.github.com/raclet/8217a012e5310e553fb2a5175d542f73
   Modifiez votre code jusqu'à ce que tous ces tests passent avec succès.

   Nous allons comparer l'exécution d'un flot de requêtes dans un contexte transactionnel par rapport à un flot de requête exécuter à l'extérieur d'une transaction.

5. Modifiez la classe « InitialisationService » de telle sorte que l'insertion de la dernière activité échoue lors de l'appel de la méthode initActivites(). Par exemple, affectez à l'activité un titre vide ou « null ».
   Lancez (ou relancez) le test « testNombreActivite » de la classe « BootstrapTest » et vérifiez que 2 activités apparaissent dans la liste des activités au lieu des 3 attendues.
6. Annotez la classe InitialisationService de l'annotation *org.springframework.transaction.annotation.Transactional*.
   Lancez le test « testNombreActivite » de la classe ActiviteServiceITest.
   Combien d'activités ont été créées ? Justifiez la réponse.

4 - Réflexions sur le SQL généré par Hibernate

Nous allons à présent observer le SQL généré par Hibernate lors de la récupération des activités.

1. Dans le fichier application.properties (dans le dossier src/main/resources), ajoutez la ligne suivante :
   logging.level.org.hibernate.SQL=DEBUG
2. Relancez l'application et observez le contenu de la console lorsque vous accéder à l'URL http://localhost:8080/activites. La console affiche maintenant le SQL généré par Hibernate pour chaque requête.
   Vous pouvez faire un « clear » de la console à période régulière afin de pouvoir observer plus facilement le SQL lors du rechargement de la page.
   Combien de requêtes sont générées lors de l'affichage de toutes les activités ? Des requêtes vous paraissent-elles « inutiles » ? Pouvez vous expliquer leur présence.
3. Modifiez l'application de telle sorte qu'une seule requête soit exécutée lors de l'accès à l'URL http://localhost:8080/activites