Gestion des transactions

1 Préalables

Depuis le premier TP, nous utilisons un accès en base de données (via les Repositories Spring) presque sans nous préoccuper de la gestion des transactions.

L'annotation `@Transactional` [1] indique à Spring d'utiliser la politique par défaut pour préparer une transaction utilisable. Cette politique revient à créer une transaction par thread et par méthode. La transaction est créée avant l'appel de la méthode et elle est validée et fermée après l'appel de la méthode.

Nous nous proposons d'étudier des variations autour de cette politique.

À faire :

- Reprenez le projet du TP précédent.
- créez le package `myboot.app2.model` : nos données.
- créez le package `myboot.app2.services` : nos services logiciels.
- créez le package `myboot.app2.test` dans le répertoire `test`.

2 Mise en oeuvre

Travail à faire : suivez les étapes ci-dessous.

- Préparez l'entité suivante. C'est un compteur identifié par un nom.

```java
package myboot.app2.model;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.Id;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Counter {

    @Id
    private String name;

    @Basic
    private Integer value;

}
```

- Ajoutez un nouveau service :

1. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Transactional.html

```java
package myboot.app2.services;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import myboot.app2.model.Counter;

@Service
@Transactional
public class CounterManager {

    @PersistenceContext
    EntityManager em;

    public Counter getCounter(String name) {
        return em.find(Counter.class, name);
    }

    public void removeCounter(String name) {
        Counter c = em.find(Counter.class, name);
        if (c != null) {
            em.remove(c);
        }
    }

    public void createCounter(String name, Integer value) {
        removeCounter(name);
        Counter c = new Counter();
        c.setName(name);
        c.setValue(value);
        em.persist(c);
    }

}
```

Remarques :

- Spring va automatiquement injecter une instance d'un `EntityManager`.
- Les transactions sont automatiquement ouvertes et fermées lors de l'appel des méthodes publiques du service.
- Lors de l'appel à `createCounter`, les deux méthodes `removeCounter` et `createCounter` travaillent sur la même transaction qui est validée à la sortie de la méthode `createCounter`.

- Testez votre service :

```java
package myboot.app2.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import myboot.app2.model.Counter;
import myboot.app2.services.CounterManager;

@SpringBootTest
public class TestCounterManager {

    @Autowired
    public CounterManager cm;

    @Test
    public void testCounterManager() {
        cm.createCounter("C1", 10);
        Counter c = cm.getCounter("C1");
        assertEquals(Integer.valueOf(10), c.getValue());
    }

}
```

- Enrichissez votre test afin de vérifier la lecture et la suppression des compteurs.

2.1 Gestion des erreurs JPA

- Ajoutez à votre service une méthode qui permet de sauvegarder deux compteurs :

```java
public void doubleSave(Counter c1, Counter c2) {
    em.persist(c1);
    em.persist(c2);
}
```

- Vérifiez dans un test unitaire que, si vous sauvegardez deux fois le même compteur, vous avez bien une erreur (violation de clé primaire) et qu'en plus, la première insertion est annulée (`rollback`).

2.2 Gestion des erreurs métier

- Imaginons que nous ajoutions à notre service un test sur la valeur du compteur et l'exception associée :

```java
public void createValidCounter(String name, Integer value) throws BadCounter {
    removeCounter(name);
    Counter c = new Counter();
    c.setName(name);
    c.setValue(value);
    em.persist(c);
    if (value < 0) {
        throw new BadCounter();
    }
}
```

```java
package myboot.app2.services;

public class BadCounter extends Exception {

    private static final long serialVersionUID = 1L;

    public BadCounter() {
        super("bad counter");
    }

}
```

Remarque : le test du compteur est volontairement placé à la fin pour générer une éventuelle exception et défaire les modifications déjà effectuées.

• **Une première vérification** : Prévoir un test unitaire afin de vérifier la génération de l'erreur et le `rollback`. Vous devriez avoir un problème avec le `rollback`. Par défaut, Spring ne réalise de retour-arrière que pour les exceptions qui héritent de `RuntimeException`. Ajoutez la clause suivante et testez à nouveau :

```java
@Transactional(rollbackFor = BadCounter.class)
public void createValidCounter(String name, Integer value) throws BadCounter {
    ...
}
```

• **Travail à faire** : Prévoir un test afin de vérifier que la valeur précédente d'un compteur n'est pas supprimée en cas d'erreur.

Remarque : La gestion des erreurs (et les éventuels retours-arrière) sont des points très importants pour construire des services transactionnels fiables.

3 Partage de transaction

• Nous venons de voir que les méthodes d'un même service partagent la transaction courante. Mais quelle est la situation entre services différents. Imaginons que nous ayons un deuxième service spécialisé dans la suppression des compteurs :

```java
package myboot.app2.services;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import myboot.app2.model.Counter;

@Service
@Transactional
public class CounterRemover {

    @PersistenceContext
    EntityManager em;

    public void removeCounter(String name) {
        Counter c = em.find(Counter.class, name);
        if (c != null) {
            em.remove(c);
        }
    }

}
```

Nous pouvons revoir la méthode `createCounter` du service `CounterManager` pour utiliser le service `CounterRemover` :

```java
    @Autowired
    CounterRemover remover;

    @Transactional(rollbackFor = BadCounter.class)
    public void createCounter2(String name, Integer value) throws BadCounter {
        remover.removeCounter(name);
        Counter c = new Counter();
        c.setName(name);
        c.setValue(value);
        em.persist(c);
        if (value < 0) {
            throw new BadCounter();
        }
    }
```

• **Travail à faire** : Vérifiez que cette deuxième version fonctionne comme la première. En clair, pour un client donné, le service appelant partage sa transaction avec le service appelé. Ainsi, l'ensemble des actions métier sont regroupées dans la même transaction.

3.1 Contrôler le partage

Le fonctionnement précédent convient dans la plupart des cas. Nous avons néanmoins besoin de contrôler ce partage. Cela passe par l'attribut `propagation` de l'annotation `@Transactional` [2].

2. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Transactional.html

3.2 Nouvelle transaction

Ajoutons une nouvelle méthode au service `CounterRemover`. L'annotation `@Transactional` [3] indique que Spring doit créer une transaction pour cette méthode et la valider à la sortie de la méthode.

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void removeCounterAndCommit(String name) {
    Counter c = em.find(Counter.class, name);
    if (c != null) {
        em.remove(c);
    }
}
```

Nous pouvons maintenant faire une troisième version de `createCounter` toujours dans le service `CounterManager` :

```
@Transactional(rollbackFor = BadCounter.class)
public void removeAndCreateCounter(String name, Integer value)
throws BadCounter {
    remover.removeCounterAndCommit(name);
    Counter c = new Counter();
    c.setName(name);
    c.setValue(value);
    em.persist(c);
    if (value < 0) {
        throw new BadCounter();
    }
}
```

Travaux à faire :

- Vérifiez par un test unitaire que la version précédente est toujours supprimée (qu'il y ait ou pas une exception).
- Vérifiez qu'un client peut directement utiliser la méthode `removeCounterAndCommit`.

3.3 Transaction existante

Après avoir lu la documentation de l'énumération `Propagation` [4], créez une méthode qui renvoie la valeur d'un compteur et qui est annotée `MANDATORY`.

Travaux à faire :

- Vérifiez que cette méthode n'est pas directement utilisable par un client (pourquoi ?)
- Vérifiez que cette méthode est utilisable via une autre méthode d'un service (il faut créer cette méthode qui permet l'accès).

3.4 Aucune transaction

Créez une méthode annotée par la propagation `NEVER`. Vérifiez par un test unitaire le fonctionnement de cette propagation.

À faire : Vous pouvez également tester les autres modes de propagation.

3. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Transactional.html
4. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Propagation.html

3.5 Transaction en lecture seule

Créez une méthode qui utilise l'attribut `readOnly` de `@Transactional` [5]. Vérifiez par un test unitaire le fonctionnement de ce mode (lisez soigneusement la documentation).

4 Isolation

Avec l'attribut `isolation` de l'annotation `@Transactional` [6], nous pouvons piloter le mode d'isolation de la transaction.

Travaux à faire :

- Commencez par lire la documentation de l'énumération `Isolation` [7].
- Il est relativement compliqué de tester ces modes. Débutez en construisant un service qui va surveiller l'évolution d'un compteur et attendre que ce dernier atteigne une valeur limite (il faut lire le compteur `em.find` puis le surveiller avec `em.refresh`).

```
@Transactional(isolation = Isolation.DEFAULT, timeout = 5)
public Counter readGreaterCounter(String name, int min) {
    ...
}
```

- Construisez un test unitaire organisé en deux threads : le premier va surveiller et le second va affecter (après avoir un peu attendu).
- Préparez maintenant une nouvelle version annotée `REPEATABLE_READ` :

```
@Transactional(isolation = Isolation.REPEATABLE_READ, timeout = 5)
public Counter readGreaterCounterRepeatableRead(String name, int min) {
    return readGreaterCounter(name, min);
}
```

- Dupliquez votre test unitaire et modifiez-le afin de vérifier que cette surveillance ne fonctionne pas. En clair, le choix de l'isolation empêche la détection des nouvelles valeurs du compteur. Nous sommes d'ailleurs dans l'obligation de fixer une limite de temps afin d'éviter les boucles infinies.

Documentation complémentaire [8].

5 Créer les transactions manuellement

Nous allons utiliser la classe `TransactionTemplate` [9] pour programmer manuellement les actions à exécuter dans une transaction. Voila un premier exemple :

5. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Transactional.html
6. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Transactional.html
7. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/annotation/Isolation.html
8. https://www.baeldung.com/spring-transactional-propagation-isolation
9. https ://docs.spring.io/spring-framework/docs/5.3.x/javadoc-api/org/springframework/transaction/support/TransactionTemplate.html

```
    @Autowired
    TransactionTemplate transactionTemplate;

    @Transactional(propagation = Propagation.NOT_SUPPORTED)
    public void addManually(Counter c) {
        transactionTemplate.executeWithoutResult(status -> {
            em.persist(c);
        });
    }
```

Travaux à faire :

- Testez le bon fonctionnement mais également l'apparition des erreurs sur violation de clé primaire.
- Ajoutez après la sauvegarde un test sur la valeur du compteur et, si il est négatif, passez la transaction en mode `rollback` (explorez l'argument `status`). Testez ce comportement.
- Utilisez la méthode `execute` qui permet de renvoyer un résultat et enchaînez deux opérations dans votre méthode (persistance et lecture par exemple). Testez le nouveau comportement.