Ejercicios sesión 6: Transacciones

Índice

Vamos a probar las distintas características de atomicidad y aislamiento de las transacciones JPA. Utilizaremos la entidad `Autor` creada en la sesión 1 y algunos programas de prueba que escribiremos en esta sesión. Puedes guardar los programas de prueba en la misma sesión 1.

1. Atomicidad

En este último ejercicio vamos a probar que las transacciones funcionan correctamente. Generaremos un error y probaremos el *rollback* de la transacción.

1. Introduce en el código la gestión de transacciones vista en los apuntes de teoría, de forma que si hay algún error, se captura y se realice un rollback();

2. Produce algún error justo después de haber creado una autor nuevo y haber llamado a `persist(autor)`, y antes de que se haga ningún volcado de sentencias en la base de datos. Comprueba que no se ha modificado la base de datos.

3. Por último, produce algún error justo después de haber realizado la consulta de la colección de mensajes y antes de hacer el commit. Comprueba que funciona correctamente el rollback.

2. Programas de prueba

Utilizaremos en los ejercicios un conjunto de programas de prueba que nos van a permitir simular los distintos problemas y niveles de aislamiento:

Programa **Escritor**:

```
public class Escritor1 {
   public static void main(String[] args) {

      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("simplejpa");
      EntityManager em = emf.createEntityManager();

      pulsaIntro("Pulsa INTRO para Begin + Find");
      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, "pepito");
      System.out.println("Valor:" + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para Set AAA");
      autor.setCorreo("AAA");
      em.flush();
      System.out.println("Nuevo valor: " + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para COMMIT");
      em.getTransaction().commit();

      System.out.println("Transacción terminada. Valor: " +
autor.getCorreo());
```

```
      em.close();
      emf.close();
   }
```

Programa **Lector**:

```
public class Lector {
   public static void main(String[] args) {

      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("simplejpa");
      EntityManager em = emf.createEntityManager();

      pulsaIntro("Pulsa INTRO para Begin + Find");
      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, "pepito");
      System.out.println(autor.getCorreo());

      pulsaIntro("Pulsa INTRO para Find");
      em.clear();
      autor = em.find(Autor.class, "pepito");
      System.out.println("Valor:" + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para COMMIT");
      em.getTransaction().commit();

      System.out.println("Transacción terminada. Valor: " +
autor.getCorreo());
      em.close();
      emf.close();
   }
```

Programa **LectorEscritor**:

```
public class LectorEscritor {
   public static void main(String[] args) {

      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("simplejpa");
      EntityManager em = emf.createEntityManager();

      pulsaIntro("Pulsa INTRO para Begin + Find");
      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, "pepito");
      String c = autor.getCorreo();
      System.out.println("Valor:" + c);

      pulsaIntro("Pulsa INTRO para Find");
      em.clear();
      autor = em.find(Autor.class, "pepito");
      System.out.println("Valor:" + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para actualizar");
      autor.setCorreo(c + "BBB");
      em.getTransaction().commit();

      System.out.println("Transacción terminada. Valor: " +
```

```
autor.getCorreo());
      em.close();
      emf.close();
   }
```

Función **private static void pulsarIntro(String msg)** que se utiliza en todos ellos:

```
private static void pulsaIntro(String msg) {
   try {
      BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
      System.out.println(msg);
      in.readLine();
   } catch (IOException e) {
   }
}
```

3. Niveles de aislamiento

Vamos a comenzar probando cómo se comporta JPA con los distintos niveles de aislamiento de la base de datos. El funcionamiento va a ser dependiente del gestor de base de datos que estemos utilizando.

Podemos definir el nivel de aislamiento SQL con la siguiente propiedad de Hibernate:

```
<property name="hibernate.connection.isolation"
          value="2"/>
```

El valor puede ser:

- 1: READ_UNCOMMITTED
- 2: READ_COMMITTED
- 4: REPEATABLE_READ
- 8: SERIALIZABLE

Vamos a comprobar cómo se consiguen estos niveles de bloqueo utilizando los programas anteriores.

1. READ_UNCOMMITTED: Debe impedirse que dos transacciones modifiquen un mismo dato, de forma que el rollback de una provoque un update perdido.

Para comprobarlo:

- Creamos los programas `Escritor1.java` y `Escritor2.java`. El primero escribirá el valor AAA y el segundo BBB en el mismo dato (el correo electrónico del autor "pepito"). Modificamos Escritor2 para haga un rollback al final.
- Ponemos el modo de aislamiento a 1 (READ_UNCOMMITED)
- Modificamos a mano el correo electrónico del autor.
- Lanzamos Escritor1 y Escritor2.
- Hacemos que escriba Escritor1. Hacemos que Escritor2 comience la transacción. Comprobamos que Escritor2 ve el registro modificado (una lectura sucia), pero que se

produce un bloqueo cuando intenta escribir.
- Escritor1 termina con un commit.
- Escritor2 hace rollback.
- Comprobamos que el valor resultante es AAA.

2. READ_COMMITED: Debe impedirse que se lean datos que no han sido confirmados.

MySQL utiliza la estrategia MVCC (MultiVersion Concurrency Control) para implementar el nivel.

Cambiamos el nivel de aislamiento a 2. Cambiamos el rollback del Escritor2 por un commit y volvemos a hacer los pasos anteriores de Escritor1 y Escritor2.

3. REPEATABLE_READ: Debe impedirse que un update confirmado por T1 modifique el valor de un dato ya leído por T2.

Primero comprobamos que con el nivel actual (2) esto no se cumple:
- Lanzamos Lector1 y hacemos que lea el dato
- Lanzamos Escritor2 y lo ejecutamos hasta el final para que modifique el valor.
- ¿Se ha cambiado el valor en el QueryBrowser?
- Continuamos con Lector1 para comprobar el segundo SELECT. Comprobamos que también ha cambiado. No se ha repetido la lectura y se ha incumplido la restricción.

Ponemos el nivel de aislamiento a 4 y repetimos la prueba.

Comprobamos cómo el lector sigue leyendo el último valor confirmado cuando empezó su transacción, independientemente de que se haya hecho commit de otra transacción.

4. SERIALIZABLE: el nivel de máximo aislamiento.

Primero comprobamos que con el nivel actual esto no se cumple:
- Cambiamos el valor a mano: GGG.
- Lanzamos LectorEscritor para que lea el valor.
- Lanzamos Escritor1 para que modifique el valor, escribiendo AAA y terminando la transacción.
- Continuamos LectorEscritor para que vuelva a leer el valor y escriba el cambio. Vuelve a leer GGG (en lugar de AAA) y escribe GGGBBB.

Cambiamos el nivel a 8 y comprobamos el funcionamiento.

4. Gestión optimista de la concurrencia

Modificamos la clase `Autor` para incluir un atributo que incluya la versión y para que JPA pueda utilizar su estrategia de gestión optimista.

```
@Entity
public class Autor {
```

```
   @Id
   private String nombre;
   @Version
   private Integer version;
   private String correo;
   ...
}
```

Ponemos el nivel de aislamiento a 2 (READ_COMMITED). Probamos un escenario en el que antes se producía un bloqueo (por ejemplo, el del REPEATABLE_READ) y comprobamos que JPA lanza una excepción cuando se intenta sobreescribir un dato con una versión posterior a la que se ha adquirido.

5. Bloqueos JPA

Por último, comprobamos que añadiendo un bloque explícito de JPA en el programa LectorEscritor evitamos que el Escritor realice la primera lectura sobre el valor:

```
pulsaIntro("Pulsa INTRO para Begin + Find");
em.getTransaction().begin();
Autor autor = em.find(Autor.class, "pepito");
em.lock(autor, LockModeType.READ);
String c = autor.getCorreo();
em.lock(autor, LockModeType.READ);
```