Gestión de Transacciones en Spring

Toda aplicación empresarial necesita hacer uso de transacciones, estas garantizan el principio ACID:

* Atomicidad: una operación compuesta por varias sentencias, se realiza completamente o no se realiza. Si alguna de ellas falla, la operación en conjunto será cancelada.
* Consistencia: se conserva la integridad de los datos.
* Aislamiento (isolation): una transacción debe ser independiente de otras transacciones.
* Durabilidad: los cambios realizados deben ser persistidos.

Veamos como Spring nos proporciona herramientas para gestionar transacciones de forma sencilla.

Para asegurar la atomicidad (y también por eficiencia), lo ideal es que una operación formada por diferentes acciones en los repositorios se constituya en una única unidad de trabajo que se ejecute en una misma conexión e implique una única transacción.

Una transacción puede gestionarse de forma local. Imaginemos un método que se encarga de asignar un paciente a una habitación, para ello primero incrementa el número de pacientes de la habitación y después le asigna el id de ésta al paciente.

|  |  |
| --- | --- |
|  | *public void assignPatientToRoom(Patient patient, Room room) {*  *...*  *try {*  *conn = dataSource.gettConnection();*  *conn.setAutocommit(false);*  *...*  *ps = conn.prepareStatement(sql);*  *room.setCurrentPatients(room.getCurrentPatients()+1);*  *...*  *ps.executeUpdate();*  *...*  *ps = conn.prepareStatement(sql);*  *patient.setRoomId(room.getId());*  *...*  *ps.executeUpdate();*  *conn.commit();*  *}*  *catch(Exception e) {*  *conn.rollback();*  *throw new RuntimeException("Error al asignar paciente",e);*  *}*    *}* |

El primer factor a tener en cuenta es que este código es muy propenso a errores, es fácil equivocarse al implementar la lógica de este tipo de flujos a medida que se van haciendo más complejos. Por otro lado, estamos tratando la transaccionalidad a nivel de la capa de acceso a datos, mezclando lógica que afecta a diferentes entidades/modelos/repositorios. En las aplicaciones reales las transacciones normalmente implican acceder a diferentes datos por lo que deberían ser manejadas desde la capa de servicio y no de forma local.

El gestor de transacciones de Spring nos permite configurar la transaccionalidad de nuestras aplicaciones mediante configuración XML, anotaciones o de forma programática y optimiza el acceso a los datos (la conexión a la base de datos se une automáticamente al hilo en ejecución).

La base de este gestor es el interfaz **[PlatformTransactionManager](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/transaction/PlatformTransactionManager.html" \t "_blank)**y existen implementaciones del mismo para diferentes tecnologías (DataSourceTransactionManager, WebLogicJtaTransactionManager, WebSphereUowTransactionManager, HibernateTransactionManager, JpaTransactionManager, JpaTransactionManager…).

Para configurar el gestor basta definir el bean eligiendo la implementación que queramos utilizar:

|  |  |
| --- | --- |
|  | <bean id="transactionManager" class="org.springframework.jdbc.  datasource.DataSourceTransactionManager">    <constructor-arg name="dataSource" ref="dataSource"/>  </bean> |
|  |  |

También es posible declarar un gestor JTA:

|  |  |
| --- | --- |
|  | <tx:jta-transaction-manager/> |

Este resolverá de forma automática a la implementación adecuada para el entorno (WebLogic, WebSphere, JTA, OC4J…).

Una vez definido, la recomendación general es configurar las transacciones de forma declarativa, ya sea mediante configuración XML o anotaciones.

**Mediante anotaciones (Proyecto Transacciones/com.curso.declarativa.anotaciones)**

Lo primero que debemos hacer es habilitar la detección de anotaciones mediante la etiqueta:

|  |  |
| --- | --- |
| 1 | <tx:annotation-driven/> |

Esto nos permitirá utilizar la anotación **@Transactional**:

|  |  |
| --- | --- |
| 1  2  3  4 | @Transactional  public void myTransactionalMethod() {   // Unidad de trabajo atómica  } |

¿Qué ocurre una vez que marcamos el método como transaccional? Antes de que comience la ejecución del método, el gestor iniciará la transacción y, una vez que el método finalice, hará commit. Si durante la ejecución del método ocurre una RuntimeException, el gestor hará rollback. Este es el comportamiento por defecto, puede ser modificado.

También es posible anotar a nivel de clase, y combinar ambos (en ese caso la configuración de los métodos sobreescribirá a la de la clase):

|  |  |
| --- | --- |
|  | @Transactional(timeout=60)  public class MyTransactionalClass {      public void myTransactionalMethod1() {     // Unidad de trabajo atómica    }      @Transactional(timeout=30)    public void myTransactionalMethod2() {     // Unidad de trabajo atómica    }  } |

**Mediante XML (Proyecto Transacciones/com.curso.declarativa.xml)**

A veces no es posible utilizar anotaciones (versiones del JDK anteriores a la 5 o cuando no tenemos acceso al código de los servicios y estos no están anotados). En este caso utilizaremos un Pointcut de AOP para definir qué métodos deben interceptarse y el namespace **tx** para definir el Advice que implementará la lógica de transacciones:

|  |  |
| --- | --- |
|  | <aop:config>   <aop:pointcut id="transactionalMethods" expression="execution(\*  org.example.services.\*.\*(..))"/>   <aop:advisor pointcut-ref="transactionalMethods" advice-ref="txAdvice"/>  </aop:config>    <tx:advice id="txAdvice">    <tx:attributtes>      <tx:method name="get\*" read-only="true" timeout="10"/>      <tx:method name="find\*" read-only="true" timeout="10"/>      <tx:method name="\*" timeout="30"/>    </tx:attributtes>  </tx:advice> |
|  |  |

**Programáticamente (Proyecto Transacciones/com.curso.programatica.\*)**

Como ya hemos dicho, lo recomendable es gestionar las transacciones de forma declarativa. No obstante, disponemos de la plantilla TransactionTemplate para hacerlo programáticamente:

|  |  |
| --- | --- |
|  | ...  txTemplate = new TransactionTemplate(txManager);  return txTemplate.execute(new TransactionCallback() {      public Object doInTransaction(TransactionStatus status) {          try {            // Código transaccional            ...          }          catch(...) {            status.setRollbackOnly();          }          return ...;      }  });  ... |

**Niveles de aislamiento**

Cuando ejecutamos varias transacciones sobre el mismo conjunto de datos existen diferentes tipos de fallos asociados a la concurrencia de las mismas: <http://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_phenomena>

* **Dirty Reads**: la transacción A lee un valor modificado por la transacción B que no ha sido confirmado. ¿Qué ocurre si B hace rollback? A habrá leído un valor no válido.
* **Non-Repeteable Reads**: A lee una fila, B modifica esa fila y hace commit, A vuelve a leer la fila y obtiene valores diferentes a los que tenía anteriormente.
* **Phantom Reads**: A ejecuta una consulta que devuelve varias filas, B realiza una operación (por ejemplo un INSERT o un DELETE) que modifica ese subconjunto de filas, A vuelve a ejecutar la misma consulta y obtiene un conjunto de filas diferente.

Las transacciones deben ser aisladas para prevenir estos errores de concurrencia. A la hora de definir la forma en la que queremos tratar estas situaciones, normalmente utilizaremos uno de estos cuatro niveles de aislamiento:

* **READ\_UNCOMMITTED**: es el nivel más permisivo y está sujeto a los tres problemas anteriores. *@Transaction(isolation=Isolation.READ\_UNCOMMITED)*
* **READ\_COMMITTED**: sólo información sobre la que se ha hecho commit es visible para el resto de transacciones, previene Dirty Reads pero no los otros dos tipos de errores. La mayoría de las veces es el comportamiento por defecto.

*@Transaction(isolation=Isolation.READ\_COMMITED)*

* **REPEATABLE\_READ**: si una fila se lee dos veces durante la misma transacción el resultado siempre será el mismo. Suele causar el bloqueo de la fila mientras dura la transacción. Previene Dirty Reads y Non-Repeteable Reads.

*@Transaction(isolation=Isolation.REPEATABLE\_READ)*

* **SERIALIZABLE**: si un conjunto de filas se lee dos veces durante la misma transacción el resultado siempre será el mismo. Suele causar el bloqueo del conjunto de filas mientras dura la transacción. Previene los tres tipos de [error. *@Transaction(isolation=Isolation.SERIALIZABLE)*](mailto:error.@Transaction(isolation=Isolation.SERIALIZABLE))

**Modos de propagación**

A veces nos encontraremos escenarios en los que un método transaccional llamará a otro método transaccional:

|  |  |
| --- | --- |
|  | *public class PatientServiceImpl implements PatientService {*  *@Autowired*  *private RoomService roomService;*    *@Transactional*  *public assignPatientRoom(Patient patient, Room room) {*  *...*  *this.roomService.incrementCurrentPatients(room);*  *}*  *...*  *}*  *...*  *public class RoomServiceImpl implements RoomService {*  *...*  *@Transactional*  *public void incrementCurrentPatients(Room room) {*  *...*  *}*    *}* |

¿Qué ocurre en estos casos, hay dos transacciones o una? En realidad dependerá del modo de propagación que configuremos. [Existen 7 diferentes](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/transaction/annotation/Propagation.html), nosotros nos centraremos en los dos más utilizados:

* **REQUIRED**: es el valor por defecto, si ya existe una transacción, el método se unirá a esta, si no existe se crea. @Transactional(propagation=Propagation.REQUIRED)
* **REQUIRES\_NEW**: siempre se crea una nueva transacción, si ya existe una en curso, se pone en espera hasta que termine la nueva. @Transactional(propagation=Propagation.REQUIRES\_NEW)

**Rollbacks**

Por defecto, siempre que se dispare cualquier tipo de RuntimeException dentro de una transacción, se hará rollback de ésta. Este compartamiento se puede modificar haciendo uso del parámetro rollbackFor (añadir estas excepciones) y noRollbackFor (no hacer rollback para estas excepciones):*@Transactional(rollbackFor=MyCheckedException.class), @Transactional(noRollbackFor={MyNonRollbackException.class})*

**Transacciones de sólo lectura**

Si tenemos transacciones que únicamente leen datos, es buena idea marcarlas con *@Transactional(readOnly=true)*, el primer motivo es para evitar los errores de concurrencia explicados anteriormente, el segundo es que, además, supondrá una mejora en el rendimiento al agrupar diferentes conexiones en una única:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public void myReadOnlyMethod() {    jdbcTemplate.queryForInt(...); // Primera conexión    jdbcTemplate.queryForLong(...); // Segunda conexión  }    @Transactional(readOnly=true)  public void myReadOnlyMethod() {    // Las dos lecturas se realizan en una única conexión    jdbcTemplate.queryForInt(...);    jdbcTemplate.queryForLong(...);  } |

**Testing**

Las transacciones son una gran ayuda a la hora de definir tests de integración puesto que podemos anotar los métodos de los tests como transaccionales para después hacer rollback y que todo vuelva a quedar en el mismo estado en el que estaba al comienzo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @ContextConfiguration(locations={"/infrastructure-test-context.xml"})  @RunWith(SpringJUnit4ClassRunner.class)  public class MyTest {      @Test    @Transactional    public void myTestMethod() {       // Por defecto este método hará rollback al finalizar    }    } |

También podemos cambiar el comportamiento por defecto utilizando la anotación @TransactionConfiguration:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @ContextConfiguration(locations={"/infrastructure-test-context.xml"})  @RunWith(SpringJUnit4ClassRunner.class)  @TransactionConfiguration(defaultRollback=false, transactionManager="myTransactionManager") // Por defecto no se hará rollback  @Transactional  public class MyTest {      @Test    public void myTestMethod1() {       // Este método NO hará rollback al finalizar    }      @Test    @Rollback(true) // Sobreescribe la configuración a nivel de clase    public void myTestMethod2() {       // Este método hará rollback al finalizar    }    } |

Si en estos tests utilizamos anotaciones @Before y @After hay que tener en cuenta que el primero se ejecuta antes del test pero después de que la transacción haya comenzado y el segundo se ejecuta después del test pero antes de que la transacción se cierre. Para ejecutar código antes o después de la transacción debemos utilizar las anotaciones @BeforeTransaction y @AfterTransaction.