

Spring: Transacciones

Antonio Espín Herranz

Transacciones

- Características.
- Gestor transaccional.
- Configuración XML.
- Atributos:
 - Propagación.
 - Aislamiento.
 - Sólo lectura.
 - Tiempo de Espera.
 - Normas de reversión.
- Transacciones declarativas.
- Transacciones programadas.
- Anotaciones.

Introducción

- Las **transacciones** son operaciones de **todo o nada**.
- Nos permiten agrupar varias operaciones en una única unidad de trabajo que si va todo bien se completa o si no se rechazan los cambios.
- Las transacciones garantizan que los datos y los recursos nunca queden en estado de inconsistencia.
- Toda aplicación empresarial debería hacer uso de transacciones.
- **Spring** permite definir **transacciones de forma programática o declarativa** (XML o anotaciones).

Características A.C.I.D

- **Atómicas:**
 - Están formadas **por una o más actividades agrupadas** entre sí formando una única unidad de trabajo.
- **Coherentes / Consistencia:**
 - Una vez **finalizada** una transacción (sea con éxito o no), **el sistema queda en un estado coherente.**
- **Independientes / Aislamiento:**
 - Deben **permitir que varios usuarios puedan trabajar con los mismos datos** sin que el trabajo de cada usuario se vea afectado por el de otros.
- **Duraderas:**
 - Cuando termina una transacción, **los resultados se deben hacer permanentes.**

Transacciones

- 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.

Compatibilidad de Transacciones en Spring

- **Spring es compatible** con la gestión de **transacciones** tanto **programáticas** como **declarativas**.
- Spring **no requiere ni está ligado a JTA** (Api Transactional Java) como ocurre en **EJB**.
- Si la **transacción** se liga a **un único recurso** (BD):
 - Spring es compatible con: JDBC, Hibernate, JPA ...
- Si son **transacciones distribuidas**:
 - Podemos utilizar implementaciones JTA a terceras partes.

Transacciones en Spring

- Elegir el **modo**:
 - **Programáticamente**:
 - Control exacto de los límites de las transacciones.
 - Más flexibles.
 - Se complica el código de las operaciones.
 - **Declarativa**:
 - Más cómodo, se describen en XML o con anotaciones.
 - **Basada en AOP**, podemos **desacoplar** las operaciones de sus reglas de transacción.

Transacciones en Spring

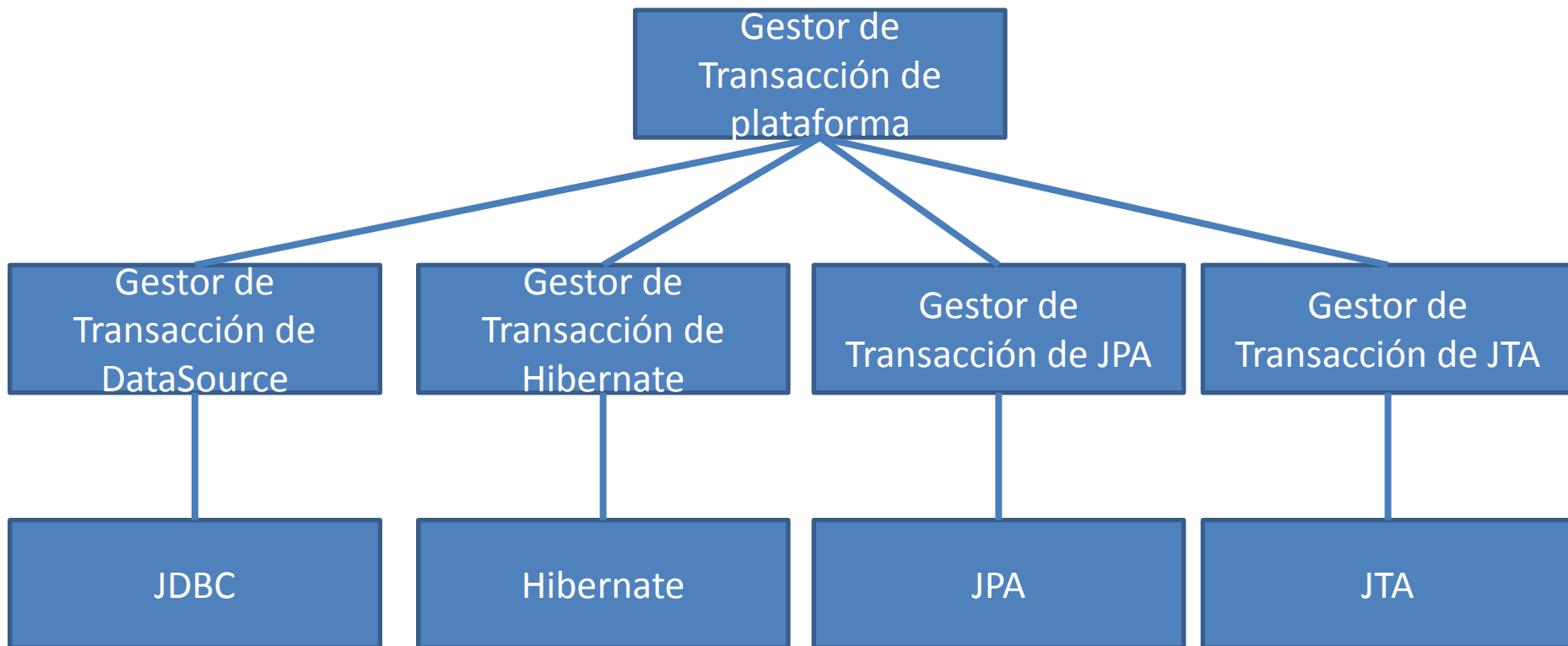
- La base de este gestor es el interfaz **PlatformTransactionManager** y existen implementaciones del mismo para diferentes tecnologías
 - DataSourceTransactionManager,
 - WebLogicJtaTransactionManager,
 - WebSphereUowTransactionManager,
 - HibernateTransactionManager,
 - JpaTransactionManager, ...

Transacciones en Spring

- Elegir un **gestor transaccional**:
 - ¿Son con un recurso o varios? ¿plataforma?
 - Para un **único recurso**:
 - jdbc.datasource.DataSourceTransactionManager:
 - » Para trabajar con **JDBC** e iBATIS.
 - orm.hibernate3.HibernateTransactionManager:
 - » Trabajar con Hibernate 3.
 - **Varios recursos**:
 - transaction.jta.WebLogicJtaTransactionManager:
 - » Aplicación que ejecuta dentro de Web Logic y además necesitan transacciones distribuidas.

Transacciones en Spring

- Los gestores transaccionales delegan la responsabilidad de la gestión de las transacciones a implementaciones específicas para cada plataforma.



Transacciones JDBC

- Para JDBC disponemos de `DataSourceTransactionManager` para la gestión de transacciones.
- Se declara en el fichero de configuración y **se le inyecta un datasource**.

```
<bean id="txManager"  
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

- **DataSourceTransactionManager**: internamente utiliza `java.sql.Connection`, hace operaciones de **commit** / **rollback**.

Programar Transacciones

- En este caso también disponemos de plantillas como en el caso del acceso a BD.

```
public void saveSpittle(final Spittle spittle) {  
    txTemplate.execute(new TransactionCallback<Void>() {  
        public Void doInTransaction(TransactionStatus txStatus) { // Todo lo que va dentro de  
                                                                    // doInTransaction está  
                                                                    // dentro de la transacción.  
  
            try {  
                spitterDao.saveSpittle(spittle); // llamamos a un método del DAO para grabar.  
            } catch (RuntimeException e) {  
                txStatus.setRollbackOnly(); // Si se produce un error de ejecución hacemos rollback().  
                throw e;  
            }  
            return null;  
        }  
    });  
}
```

Para utilizar **TransactionTemplate** se comienza implementando la interfaz **Callback**. Se suele hacer con una clase interna Anónima porque sólo tiene un método que es **doInTransaction**.

En el XML necesitamos

- **DataSource:**
 - configurado con las propiedades de la conexión.
- **DataSourceTransactionManager:**
 - Es el gestor para poder ejecutar transacciones dentro de JDBC.
 - Necesita rellenar una propiedad: dataSource.
- **TransactionTemplate:**
 - Es la plantilla sobre la que podemos ejecutar las transacciones.
 - Necesita rellenar una propiedad: transactionManager.
- A partir de estos irán nuestro objetos de Servicio.

Ejemplo

```
<bean id="datasource"  
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name="url" value="jdbc:mysql://localhost:3306/ejemplos"></property>  
    <property name="username" value="root"></property>  
    <property name="password" value="antonio"></property>  
    <property name="driverClassName" value="com.mysql.jdbc.Driver">  
    </property>  
</bean>  
  
<bean id="txManager"  
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource" />  
</bean>  
  
<!-- Resto de beans ... -->
```

Propiedades de las Transacciones

- Todas las propiedades de la Transacciones se pueden configurar de forma programática, para ello tendremos que utilizar un objeto de la clase **TransactionTemplate**.
- `transactionTemplate.setXXX`
- Las propiedades de las constantes están definidas dentro de la clase:
`org.springframework.transaction.TransactionDefinition`
- Las propiedades las detallaremos más adelante.

Declaración de Transacciones

- La otra posibilidad es declarar las transacciones dentro del fichero de configuración → **transacciones declarativas**.
- Estas **se definen utilizando la AOP** ya que pertenecen a una capa superior que la funcionalidad de la Aplicación.
- Es necesario utilizar el **espacio de nombres tx** y la **anotación @Transactional**.

Atributos de la Transacción

- Las transacciones se definen en términos de:
 - **Propagación.**
 - **Aislamiento.**
 - **¿Sólo lectura?**
 - **Tiempo de Espera.**
 - **Normas de reversión.**
- Hay constantes predefinidas para cada valor.

Atributos de la Transacción

Atributo	Función
isolation	Especifica el nivel de aislamiento de la Transacción. Las transacciones deben ser aisladas para prevenir errores de concurrencia.
propagation	Define las normas de propagación de la transacción. La propagación se da cuando pasamos de un método transaccional o no transaccional a otro que también lo es o que no es transaccional
read-only	Transacción de sólo lectura. El primer motivo es para evitar los errores de concurrencia y el segundo es que, además, supondrá una mejora en el rendimiento al agrupar diferentes conexiones en una única
rollback-for	Especifica las excepciones comprobadas que deben hacer que una transacción se revierta y no se aplique.
no-rollback-for	Especifican las excepciones que no deben hacer que una transacción se revierta.
timeout	Tiempo de espera para una transacción.

Niveles de Aislamiento

- **Definen que parte de una transacción va a verse afectada por las actividades de otras transacciones** que tengan lugar a la vez.
- El trabajo simultáneo sobre los datos puede dar lugar a:
 - **Lectura de datos sucios:**
 - Una tx lee datos que han sido escritos por otra tx pero no se han aplicado. Si los cambios se cancelan los datos de la primera tx no serán válidos.
 - **Lectura no repetible:**
 - Una tx realiza la misma consulta 2 o mas veces y cada vez los datos son diferentes, puede ocurrir porque otra tx está actualizando los datos entre las consultas.
 - **Lectura fantasma:**
 - Una tx lee varias filas y a continuación otra inserta filas, después de varias consultas la primera tx encuentra filas que no estaban ahí.

Niveles de Aislamiento

Las ctes. en `org.springframework.transaction.TransactionDefinition`

<code>ISOLATION_DEFAULT</code>	Utiliza el nivel de aislamiento predeterminado del almacenamiento subyacente.
<code>ISOLATION_READ_UNCOMMITTED</code>	Se puede leer cambios que no se han aplicado, pero ocasiona los 3 tipos de problemas.
<code>ISOLATION_READ_COMMITTED</code>	Permite las lecturas de transacciones simultáneas que se han aplicado, evita datos sucios pero no las otras dos.
<code>ISOLATION_REPEATABLE_READ</code>	Puede producirse lectura de datos fantasma pero no las otras dos.
<code>ISOLATION_SERIALIZABLE</code>	Garantiza que no se producen lecturas fantasma, datos sucios y repetibles.

`ISOLATION_READ_UNCOMMITTED`: es el nivel de aislamiento mas eficiente, pero deja expuesta la lectura de datos sucios, no repetibles y fantasmas.

En el otro extremo **`ISOLATION_SERIALIZABLE`** evita todos estos problemas pero es la menos eficiente.

Comportamiento de Propagación

Las ctes. en `org.springframework.transaction.TransactionDefinition`

PROPAGATION_MANDATORY	El método se tiene que ejecutar en una tx si no se produce excepción.
PROPAGATION_NESTED	El método debe ejecutarse en una tx anidada si hay una progreso.
PROPAGATION_NEVER	El método actual no debe ejecutarse dentro de un contexto transaccional, si hay una tx se genera excepción.
PROPAGATION_NOT_SUPPORTED	Indica que el método no debe ejecutarse dentro de una transacción en caso de que haya una se suspenderá.
PROPAGATION_REQUIRED	El método debe ejecutarse en una tx, si no hay una se creará una nueva.
PROPAGATION_REQUIRES_NEW	El método debe ejecutarse en su propia transacción, se inicia una nueva y si hay una curso se suspenderá.
PROPAGATION_SUPPORTS	El método no requiere un contexto transaccional pero puede ejecutarse dentro de una tx.

Las normas de propagación responden a la pregunta si una nueva transacción debe iniciarse o suspenderse o el método debe ejecutarse o no dentro de un contexto transaccional.

Sólo lectura

- Si una transacción sólo realiza operaciones de lectura en el almacenamiento de datos subyacente, este puede aplicar ciertas optimizaciones que van a sacar partido al carácter de sólo lectura de la transacción.

Tiempo de espera

- Las transacciones no debería de tardar mucho tiempo en ejecutarse.
- Se puede configurar una transacción de forma que se anule de forma automática transcurrido un número específico de segundos.
- Esto sólo cuenta cuando se inicia la transacción, sólo tiene sentido con comportamiento de propagación que inicien una transacción nueva.


Normas de Reversión

- Se pueden declarar una transacción que se revierta si se producen determinadas excepciones comprobadas, así como las excepciones del tiempo de ejecución.
- Se puede declarar una transacción que no se revierta con las excepciones especificadas, incluso aunque se produzcan durante el tiempo de ejecución.

Declaración en XML

- Para definir las transacciones dentro del fichero de configuración debemos **añadir los espacios de nombres tx y aop**.
- Utilizamos **tx:advice** para **definir los atributos y los métodos de la transacción** y aparte necesitamos un **gestor de transacciones** y los **puntos de corte de AOP**.

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
```



```
- <tx:advice id="txAdvice">  
  - <tx:attributes>  
    <tx:method propagation="REQUIRED" name="add*"/>  
    <tx:method propagation="REQUIRED" name="save*"/>  
    <tx:method propagation="SUPPORTS" name="*" read-only="true"/>  
  </tx:attributes>  
</tx:advice>  
- <aop:config>  
  <aop:advisor advice-ref="txAdvice" pointcut="execution(* *..SpitterService.*(..))"/>  
</aop:config>
```

*** *..SpitterService.*(..)** → Cualquier método que se ejecute de la **interface SpitterService**.

Ejemplo

```
<aop:config>
  <aop:pointcut id="transactionalMethods" expression="execution(*
    org.example.services.*.*(..))"/>
  <aop:advisor pointcut-ref="transactionalMethods"
    adviceref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" timeout="10"/>
    <tx:method name="find*" read-only="true" timeout="10"/>
    <tx:method name="*" timeout="30"/>
  </tx:attributes>
</tx:advice>
```

Declaración con Anotaciones

- **Para poder utilizar anotaciones** hay que especificar:
- `<tx:annotation-driven transaction-manager="txManager" />`
- A nivel del fichero configuración, con esto se indica a Spring que examine todos los beans del contexto de aplicación y que **busque** todos los **métodos** que estén **anotados con @Transactional**.

Declaración con anotaciones

@Transactional

```
public void myTransactionalMethod() {  
    // Unidad de trabajo atómica  
}
```

- 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.

Declaración con anotaciones

- También es posible anotar a nivel de clase, y combinar ambos (en ese caso la configuración de los métodos sobrescribirá 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  
    }  
}
```

Ejemplo

```
@Service("spitterService")  
@Transactional(propagation=Propagation.REQUIRED)  
public class SpitterServiceImpl implements SpitterService {  
    // OJO, trabajar con interfaces.
```

```
    @Transactional(propagation=Propagation.SUPPORTS, readOnly=true)  
    public List<Spittle> getRecentSpittles(int count) {  
        List<Spittle> recentSpittles = spitterDao.getRecentSpittle();  
        reverse(recentSpittles);  
        return recentSpittles.subList(0, min(49, recentSpittles.size()));  
    }
```

Propiedades de las anotaciones

- **Niveles de aislamiento:**

@Transaction(isolation=Isolation.READ_UNCOMMITTED)

@Transaction(isolation=Isolation.READ_COMMITTED)

@Transaction(isolation=Isolation.REPEATABLE_READ)

@Transaction(isolation=Isolation.SERIALIZABLE)

- **Rollback:**

@Transactional(rollbackFor=MyCheckedException.class),

@Transactional(noRollbackFor={MyNonRollbackException.class})

- **Solo lectura:**

@Transactional(readOnly=true)