Table of Contents

Spring AOP

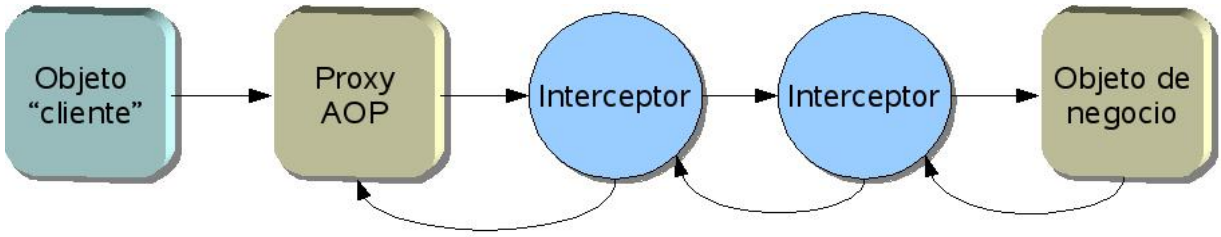
Aspecto: funcionalidad transversal a todas las aplicaciones (por ejemplo, gestión de transacciones o seguridad).

Un aspecto se integra en una aplicación sin necesidad de modificar el código fuente de las clases a las que se aplica.

Esto se consigue o bien mediante interceptores, o bien modificando el bytecode de la clase.

Spring AOP se apoya en @AspectJ

La cadena de ejecución de apectos en spring es la siguiente:



Conceptos

- Aspect (Aspecto) es una funcionalidad transversal (cross-cutting) que se va a implementar de forma modular y separada del resto del sistema. El ejemplo más común y simple de un aspecto es el logging (registro de sucesos) dentro del sistema, ya que necesariamente afecta a todas las partes del sistema que generan un suceso.
- Join point (Punto de Cruce o de Unión) es un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como una llamada a un método, el lanzamiento de una excepción o la modificación de un campo. El código del aspecto será insertado en el flujo de ejecución de la aplicación para añadir su funcionalidad. En Spring AOP un Join Point siempre representa la ejecución de un método.
- Advice (Consejo) es la implementación del aspecto, es decir, contiene el código que implementa la nueva funcionalidad. Se insertan en la aplicación en los Puntos de Cruce.
- Pointcut (Puntos de Corte) define los Consejos que se aplicarán a cada Punto de Cruce. Se especifica mediante Expresiones Regulares o mediante patrones de nombres (de clases, métodos o campos), e incluso dinámicamente en tiempo de ejecución según el valor de ciertos parámetros.
- Introduction (Introducción) permite añadir métodos o atributos a clases ya existentes. Un ejemplo en el que resultaría útil es la creación de un Consejo de Auditoría que mantenga la fecha de la última modificación de un objeto, mediante una variable y un método setUltimaModificacion(fecha), que podrían ser introducidos en todas las clases (o sólo en algunas) para proporcionarlas esta nueva funcionalidad.
- Target (Destinatario) es la clase aconsejada, la clase que es objeto de un consejo. Sin AOP, esta clase debería contener su lógica, además de la lógica del aspecto.

- Proxy (Resultante) es el objeto creado después de aplicar el Consejo al Objeto Destinatario. El resto de la aplicación únicamente tendrá que soportar al Objeto Destinatario (pre-AOP) y no al Objeto Resultante (post-AOP).
- Weaving (Tejido) es el proceso de aplicar Aspectos a los Objetos Destinatarios para crear los nuevos Objetos Resultantes en los especificados Puntos de Cruce. En el caso de Spring, es aplicado en tiempo de ejecución.

Tipos de advice

- Before advice: Se ejecuta antes de invocar a un método.
- After returning advice: Después de invocar un método, si retornó OK.
- After throwing advice: Después de invocar un método, si este lanzó una excepción.
- After (finally) advice: Después de invocar un método lance o no una excepción.
- Around advice: Antes y después de invocar a un método, decidiendo además si se ejecuta o no dicho método o incluso lanzar una excepción.

Proxies dinámicos, CGLib y AspectJ LTW

Tenemos tres mecanismos diferentes para enganchar un aspecto con nuestro código

- Proxy dinámico: Es el mecanismo por defecto. Se trata de un proxy real, generado a partir del interface que implementa el *`target`*. Cuando el objeto *`target`* se invoca a sí mismo, esta segunda llamada no activa los aspectos (ver *Aspectos spring y transacciones* más abajo).
- Si el objeto *`target`* no implementa ningún interfaz Spring utiliza CGLib para modificar el bytecode y generar proxies estáticos. También podemos forzar su uso (código a continuación) en el caso de que queramos aplicar aspectos sobre métodos del objeto *`target`* que no pertenezcan al interfaz que implementa. Tenemos el mismo problema en caso de que un objeto *`target`* se invoque a sí mismo:

  ```
  <!-- Usando config de aspectos declarativa -->
  <aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
  </aop:config>
  ```

  o

  ```
  <!-- Usando anotaciones -->
  <aop:aspectj-autoproxy proxy-target-class="true"/>
  ```

- También podemos usar *AspectJ Weaving* (LTW o CTW). Esto no utiliza proxies spring sino que el código de aspectos se entrelaza directamente en el código final (en tiempo de compilación o en tiempo de ejecución, según decidamos). No tenemos la limitación de las autoinvocaciones del *`target`* sobre sí mismo que tenemos si utilizamos proxies.

  Vamos a ver los pasos que tenemos que dar para utilizar LTW en nuestra aplicación.

  Lo primero es configurarlo en *`beans.xml`* de esta manera,

  ```
  <aop:aspectj-autoproxy/>
  <context:load-time-weaver aspectj-weaving="on"/>
  <tx:annotation-driven transaction-manager="txManagerJpa" mode="aspectj"/>
  ```

Hay que pararse en esta última línea. SI QUEREMOS USAR TRANSACCIONES CON ASPECTJ SÓLO PODEMOS USAR TRANSACCIONES CON ANOTACIONES, XQ EL FICHERO `META-INF/aop.xml` de `spring-aspect.jar`, DONDE SE DECLARAN LOS ASPECTOS PROPIOS DE SPRING, INCLUYE LO SIGUIENTE:

```
<aspect name="org.springframework.transaction.aspectj.AnnotationTransactionAspect"/>
```

QUE COMO VEMOS HACE REFERENCIA A LAS TRANSACCIONES ANOTADAS, PERO NO A LAS DECLARADAS

El valor por defecto del atributo aspectj-weaving es autodetect, que pondrá a on este atributo si spring detecta algún fichero aop.xml en el sistema subyacente, y off en caso contrario. Lo ponemos a on de forma explícita para que no haya confusión.

Necesitamos añadir además un fichero *META-INF/aop.xml* con información sobre las clases a las que queremos aplicar aspectos, y los aspectos (propios) que vamos a aplicar:

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
      "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

  <weaver options="-showWeaveInfo">
    <include within="profe.empleados.negocio.*" />
    <include within="profe.empleados.aspectos.LoggingAspect" />
  </weaver>

  <aspects>
    <aspect name="profe.empleados.aspectos.LoggingAspect"/>
  </aspects>
</aspectj>
```

Para que nuestros aspectos propios funcionen los tenemos que incluir como parte del weaving (en la parte include, como en este caso, el `LoggingAspect`).

No es necesario declarar los aspectos Spring (por ejemplo, transacciones), ya que vienen declarados en el `META-INF/aop.xml` de `spring-aspect.jar`. Lo que sí tendríamos que declarar en este caso es la parte include con las clases anotadas transaccionalmente. Si no declaramos el include, o directamente no aportamos el fichero `aop.xml`, las transacciones siguen funcionando, porque lo que hace aspectj es hacer el weaving sobre todas las clases que se vayan cargando en memoria (podemos comprobarlo modificando la siguiente línea):

```
<weaver options="-showWeaveInfo -debug -verbose">
```

Esto lo que hace es mostrar el weaving por consola. Si, por ejemplo, el include lo hacemos de algún paquete que no existe, entonces los aspectos propios de spring (por ejemplo, los transaccionales) no se aplican. Si no existe el fichero o el include hace referencia a clases reales (por ejemplo, las de negocio) los aspectos se aplican únicamente sobre estas clases.

Por último, al usar LTW necesitamos cargar de forma específica la librería `spring-instrumentation` al arrancar la aplicación, como mostramos en el siguiente código, o configurar nuestro servidor de aplicaciones (Tomcat, JBoss, etc) con código específico:

```
-javaagent:/path/to/spring-instrument.jar
```

Nota: por alguna razón, el weaving no se realiza sobre la clase de entrada a nuestra aplicación (normalmente *Gestoralgo*). QUIZÁS LOS MÉTODOS SOBRE LOS QUE HEMOS PROBADO

ERAN PRIVATE. Esto se puede comprobar añadiendo las siguientes opciones a la etiqueta `<weaver>` del fichero anterior:

```
<weaver options="-showWeaveInfo -debug -verbose">
```

Aspectos spring y transacciones

Resulta que cuando aplicamos aspectos en spring normalmente engancha un proxy que es el que intercepta la llamada y aplica el aspecto. Esto, aplicado a transacciones, resulta en lo siguiente:

> In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual transaction at runtime even if the invoked method is marked with @Transactional.

Por consiguiente, hacer pruebas con atributos transaccionales diferentes en métodos de una misma clase en la que uno de los métodos llama al otro no funciona (tratándose del mismo objeto), porque sería una self-invocation. Esto sucede tanto si estamos usando proxies dinámicos jdk (el objeto sobre el que se aplica el proxy implementa al menos un interfaz) como si usamos cglib (no implementamos ninguna interfaz o forzamos la utilización de cglib de esta manera):

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

Solución para ver ejemplos de atributos transaccionales: utilizar dos objetos diferentes o forzar la utilización de aspectj. En este último caso las transacciones deben configurarse mediante anotaciones.

Cuando usamos algún ORM (Hibernate, JPA) debemos ejecutar todas las modificaciones de la base de datos como parte de una transacción, si no lanzará una excepcion o simplemente no hará nada. Sin embargo, parece que hay alguna incongruencia en el proceso: si no hay ninguna anotación `@Transactional` en el código que se ejecuta lanza una excepción de que no hay transacción activa, pero si hay alguna incluso con atributo de propagación `NEVER` o `NOT_SUPPORTED` parece como si las operaciones fueran autocommit (ver ejemplo al ejecutar `negocio.insertaEmpleados()` sin ningún atributo `@Transaction`, lo que lanza una excepción, y con `@Transaction.NEVER` en el `dao.insertaEmpleado`, que sí que lo inserta aunque en teoría no haya ninguna transacción). Esto lo he probado únicamente con interfaces dinámicos.

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Transactional annotation, as opposed to annotating interfaces. You certainly can place the @Transactional annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are not inherited from interfaces means that if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a transactional proxy, which would be decidedly bad.

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual transaction at runtime even if the invoked method is marked with @Transactional.

When using proxies, you should apply the @Transactional annotation only to methods with public visibility. If you do annotate protected, private or package-visible methods with the @Transactional annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings. Consider the use of AspectJ if you need to annotate non-public methods.

When you use an ORM-based framework, the read-only flag is quite useless and in most cases is ignored. But if you still insist on using it, always set the propagation mode to SUPPORTS, as shown in Listing 9, so no transaction is started.

What are transaction attributes?

Spring transactions allow setting up the propagation behavior, isolation, timeout and read only settings of a transaction. Before we delve into the details, here are some points that need to be kept in mind

Isolation level and timeout settings get applied only after the transaction starts.
Not all transaction managers specify all values and may throw exception with some non default values

Propagation

Transaction Attribute	Client's Transaction	Business Method's Transaction
`Required`	None	T2
	T1	T1
`RequiresNew`	None	T2
	T1	T2
`Mandatory`	None	error
	T1	T1
`NotSupported`	None	None
	T1	None
`Supports`	None	None
	T1	T1
`Never`	None	None
	T1	Error

PROPAGATION_REQUIRED
This attribute tells that the code needs to be run in a transactional context. If a transaction already exists then the code will use it otherwise a new transaction is created. This is the default and mostly widely used transaction setting.

PROPAGATION_SUPPORTS
If a transaction exists then the code will use it, but the code does not require a new one. As an example, consider a ticket reservation system. A query to get total seats available can be executed non-transactionally. However, if used within a transaction context it will deduct tickets already selected and reduce them from the total count, and hence may give a better picture. This attribute should be used with care especially when PROPAGATION_REQUIRED or PROPAGATION_REQUIRES_NEW is used within a PROPAGATION_SUPPORTS context.

PROPAGATION_MANDATORY
Participates in an existing transaction, however if no transaction context is present then it throws a TransactionRequiredException

PROPAGATION_REQUIRES_NEW
Creates a new transaction and if an existing transaction is present then it is suspended. In other words a new transaction is always started. When the new transaction is complete then the original transaction resumes. This transaction type is useful when a sub activity needs to be completed irrespective of the containing transaction. The best example of this is logging. Even if a transaction roll backs you still want to preserve the log statements. Transaction suspension may not work out of the box with all transaction managers, so make sure that the transaction manager supports transaction suspension

PROPAGATION_NOT_SUPPORTED
This attribute says that transaction is not supported. In other words the activity needs to be performed non-transactionally. If an existing transaction is present then it is suspended till the activity finishes.

PROPAGATION_NEVER
This attributes says that the code cannot be invoked within a transaction. However, unlike PROPAGATION_NOT_SUPPORTED, if an existing transaction is present then an exception will be thrown

PROPAGATION_NESTED
The code is executed within a nested transaction if existing transaction is present, if no transaction is present then a new transaction is created. Nested transaction is supported out of the box on only certain transaction managers (i.e., jdbc, ya que se basa en los savepoints de jdbc).

Isolation

Isolation is a property of a transaction that determines what effect a transaction has on other concurrent transactions. To completely isolate the transaction the database may apply locks to rows or tables. Before we go through the transaction levels, let us look at some problems that occur when transaction 1 reads data that is being modified by transaction 2.

Dirty Reads- Dirty reads occur when transaction 2 reads data that has been modified by transaction 1 but not committed. The problem occurs when transaction 1 rollbacks the transaction, in which case the data read by transaction 2 will be invalid.
Non Repeatable Reads- Nonrepeatable reads happen when a transaction fires the same query multiple times but receives different data each time for the same query. This may happen when another transaction has modified the rows while this query is in progress.
Phantom Reads - Phantom reads occur when the collection of rows returned is different when a same query is executed multiple times in a transaction. Phantom reads occur when transaction 2 adds rows to a table between the multiple queries of transaction 1.

The following isolation levels are supported by spring

ISOLATION_DEFAULT
Use the isolation level of the underlying database.

ISOLATION_READ_UNCOMMITTED
This is the lowest level of isolation and says that a transaction is allowed to read rows that have been added but not committed by another transaction. This level allows dirty reads, phantom reads and non repeatable reads.

ISOLATION_READ_COMMITTED
This level allows multiple transactions on the same data but does not allow uncommited transaction of one transaction to be read by another. This level, therefore, prevents dirty reads but allows phantom reads and nonrepeatable reads. This is the default isolation setting for most database and is supported by most databases.

ISOLATION_REPEATABLE_READ
This level ensures that the data set read during a transaction remains constant even if another transaction modifies and commits changes to the data. Therefore if transaction 1 reads 4 rows of data and transaction 2 modifies and commits the fourth row and then transaction 1 reads the four rows again then it does not see the modifications made by transaction 2. (It does not see the changes made in the fourth row by the second transaction). This level prevents dirty reads and non repeatable reads but allows phantom reads.

ISOLATION_SERIALIZABLE
This is the highest isolation level. It prevents dirty reads, non repeatable reads and phantom reads. This level prevents the situation when transaction 1 performs a query with a certain where clause and retrieves say four rows, transaction 2 inserts a row that forms part of the same where clause and then transaction 1 reruns the query with the same where clause but still sees only four rows (does not see the row added by the second transaction)

Read Only

The read only attribute specifies that the transaction is only going to read data from a database. The advantage is that the database may apply certain optimization to the transaction when it is declared to be read only. Since read only attribute comes in action as soon as the transaction starts, it may be applied to only those propagation settings that start a transaction. i.e. PROPAGATION_REQUIRED,PROPAGATION_REQUIRES_NEW and PROPAGATION_NESTED.

Timeout

Timeout specifies the maximum time allowed for a transaction to run. This may be required since transactions that run for a very long time may unnecessarily hold locks for a long time. When a transaction reaches the timeout period, it is rolled back. Timeout needs to be specified only on propagation settings that start a new transaction

Rollback

Por defecto, si nuestras transacciones son declarativas Spring hará rollback automático cuando se produzca una excepción Runtime o no chequeada, o un Error. It is also possible to specify that

transactions roll back on certain exceptions and do not rollback on other exceptions by specifying the rollback rules.

```
<tx:advice id="txAdvice">
    <tx:attributes>
    <tx:method name="*" rollback-for="Throwable" no-rollback-
for="InstrumentNotFoundException"/>
    </tx:attributes>
</tx:advice>
```

Estas reglas determinan que el framework hará rollback en cualquier *`Throwable`* que no sea *`InstrumentNotFoundException`*. Las reglas de rollback admiten patrones separados por comas, y matchean cualquier parte del nombre completo de la excepción:

```
rollback-for="EmpleadosDAOException,profe.spring.empleados.negocio”
```

Los métodos caracterizados por esta regla harán rollback automático cuando se lance la excepción *`EmpleadosDAOException`* o cualquier excepción del paquete *`profe.spring.empleados.negocio`* o sus subpaquetes.

Podemos forzar rollbacks programáticos de esta forma:

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

Transacciones Programáticas

TransactionTemplate

```
public class SimpleService implements Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private final TransactionTemplate transactionTemplate;

    // use constructor-injection to supply the PlatformTransactionManager
    public SimpleService(PlatformTransactionManager transactionManager) {
      this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
      return transactionTemplate.execute(new TransactionCallback() {
        // the code in this method executes in a transactional context
        public Object doInTransaction(TransactionStatus status) {
          updateOperation1();
          return resultOfUpdateOperation2();
        }
```

```
    });
  }
}
```

If there is no return value, use the convenient `TransactionCallbackWithoutResult` class with an anonymous class as follows:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
  protected void doInTransactionWithoutResult(TransactionStatus status) {
    updateOperation1();
    updateOperation2();
  }
});
```

Code within the callback can roll the transaction back by calling the `setRollbackOnly()` method on the supplied `TransactionStatus` object:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

  protected void doInTransactionWithoutResult(TransactionStatus status) {
    try {
      updateOperation1();
      updateOperation2();
    } catch (SomeBusinessExeption ex) {
      status.setRollbackOnly();
    }
  }
});
```

Specifying transaction settings

You can specify transaction settings such as the propagation mode, the isolation level, the timeout, and so forth on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the default transactional settings. The following example shows the programmatic customization of the transactional settings for a specific `TransactionTemplate:`

```
public class SimpleService implements Service {

   private final TransactionTemplate transactionTemplate;

   public SimpleService(PlatformTransactionManager transactionManager) {
      this.transactionTemplate = new TransactionTemplate(transactionManager);

      // the transaction settings can be set here explicitly if so desired
      this.transactionTemplate.setIsolationLevel(
                TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
      this.transactionTemplate.setTimeout(30); // 30 seconds
      // and so forth...
   }
}
```

The following example defines a `TransactionTemplate` with some custom transactional settings, using Spring XML configuration.
The `sharedTransactionTemplate` can then be injected into as many services as are required.

```
<bean id="sharedTransactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
   <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
   <property name="timeout" value="30"/>
</bean>"
```

Finally, instances of the `TransactionTemplate` class are threadsafe, in that instances do not maintain any conversational state. `TransactionTemplate` instances do however maintain configuration state, so while a number of classes may share a single instance of a `TransactionTemplate`, if a class needs to use a `TransactionTemplate` with different settings (for example, a different isolation level), then you need to create two distinct `TransactionTemplate` instances.

PlatformTransactionManager

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
```

```
// explicitly setting the transaction name is something that can only be done
programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
   // execute your business logic here
}
catch (MyException ex) {
   txManager.rollback(status);
   throw ex;
}
txManager.commit(status);
```