Chapter 9. Transaction management

9.1. Introduction

One of the most compelling reasons to use the Spring Framework is the comprehensive transaction support. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes up with some discussion of best practices surrounding transaction management (for example, choosing between declarative and programmatic transaction management).

- The first section, entitled Motivations, describes *why* one would want to use the Spring Framework's transaction abstraction as opposed to EJB CMT or driving transactions via a proprietary API such as Hibernate.
- The second section, entitled Key abstractions outlines the core classes in the Spring Framework's transaction support, as well as how to configure and obtain `DataSource` instances from a variety of sources.
- The third section, entitled Declarative transaction management, covers the Spring Framework's support for declarative transaction management.
- The fourth section, entitled Programmatic transaction management, covers the Spring Framework's support for programmatic (that is, explicitly coded) transaction management.

9.2. Motivations

Traditionally, J2EE developers have had two choices for transaction management: *global* or *local* transactions. Global transactions are managed by the application server, using the Java Transaction API (JTA). Local transactions are resource-specific: the most common example would be a transaction associated with a JDBC connection. This choice has profound implications. For instance, global transactions provide the ability to work with multiple transactional resources (typically relational databases and message queues). With local transactions, the application server is not involved in transaction management and cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.)

Global Transactions. Global transactions have a significant downside, in that code needs to use JTA, and JTA is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be sourced from JNDI: meaning that we need to use *both* JNDI *and* JTA to use JTA. Obviously all use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment. Previously, the preferred way to use global transactions was via EJB *CMT* (*Container Managed Transaction*): CMT is a form of **declarative transaction management** (as distinguished from **programmatic transaction management**). EJB CMT removes the need for transaction-related JNDI lookups - although of course the use of EJB itself necessitates the use of JNDI. It removes most of the need (although not entirely) to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is

> **Is an application server needed for transaction management?**
>
> The Spring Framework's transaction management support significantly changes traditional thinking as to when a J2EE application requires an application server.
>
> In particular, you don't need an application server just to have declarative transactions via EJB. In fact, even if you have an application server with powerful JTA capabilities, you may well decide that the Spring Framework's declarative transactions offer more power and a much more productive programming model than EJB CMT.
>
> Typically you need an application server's JTA capability only if you need to enlist multiple transactional resources, and for many applications being able to handle transactions across multiple resources isn't a requirement. For example, many high-end applications use a single, highly scalable database (such as

only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives around EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

Local Transactions. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Another downside is that local transactions tend to be invasive to the programming model.

Spring resolves these problems. It enables application developers to use a *consistent* programming model *in any environment*. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence don't depend on the Spring Framework's transaction API (or indeed on any other transaction API).

Oracle 9i RAC). Standalone transaction managers such as Atomikos Transactions and JOTM are other options. (Of course you may need other application server capabilities such as JMS and JCA.)

The most important point is that with the Spring Framework *you can choose when to scale your application up to a full-blown application server*. Gone are the days when the only alternative to using EJB CMT or JTA was to write code using local transactions such as those on JDBC connections, and face a hefty rework if you ever needed that code to run within global, container-managed transactions. With the Spring Framework, only configuration needs to change so that your code doesn't have to.

9.3. Key abstractions

The key to the Spring transaction abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by the `org.springframework.transaction.PlatformTransactionManager` interface, shown below:

```
public interface PlatformTransactionManager {

  TransactionStatus getTransaction(TransactionDefinition definition)
    throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;

  void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with the Spring Framework's philosophy, `PlatformTransactionManager` is an *interface*, and can thus be easily mocked or stubbed as necessary. Nor is it tied to a lookup strategy such as JNDI: `PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework's IoC container. This benefit alone makes it a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring's philosophy, the `TransactionException` that can be thrown by any of the `PlatformTransactionManager` interface's methods is *unchecked* (that is it extends the `java.lang.RuntimeException` class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle `TransactionException`. The salient point is that developers are not *forced* to do so.

The `getTransaction(..)` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned `TransactionStatus` might represent a new or existing transaction (if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a `TransactionStatus` is associated with a **thread** of execution).

The `TransactionDefinition` interface specifies:

- **Isolation**: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Propagation**: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case);

or suspending the existing transaction and creating a new transaction. *Spring offers all of the transaction propagation options familiar from EJB CMT*. (Some details regarding the semantics of transaction propagation in Spring can be found in the section entitled Section 9.5.7, “Transaction propagation”.

- **Timeout**: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).
- **Read-only status**: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

These settings reflect standard transactional concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed any other transaction management solution.

The `TransactionStatus` interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus {

  boolean isNewTransaction();

  void setRollbackOnly();

  boolean isRollbackOnly();
}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `PlatformTransactionManager` implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

`PlatformTransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, etc The following examples from the `dataAccessContext-local.xml` file from Spring's **jPetStore** sample application show how a local `PlatformTransactionManager` implementation can be defined. (This will work with plain JDBC.)

We must define a JDBC `DataSource`, and then use the Spring `DataSourceTransactionManager`, giving it a reference to the `DataSource`.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

The related `PlatformTransactionManager` bean definition will look like this:

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

If we use JTA in a J2EE container, as in the `'dataAccessContext-jta.xml'` file from the same sample application, we use a container `DataSource`, obtained via JNDI, in conjunction with Spring's `JtaTransactionManager`. The `JtaTransactionManager` doesn't need to know about the `DataSource`, or any other specific resources, as it will use the container's global transaction management infrastructure.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

  <!-- other <bean/> definitions here -->

</beans>
```

Note

The above definition of the `'dataSource'` bean uses the `<jndi-lookup/>` tag from the `'jee'` namespace. For more information on schema-based configuration, see [Appendix A, *XML Schema-based configuration*](), and for more information on the `<jee/>` tags see the section entitled [Section A.2.3, "The jee schema"]().

We can also use Hibernate local transactions easily, as shown in the following examples from the Spring Framework's **PetClinic** sample application. In this case, we need to define a Hibernate `LocalSessionFactoryBean`, which application code will use to obtain Hibernate `Session` instances.

The `DataSource` bean definition will be similar to the one shown previously (and thus is not shown). If the `DataSource` is managed by the JEE container it should be non-transactional as the Spring Framework, rather than the JEE container, will manage transactions.

The `'txManager'` bean in this case is of the `HibernateTransactionManager` type. In the same way as the `DataSourceTransactionManager` needs a reference to the `DataSource`, the `HibernateTransactionManager` needs a reference to the `SessionFactory`.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
  <list>
    <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
  </list>
  </property>
  <property name="hibernateProperties">
  <value>
          hibernate.dialect=${hibernate.dialect}
        </value>
  </property>
</bean>

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

With Hibernate and JTA transactions, we can simply use the `JtaTransactionManager` as with JDBC or any other resource strategy.

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Note that this is identical to JTA configuration for any resource, as these are global transactions, which can enlist any transactional resource.

In all these cases, application code will not need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

9.4. Resource synchronization with transactions

It should now be clear how different transaction managers are created, and how they are linked to related resources which need to be synchronized to transactions (for example `DataSourceTransactionManager` to a JDBC `DataSource`, `HibernateTransactionManager` to a Hibernate `SessionFactory`, and so forth). There remains the question however of how the application code, directly or indirectly using a persistence API (such as JDBC, Hibernate, and JDO), ensures that these resources are obtained and handled properly in terms of proper creation/reuse/cleanup and trigger (optionally) transaction synchronization via the relevant `PlatformTransactionManager`.

9.4.1. High-level approach

The preferred approach is to use Spring's highest level persistence integration APIs. These do not replace the native APIs, but internally handle resource creation/reuse, cleanup, optional transaction synchronization of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same *template* approach is used for all persistence APIs, with examples including the `JdbcTemplate`, `HibernateTemplate`, and `JdoTemplate` classes (detailed in subsequent chapters of this reference documentation.

9.4.2. Low-level approach

At a lower level exist classes such as `DataSourceUtils` (for JDBC), `SessionFactoryUtils` (for Hibernate), `PersistenceManagerFactoryUtils` (for JDO), and so on. When it is preferable for application code to deal directly with the resource types of the native persistence APIs, these classes ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions which happen in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you would instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

If an existing transaction exists, and already has a connection synchronized (linked) to it, that instance will be returned. Otherwise, the method call will trigger the creation of a new connection, which will be (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, this has the added advantage that any `SQLException` will be wrapped in a Spring Framework `CannotGetJdbcConnectionException` - one of the Spring Framework's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the `SQLException`, and ensures portability across databases: even across different persistence technologies.

It should be noted that this will also work fine without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you are using Spring for transaction management.

Of course, once you've used Spring's JDBC support or Hibernate support, you will generally prefer not to use `DataSourceUtils` or the other helper classes, because you'll be much happier working via the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring `JdbcTemplate` or `jdbc.object` package to simplify your use of JDBC, correct connection retrieval happens behind the scenes and you won't need to write any special code.

9.4.3. `TransactionAwareDataSourceProxy`

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a J2EE server.

It should almost never be necessary or desirable to use this class, except when existing code exists which must be called and passed a standard JDBC `DataSource` interface implementation. In that case, it's possible to still have this code be usable, but participating in Spring managed transactions. It is preferable to write your new code using the higher level abstractions mentioned above.

9.5. Declarative transaction management

Most users of the Spring Framework choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

The Spring Framework's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with the Spring Framework's declarative transaction management. The basic approach is similar: it is possible to specify transaction behavior (or lack of it) down to individual method level. It is possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences are:

- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.
- The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*: this is a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.
- The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than `setRollbackOnly()`.
- The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because normally, one does not want transactions to span remote calls.

The concept of rollback rules is important: they enable us to specify which exceptions (and throwables) should cause automatic roll back. We specify this declaratively, in configuration, not in Java code. So, while we can still call `setRollbackOnly()`on the `TransactionStatus` object to roll the current transaction back programmatically, most often we can specify a rule that `MyApplicationException` must always result in rollback. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other.

While the EJB default behavior is for the EJB container to automatically roll back the transaction on a *system exception* (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an *application exception* (that is, a checked exception other than `java.rmi.RemoteException`). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this.

Where is `TransactionProxyFactoryBean`?

Declarative transaction configuration in versions of Spring 2.0 and above differs considerably from previous versions of Spring. The main difference is that there is no longer any need to configure `TransactionProxyFactoryBean` beans.

The old, pre-Spring 2.0 configuration style is still 100% valid configuration; think of the new `<tx:tags/>` as simply defining `TransactionProxyFactoryBean` beans on your behalf.

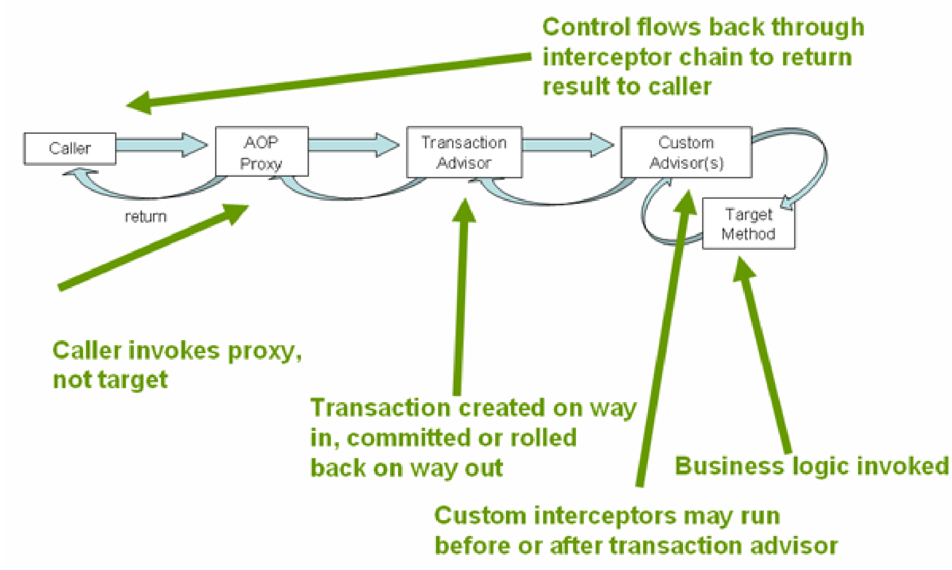
9.5.1. Understanding the Spring Framework's declarative transaction implementation

The aim of this section is to dispel the mystique that is sometimes associated with the use of declarative transactions. It is all very well for this reference documentation simply to tell you to annotate your classes with the `@Transactional` annotation, add the line ('`<tx:annotation-driven/>`') to your configuration, and then expect you to understand how it all works. This section will explain the inner workings of the Spring Framework's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled *via AOP proxies*, and that the transactional advice is driven by *metadata* (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a `TransactionInterceptor` in conjunction with an appropriate `PlatformTransactionManager` implementation to drive transactions *around method invocations*.

> **Note**
>
> Although knowledge of Spring AOP is not required to use Spring's declarative transaction support, it can help. Spring AOP is thoroughly covered in the chapter entitled *Chapter 6, Aspect Oriented Programming with Spring*.

Conceptually, calling a method on a transactional proxy looks like this...



9.5.2. A first example

Consider the following interface, and its attendant implementation. (The intent is to convey the concepts, and using the rote `Foo` and `Bar` tropes means that you can concentrate on the transaction usage and not have to worry about the domain model.)

```
// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

  Foo getFoo(String fooName);

  Foo getFoo(String fooName, String barName);

  void insertFoo(Foo foo);

  void updateFoo(Foo foo);

}
```

```
// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    throw new UnsupportedOperationException();
  }

  public Foo getFoo(String fooName, String barName) {
    throw new UnsupportedOperationException();
```

```
  }

  public void insertFoo(Foo foo) {
    throw new UnsupportedOperationException();
  }

  public void updateFoo(Foo foo) {
    throw new UnsupportedOperationException();
  }

}
```

(For the purposes of this example, the fact that the `DefaultFooService` class throws `UnsupportedOperationException` instances in the body of each implemented method is good; it will allow us to see transactions being created and then rolled back in response to the `UnsupportedOperationException` instance being thrown.)

Let's assume that the first two methods of the `FooService` interface (`getFoo(String)` and `getFoo(String, String)`) have to execute in the context of a transaction with read-only semantics, and that the other methods (`insertFoo(Foo)` and `updateFoo(Foo)`) have to execute in the context of a transaction with read-write semantics. Don't worry about taking the following configuration in all at once; everything will be explained in detail in the next few paragraphs.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
     http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
     http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true"/>
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*"/>
  </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
    of an operation defined by the FooService interface -->
  <aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>
```

```
  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Let's pick apart the above configuration. We have a service object (the `'fooService'` bean) that we want to make transactional. The transaction semantics that we want to apply are encapsulated in the `<tx:advice/>` definition. The `<tx:advice/>` definition reads as "*... all methods on starting with `'get'` are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics*". The `'transaction-manager'` attribute of the `<tx:advice/>` tag is set to the name of the `PlatformTransactionManager` bean that is going to actually *drive* the transactions (in this case the `'txManager'` bean).

Tip

You can actually omit the `'transaction-manager'` attribute in the transactional advice (`<tx:advice/>`) if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the `PlatformTransactionManager` bean that you want to wire in has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

The `<aop:config/>` definition ensures that the transactional advice defined by the `'txAdvice'` bean actually executes at the appropriate points in the program. First we define a pointcut that matches the execution of any operation defined in the `FooService` interface (`'fooServiceOperation'`). Then we associate the pointcut with the `'txAdvice'` using an advisor. The result indicates that at the execution of a `'fooServiceOperation'`, the advice defined by `'txAdvice'` will be run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression; see the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring* for more details on pointcut expressions in Spring 2.0.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

```
<aop:config>
  <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

(This example assumes that all your service interfaces are defined in the `'x.y.service'` package; see the chapter entitled Chapter 6, Aspect Oriented Programming with Spring for more details.)

Now that we've analyzed the configuration, you may be asking yourself, "*Okay... but what does all this configuration actually do?*".

The above configuration is going to effect the creation of a transactional proxy around the object that is created from the `'fooService'` bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked *on the proxy*, a transaction *may* be started, suspended, be marked as read-only, etc., depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration.

```
public final class Boot {

  public static void main(final String[] args) throws Exception {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
    FooService fooService = (FooService) ctx.getBean("fooService");
    fooService.insertFoo (new Foo());
  }
}
```

The output from running the above program will look something like this. *(Please note that the Log4J output and the stacktrace from the `UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated in the interest of clarity.)*

```
  <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
    for bean 'fooService' with 0 common interceptors and 1 specific interceptors
  <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

  <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo
  <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
    [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

  <!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should
    rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo
    due to throwable [java.lang.UnsupportedOperationException]

   <!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
    [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

9.5.3. Rolling back

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled `Exception` as it bubbles up the call stack, and will mark the transaction for rollback.

Note however that the Spring Framework's transaction infrastructure code will, by default, *only* mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Errors` will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional method will *not* result in the transaction being rolled back.

Exactly which `Exception` types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific `Exception` type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
  <tx:method name="get*" read-only="true" rollback-for="NoProductInStockException"/>
  <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

It is also possible to specify 'no rollback rules', for those times when you do *not* want a transaction to be marked for rollback when an exception is thrown. In the example configuration below, we effectively are telling the Spring Framework's transaction infrastructure to commit the attendant transaction even in the face of an unhandled `InstrumentNotFoundException`.

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

When the Spring Framework's transaction infrastructure has caught an exception and is consulting any configured rollback rules to determine whether or not to mark the transaction for rollback, the *strongest* matching rule wins. So in the case of the following configuration, any exception other than an `InstrumentNotFoundException` would result in the attendant transaction being marked for rollback.

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

The second way to indicate that a rollback is required is to do so *programmatically*. Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure, as can be seen below:

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

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you absolutely need it, but its usage flies in the face of achieving a nice, clean POJO-based architecture.

9.5.4. Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply *totally different* transactional configuration to each of them. This is achieved by defining distinct `<aop:advisor/>` elements with differing `'pointcut'` and `'advice-ref'` attribute values.

Let's assume that all of your service layer classes are defined in a root `'x.y.service'` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `'Service'` have the default transactional configuration, you would write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <aop:config>

    <aop:pointcut id="serviceOperation"
          expression="execution(* x.y.service..*Service.*(..))"/>

    <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

  </aop:config>

  <!-- these two beans will be transactional... -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>
  <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

  <!-- ... and these two beans won't -->
  <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->
  <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

  <tx:advice id="txAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

Find below an example of configuring two distinct beans with totally different transactional settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <aop:config>

    <aop:pointcut id="defaultServiceOperation"
          expression="execution(* x.y.service.*Service.*(..))"/>

    <aop:pointcut id="noTxServiceOperation"
          expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>

    <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

    <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

  </aop:config>

  <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- this bean will also be transactional, but with totally different transactional settings -->
  <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

  <tx:advice id="defaultTxAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <tx:advice id="noTxAdvice">
    <tx:attributes>
      <tx:method name="*" propagation="NEVER"/>
    </tx:attributes>
  </tx:advice>

  <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

9.5.5. `<tx:advice/>` settings

This section summarises the various transactional settings that can be specified using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are:

- The propagation setting is `REQUIRED`
- The isolation level is `DEFAULT`
- The transaction is read/write

- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can be changed; the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and `<tx:attributes/>` tags are summarized below:

Table 9.1. `<tx:method/>` settings

Attribute	*Required?*	*Default*	*Description*
`name`	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `'get*'`, `'handle*'`, `'on*Event'`, and so forth.
`propagation`	No	REQUIRED	The transaction propagation behavior
`isolation`	No	DEFAULT	The transaction isolation level
`timeout`	No	-1	The transaction timeout value (in seconds)
`read-only`	No	false	Is this transaction read-only?
`rollback-for`	No		The `Exception(s)` that will trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'`
`no-rollback-for`	No		The `Exception(s)` that will *not* trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'`

At the time of writing it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example `'com.foo.BusinessService.handlePayment'`.

9.5.6. Using `@Transactional`

Note

The functionality offered by the `@Transactional` annotation and the support classes is only available to you if you are using at least Java 5 (Tiger).

In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the `@Transactional` annotation is best illustrated with an example, after which all of the details will be explained. Consider the following class definition:

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

  Foo getFoo(String fooName);

  Foo getFoo(String fooName, String barName);
```

```
  void insertFoo(Foo foo);

  void updateFoo(Foo foo);
}
```

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely *one* line of XML configuration, like so:

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
     http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
     http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <!-- (this dependency is defined somewhere else) -->
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Tip

You can actually omit the `'transaction-manager'` attribute in the `<tx:annotation-driven/>` tag if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the `PlatformTransactionManager` bean that you want to dependency inject has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

The `@Transactional` annotation may be placed before an interface definition, a method on an interface, a class definition, or a *public* method on a class. However, please note that the mere presence of the `@Transactional` annotation is not enough to actually turn on the transactional behavior - the `@Transactional` annotation *is simply metadata* that can be consumed by something that is `@Transactional`-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the case of the above example, it is the presence of the `<tx:annotation-driven/>` element that *switches on* the transactional behavior.

The Spring team's recommendation is that you only annotate concrete classes with the `@Transactional` annotation, as opposed

Method visibility and `@Transactional`

When using proxies, the `@Transactional` annotation should only be applied to methods with *public* visibility. If you do annotate protected, private or package-visible methods with the `@Transactional` annotation, no error will be raised, but the annotated method will not exhibit the configured transactional settings. Consider the use of AspectJ to annotating interfaces. You certainly can place the `@Transactional` annotation on an interface (or an interface method), but this will only work as you would expect it to if you are using interface-based proxies. The fact that annotations are *not inherited* means that if you are using class-based proxies (`proxy-target-class="true"`) or the weaving-based aspect (`mode="aspectj"`) then the transaction settings will not be recognised by the proxying/weaving infrastructure and the object will not be wrapped in a transactional proxy (which would be decidedly *bad*). So please do take the Spring team's advice and only annotate concrete classes (and the methods of concrete classes) with the `@Transactional` annotation.

(see below) if you need to annotate non-public methods.

Note: In proxy mode (which is the default), only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with `@Transactional`!

Consider the use of AspectJ mode (see below) if you expect self-invocations to be wrapped with transactions as well. In this case, there won't be a proxy in the first place; instead, the target class will be 'weaved' (i.e. its byte code will be modified) in order to turn `@Transactional` into runtime behavior on any kind of method.

Table 9.2. `<tx:annotation-driven/>` settings

Attribute	*Default*	*Description*
`transaction-manager`	transactionManager	The name of transaction manager to use. Only required if the name of the transaction manager is not `transactionManager`, as in the example above.
`mode`	proxy	The default mode "proxy" will process annotated beans to be proxied using Spring's AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" will instead weave the affected classes with Spring's AspectJ transaction aspect (modifying the target class byte code in order to apply to any kind of method call). AspectJ weaving requires spring-aspects.jar on the classpath as well as load-time weaving (or compile-time weaving) enabled. (See the section entitled Section 6.8.4.5, "Spring configuration" for details on how to set up load-time weaving.)
`proxy-target-class`	false	Applies to proxy mode only. Controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled Section 6.6, "Proxying mechanisms" for a detailed examination of the different proxy types.)
`order`	Ordered.LOWEST_PRECEDENCE	Defines the order of the transaction advice that will be applied to beans annotated with `@Transactional`. More on the rules related to ordering of AOP advice can be found in the AOP chapter (see section Section 6.2.4.7, "Advice ordering"). Note that not specifying any ordering will leave the decision as to what order advice is run in to the AOP subsystem.

> **Note**
>
> The "`proxy-target-class`" attribute on the `<tx:annotation-driven/>` element controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled [Section 6.6, "Proxying mechanisms"](#) for a detailed examination of the different proxy types.)

> **Note**
>
> Note that `<tx:annotation-driven/>` only looks for `@Transactional` on beans in the same application context it is defined in. This means that, if you put `<tx:annotation-driven/>` in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Transactional` beans in your controllers, and not your services. See [Section 13.2, "The DispatcherServlet"](#) for more information.

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the `DefaultFooService` class is annotated at the class level with the settings for a read-only transaction, but the `@Transactional` annotation on the `updateFoo(Foo)` method in the same class takes precedence over the transactional settings defined at the class level.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    // do something
  }

  // these settings have precedence for this method
  @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
    // do something
  }
}
```

9.5.6.1. `@Transactional` settings

The `@Transactional` annotation is metadata that specifies that an interface, class, or method must have transactional semantics; for example, "*start a brand new read-only transaction when this method is invoked, suspending any existing transaction*". The default `@Transactional` settings are:

- The propagation setting is `PROPAGATION_REQUIRED`
- The isolation level is `ISOLATION_DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can be changed; the various properties of the `@Transactional` annotation are summarized in the following table:

Table 9.3. `@Transactional` properties

Property	Type	Description
[propagation](#)	enum: `Propagation`	optional propagation setting

Property	Type	Description
`isolation`	enum: `Isolation`	optional isolation level
`readOnly`	boolean	read/write vs. read-only transaction
`timeout`	int (in seconds granularity)	the transaction timeout
`rollbackFor`	an array of `Class` objects, which must be derived from `Throwable`	an optional array of exception classes which **must** cause rollback
`rollbackForClassname`	an array of class names. Classes must be derived from `Throwable`	an optional array of names of exception classes that **must** cause rollback
`noRollbackFor`	an array of `Class` objects, which must be derived from `Throwable`	an optional array of exception classes that **must not** cause rollback.
`noRollbackForClassname`	an array of `String` class names, which must be derived from `Throwable`	an optional array of names of exception classes that **must not** cause rollback

Currently it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example, if the `handlePayment(..)` method of the `BusinessService` class started a transaction, the name of the transaction would be:

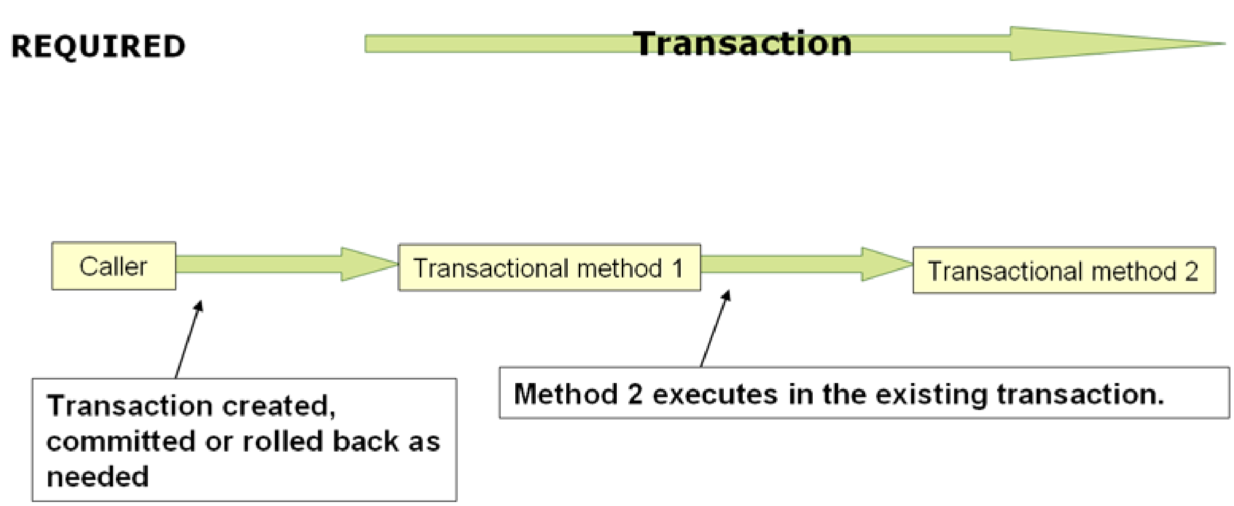
```
com.foo.BusinessService.handlePayment
```

9.5.7. Transaction propagation

Please note that this section of the Spring reference documentation is not an introduction to transaction propagation proper; rather it details some of the semantics regarding transaction propagation in Spring.

In the case of Spring-managed transactions, please be aware of the difference between *physical* and *logical* transactions, and how the propagation setting applies to this difference.

9.5.7.1. Required

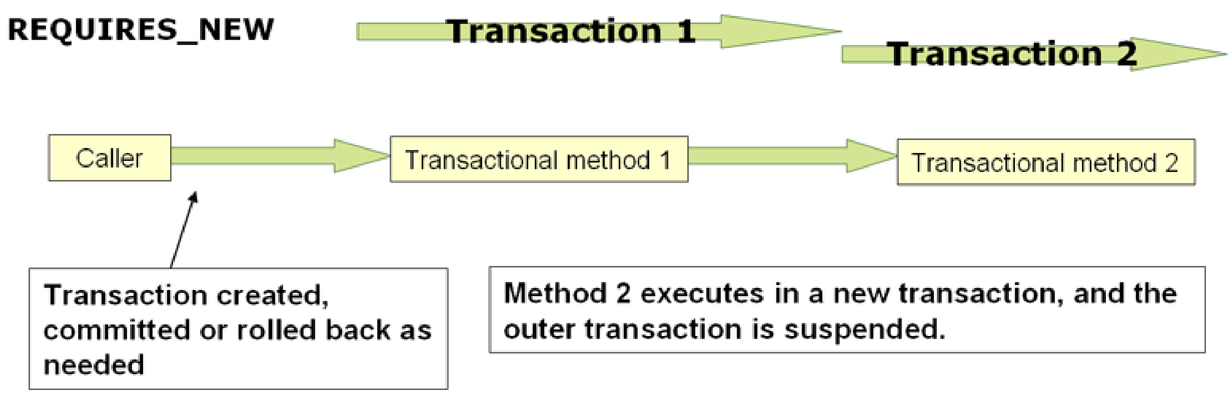


PROPAGATION_REQUIRED

When the propagation setting is `PROPAGATION_REQUIRED`, a *logical* transaction scope is created for each method that it gets applied to. Each such logical transaction scope can individually decide on rollback-only status, with an outer transaction scope being logically independent from the inner transaction scope. Of course, in case of standard `PROPAGATION_REQUIRED` behavior, they will be mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transactions chance to actually commit (as you would expect it to).

However, in the case where an inner transaction scopes sets the rollback-only marker, the outer transaction itself has not decided on the rollback itself, and so the rollback (silently triggered by the inner transaction scope) is unexpected: a corresponding `UnexpectedRollbackException` will be thrown at that point. This is *expected behavior* so that the caller of a transaction can never be misled to assume that a commit was performed when it really was not. So if an inner transaction (that the outer caller is not aware of) silently marks a transaction as rollback-only, the outer caller would still innocently call commit - and needs to receive an `UnexpectedRollbackException` to indicate clearly that a rollback was performed instead.

9.5.7.2. RequiresNew



PROPAGATION_REQUIRES_NEW

`PROPAGATION_REQUIRES_NEW`, in contrast, uses a *completely* independent transaction for each affected transaction scope. In that case, the underlying physical transactions will be different and hence can commit or rollback independently, with an outer transaction not affected by an inner transaction's rollback status.

9.5.7.3. Nested

`PROPAGATION_NESTED` is different again in that it uses a *single* physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks allow an inner transaction scope to trigger a rollback *for its scope*, with the outer transaction being able to continue the physical transaction despite some operations having been rolled back. This is typically mapped onto JDBC savepoints, so will only work with JDBC resource transactions (see Spring's `DataSourceTransactionManager`).

9.5.8. Advising transactional operations

Consider the situation where you would like to execute *both* transactional *and* (to keep things simple) some basic profiling advice. How do you effect this in the context of using `<tx:annotation-driven/>`?

What we want to see when we invoke the `updateFoo(Foo)` method is:

- the configured profiling aspect starting up,
- then the transactional advice executing,
- then the method on the advised object executing
- then the transaction committing (we'll assume a sunny day scenario here),
- and then finally the profiling aspect reporting (somehow) exactly how long the whole transactional method invocation took

> **Note**
>
> This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). Please see the chapter entitled [Chapter 6, *Aspect Oriented Programming with Spring*] for detailed coverage of the various bits and pieces of the following AOP configuration (and AOP in general).

Here is the code for a simple profiling aspect. The ordering of advice is controlled via the `Ordered` interface. For full details on advice ordering, see [Section 6.2.4.7, "Advice ordering"](#).

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

  private int order;

  // allows us to control the ordering of advice
  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  // this method is the around advice
  public Object profile(ProceedingJoinPoint call) throws Throwable {
    Object returnValue;
    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(call.toShortString());
      returnValue = call.proceed();
    } finally {
      clock.stop();
      System.out.println(clock.prettyPrint());
    }
    return returnValue;
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- this is the aspect -->
  <bean id="profiler" class="x.y.SimpleProfiler">
```

```
    <!-- execute before the transactional advice (hence the lower order number) -->
    <property name="order" value="1"/>
  </bean>

  <tx:annotation-driven transaction-manager="txManager" order="200"/>

  <aop:config>
    <!-- this advice will execute around the transactional advice -->
    <aop:aspect id="profilingAspect" ref="profiler">
      <aop:pointcut id="serviceMethodWithReturnValue"
              expression="execution(!void x.y..*Service.*(..))"/>
      <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
    </aop:aspect>
  </aop:config>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>

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

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. The configuration of any number of additional aspects is effected in a similar fashion.

Finally, find below some example configuration for effecting the same setup as above, but using the purely XML declarative approach.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the profiling advice -->
  <bean id="profiler" class="x.y.SimpleProfiler">
    <!-- execute before the transactional advice (hence the lower order number) -->
    <property name="order" value="1"/>
  </bean>

  <aop:config>
```

```
    <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>

    <!-- will execute after the profiling advice (c.f. the order attribute) -->
    <aop:advisor
        advice-ref="txAdvice"
        pointcut-ref="entryPointMethod"
        order="2"/> <!-- order value is higher than the profiling aspect -->

    <aop:aspect id="profilingAspect" ref="profiler">
      <aop:pointcut id="serviceMethodWithReturnValue"
              expression="execution(!void x.y..*Service.*(..))"/>
      <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
    </aop:aspect>

  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. If we wanted the profiling advice to execute *after* the transactional advice on the way in, and *before* the transactional advice on the way out, then we would simply swap the value of the profiling aspect bean's `'order'` property such that it was higher than the transactional advice's order value.

The configuration of any number of additional aspects is achieved in a similar fashion.

9.5.9. Using `@Transactional` with AspectJ

It is also possible to use the Spring Framework's `@Transactional` support outside of a Spring container by means of an AspectJ aspect. To use this support you must first annotate your classes (and optionally your classes' methods with the `@Transactional` annotation, and then you must link (weave) your application with the `org.springframework.transaction.aspectj.AnnotationTransactionAspect` defined in the `spring-aspects.jar` file. The aspect must also be configured with a transaction manager. You could of course use the Spring Framework's IoC container to take care of dependency injecting the aspect. The simplest way to configure the transaction management aspect is to use the `'<tx:annotation-driven/>'` element and specify the `mode` attribute to `asepctj` as described in Section 9.5.6, "Using @Transactional". Since we're focusing here on applications running outside of a Spring container, we'll show you how to do it programmatically.

> **Note**
>
> Prior to continuing, you may well want to read the previous sections entitled Section 9.5.6, "Using @Transactional" and Chapter 6, *Aspect Oriented Programming with Spring* respectively.

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

> **Note**
>
> When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

The `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any method in the class.

The `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Any method may be annotated, regardless of visibility.

To weave your applications with the `AnnotationTransactionAspect` you must either build your application with AspectJ (see the AspectJ Development Guide) or use load-time weaving. See the section entitled Section 6.8.4, "Load-time weaving with AspectJ in the Spring Framework" for a discussion of load-time weaving with AspectJ.

9.6. Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

- Using the `TransactionTemplate`.
- Using a `PlatformTransactionManager` implementation directly.

If you are going to use programmatic transaction management, the Spring team generally recommends using the `TransactionTemplate`. The second approach is similar to using the JTA `UserTransaction` API (although exception handling is less cumbersome).

9.6.1. Using the `TransactionTemplate`

The `TransactionTemplate` adopts the same approach as other Spring *templates* such as the `JdbcTemplate`. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of transactional resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do.

> **Note**
>
> As you will immediately see in the examples that follow, using the `TransactionTemplate` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transactional context, and that will use the `TransactionTemplate` explicitly, looks like this. You, as an application developer, will write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that will contain all of the code that you need to have execute in the context of a transaction. You will then pass an instance of your custom `TransactionCallback` to the `execute(..)` method exposed on the `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) {
    Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
    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 via an anonymous class like so:

```
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();
    }
  }
});
```

9.6.1.1. Specifying transaction settings

Transaction settings such as the propagation mode, the isolation level, the timeout, and so forth can be set on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the default transactional settings. Find below an example of programmatically customizing the transactional settings for a specific `TransactionTemplate`.

```
public class SimpleService implements Service {

  private final TransactionTemplate transactionTemplate;

  public SimpleService(PlatformTransactionManager transactionManager) {
    Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
    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...
  }
}
```

Find below an example of defining 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 choose to share a single instance of a `TransactionTemplate`, if a class needed to use a `TransactionTemplate` with different settings (for example, a different isolation level), then two distinct `TransactionTemplate` instances would need to be created and used.

9.6.2. Using the `PlatformTransactionManager`

You can also use the `org.springframework.transaction.PlatformTransactionManager` directly to manage your transaction. Simply pass the implementation of the `PlatformTransactionManager` you're using to your bean via a bean reference. Then, using the `TransactionDefinition` and `TransactionStatus` objects you can initiate transactions, rollback and commit.

```
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);
```

9.7. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the `TransactionTemplate` *may* be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

9.8. Application server-specific integration

Spring's transaction abstraction generally is application server agnostic. Additionally, Spring's `JtaTransactionManager` class, which can optionally perform a JNDI lookup for the JTA `UserTransaction` and `TransactionManager` objects, autodetects the location for the latter object, which varies by application server. Having access to the JTA `TransactionManager` allows for enhanced transaction semantics, in particular supporting transaction suspension. Please see the `JtaTransactionManager` Javadocs for details.

Spring's `JtaTransactionManager` is the standard choice when running on J2EE application servers, known to work on all common servers. Its advanced functionality such as transaction suspension is known to work on many servers as well - including GlassFish, JBoss, Geronimo and Oracle OC4J - without any special configuration required. However, for fully supported transaction suspension and further advanced integration, Spring ships special adapters for IBM WebSphere and BEA WebLogic and also for Oracle OC4J. We'll discuss these adapters in the following sections.

For standard scenarios, including WebLogic, WebSphere and OC4J, consider using the convenient '`<tx:jta-transaction-manager/>`' configuration element. This will automatically detect the underlying server and choose the best transaction manager available for the platform. This means that you won't have to configure server-specific adapter classes (as discussed in the following sections) explicitly; they will rather be chosen automatically, with the standard `JtaTransactionManager` as default fallback.

9.8.1. IBM WebSphere

On WebSphere 6.0 and above, the recommended Spring JTA transaction manager to use is `WebSphereUowTransactionManager`. This special adapter leverages IBM's `UOWManager` API which is available in WebSphere Application Server 6.0.2.19 or above and 6.1.0.9 or above. With this adapter, Spring-driven transaction suspension (suspend/resume as initiated by `PROPAGATION_REQUIRES_NEW`) is officially supported by IBM!

In a WebSphere 5.1 environment, you may wish to use Spring's `WebSphereTransactionManagerFactoryBean` class. This is a factory bean which retrieves the JTA `TransactionManager` in a WebSphere environment, which is done via WebSphere's `static` access methods. Once the JTA `TransactionManager` instance has been obtained via this factory bean, Spring's `JtaTransactionManager` may be configured with a reference to it, for enhanced transaction semantics over the use of only the JTA `UserTransaction` object. Please see the Javadocs for full details.

Note that `WebSphereTransactionManagerFactoryBean` usage is known to work on WAS 5.1 and 6.0 but is not officially supported by IBM. Prefer `WebSphereUowTransactionManager` when running on WAS 6.0 or higher (see above).

9.8.2. BEA WebLogic

On WebLogic 8.1 or above, you will generally prefer to use the `WebLogicJtaTransactionManager` instead of the stock `JtaTransactionManager` class. This special WebLogic-specific subclass of the normal `JtaTransactionManager` supports the full power of Spring's transaction definitions in a WebLogic-managed transaction environment, beyond standard JTA semantics: Features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.

9.8.3. Oracle OC4J

Spring ships a special adapter class for OC4J 10.1.3 or above: `OC4JJtaTransactionManager`. This is analogous to the `WebLogicJtaTransactionManager` class discussed in the previous section, providing similar value-adds on OC4J: transaction names and per-transaction isolation levels.

Note that the full JTA functionality, including transaction suspension, works fine with Spring's `JtaTransactionManager` on OC4J as well. The special `OC4JJtaTransactionManager` adapter simply provides value-adds beyond standard JTA.

9.9. Solutions to common problems

9.9.1. Use of the wrong transaction manager for a specific `DataSource`

You should take care to use the *correct* `PlatformTransactionManager` implementation for their requirements. Used properly, the Spring Framework merely provides a straightforward and portable abstraction. If you are using global transactions, you *must* use the `org.springframework.transaction.jta.JtaTransactionManager` class (or an application server-specific subclass of it) for all your transactional operations. Otherwise the transaction infrastructure will attempt to perform local transactions on resources such as container `DataSource` instances. Such local transactions do not make sense, and a good application server will treat them as errors.

9.10. Further Resources

Find below links to further resources about the Spring Framework's transaction support.

- Java Transaction Design Strategies is a book available from InfoQ that provides a well-paced introduction to transactions in Java. It also includes side-by-side examples of how to configure and use transactions using both the Spring Framework and EJB3.