12. Transaction Management

12.1 Introduction to Spring Framework transaction management

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

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).
- Support for declarative transaction management.
- Simpler API for programmatic transaction management than complex transaction APIs such as JTA.
- Excellent integration with Spring's data access abstractions.

The following sections describe the Spring Framework's transaction value-adds and technologies. (The chapter also includes discussions of best practices, application server integration, and solutions to common problems.)

- Advantages of the Spring Framework's transaction support model describes *why* you would use the Spring Framework's transaction abstraction instead of EJB Container-Managed Transactions (CMT) or choosing to drive local transactions through a proprietary API such as Hibernate.
- Understanding the Spring Framework transaction abstraction outlines the core classes and describes how to configure and obtain `DataSource` instances from a variety of sources.
- Synchronizing resources with transactions describes how the application code ensures that resources are created, reused, and cleaned up properly.
- Declarative transaction management describes support for declarative transaction management.

- Programmatic transaction management covers support for programmatic (that is, explicitly coded) transaction management.

12.2 Advantages of the Spring Framework's transaction support model

Traditionally, Java EE developers have had two choices for transaction management: *global* or *local* transactions, both of which have profound limitations. Global and local transaction management is reviewed in the next two sections, followed by a discussion of how the Spring Framework's transaction management support addresses the limitations of the global and local transaction models.

12.2.1 Global transactions

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which 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 you *also* need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse 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 but not all of the need 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 only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

12.2.2 Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. 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. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

12.2.3 Spring Framework's consistent programming model

Spring resolves the disadvantages of global and local transactions. 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. Most users prefer declarative transaction management, which 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 do not depend on the Spring Framework transaction API, or any other transaction API.

Do you need an application server for transaction management?

The Spring Framework's transaction management support changes traditional rules as to when an enterprise Java application requires an application server.

In particular, you do not need an application server simply for declarative transactions through EJBs. In fact, even if your application server has powerful JTA capabilities, you may decide that the Spring Framework's declarative transactions offer more power and a more productive programming model than EJB CMT.

Typically you need an application server's JTA capability only if your application needs to handle transactions across multiple resources, which is not a requirement for many applications. Many high-end applications use a single, highly scalable database (such as Oracle RAC) instead. Standalone transaction managers such as Atomikos Transactions and JOTM are other options. Of course, you may need other application server capabilities such as Java Message Service (JMS) and J2EE Connector Architecture (JCA).

The Spring Framework *gives you the choice of when to scale your application to a fully loaded application server*. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions such as those on JDBC connections, and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file, rather than your code, need to change.

12.3 Understanding the Spring Framework transaction abstraction

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:

```
public interface PlatformTransactionManager {

  TransactionStatus getTransaction(TransactionDefinition definition)
    throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;

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

This is primarily a service provider interface (SPI), although it can be used programmatically from your application code. Because `PlatformTransactionManager` is an *interface*, it can be easily mocked or stubbed as necessary. It is not tied to a lookup strategy such as JNDI. `PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework IoC container. This benefit alone makes Spring Framework transactions a worthwhile abstraction even when you work 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 transaction, or can represent an existing transaction if a matching transaction exists in the current call stack. The implication in this latter case is that, as with Java EE transaction contexts, a `TransactionStatus` is associated with a **thread** of execution.

The `TransactionDefinition` interface specifies:

- **Isolation**: The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Propagation**: Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. *Spring offers all of the transaction propagation options familiar from EJB CMT*. To read about the semantics of transaction propagation in Spring, see Section 12.5.7, “Transaction propagation”.
- **Timeout**: How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.
- **Read-only status**: A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

These settings reflect standard transactional concepts. If necessary, refer to resources that discuss transaction isolation levels and other core transaction concepts. Understanding these concepts is essential to using the Spring Framework or any 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 extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();
```

```
    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `PlatformTransactionManager` implementation is absolutely essential. You typically define this implementation through dependency injection.

`PlatformTransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on. The following examples show how you can define a local `PlatformTransactionManager` implementation. (This example works with plain JDBC.)

You define a JDBC `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 then have a reference to the `DataSource` definition. It will look like this:

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

If you use JTA in a Java EE container then you use a container `DataSource`, obtained through JNDI, in conjunction with Spring's `JtaTransactionManager`. This is what the JTA and JNDI lookup version would look like:

```
<?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.xsd
     http://www.springframework.org/schema/jee
     http://www.springframework.org/schema/jee/spring-jee.xsd">

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

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

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

</beans>
```

The `JtaTransactionManager` does not need to know about the `DataSource`, or any other specific resources, because it uses the container's global transaction management infrastructure.

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 E, *XML Schema-based configuration*, and for more information on the `<jee/>` tags see the section entitled Section E.2.3, “The `jee` schema”.

You can also use Hibernate local transactions easily, as shown in the following examples. In this case, you need to define a Hibernate `LocalSessionFactoryBean`, which your application code will use to obtain Hibernate `Session` instances.

The `DataSource` bean definition will be similar to the local JDBC example shown previously and thus is not shown in the following example.

If the `DataSource`, used by any non-JTA transaction manager, is looked up via JNDI and managed by a Java EE container, then it should be non-transactional because the Spring Framework, rather than the Java EE container, will manage the 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>
```

If you are using Hibernate and Java EE container-managed JTA transactions, then you should simply use the same `JtaTransactionManager` as in the previous JTA example for JDBC.

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

If you use JTA , then your transaction manager definition will look the same regardless of what data access technology you use, be it JDBC, Hibernate JPA or any other supported technology. This is due to the fact that JTA transactions are global transactions, which can enlist any transactional resource.

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

12.4 Synchronizing resources with transactions

It should now be clear how you create different transaction managers, and how they are linked to related resources that need to be synchronized to transactions (for example `DataSourceTransactionManager` to a JDBC `DataSource`, `HibernateTransactionManager` to a Hibernate `SessionFactory`, and so forth). This section describes how the application code, directly or indirectly using a persistence API such as JDBC, Hibernate, or JDO, ensures that these resources are created, reused, and cleaned up properly. The section also discusses how transaction synchronization is triggered (optionally) through the relevant `PlatformTransactionManager`.

12.4.1 High-level synchronization approach

The preferred approach is to use Spring's highest level template based persistence integration APIs or to use native ORM APIs with transaction- aware factory beans or proxies for managing the native resource factories. These transaction-aware solutions internally handle resource creation and reuse, cleanup, optional transaction synchronization of the resources, and exception mapping. Thus user data access code does not have to address these tasks, but can be focused purely on non-boilerplate persistence logic. Generally, you use the native ORM API or take a *template* approach for JDBC access by using the `JdbcTemplate`. These solutions are detailed in subsequent chapters of this reference documentation.

12.4.2 Low-level synchronization approach

Classes such as `DataSourceUtils` (for JDBC), `EntityManagerFactoryUtils` (for JPA), `SessionFactoryUtils` (for Hibernate), `PersistenceManagerFactoryUtils` (for JDO), and so on exist at a lower level. When you want the application code to deal directly with the resource types of the native persistence APIs, you use these classes to ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions that occur 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 instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

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

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

This approach also works 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 have used Spring's JDBC support, JPA support or Hibernate support, you will generally prefer not to use `DataSourceUtils` or the other helper classes, because you will be much happier working through 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 occurs behind the scenes and you won't need to write any special code.

12.4.3 `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 Java EE server.

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

12.5 Declarative transaction management

Most Spring Framework users choose declarative transaction management. This option has 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 aspect-oriented programming (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.

The Spring Framework's declarative transaction management is similar to EJB CMT in that you can 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 between the two types of transaction management are:

- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA, Hibernate or JDO by simply adjusting the configuration files.
- You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*, a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.

- The Spring Framework enables you to customize transactional behavior, by using AOP. For example, you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you cannot influence the container's transaction management except with `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.

> **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 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.

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

Although EJB container default behavior automatically rolls 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 behavior.

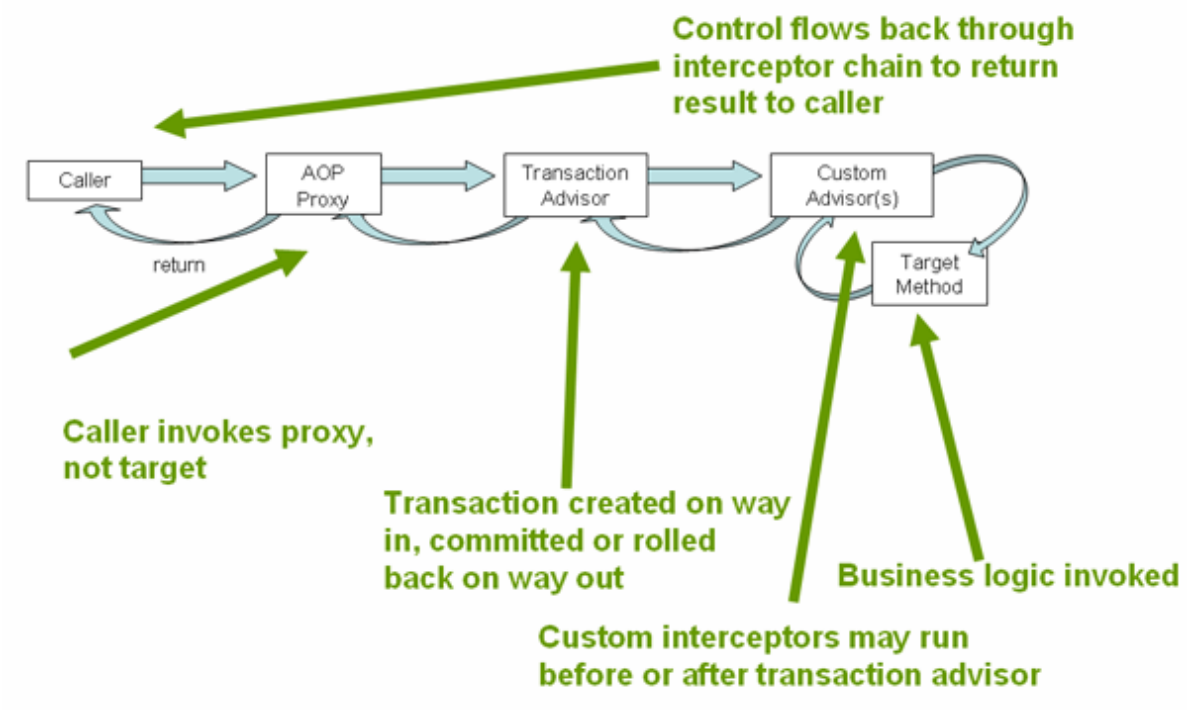
12.5.1 Understanding the Spring Framework's declarative transaction implementation

It is not sufficient to tell you simply to annotate your classes with the `@Transactional` annotation, add `@EnableTransactionManagement` to your configuration, and then expect you to understand how it all works. This section explains the inner workings of the Spring Framework's declarative transaction infrastructure 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*.

Spring AOP is covered in Chapter 9, *Aspect Oriented Programming with Spring*.

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



12.5.2 Example of declarative transaction implementation

Consider the following interface, and its attendant implementation. This example uses `Foo` and `Bar` classes as placeholders so that you can concentrate on the transaction usage without focusing on a particular domain model. 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 allows you to see transactions created and then rolled back in response to the `UnsupportedOperationException` instance.

```
// 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();
  }

}
```

Assume that the first two methods of the `FooService` interface`, getFoo(String)` and `getFoo(String, String),` must execute in the context of a transaction with read-only semantics, and that the other methods`,insertFoo(Foo)` and `updateFoo(Foo),` must execute in the context of a transaction with read-write semantics. The following configuration is 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.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.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>
```

Examine the preceding configuration. You want to make a service object, the `fooService` bean, transactional. The transaction semantics 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 *drive* the transactions, in this case, the `txManager` bean.

You can 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 must use the `transaction-manager` attribute explicitly, as in the preceding example.

The `<aop:config/>` definition ensures that the transactional advice defined by the `txAdvice` bean executes at the appropriate points in the program. First you define a pointcut that matches the execution of any operation defined in the `FooService` interface (`fooServiceOperation`). Then you 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 Chapter 9, *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>
```

In this example it is assumed that all your service interfaces are defined in the `x.y.service` package; see Chapter 9, 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 will be used to create 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 is started, suspended, marked as read-only, and so on, 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 preceding program will resemble the following. (The Log4J output and the stack trace from the UnsupportedOperationException thrown by the insertFoo(..) method of the DefaultFooService class have been truncated for 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)
```

12.5.3 Rolling back a declarative transaction

The previous section outlined the basics of how to specify transactional settings for classes, typically service layer classes, declaratively in your application. 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 make a determination whether to mark the transaction for rollback.

In its default configuration, the Spring Framework's transaction infrastructure code *only* marks a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Error`s will also - by default - result in a rollback). Checked exceptions that are thrown from a transactional method do *not* result in rollback in the default configuration.

You can configure exactly which `Exception` types mark a transaction for rollback, including checked exceptions. The following XML snippet demonstrates how you 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>
```

You can also specify 'no rollback rules', if you do *not* want a transaction rolled back when an exception is thrown. The following example tells 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 catches an exception and is consults configured rollback rules to determine whether 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` results in a rollback of the attendant transaction.

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

```
  </tx:attributes>
</tx:advice>
```

You can also indicate a required rollback *programmatically*. Although very simple, this process is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure:

```
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 clean POJO-based architecture.

12.5.4 Configuring different transactional semantics for different beans

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

As a point of comparison, first 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.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.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>
```

The following example shows how to configure 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.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.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>
```

12.5.5 `<tx:advice/>` settings

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

- Propagation setting is `REQUIRED.`
- Isolation level is `DEFAULT.`
- Transaction is read/write.
- Transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported.
- Any `RuntimeException` triggers rollback, and any checked `Exception` does not.

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

Table 12.1. `<tx:method/>` settings

Attribute	Required?	Default	Description
`name`	Yes		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	Transaction propagation behavior.
`isolation`	No	DEFAULT	Transaction isolation level.
`timeout`	No	-1	Transaction timeout value (in seconds).
`read-only`	No	false	Is this transaction read-only?
`rollback-for`	No		`Exception(s)` that trigger rollback; comma-delimited. For example, `com.foo.MyBusinessException,ServletException.`
`no-rollback-for`	No		`Exception(s)` that do *not* trigger rollback; comma-delimited. For example, `com.foo.MyBusinessException,ServletException.`

12.5.6 Using `@Transactional`

In addition to the XML-based declarative approach to transaction configuration, you can use an annotation-based approach. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code. There is not much danger of undue coupling, because 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, which is explained in the text that follows. 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:

```
<!-- 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.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.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>
```

You can 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 use the `transaction-manager` attribute explicitly, as in the preceding example.

The `@EnableTransactionManagement` annotation provides equivalent support if you are using Java based configuration. Simply add the annotation to a `@Configuration` class. See Javadoc for full details.

Method visibility and `@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 (see below) if you need to annotate non-public methods.

You can place the `@Transactional` annotation before an interface definition, a method on an interface, a class definition, or a *public* method on a class. However, the mere presence of the `@Transactional` annotation is not enough to activate the transactional behavior. The `@Transactional` annotation is simply metadata that can be consumed by some runtime infrastructure that is `@Transactional`-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the preceding example, the `<tx:annotation-driven/>` element *switches on* the transactional behavior.

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`.

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

Table 12.2. Annotation driven transaction settings

XML Attribute	Annotation Attribute	Default	Description

XML Attribute	Annotation Attribute	Default	Description
`transaction-manager`	N/A (See `TransactionManagementConfigurer` Javadoc)	transactionManager	Name of transaction manager to use. Only required if the name of the transaction manager is not `transactionManager`, as in the example above.
`mode`	`mode`	proxy	The default mode "proxy" processes 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" instead weaves the affected classes with Spring's AspectJ transaction aspect, modifying the target class byte code to apply to any kind of

XML Attribute	Annotation Attribute	Default	Description
			method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See the section called “Spring configuration” for details on how to set up load-time weaving.)

XML Attribute	Annotation Attribute	Default	Description
`proxy-target-class`	`proxyTargetClass`	false	Applies to proxy mode only. Controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If the `proxy-target-class` attribute is set to `true`, then class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, then standard JDK interface-based proxies are created. (See Section 9.6, “Proxying mechanisms” for a detailed examination of the different proxy types.)

XML Attribute	Annotation Attribute	Default	Description
`order`	`order`	Ordered.LOWEST_PRECEDENCE	Defines the order of the transaction advice that is applied to beans annotated with `@Transactional`. (For more information about the rules related to ordering of AOP advice, see the section called "Advice ordering".) No specified ordering means that the AOP subsystem determines the order of the advice.

The `proxy-target-class` attribute controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If `proxy-target-class` is set to `true`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, standard JDK interface-based proxies are created. (See Section 9.6, "Proxying mechanisms" for a discussion of the different proxy types.)

`@EnableTransactionManagement` and `<tx:annotation-driven/>` only looks for `@Transactional` on beans in the same application context they are defined in. This means that, if you put annotation driven configuration in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Transactional` beans in your controllers, and not your services. See Section 17.2, "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
  }
}
```

`@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 as follows:

- Propagation setting is `PROPAGATION_REQUIRED.`
- Isolation level is `ISOLATION_DEFAULT.`
- Transaction is read/write.
- Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
- Any `RuntimeException` triggers rollback, and any checked `Exception` does not.

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

Table 12.3. `@Transactional` properties

Property	Type	Description
`value`	String	Optional qualifier specifying the transaction manager to be used.
`propagation`	enum: `Propagation`	Optional propagation setting.
`isolation`	enum: `Isolation`	Optional isolation level.
`readOnly`	boolean	Read/write vs. read-only transaction
`timeout`	int (in seconds granularity)	Transaction timeout.
`rollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception classes that **must** cause rollback.
`rollbackForClassName`	Array of class names. Classes must be derived from `Throwable.`	Optional array of names of exception classes that **must** cause rollback.
`noRollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception classes that **must not** cause rollback.
`noRollbackForClassName`	Array of `String` class names, which must be derived from `Throwable.`	Optional array of names of exception classes that **must not** cause rollback.

Currently you cannot 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`.

Multiple Transaction Managers with `@Transactional`

Most Spring applications only need a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. The value attribute of the `@Transactional` annotation can be used to optionally specify the identity of the `PlatformTransactionManager` to be used. This can either be the bean name or the qualifier value of the transaction manager bean. For example, using the qualifier notation, the following Java code

```
public class TransactionalService {

  @Transactional("order")
  public void setSomething(String name) { ... }

  @Transactional("account")
  public void doSomething() { ... }
}
```

could be combined with the following transaction manager bean declarations in the application context.

```
<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.DataSourceTransactionManager">
  ...
  <qualifier value="order"/>
</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.DataSourceTransactionManager">
  ...
  <qualifier value="account"/>
```

```
</bean>
```

In this case, the two methods on `TransactionalService` will run under separate transaction managers, differentiated by the "order" and "account" qualifiers. The default `<tx:annotation-driven>` target bean name `transactionManager` will still be used if no specifically qualified PlatformTransactionManager bean is found.

Custom shortcut annotations

If you find you are repeatedly using the same attributes with `@Transactional` on many different methods, then Spring's meta-annotation support allows you to define custom shortcut annotations for your specific use cases. For example, defining the following annotations

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("order")
public @interface OrderTx {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("account")
public @interface AccountTx {
}
```

allows us to write the example from the previous section as

```
public class TransactionalService {

  @OrderTx
  public void setSomething(String name) { ... }

  @AccountTx
```

```
    public void doSomething() { ... }
}
```

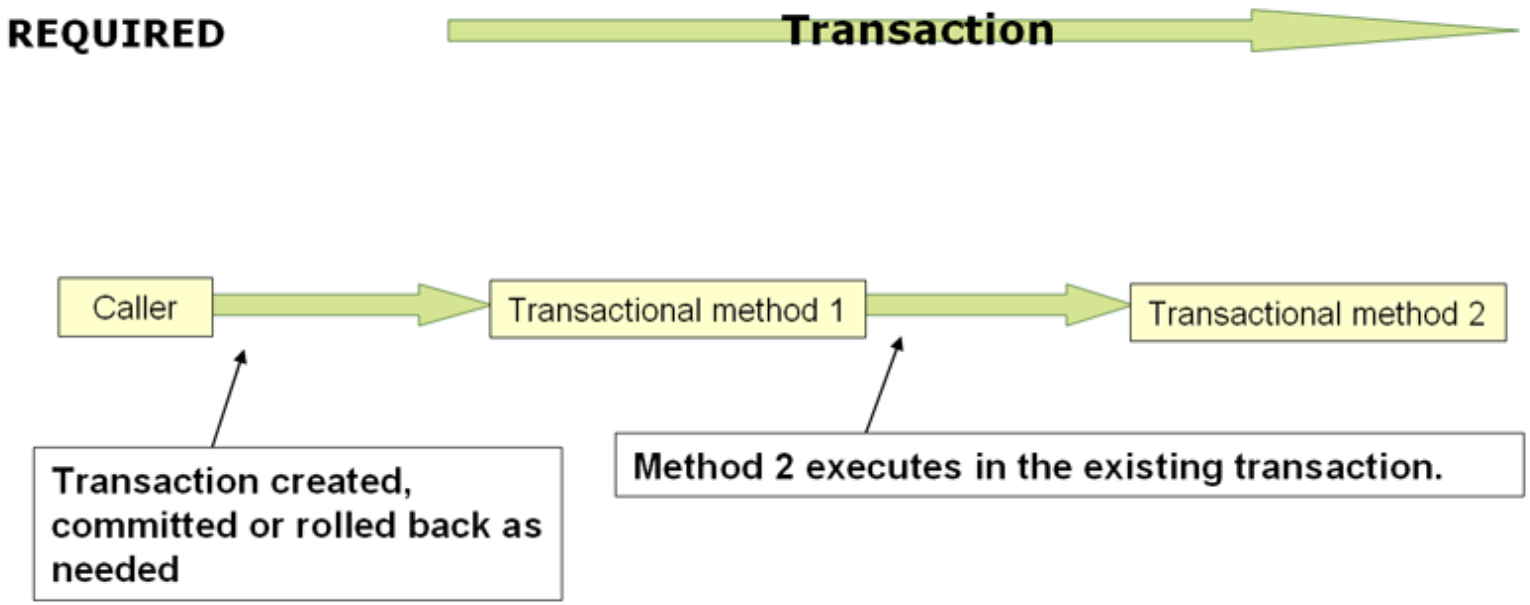
Here we have used the syntax to define the transaction manager qualifier, but could also have included propagation behavior, rollback rules, timeouts etc.

12.5.7 Transaction propagation

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

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

Required

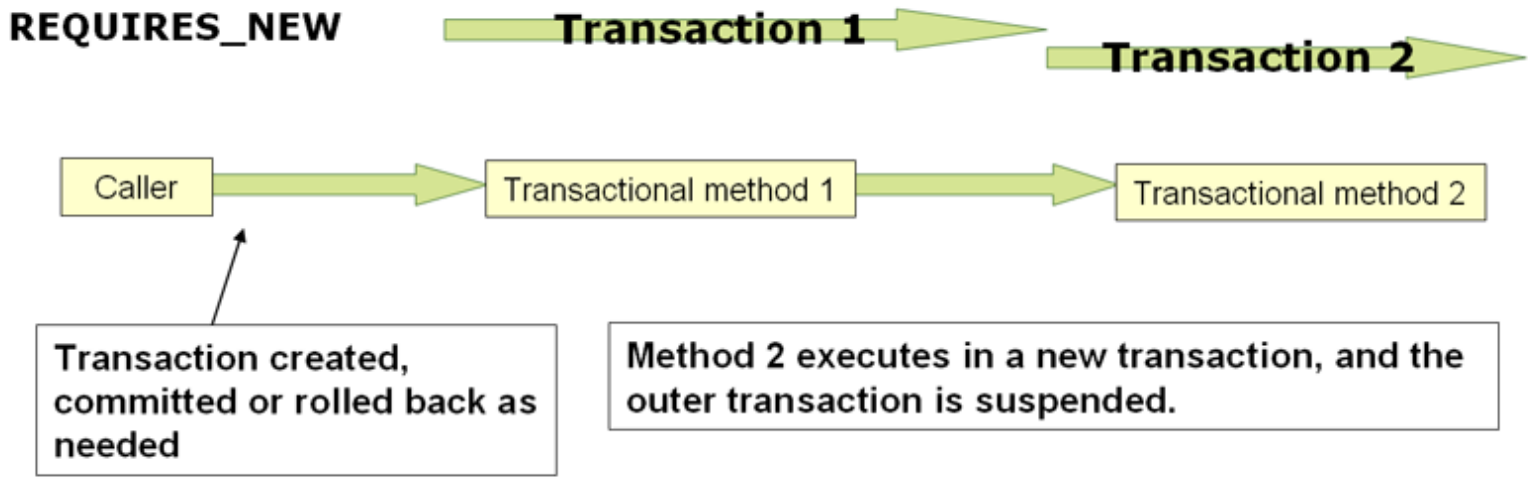


PROPAGATION_REQUIRED

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

However, in the case where an inner transaction scope sets the rollback-only marker, the outer transaction has not decided on the rollback itself, and so the rollback (silently triggered by the inner transaction scope) is unexpected. A corresponding `UnexpectedRollbackException` is 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 (of which the outer caller is not aware) silently marks a transaction as rollback-only, the outer caller still calls commit. The outer caller needs to receive an `UnexpectedRollbackException` to indicate clearly that a rollback was performed instead.

RequiresNew



PROPAGATION_REQUIRES_NEW

`PROPAGATION_REQUIRES_NEW`, in contrast to PROPAGATION_REQUIRED, uses a *completely* independent transaction for each affected transaction scope. In that case, the underlying physical transactions are different and hence can commit or roll back

independently, with an outer transaction not affected by an inner transaction's rollback status.

Nested

`PROPAGATION_NESTED` 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 setting is typically mapped onto JDBC savepoints, so will only work with JDBC resource transactions. See Spring's `DataSourceTransactionManager`.

12.5.8 Advising transactional operations

Suppose you want to execute *both* transactional *and* some basic profiling advice. How do you effect this in the context of `<tx:annotation-driven/>`?

When you invoke the `updateFoo(Foo)` method, you want to see the following actions:

1. Configured profiling aspect starts up.
2. Transactional advice executes.
3. Method on the advised object executes.
4. Transaction commits.
5. Profiling aspect reports exact duration of the whole transactional method invocation.

This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). See Chapter 9, *Aspect Oriented Programming with Spring* for detailed coverage of the following AOP configuration and AOP in general.

Here is the code for a simple profiling aspect discussed above. The ordering of advice is controlled through the `Ordered` interface. For full details on advice ordering, see the section called “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
<?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.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.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 is a `fooService` bean that has profiling and transactional aspects applied to it *in the desired order*. You configure any number of additional aspects in similar fashion.

The following example effects the same setup as above, but uses 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.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.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 you want the profiling advice to execute *after* the transactional advice on the way in, and *before* the transactional advice on the way out, then you simply swap the value of the profiling aspect bean's `order` property so that it is higher than the transactional advice's order value.

You configure additional aspects in similar fashion.

12.5.9 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 do so, you first annotate your classes (and optionally your classes' methods) with the `@Transactional` annotation, and then you 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 can 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 `aspectj` as described in Section 12.5.6, “Using `@Transactional`”. Because we're focusing here on applications running outside of a Spring container, we'll show you how to do it programmatically.

Prior to continuing, you may want to read Section 12.5.6, “Using `@Transactional`” and Chapter 9, *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 met
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

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 Section 9.8.4, "Load-time weaving with AspectJ in the Spring Framework" for a discussion of load-time weaving with AspectJ.

12.6 Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

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

The Spring team generally recommends the `TransactionTemplate` for programmatic transaction management. The second approach is similar to using the JTA `UserTransaction` API, although exception handling is less cumbersome.

12.6.1 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.

As you will 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 the following. You, as an application developer, write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that contains the code that you need to execute in the context of a transaction. You 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 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) {
    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...
  }
}
```

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.

12.6.2 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 are using to your bean through a bean reference. Then, using the `TransactionDefinition` and `TransactionStatus` objects you can initiate transactions, roll back, 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);
```

12.7 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.

12.8 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. See the `JtaTransactionManager` Javadocs for details.

Spring's `JtaTransactionManager` is the standard choice to run on Java EE application servers, and is known to work on all common servers. Advanced functionality such as transaction suspension works 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, BEA WebLogic Server, and Oracle OC4J. These adapters are discussed in the following sections.

For standard scenarios, including WebLogic Server, WebSphere and OC4J, consider using the convenient `<tx:jta-transaction-manager/>` *configuration element.* When configured, this element automatically detects the underlying server and chooses 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; rather, they are chosen automatically, with the standard `JtaTransactionManager` as default fallback.

12.8.1 IBM WebSphere

On WebSphere 6.1.0.9 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 and later and 6.1.0.9 and later. With this adapter, Spring-driven transaction suspension (suspend/resume as initiated by `PROPAGATION_REQUIRES_NEW`) is officially supported by IBM!

12.8.2 BEA WebLogic Server

On WebLogic Server 9.0 or above, you typically would 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.

12.8.3 Oracle OC4J

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

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.

12.9 Solutions to common problems

12.9.1 Use of the wrong transaction manager for a specific `DataSource`

Use the *correct* `PlatformTransactionManager` implementation based on your choice of transactional technologies and 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 attempts to perform local transactions on resources such as container `DataSource` instances. Such local transactions do not make sense, and a good application server treats them as errors.

12.10 Further Resources

For more information about the Spring Framework's transaction support:

- Distributed transactions in Spring, with and without XA is a JavaWorld presentation in which SpringSource's David Syer guides you through seven patterns for distributed transactions in Spring applications, three of them with XA and four without.
- 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 with both the Spring Framework and EJB3.

Prev Up Next

Part IV. Data Access Home 13. DAO support