covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

Chapter 6. Managing transactions.. 1

Section 6.1. Understanding transactions.. 2

Section 6.2. Choosing a transaction manager.. 5

Section 6.3. Programming transactions in Spring.. 8

Section 6.4. Declaring transactions.. 10

Section 6.5. Summary.. 18

6 Managing transactions

This chapter covers

- Integrating with transaction managers
- Managing transactions programmatically
- Using declarative transactions
- Describing transactions using annotations

Take a moment to recall your younger days. If you were like many children, you spent more than a few carefree moments on the playground swinging on the swings, traversing the monkey bars, getting dizzy while spinning on the merry-go-round, and going up and down on the teeter-totter.

The problem with the teeter-totter is that it's practically impossible to enjoy on your own. To truly enjoy a teeter-totter, you need another person: you and a friend both have to agree to play on the teeter-totter. This agreement is an all-or-nothing proposition. Either both of you will teeter-totter or you won't. If either of you fails to take your respective seat on each end of the teeter-totter, then there will be no teeter-tottering—just a sad kid sitting motionless on the end of a slanted board.[1]

[1] Since the first edition of this book, I've confirmed that this qualifies as the most uses of the word *teeter-totter* in a technical book. That's a bit of trivia to challenge your friends with.

Chapter 6. Managing transactions. Spring in Action, Third Edition, ISBN: 9781935182351

In software, all-or-nothing operations are called *transactions*. Transactions allow you to group several operations into a single unit of work that either fully happens or fully doesn't happen. If everything goes well, then the transaction is a success. But if anything goes wrong, the slate is wiped clean and it's as if nothing ever happened.

Probably the most common example of a real-world transaction is a money transfer. Imagine that you were to transfer $100 from your savings account to your checking account. The transfer involves two operations: $100 is deducted from the savings account and $100 is added to the checking account. The money transfer must be performed completely or not at all. If the deduction from the savings account works but the deposit into the checking account fails, you'll be out $100 (good for the bank, bad for you). On the other hand, if the deduction fails but the deposit succeeds, you'll be ahead $100 (good for you, bad for the bank). It's best for both parties involved if the entire transfer is rolled back if either operation fails.

In the previous chapter, we examined Spring's data access support and saw several ways to read from and write data to the database. When writing to a database, we must ensure that the integrity of the data is maintained by performing the updates within a transaction. Spring has rich support for transaction management, both programmatic and declarative. In this chapter, we'll see how to apply transactions to your application code so that when things go right, they're made permanent. And when things go wrong… nobody needs to know. (Almost nobody. You may still want to log the problem for the sake of auditing.)

6.1 Understanding transactions

To illustrate transactions, consider the purchase of a movie ticket. Purchasing a ticket typically involves the following actions:

- The number of available seats will be examined to verify that enough seats are available for your purchase.
- The number of available seats is decremented by one for each ticket purchased.
- You provide payment for the ticket.
- The ticket is issued to you.

If everything goes well, you'll be enjoying a blockbuster movie and the theater will be a few dollars richer. But what if something goes wrong? For instance, what if you paid with a credit card that had reached its limit? Certainly, you wouldn't receive a ticket and the theater wouldn't receive payment. If the number of seats isn't reset to its value before the purchase, the movie may artificially run out of seats (and thus lose sales). Or consider what would happen if everything else works fine but the ticket issue fails. You'd be short a few dollars and be stuck at home watching reruns on cable TV.

To ensure that neither you nor the theater loses out, these actions should be wrapped in a transaction. As a transaction, they're all treated as a single action, guaranteeing that either they'll all fully succeed or all be rolled back as if these steps never happened. Figure 6.1 illustrates how this transaction plays out.

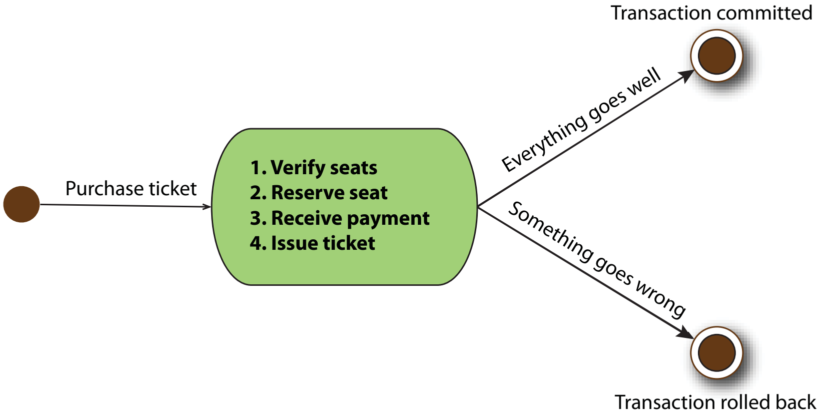

Figure 6.1 The steps involved when purchasing a movie ticket should be all or nothing. If every step is successful, then the entire transaction is successful. Otherwise, the steps should be rolled back—as if they never happened.

Transactions play an important role in software, ensuring that data and resources are never left in an inconsistent state. Without them, there's potential for data to be corrupted or inconsistent with the business rules of the application.

Before we get too carried away with Spring's transaction support, it's important to understand the key ingredients of a transaction. Let's take a quick look at the four factors that guide transactions and how they work.

6.1.1 *Explaining transactions in only four words*

In the grand tradition of software development, an acronym has been created to describe transactions: *ACID*. In short, ACID stands for

- *Atomic*—Transactions are made up of one or more activities bundled together as a single unit of work. Atomicity ensures that all the operations in the transaction happen or that none of them happen. If all the activities succeed, the transaction is a success. If any of the activities fails, the entire transaction fails and is rolled back.
- *Consistent*—Once a transaction ends (whether successful or not), the system is left in a state consistent with the business that it models. The data shouldn't be corrupted with respect to reality.
- *Isolated*—Transactions should allow multiple users to work with the same data, without each user's work getting tangled up with the others. Therefore, transactions should be isolated from each other, preventing concurrent reads and writes to the same data from occurring. (Note that isolation typically involves locking rows and/or tables in a database.)
- *Durable*—Once the transaction has completed, the results of the transaction should be made permanent so that they'll survive any sort of system crash. This typically involves storing the results in a database or some other form of persistent storage.

In the movie ticket example, a transaction could ensure atomicity by undoing the result of all the steps if any step fails. Atomicity supports consistency by ensuring that the system's data is never left in an inconsistent, partially done state. Isolation also supports consistency by preventing another concurrent transaction from stealing seats out from under you while you're still in the process of purchasing them.

Finally, the effects are durable because they'll have been committed to some persistent storage. In the event of a system crash or other catastrophic event, you shouldn't have to worry about results of the transaction being lost.

For a more detailed explanation of transactions, I suggest that you read Martin Fowler's *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002). Specifically, chapter 5 discusses concurrency and transactions.

Now that you know the makings of a transaction, let's see the transaction capabilities available to a Spring application.

6.1.2 Understanding Spring's transaction management support

Spring, like EJB, provides support for both programmatic and declarative transaction management. But Spring's transaction management capabilities exceed those of EJB.

Spring's support for programmatic transaction management differs greatly from that of EJB. Unlike EJB, which is coupled with a Java Transaction API (JTA) implementation, Spring employs a callback mechanism that abstracts away the actual transaction implementation from the transactional code. In fact, Spring's transaction management support doesn't even require a JTA implementation. If your application uses only a single persistent resource, Spring can use the transactional support offered by the persistence mechanism. This includes JDBC, Hibernate, and the Java Persistence API (JPA). But if your application has transaction requirements that span multiple resources, Spring can support distributed (XA) transactions using a third-party JTA implementation. We'll discuss Spring's support for programmatic transactions in section 6.3.

Where programmatic transaction management affords you flexibility in precisely defining transaction boundaries in your code, declarative transactions (which are based on Spring AOP) help you decouple an operation from its transaction rules. Spring's support for declarative transactions is reminiscent of EJB's *container-managed transactions (CMTs)*. Both allow you to define transaction boundaries declaratively. But Spring's declarative transactions go beyond CMTs by allowing you to declare additional attributes such as isolation level and timeouts. We'll begin working with Spring's declarative transaction support in section 6.4.

Choosing between programmatic and declarative transaction management is largely a decision of fine-grained control versus convenience. When you program transactions into your code, you gain precise control over transaction boundaries, beginning and ending them precisely where you want. Typically, you won't require the fine-grained control offered by programmatic transactions and will choose to declare your transactions in the context definition file.

Regardless of whether you choose to program transactions into your beans or to declare them as aspects, you'll be using a Spring transaction manager to interface with a platform-specific transaction implementation. Let's see how Spring's transaction managers free you from dealing directly with platform-specific transaction implementations.

6.2 Choosing a transaction manager

Spring doesn't directly manage transactions. Instead, it comes with a selection of transaction managers that delegate responsibility for transaction management to a platform-specific transaction implementation provided by either JTA or the persistence mechanism. Spring's transaction managers are listed in table 6.1.

Each of these transaction managers acts as a facade to a platform-specific transaction implementation. (Figure 6.2 illustrates the relationship between transaction managers and the underlying platform implementations for a few of the transaction managers.) This makes it possible for you to work with a transaction in Spring with little regard to what the actual transaction implementation is.

Table 6.1 Spring has transaction managers for every occasion.

Transaction manager (`org.springframework.*`)	Use it when...
`jca.cci.connection.CciLocalTransactionManager`	Using Spring's support for Java EE Connector Architecture (JCA) and the Common Client Interface (CCI).
`jdbc.datasource.DataSourceTransactionManager`	Working with Spring's JDBC abstraction support. Also useful when using iBATIS for persistence.
`jms.connection.JmsTransactionManager`	Using JMS 1.1+.
`jms.connection.JmsTransactionManager102`	Using JMS 1.0.2.
`orm.hibernate3.HibernateTransactionManager`	Using Hibernate 3 for persistence.
`orm.jdo.JdoTransactionManager`	Using JDO for persistence.
`orm.jpa.JpaTransactionManager`	Using the Java Persistence API (JPA) for persistence.
`transaction.jta.JtaTransactionManager`	You need distributed transactions or when no other transaction manager fits the need.
`transaction.jta.OC4JJtaTransactionManager`	Using Oracle's OC4J JEE container.
`transaction.jta.WebLogicJtaTransactionManager`	You need distributed transactions and your application is running within WebLogic.
`transaction.jta.WebSphereUowTransactionManager`	You need transactions managed by a `UOWManager` in WebSphere.

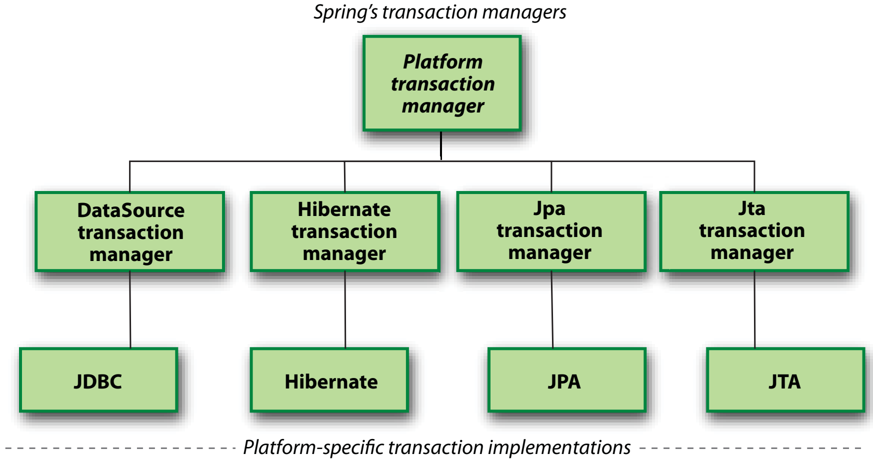
Spring's transaction managers
Platform transaction manager
DataSource transaction manager
Hibernate transaction manager
Jpa transaction manager
Jta transaction manager
JDBC
Hibernate
JPA
JTA
Platform-specific transaction implementations

Figure 6.2 Spring's transaction managers delegate transaction-management responsibility to platform-specific transaction implementations.

To use a transaction manager, you'll need to declare it in your application context. In this section, you'll learn how to configure a few of Spring's most commonly used transaction managers, starting with DataSourceTransactionManager, which provides transaction support for plain JDBC and iBATIS.

6.2.1 JDBC transactions

If you're using straight JDBC for your application's persistence, DataSourceTransactionManager will handle transactional boundaries for you. To use DataSourceTransactionManager, wire it into your application's context definition using the following XML:

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

Note that the dataSource property is set with a reference to a bean named dataSource. Presumably, the dataSource bean is a javax.sql.DataSource bean defined elsewhere in your context definition file.

Behind the scenes, DataSourceTransactionManager manages transactions by making calls on the java.sql.Connection object retrieved from the DataSource. For instance, a successful transaction is committed by calling the commit() method on the connection. Likewise, a failed transaction is rolled back by calling the rollback() method.

6.2.2 Hibernate transactions

If your application's persistence is handled by Hibernate then you'll want to use HibernateTransactionManager. For Hibernate 3, you'll need to add the following <bean> declaration to the Spring context definition:

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

The sessionFactory property should be wired with a Hibernate SessionFactory, here cleverly named sessionFactory. See the previous chapter for details on setting up a Hibernate session factory.

What if I'm using Hibernate 2?

If you're using the older Hibernate 2 for persistence, you won't be able to use the HibernateTransactionManager in Spring 3.0 or even Spring 2.5. Those versions of Spring don't include support for Hibernate 2. You'll have to go back to using Spring 2.0 if you insist on using an older version of Hibernate.

But, if you roll back to using an older version of Spring with your older version of Hibernate, you should realize that you'll be giving up a lot of Spring features that we'll be talking about in this book. So, rather than roll back to an older version of Spring, I recommend upgrading to Hibernate 3.

HibernateTransactionManager delegates responsibility for transaction management to an org.hibernate.Transaction object that it retrieves from the Hibernate session. When a transaction successfully completes, HibernateTransactionManager will call the commit() method on the Transaction object. Similarly, when a transaction fails, the rollback() method will be called on the Transaction object.

6.2.3 *Java Persistence API transactions*

Hibernate has been Java's de facto persistence standard for a few years, but now the Java Persistence API (JPA) has entered the scene as the true standard for Java persistence. If you're ready to move up to JPA then you'll want to use Spring's JpaTransactionManager to coordinate transactions. Here's how you might configure JpaTransactionManager in Spring:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

JpaTransactionManager only needs to be wired with a JPA entity manager factory (any implementation of javax.persistence.EntityManagerFactory). JpaTransactionManager will collaborate with the JPA EntityManager produced by the factory to conduct transactions.

In addition to applying transactions to JPA operations, JpaTransactionManager also supports transactions on simple JDBC operations on the same DataSource used by EntityManagerFactory. For this to work, JpaTransactionManager must also be wired with an implementation of JpaDialect. For example, suppose that you've configured EclipseLinkJpaDialect as follows:

```
<bean id="jpaDialect"
    class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect" />
```

Then you must wire the `jpaDialect` bean into the `JpaTransactionManager` like this:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
  <property name="jpaDialect" ref="jpaDialect" />
</bean>
```

It's important to note that the `JpaDialect` implementation must support mixed JPA/JDBC access for this to work. All of Spring's vendor-specific implementations of `JpaDialect` (`EclipseLinkJpaDialect`, `HibernateJpaDialect`, `OpenJpaDialect`, and `TopLinkJpaDialect`) provide support for mixing JPA with JDBC. `DefaultJpaDialect` doesn't.

6.2.4 Java transaction API transactions

If none of the aforementioned transaction managers meet your needs or if your transactions span multiple transaction sources (for example, two or more different databases), you'll need to use `JtaTransactionManager`:

```
<bean id="transactionManager" class="org.springframework.
      ➥transaction.jta.JtaTransactionManager">
  <property name="transactionManagerName"
      value="java:/TransactionManager" />
</bean>
```

`JtaTransactionManager` delegates transaction management responsibility to a JTA implementation. JTA specifies a standard API to coordinate transactions between an application and one or more data sources. The `transactionManagerName` property specifies a JTA transaction manager to be looked up via JNDI.

`JtaTransactionManager` works with `javax.transaction.UserTransaction` and `javax.transaction.TransactionManager` objects, delegating responsibility for transaction management to those objects. A successful transaction will be committed with a call to the `UserTransaction.commit()` method. Likewise, if the transaction fails, the `UserTransaction`'s `rollback()` method will be called.

By now, it should be clear which of Spring's transaction managers is a best fit for the Spitter application—insomuch as we've chosen a persistence mechanism. Now it's time to put that transaction manager to work. We'll start by using it to program transactions manually.

6.3 Programming transactions in Spring

There are two kinds of people: those who are control freaks and those who aren't. Control freaks like complete control over everything that happens and don't take anything for granted. If you're a developer and a control freak, you're probably the kind of person who prefers the command line and would rather write your own getter and setter methods than to delegate that work to an IDE.

Control freaks also like to know exactly what's going on in their code. When it comes to transactions, they want full control over where a transaction starts, where it commits, and where it ends. Declarative transactions aren't precise enough for them.

This isn't a bad thing, though. The control freaks are at least partially right. As you'll see later in this chapter, you're limited to declaring transaction boundaries at the method level. If you need more fine-grained control over transactional boundaries, programmatic transactions are the only way to go.

Consider the saveSpittle() method of SpitterServiceImpl as an example of a transactional method.

Listing 6.1 saveSpittle() saves a Spittle

```
public void saveSpittle(Spittle spittle) {
  spitterDao.saveSpittle(spittle);
}
```

Although this method appears rather simple, there may be more than meets the eye. As the Spittle is saved, the underlying persistence mechanism may have a lot to do. Even if it ends up being a simple matter of inserting a row into a database table, it's important to make sure that whatever happens takes place within the confines of a transaction. If it succeeds, the work should be committed. If it fails, then it should be rolled back.

One approach to adding transactions is to programmatically add transactional boundaries directly within the saveSpittle() method using Spring's TransactionTemplate. Like other template classes in Spring (such as JdbcTemplate, discussed in the previous chapter), TransactionTemplate utilizes a callback mechanism. Here's an updated saveSpittle() method to show how to add a transactional context using a TransactionTemplate.

Listing 6.2 Programmatically adding transactions to saveSpittle()

```
public void saveSpittle(final Spittle spittle) {
  txTemplate.execute(new TransactionCallback<Void>() {
    public Void doInTransaction(TransactionStatus txStatus) {
      try {
      spitterDao.saveSpittle(spittle);
      } catch (RuntimeException e) {
        txStatus.setRollbackOnly();
        throw e;
      }
      return null;
    }
  });
}
```

To use the TransactionTemplate, you start by implementing the TransactionCallback interface. Because TransactionCallback has only one method to implement, it's often easiest to implement it as an anonymous inner class, as shown in listing 6.2. As for the code that needs to be transactional, place it within the doInTransaction() method.

Calling the execute() method on the TransactionTemplate instance will execute the code contained within the TransactionCallback instance. If your code encounters a problem, calling setRollbackOnly() on the TransactionStatus object will roll back the transaction. Otherwise, if the doInTransaction() method returns successfully, the transaction will be committed.

Where does the TransactionTemplate instance come from? Good question. It should be injected into SpitterServiceImpl, as follows:

```
<bean id="spitterService"
      class="com.habuma.spitter.service.SpitterServiceImpl">
  ...
  <property name="transactionTemplate ">
    <bean class="org.springframework.transaction.support.
            TransactionTemplate">
      <property name="transactionManager"
          ref="transactionManager" />
    </bean>
  </property>
</bean>
```

Note that the TransactionTemplate is injected with a transactionManager. Under the hood, TransactionTemplate uses an implementation of PlatformTransactionManager to handle the platform-specific details of transaction management. Here we've wired in a reference to a bean named transactionManager, which could be any of the transaction managers listed in table 6.1.

Programmatic transactions are good when you want complete control over transactional boundaries. But, as you can see from the code in listing 6.1, they're intrusive. You had to alter the implementation of saveSpittle()—using Spring-specific classes—to employ Spring's programmatic transaction support.

Usually your transactional needs won't require such precise control over transactional boundaries. That's why you'll typically choose to declare your transactions outside your application code (in the Spring configuration file, for instance). The rest of this chapter will cover Spring's declarative transaction management.

6.4 Declaring transactions

Not long ago, declarative transaction management was only available in EJB containers. But now Spring offers support for declarative transactions to POJOs. This is a significant feature of Spring because you now have an alternative to EJB for declaring atomic operations.

Spring's support for declarative transaction management is implemented through Spring's AOP framework. This is a natural fit because transactions are a system-level service above an application's primary functionality. You can think of a Spring transaction as an aspect that "wraps" a method with transactional boundaries.

Spring provides three ways to declare transactional boundaries. Historically, Spring has always supported declarative transactions by proxying beans using Spring AOP and `TransactionProxyFactoryBean`. But since Spring 2.0, the preferred ways to declare transactions are to use Spring's `tx` configuration namespace and to use the `@Transactional` annotation.

Although the legacy `TransactionProxyFactoryBean` is still available in modern versions of Spring, it's effectively obsolete and so we won't look at it in any detail. Instead, we'll focus on the `tx` namespace and annotation-oriented declarative transactions later in this section. But first let's examine the attributes that define transactions.

6.4.1 Defining transaction attributes

In Spring, declarative transactions are defined with *transaction attributes*. A transaction attribute is a description of how transaction policies should be applied to a method. There are five facets of a transaction attribute, as illustrated in figure 6.3.

Although Spring provides several mechanisms for declaring transactions, all of them rely on these five parameters to govern how transaction policies are administered. Therefore, it's essential to understand these parameters in order to declare transaction policies in Spring.

Regardless of which declarative transaction mechanism you use, you'll have the opportunity to define these attributes. Let's examine each attribute to understand how it shapes a transaction.

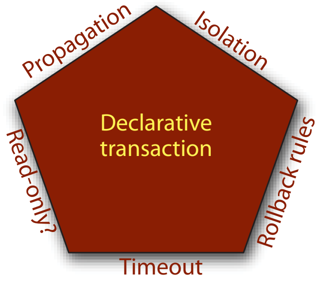

Figure 6.3 **Declarative transactions are defined in terms of propagation behavior, isolation level, read-only hints, timeout, and rollback rules.**

PROPAGATION BEHAVIOR

The first facet of a transaction is *propagation behavior*. Propagation behavior defines the boundaries of the transaction with respect to the client and to the method being called. Spring defines seven distinct propagation behaviors, as described in table 6.2.

> **PROPAGATION CONSTANTS** The propagation behaviors described in table 6.2 are defined as constants in the `org.springframework.transaction.TransactionDefinition` interface.

The propagation behaviors in table 6.2 may look familiar. That's because they mirror the propagation rules available in EJB's container-managed transactions (CMTs). For instance, Spring's `PROPAGATION_REQUIRES_NEW` is equivalent to CMT's `RequiresNew`. Spring adds an additional propagation behavior not available in CMT, `PROPAGATION_NESTED`, to support nested transactions.

Propagation rules answer the question of whether a new transaction should be started or suspended, or if a method should even be executed within a transactional context at all.

Table 6.2 Propagation rules define when a transaction is created or when an existing transaction can be used. Spring provides several propagation rules to choose from.

Propagation behavior	What it means
PROPAGATION_MANDATORY	Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
PROPAGATION_NESTED	Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.
PROPAGATION_NEVER	Indicates that the current method shouldn't run within a transactional context. If an existing transaction is in progress, an exception will be thrown.
PROPAGATION_NOT_SUPPORTED	Indicates that the method shouldn't run within a transaction. If an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_REQUIRED	Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise, a new transaction will be started.
PROPAGATION_REQUIRES_NEW	Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_SUPPORTS	Indicates that the current method doesn't require a transactional context, but may run within a transaction if one is already in progress.

For example, if a method is declared to be transactional with PROPAGATION_REQUIRES_NEW behavior, it means that the transactional boundaries are the same as the method's own boundaries: a new transaction is started when the method begins and the transaction ends when the method returns or throws an exception. If the method has PROPAGATION_REQUIRED behavior, the transactional boundaries depend on whether a transaction is already under way.

ISOLATION LEVELS

The second dimension of a declared transaction is the *isolation level*. An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions. Another way to look at a transaction's isolation level is to think of it as how selfish the transaction is with the transactional data.

In a typical application, multiple transactions run concurrently, often working with the same data to get their jobs done. Concurrency, while necessary, can lead to the following problems:

- *Dirty reads* occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.
- *Nonrepeatable reads* happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between the queries.
- *Phantom reads* are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries, the first transaction (T1) finds additional rows that weren't there before.

In an ideal situation, transactions would be completely isolated from each other, thus avoiding these problems. But perfect isolation can affect performance because it often involves locking rows (and sometimes complete tables) in the data store. Aggressive locking can hinder concurrency, requiring transactions to wait on each other to do their work.

Realizing that perfect isolation can impact performance and because not all applications will require perfect isolation, sometimes it's desirable to be flexible with regard to transaction isolation. Therefore, several levels of isolation are possible, as described in table 6.3.

Table 6.3 Isolation levels determine to what degree a transaction may be impacted by other transactions being performed in parallel.

Isolation level	What it means
`ISOLATION_DEFAULT`	Use the default isolation level of the underlying data store.
`ISOLATION_READ_UNCOMMITTED`	Allows you to read changes that haven't yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads.
`ISOLATION_READ_COMMITTED`	Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeatable reads may still occur.
`ISOLATION_REPEATABLE_READ`	Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeatable reads are prevented, but phantom reads may still occur.
`ISOLATION_SERIALIZABLE`	This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all prevented. This is the slowest of all isolation levels because it's typically accomplished by doing full table locks on the tables involved in the transaction.

ISOLATION LEVEL CONSTANTS The isolation levels described in table 6.3 are defined as constants in the `org.springframework.transaction.TransactionDefinition` interface.

`ISOLATION_READ_UNCOMMITTED` is the most efficient isolation level, but isolates the transaction the least, leaving the transaction open to dirty, nonrepeatable, and phantom reads. At the other extreme, `ISOLATION_SERIALIZABLE` prevents all forms of isolation problems but is the least efficient.

Be aware that not all data sources support all the isolation levels listed in table 6.3. Consult the documentation for your resource manager to determine what isolation levels are available.

READ-ONLY

The third characteristic of a declared transaction is whether it's a read-only transaction. If a transaction performs only read operations against the underlying data store, the data store may be able to apply certain optimizations that take advantage of the read-only nature of the transaction. By declaring a transaction as read-only, you give the underlying data store the opportunity to apply those optimizations as it sees fit.

Because read-only optimizations are applied by the underlying data store when a transaction begins, it only makes sense to declare a transaction as read-only on methods with propagation behaviors that may start a new transaction (`PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRES_NEW`, and `PROPAGATION_NESTED`).

Furthermore, if you're using Hibernate as your persistence mechanism, declaring a transaction as read-only will result in Hibernate's flush mode being set to `FLUSH_NEVER`. This tells Hibernate to avoid unnecessary synchronization of objects with the database, thus delaying all updates until the end of the transaction.

TRANSACTION TIMEOUT

For an application to perform well, its transactions can't carry on for a long time. Therefore, the next trait of a declared transaction is its *timeout.*

Suppose that your transaction becomes unexpectedly long-running. Because transactions may involve locks on the underlying data store, long-running transactions can tie up database resources unnecessarily. Instead of waiting it out, you can declare a transaction to automatically roll back after a certain number of seconds.

Because the timeout clock begins ticking when a transaction starts, it only makes sense to declare a transaction timeout on methods with propagation behaviors that may start a new transaction (`PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRES_NEW`, and `PROPAGATION_NESTED`).

ROLLBACK RULES

The final facet of the transaction pentagon is a set of rules that define which exceptions prompt a rollback and which ones don't. By default, transactions are rolled back only on runtime exceptions and not on checked exceptions. (This behavior is consistent with rollback behavior in EJBs.)

But you can declare that a transaction be rolled back on specific checked exceptions as well as runtime exceptions. Likewise, you can declare that a transaction not roll back on specified exceptions, even if those exceptions are runtime exceptions.

Now that you've had an overview of how transaction attributes shape the behavior of a transaction, let's see how to use these attributes when declaring transactions in Spring.

6.4.2 Declaring transactions in XML

In early version of Spring, declaring transactions involved wiring a special bean called TransactionProxyFactoryBean. The problem with TransactionProxyFactoryBean was that using it resulted in extremely verbose Spring configuration files. Fortunately, those days are gone and Spring now offers a tx configuration namespace that greatly simplifies declarative transactions in Spring.

Using the tx namespace involves adding it to your Spring configuration XML file:

```
<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-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

Note that the aop namespace should also be included. This is important, because some of the declarative transaction configuration elements rely on a few of Spring's AOP configuration elements (as discussed in chapter 4).

The tx namespace provides a handful of new XML configuration elements, most notably the <tx:advice> element. The following XML snippet shows how <tx:advice> can be used to declare transactional policies similar to those we defined for the Spitter service in listing 6.2:

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS"
        read-only="true"/>
  </tx:attributes>
</tx:advice>
```

With <tx:advice>, the transaction attributes are defined in a <tx:attributes> element, which contains one or more <tx:method> elements. The <tx:method> element defines the transaction attributes for a given method (or methods) as defined by the name attribute (using wildcards).

<tx:method> has several attributes that help define the transaction policies for the method(s), as defined in table 6.4.

Table 6.4 The five facets of the transaction pentagon (see figure 6.3) are specified in the attributes of the element.

Attribute	Purpose
`isolation`	Specifies the transaction isolation level.
`propagation`	Defines the transaction's propagation rule.
`read-only`	Specifies that a transaction be read-only.

Rollback rules:
`rollback-for`
`no-rollback-for`
	`rollback-for` specifies checked exceptions for which a transaction should be rolled back and not committed.
	`no-rollback-for` specifies exceptions for which the transaction should continue and not be rolled back.

`timeout`	Defines a timeout for a long-running transaction.

As defined in the `txAdvice` transaction advice, the transactional methods configured are divided into two categories: those whose names begin with `add` and everything else. The `saveSpittle()` method falls into the first category and is declared to require a transaction. The other methods are declared with `propagation="supports"`—they'll run in a transaction if one already exists, but they don't need to run within a transaction.

When declaring a transaction using `<tx:advice>`, you'll still need a transaction manager just like you did when using `TransactionProxyFactoryBean`. Choosing convention over configuration, `<tx:advice>` assumes that the transaction manager will be declared as a bean whose `id` is `transactionManager`. If you happen to give your transaction manager a different `id` (`txManager`, for instance), you'll need to specify the `id` of the transaction manager in the `transactionmanager` attribute:

```
<tx:advice id="txAdvice"
      transaction-manager="txManager">
...
</tx:advice>
```

On its own, `<tx:advice>` only defines an AOP advice for advising methods with transaction boundaries. But this is only transaction advice, not a complete transactional aspect. Nowhere in `<tx:advice>` did we indicate which beans should be advised—we need a pointcut for that. To completely define the transaction aspect, we must define an advisor. This is where the `aop` namespace gets involved. The following XML defines an advisor that uses the `txAdvice` advice to advise any beans that implement the `SpitterService` interface:

```
<aop:config>
  <aop:advisor
      pointcut="execution(* *..SpitterService.*(..))"
      advice-ref="txAdvice"/>
</aop:config>
```

The pointcut attribute uses an AspectJ pointcut expression to indicate that this advisor should advise all methods of the SpitterService interface. The transaction advice, which is referenced with the advice-ref attribute to be the advice named txAdvice, defines which methods are actually run within a transaction as well as the transactional attributes for those methods.

Although the <tx:advice> element goes a long way toward making declarative transactions more palatable for Spring developers, Spring 2.0 has one more feature that makes it even nicer for those working in a Java 5 environment. Let's look at how Spring transactions can be annotation driven.

6.4.3 Defining annotation-driven transactions

The <tx:advice> configuration element greatly simplifies the XML required for declarative transactions in Spring. What if I told you that it could be simplified even further? What if I told you that you only need to add a single line of XML to your Spring context in order to declare transactions?

In addition to the <tx:advice> element, the tx namespace provides the <tx:annotation-driven> element. Using <tx:annotation-driven> is often as simple as the following line of XML:

```
<tx:annotation-driven />
```

That's it! If you were expecting more, I apologize. I could make it slightly more interesting by specifying a specific transaction manager bean with the transactionmanager attribute (which defaults to transactionManager):

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

Otherwise, there's not much more to it. That single line of XML packs a powerful punch that lets you define transaction rules where they make the most sense: on the methods that are to be transactional.

Annotations are one of the biggest and most debated new features of Java 5. Annotations let you define metadata directly in your code rather than in external configuration files. I think they're a perfect fit for declaring transactions.

The <tx:annotation-driven> configuration element tells Spring to examine all beans in the application context and to look for beans that are annotated with @Transactional, either at the class level or at the method level. For every bean that is @Transactional, <tx:annotation-driven> will automatically advise it with transaction advice. The transaction attributes of the advice will be defined by parameters of the @Transactional annotation.

For example, the following shows SpitterServiceImpl, updated to include the @Transactional annotations.

Listing 6.3 Annotating the spitter service to be transactional

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class SpitterServiceImpl implements SpitterService {
...
  @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
  public void addSpitter(Spitter spitter) {
...
  }
...
}
```

At the class level, `SpitterServiceImpl` has been annotated with an `@Transactional` annotation that says that all methods will support transaction and be read-only. At the method level, the `saveSpittle()` method has been annotated to indicate that this method requires a transactional context.

6.5 Summary

Transactions are an important part of enterprise application development that leads to more robust software. They ensure an all-or-nothing behavior, preventing data from being inconsistent should the unexpected occur. They also support concurrency by preventing concurrent application threads from getting in each other's way as they work with the same data.

Spring supports both programmatic and declarative transaction management. In either case, Spring shields you from having to work directly with a specific transaction management implementation by abstracting the transaction management platform behind a common API.

Spring employs its own AOP framework to support declarative transaction management. Spring's declarative transaction support rivals that of EJB's CMT, enabling you to declare more than just propagation behavior on POJOs, including isolation levels, read-only optimizations, and rollback rules for specific exceptions.

This chapter showed you how to bring declarative transactions into the Java 5 programming model using annotations. With the introduction of Java 5 annotations, making a method transactional is simply a matter of tagging it with the appropriate transaction annotation.

As you've seen, Spring extends the power of declarative transactions to POJOs. This is an exciting development—declarative transactions were previously only available to EJBs. But declarative transactions are only the beginning of what Spring has to offer to POJOs. In the next chapter, you'll see how Spring extends declarative security to POJOs.