**Spring Data Access**

**Transaction Management:**

***Introduction to Spring Framework transaction management***

**Benefits of Spring Transaction Management?**

* Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, and Java Persistence API (JPA).
* Support for [declarative transaction management](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-declarative).
* Simpler API for [programmatic](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-programmatic) transaction management than complex transaction APIs such as JTA.
* Excellent integration with Spring’s data access abstractions.

**Spring Framework’s transaction value-adds and technologies.**

* [Advantages of the Spring Framework’s transaction support model](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-motivation) 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](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-strategies) outlines the core classes and describes how to configure and obtain DataSource instances from a variety of sources.
* [Synchronizing resources with transactions](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#tx-resource-synchronization)describes how the application code ensures that resources are created, reused, and cleaned up properly.
* [Declarative transaction management](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-declarative) describes support for declarative transaction management.
* [Programmatic transaction management](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-programmatic) covers support for programmatic (that is, explicitly coded) transaction management.
* [Transaction bound event](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-event) describes how you could use application events within a transaction.

***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 theorg.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;

}

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).

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 TransactionDefinition interface specifies:

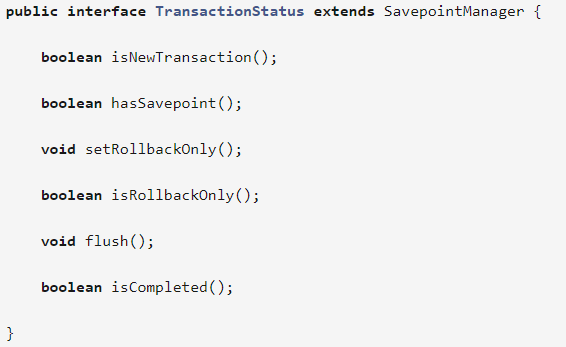
Propagation:

Isolation

Timeout:

Read-only status

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status.



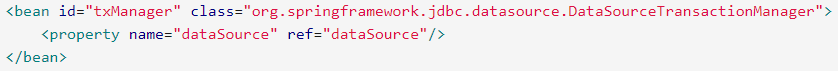
Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct PlatformTransactionManager implementation is absolutely essential.

PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on.

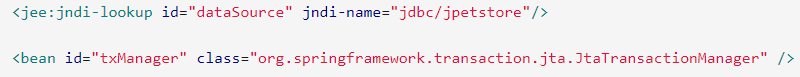
For **standalone applications**, you can configure DataSource as shown below.



The related PlatformTransactionManager bean definition will then have a reference to the DataSource definition. It will look like this:



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:



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.

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.

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.



**Synchronizing resources with transactions**

This section describes how the application code, directly or indirectly using a persistence API such as JDBC, Hibernate, or JPA, 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.

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.

Low-level synchronization approach

Classes such as DataSourceUtils (for JDBC), EntityManagerFactoryUtils (for JPA), SessionFactoryUtils (for Hibernate), 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:



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.

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 JNDIDataSource as provided by a Java EE server.

**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 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 or Hibernate 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,](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#transaction-declarative-rolling-back)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.

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.

***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.

@Transactional

class MyClass {

}

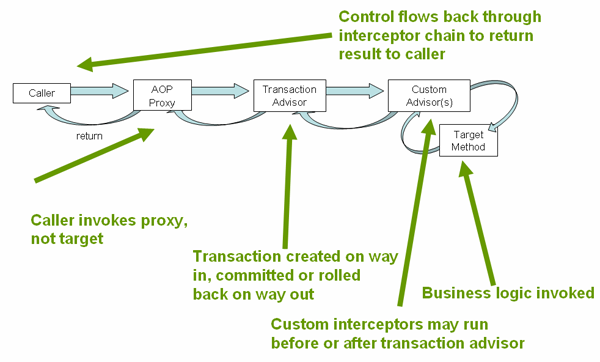
@EnableTransactionManagement

@Configuration

class MyApp {

}

Conceptually, calling a method on a transactional proxy looks like this…​



|  |  |
| --- | --- |
| Caller | Target Method |
| **public** **class** MyClass {  @Inject  MyDao dao;    **public** **void** methodA() {  dao.selectAllRecords();  }  } | @Transactional  **public** **class** MyDao {  **public** **void** selectAllRecords() {  //Do impl  }  } |

***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. (Refer SpringJdbcDemo [BootM ain.java])

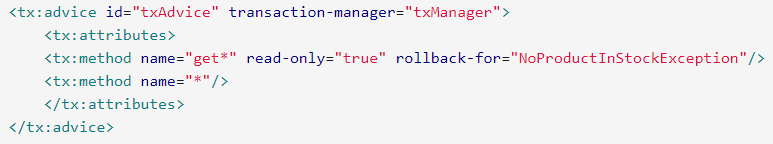
This section describes how you can control the rollback of transactions in a simple declarative fashion.

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.

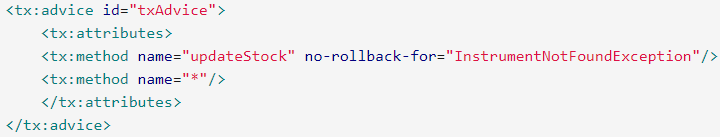
The Spring Framework’s transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and make a determination whether to mark the transaction for rollback.

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. (Errors 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.

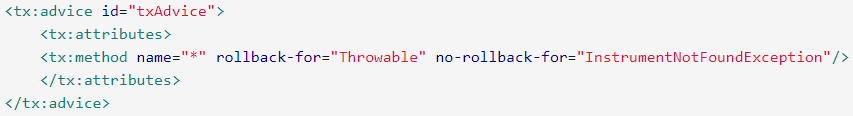
You can configure exactly which Exception types mark a transaction for rollback, including checked exceptions. The following XML snippet demonstrates how you configure rollback for a checked, application-specific Exception type.



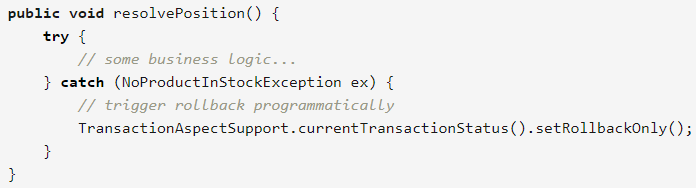
You can also specify 'no rollback rules', if you do not want a transaction rolled back when an exception is thrown. The following example tells the Spring Framework’s transaction infrastructure to commit the attendant transaction even in the face of an unhandled InstrumentNotFoundException.



When the Spring Framework’s transaction infrastructure catches an exception and it 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.



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:



***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:

Refer: SpringJdbcDemo (context2.xml)

The following example shows how to configure two distinct beans with totally different transactional settings.

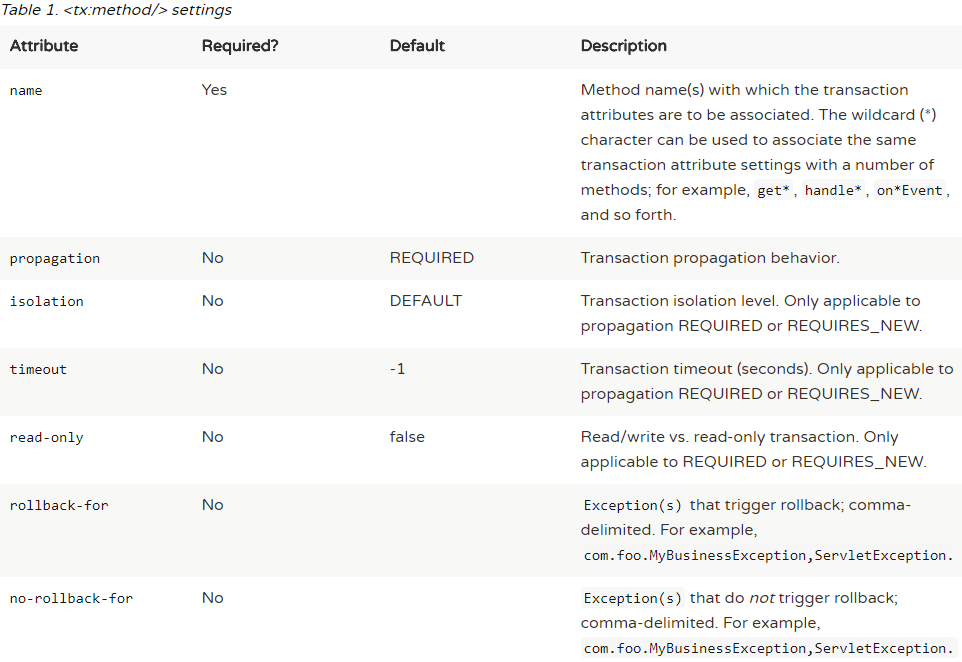
Refer: SpringJdbcDemo (context3.xml)

***<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](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html#tx-propagation) 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:



***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 standard javax.transaction.Transactional annotation is also supported as a drop-in replacement to Spring’s own annotation. Please refer to JTA 1.2 documentation for more details.

Refer: SpringJdbcDemo com.techstack.spring.declarativeTransactionManagement.annotation.\*

@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.

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

Refer: TrasnactionalService.java and AppConfig2.java

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](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-meta-annotations) allows you to define custom shortcut annotations for your specific use cases. For example, defining the following annotations

Refer: com.techstack.spring.declarativeTransactionManagement.annotation.custom

***Transaction propagation***

**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.

***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.

**Transaction bound event**

As of Spring 4.2, the listener of an event can be bound to a phase of the transaction. The typical example is to handle the event when the transaction has completed successfully: this allows events to be used with more flexibility when the outcome of the current transaction actually matters to the listener.

Registering a regular event listener is done via the @EventListener annotation. If you need to bind it to the transaction use @TransactionalEventListener. When you do so, the listener will be bound to the commit phase of the transaction by default.