**Global Transaction**

Global transactions let you 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 (partly due to its exception model). Furthermore, a JTA User Transaction normally needs to be sourced from JNDI, meaning that you also need to use JNDI in order to use JTA. The use of global transactions limits 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 through EJB CMT (Container Managed Transaction).

EJB CMT removes the need for transaction-related JNDI lookups, although the use of EJB itself necessitates the use of JNDI. (EJB internally uses JNDI).

The significant downside is that CMT is tied to JTA and an application server environment.

**Local Transaction**

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use but have a significant disadvantage: They cannot work across multiple transactional resources.

code that manages transactions by using a JDBC connection cannot run within a global JTA transaction.

Because the application server is not involved in transaction management, it cannot help ensuring correctness across multiple resources.

It is worth noting that most applications use a single transaction resource.

**Springs consistent Programmatic Model for Transaction Management**

Spring resolves the disadvantages of global and local transactions. It lets application developers 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.

Spring Provides 2 types of Transaction Management

**1) Declarative**

**2) Programmatic**

**Declarative:**

* Most preferred transaction management, which is recommend in most cases.
* 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.

Declarative transactions transaction logic is separated from the business logic.

Example : Declarative is done through Transaction

**Programmatic**

* With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure.
* Programmatic transaction management is usually a good idea only if you have a small number of transactional operations.
* 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.

Example : handling Transaction programmatically

**Application server in Transaction**

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.

Spring Framework, only some of the bean definitions in your configuration file need to change (rather than your code) for moving from local transaction to global transaction.

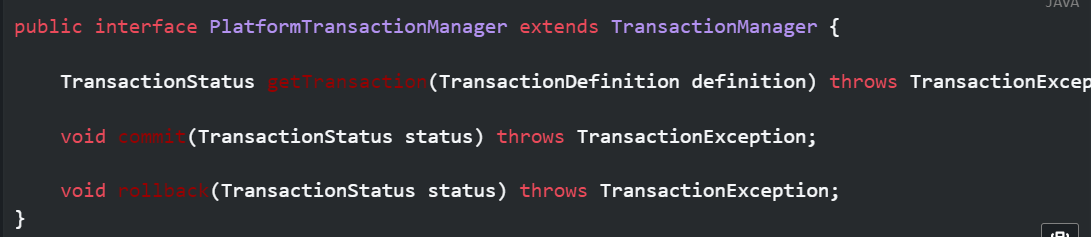
you may need application server for other application server capabilities, such as Java Message Service (JMS) and Jakarta EE Connector Architecture (JCA).

JtaTransactionManager is the standard choice to run on Jakarta 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 and Geronimo) without any special configuration required.

**Spring Framework traction Internals.**

A transaction strategy is defined by a TransactionManager.

1. org.springframework.transaction.PlatformTransactionManager (interface for imperative transaction management)
2. org.springframework.transaction.ReactiveTransactionManager (interface for reactive transaction management)



**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.

**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.

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.

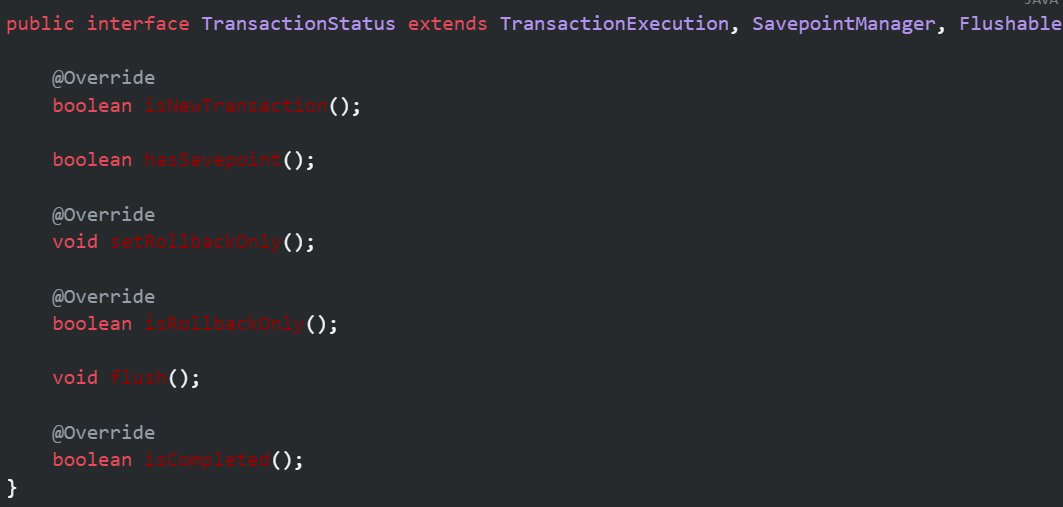
**The TransactionDefinition interface specifies:**

**Propagation:** Typically, all code within a transaction scope runs in that transaction. However, you can specify the behavior if a transactional method is run 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 Transaction Propagation.

**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?

**Timeout:** How long this transaction runs before timing out and being automatically rolled back by the underlying transaction infrastructure.

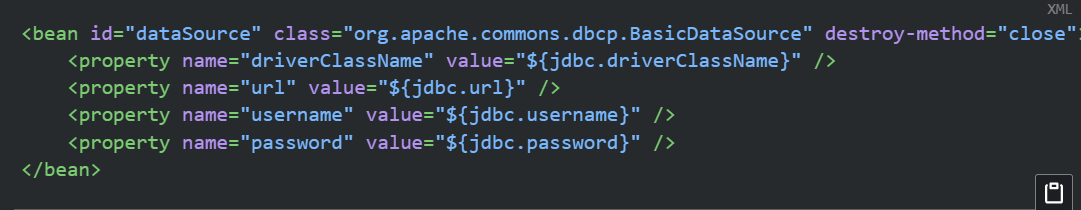
**Read-only status:** You can use a read-only transaction when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you use Hibernate.

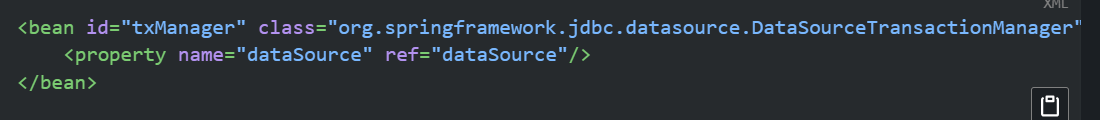


Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct **TransactionManager** implementation is absolutely essential. You typically define this implementation through dependency injection.

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

**local PlatformTransactionManager implementation Using JDBC**





**JTA Transaction Manager implementation**

For JTA in a Jakarta EE container, then you use a container DataSource, obtained through JNDI, in conjunction with Spring’s JtaTransactionManager.



**High level approach**

resource creation and reuse, cleanup, optional transaction synchronization of the resources, and exception mapping.

Example ORM ,JPA

**Low level approach**

**DataSourceUtils** (for JDBC), **EntityManagerFactoryUtils** (for JPA), **SessionFactoryUtils** are used for

Low level transaction management.

Users must handle entire process using native support for transaction synchronization.

This approach also works without Spring transaction management (transaction synchronization is optional), so you can use it whether you use Spring for transaction management.

Example : JDBC transaction related classes

**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 Jakarta EE server**.**

Should never be used.

**Declarative Transaction**

The Spring Framework’s declarative transaction management is made possible with Spring aspect-oriented programming (AOP).

**Difference between**

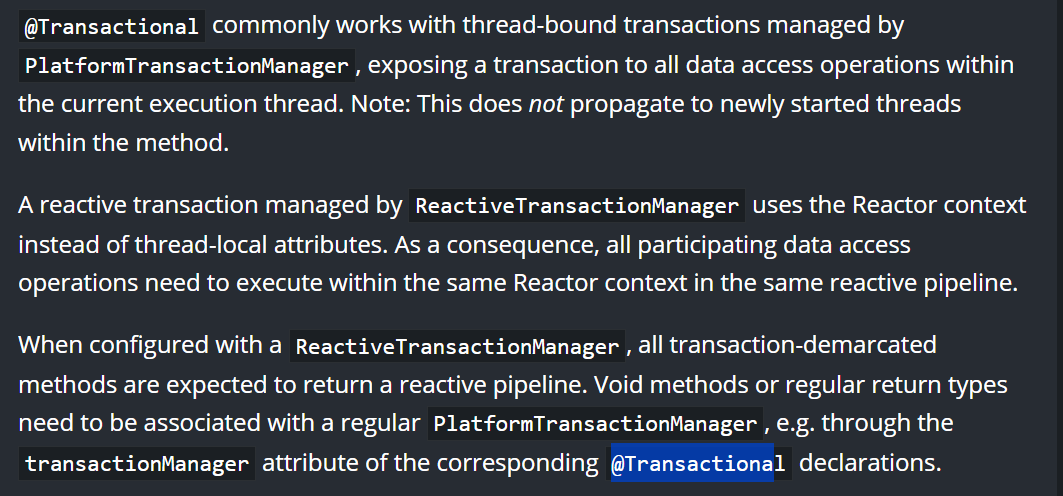
* Unlike EJB CMT, which is tied to JTA, the Spring Framework’s declarative transaction management works in any environment. It can work with JTA transactions or local transactions by using JDBC, JPA, or Hibernate by adjusting the configuration files.
* You can apply the Spring Framework declarative transaction 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 lets you 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 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 high-end application servers do. 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.

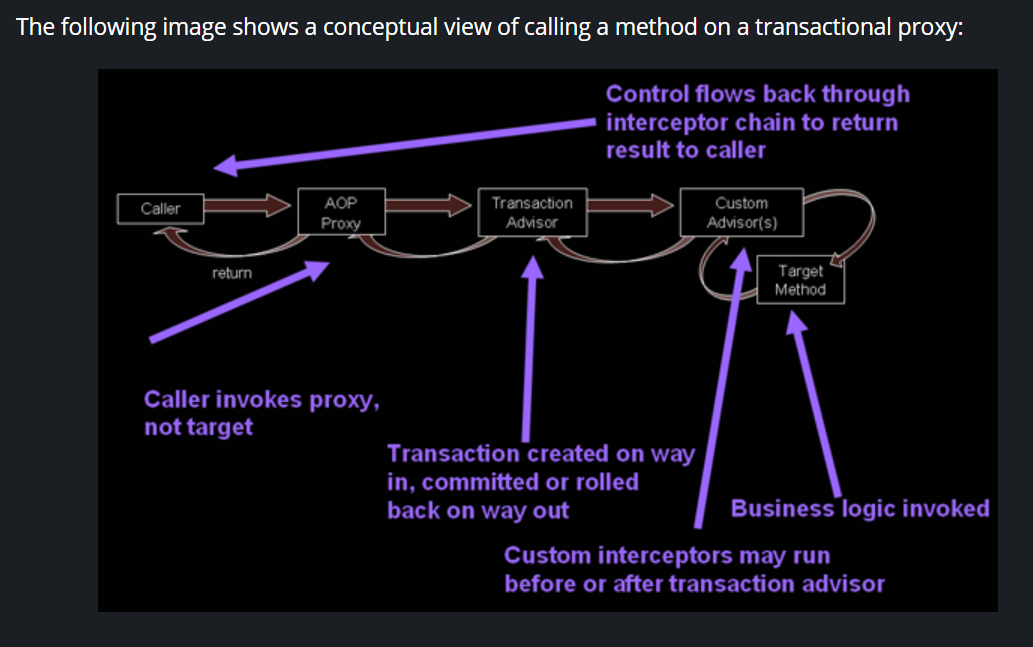
EJB container default behavior automatically rolls back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an application exception (that is, a checked exception other than java.rmi.RemoteException). While the Spring default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

**How Transaction works in declarative model**

@Transactional annotation, add @EnableTransactionManagement to your configuration

* 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 **TransactionManager** implementation to drive transactions around method invocations.
* Spring Framework’s TransactionInterceptor provides transaction management for imperative and reactive programming models. The interceptor detects the desired flavour of transaction management by inspecting the method return type. Methods returning a reactive type such as Publisher or Kotlin Flow (or a subtype of those) qualify for reactive transaction management. All other return types including void use the code path for imperative transaction management.
* Transaction management flavours impact which transaction manager is required. Imperative transactions require a **PlatformTransactionManager**, while reactive transactions use **ReactiveTransactionManager** implementations.





**@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:

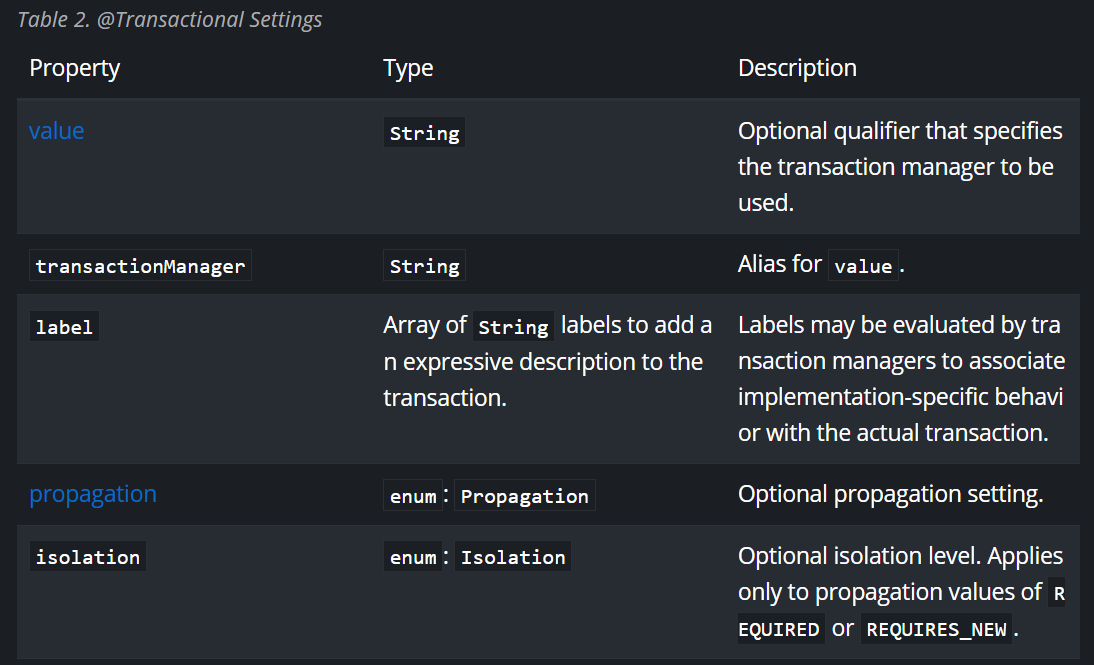
• The propagation setting is PROPAGATION\_REQUIRED.

• The isolation level is ISOLATION\_DEFAULT.

• The transaction is read-write.

• The transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.

• Any **RuntimeException** or Error triggers rollback, and any checked Exception does not.



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Currently, you cannot have explicit control over the name of a transaction, where 'name' means the transaction name that appears in a transaction monitor and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + . + the 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.example.BusinessService.handlePayment.

**Multiple Transaction Managers with @Transactional**

Most Spring applications need only a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. You can use the value or transactionManager attribute of the @Transactional annotation to optionally specify the identity of the TransactionManager 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, you can combine the following Java code with the following transaction manager bean declarations in the application context:

• Java

• Kotlin

public class TransactionalService {

@Transactional("order")

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

@Transactional("account")

public void doSomething() { ... }

@Transactional("reactive-account")

public Mono<Void> doSomethingReactive() { ... }

}

The following listing shows the bean declarations:

<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.support.JdbcTransactionManager">

...

<qualifier value="order"/>

</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.support.JdbcTransactionManager">

...

<qualifier value="account"/>

</bean>

<bean id="transactionManager3" class="org.springframework.data.r2dbc.connection.R2dbcTransactionManager">

...

<qualifier value="reactive-account"/>

</bean>

Copied!

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

**Custom Composed Annotations**

If you find you repeatedly use the same attributes with @Transactional on many different methods, Spring’s meta-annotation support lets you define custom composed annotations for your specific use cases. For example, consider the following annotation definitions:

• Java

• Kotlin

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Transactional(transactionManager = "order", label = "causal-consistency")

public @interface OrderTx {

}

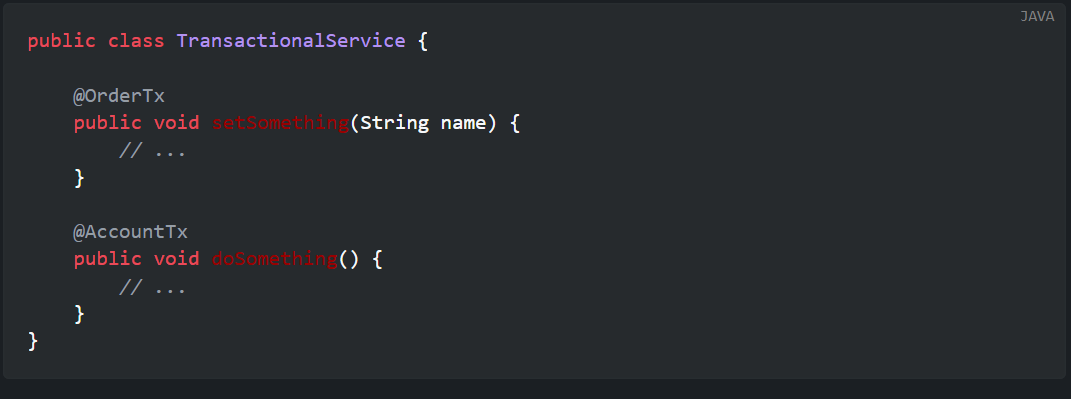
@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Transactional(transactionManager = "account", label = "retryable")

public @interface AccountTx {

}



**Types of Propagation**

**1)REQUIRED**

REQUIRED is the default propagation. Spring checks if there is an active transaction, and if nothing exists, it creates a new one. Otherwise, the business logic appends to the currently active transaction:

**2)** **SUPPORTS**

For SUPPORTS, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn’t a transaction, it is executed non-transactional:

***3)* MANDATORY**

When the propagation is MANDATORY, if there is an active transaction, then it will be used. If there isn’t an active transaction, then Spring throws an exception:

**4)NEVER**

For transactional logic with NEVER propagation, Spring throws an exception if there’s an active transaction:

**5)REQUIRES\_NEW**

When the propagation is REQUIRES\_NEW, Spring suspends the current transaction if it exists, and then creates a new one:

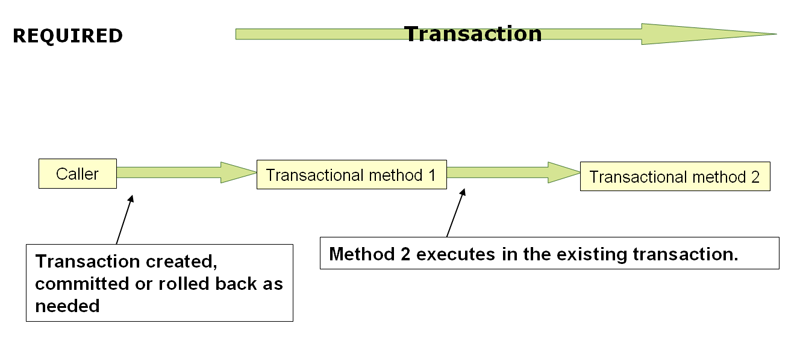
**6)NESTED**

NESTED propagation, Spring checks if a transaction exists, and if so, it marks a save point. This means that if our business logic execution throws an exception, then the transaction rollbacks to this save point. If there’s no active transaction, it works like REQUIRED.

**DataSourceTransactionManager supports this propagation out-of-the-box. Some implementations of JTATransactionManager may also support this.**

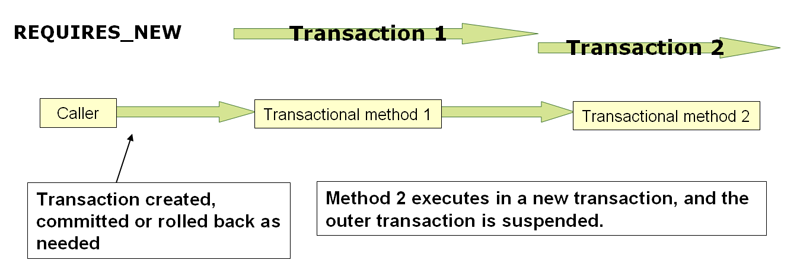
[**JpaTransactionManager**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaTransactionManager.html)**supports NESTED only for JDBC connections. However, if we set the nestedTransactionAllowed flag to true, it also works for JDBC access code in JPA transactions if our JDBC driver supports save points.**

**Understanding PROPAGATION\_REQUIRED**



* PROPAGATION\_REQUIRED enforces a physical transaction, either locally for the current scope if no transaction exists yet or participating in an existing 'outer' transaction defined for a larger scope.
* Outer transactions in read-only mode inner transactions in read-write mode will fail.
* By default, a participating transaction joins the characteristics of the outer scope, silently ignoring the local isolation level, timeout value, or read-only flag.
* rollback-only marker set in the inner transaction scope does affect the outer transaction’s chance to actually commit.
* in the case where an inner transaction scope sets the rollback-only marker, the outer transaction has not decided on the rollback itself, 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. 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.

**PROPAGATION\_REQUIRES\_NEW**



PROPAGATION\_REQUIRES\_NEW, in contrast to PROPAGATION\_REQUIRED, always uses an independent physical transaction for each affected transaction scope, never participating in an existing transaction for an outer scope. In such an arrangement, the underlying resource transactions are different and, hence, can commit or roll back independently, with an outer transaction not affected by an inner transaction’s rollback status and with an inner transaction’s locks released immediately after its completion. Such an independent inner transaction can also declare its own isolation level, timeout, and read-only settings and not inherit an outer transaction’s characteristics.

|  |  |
| --- | --- |
|  | The resources attached to the outer transaction will remain bound there while the inner transaction acquires its own resources such as a new database connection. This may lead to exhaustion of the connection pool and potentially to a deadlock if several threads have an active outer transaction and wait to acquire a new connection for their inner transaction, with the pool not being able to hand out any such inner connection anymore. Do not use PROPAGATION\_REQUIRES\_NEW unless your connection pool is appropriately sized, exceeding the number of concurrent threads by at least 1. |

PROPAGATION\_NESTED

PROPAGATION\_NESTED uses a single physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks let an inner transaction scope 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 it works only with JDBC resource transactions. See Spring’s [DataSourceTransactionManager](https://docs.spring.io/spring-framework/docs/6.1.4/javadoc-api/org/springframework/jdbc/datasource/DataSourceTransactionManager.html).

**Advising Transactional Operations**

* The configured profiling aspect starts.
* The transactional advice runs.
* The method on the advised object runs.
* The transaction commits.
* The profiling aspect reports the exact duration of the whole transactional method invocation.

@with AspectJ

You can also use the Spring Framework’s @Transactional support outside of a Spring container by means of an AspectJ aspect. To do so, first annotate your classes (and optionally your classes' methods) with the @Transactional annotation, and then link (weave) your application with the org.springframework.transaction.aspectj.AnnotationTransactionAspect defined in the spring-aspects.jar file.

**@TransactionalEventListener**

* The @TransactionalEventListener annotation is used to register a listener method that will be invoked after a transaction has committed successfully. This can be useful for ensuring that certain operations are only performed if the transaction succeeds.
* To use the @TransactionalEventListener annotation, you first need to create a listener method. This method must be annotated with @TransactionalEventListener and must take a single parameter of the type of event that you want to listen for.
* Once you have created your listener method, you need to register it with Spring. This can be done by annotating the method with @Component.
* When a transaction commits successfully, Spring will check to see if any listener methods have been registered for the event type that was published. If any listener methods have been registered, Spring will invoke them.
* It is important to note that the @TransactionalEventListener annotation is only supported by Spring Boot 2.0 and higher.
* In this example, the onOrderCreated() method is a listener method that will be invoked after an OrderCreatedEvent has been published and the transaction has committed successfully.
* The @TransactionalEventListener annotation can be a useful tool for ensuring that certain operations are only performed if the transaction succeeds.
* The @TransactionalEventListener annotation exposes a phase attribute that lets you customize the phase of the transaction to which the listener should be bound. The valid phases are BEFORE\_COMMIT, AFTER\_COMMIT (default), AFTER\_ROLLBACK, as well as AFTER\_COMPLETION which aggregates the transaction completion (be it a commit or a rollback).

Here is an example of how to use the @TransactionalEventListener annotation:

@Component

public class OrderCreatedEventListener {

@TransactionalEventListener

public void onOrderCreated(OrderCreatedEvent event) {

// Do something with the order created event

}

}

* The @TransactionalEventListener annotation is used to register a method to be called after a transaction has completed. The method is called by the Spring Framework's transaction synchronization mechanism.
* The transaction synchronization mechanism is responsible for ensuring that all of the changes made to the database within a transaction are committed or rolled back as a unit. The mechanism does this by registering callback methods to be called at various points in the transaction lifecycle.
* The @TransactionalEventListener annotation can be used to register a method to be called at any of the following points in the transaction lifecycle:
* AFTER\_COMMIT: The method is called after the transaction has successfully committed.
* AFTER\_ROLLBACK: The method is called after the transaction has rolled back.
* AFTER\_COMPLETION: The method is called after the transaction has completed, regardless of whether it committed or rolled back.
* BEFORE\_COMMIT: The method is called before the transaction commits.
* The @TransactionalEventListener annotation can be used on any method, but it is typically used on methods that perform some type of cleanup operation, such as sending an email or logging a message.
* Here is an example of a method that is annotated with the @TransactionalEventListener annotation:

@TransactionalEventListener

public void onTransactionCommit(TransactionSynchronization synchronization) {

// Perform some cleanup operation

}

This method will be called after the transaction has successfully committed.

The @TransactionalEventListener annotation can also be used to register a method to be called asynchronously. To do this, you can annotate the method with the @Async annotation.

Here is an example of a method that is annotated with the @TransactionalEventListener and @Async annotations:

@TransactionalEventListener

@Async

public void onTransactionCommit(TransactionSynchronization synchronization) {

// Perform some cleanup operation

}

Use code with caution.

Learn more

This method will be called asynchronously after the transaction has successfully committed.

The @TransactionalEventListener annotation is a powerful tool that can be used to ensure that all of the changes made to the database within a transaction are cleaned up properly.

A screenshot of a computer program

Description automatically generated



Spring supports various isolation levels, each providing a different balance between performance and data consistency:

## 1. DEFAULT

* Uses the default isolation level of the underlying database.

## 2. READ\_UNCOMMITTED

* Allows dirty reads, meaning that changes made by one transaction can be read by other transactions before being committed.
* This level provides the highest performance but the lowest data consistency.

## 3. READ\_COMMITTED

* Prevents dirty reads by allowing other transactions to read only committed changes.
* This level provides a balance between performance and consistency.

## 4. REPEATABLE\_READ

* Ensures that if a value is read multiple times within the same transaction, the result will always be the same, as long as the value hasn’t been changed by the same transaction.
* This level provides higher consistency but may impact performance.
* Repeatable read is the most common solution for the problems we faced in reading committed isolation levels. It is by default choice for MySQL. In this, each transaction reads from a snapshot of the database at the start of the transaction. I.e. all the transactions that were committed till the point of time.

## 5. SERIALIZABLE

* Ensures complete isolation from other transactions, meaning that transactions are executed in a way that produces the same outcome as if they were executed serially.
* This level provides the highest consistency but may have a significant impact on performance.

**810**

Read committed is an isolation level that guarantees that any data read was *committed* at the moment is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, will find the *Same* data, data is free to change after it was read.

Repeatable read is a higher isolation level, that in addition to the guarantees of the read committed level, it also guarantees that any data read *cannot change*, if the transaction reads the same data again, it will find the previously read data in place, unchanged, and available to read.

The next isolation level, serializable, makes an even stronger guarantee: in addition to everything repeatable read guarantees, it also guarantees that *no****new****data* can be seen by a subsequent read.

Say you have a table T with a column C with one row in it, say it has the value '1'. And consider you have a simple task like the following:

BEGIN TRANSACTION;

SELECT \* FROM T;

WAITFOR DELAY '00:01:00'

SELECT \* FROM T;

COMMIT;

That is a simple task that issue two reads from table T, with a delay of 1 minute between them.

* under READ COMMITTED, the second SELECT may return *any* data. A concurrent transaction may update the record, delete it, insert new records. The second select will always see the *new* data.
* under REPEATABLE READ the second SELECT is guaranteed to display at least the rows that were returned from the first SELECT *unchanged*. New rows may be added by a concurrent transaction in that one minute, but the existing rows cannot be deleted nor changed.
* under SERIALIZABLE reads the second select is guaranteed to see *exactly* the same rows as the first. No row can change, nor deleted, nor new rows could be inserted by a concurrent transaction.

If you follow the logic above you can quickly realize that SERIALIZABLE transactions, while they may make life easy for you, are always *completely blocking* every possible concurrent operation, since they require that nobody can modify, delete nor insert any row. The default transaction isolation level of the .Net System.Transactions scope is serializable, and this usually explains the abysmal performance that results.

**Annotations Used to Configure DAO or Repository Classes**

The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the @Repository annotation. This annotation also lets the component scanning support find and configure your DAOs and repositories without having to provide XML configuration entries for them. The following example shows how to use the @Repository annotation:

Java

Kotlin

@Repository

public class SomeMovieFinder implements MovieFinder {

// ...

}

The @Repository annotation.

Any DAO or repository implementation needs access to a persistence resource, depending on the persistence technology used. For example, a JDBC-based repository needs access to a JDBC DataSource, and a JPA-based repository needs access to an EntityManager. The easiest way to accomplish this is to have this resource dependency injected by using one of the @Autowired, @Inject, @Resource or @PersistenceContext annotations.

**Rollback Transaction For Unchecked Exceptions**

@Transactional(**rollbackFor = InvalidInsuranceAmountException.class**)

Automatic rollback is applicable of unchecked exception . for checked exception transaction will not rollback automatically . need to be handled using rollbackFor.

**AOP**

Cross-cutting concerns are parts of a program that affect many other parts of the system.

**AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.** It does this by adding additional behaviour to existing code without modifying the code itself.

Instead, we can declare the new code and the new behaviors separately.

Spring’s [AOP framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop) helps us implement these cross-cutting concerns.