**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.

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.

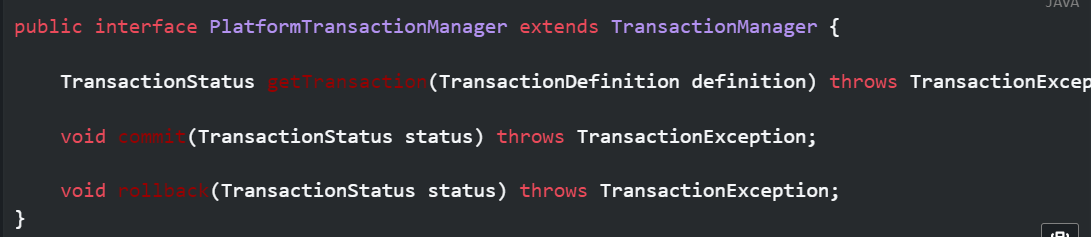
Spring Framework, only some of the bean definitions in your configuration file need to change (rather than your code) for moving from local transaction to global transaction.

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

**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.

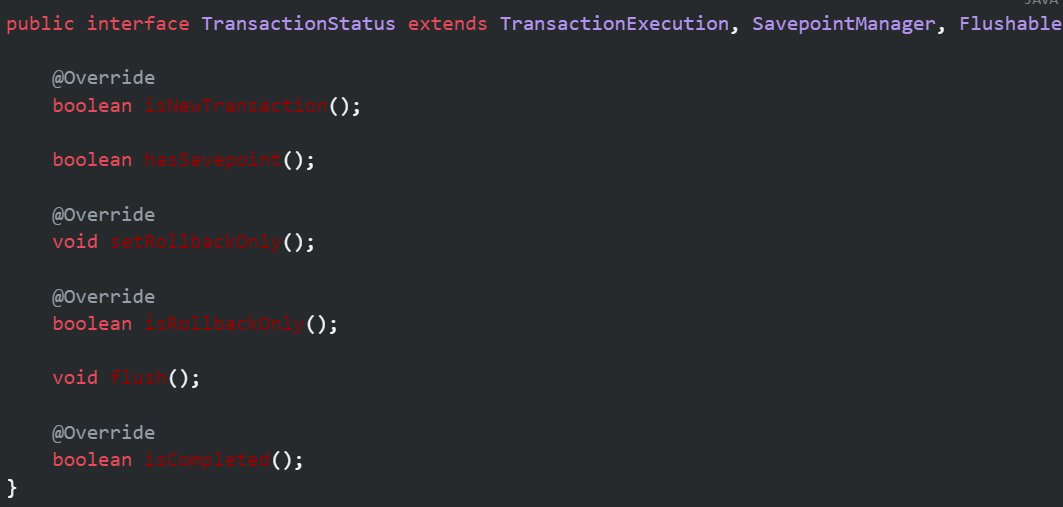
**The TransactionDefinition interface specifies:**

**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.

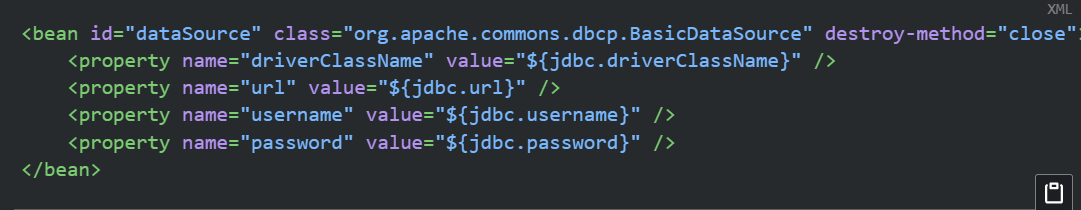
**Read-only status:** You can use a read-only transaction when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you use Hibernate.

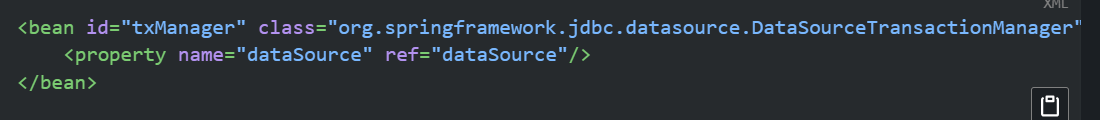


Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct **TransactionManager** implementation is absolutely essential. You typically define this implementation through dependency injection.

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 have to 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 or not 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**

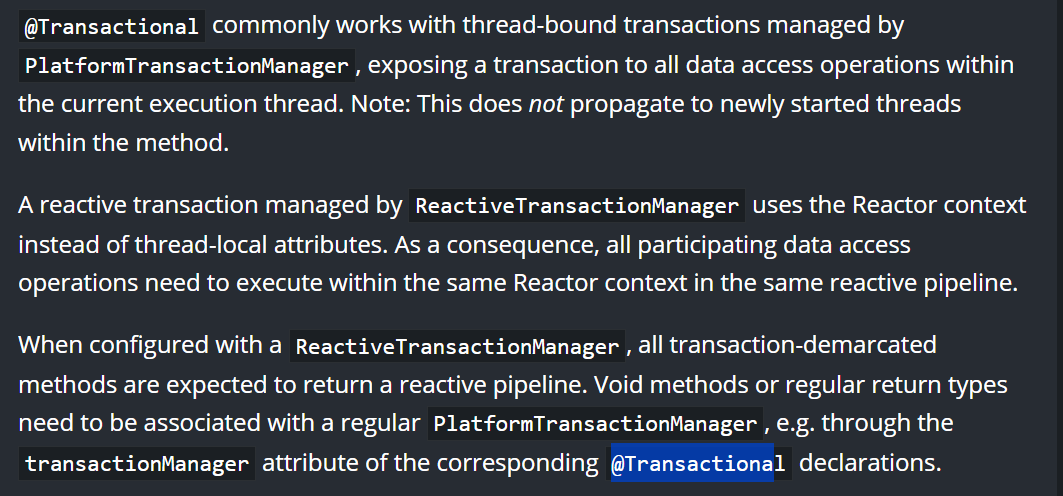
* Unlike EJB CMT, which is tied to JTA, the Spring Framework’s declarative transaction management works in any environment. It can work with JTA transactions or local transactions by using JDBC, JPA, or Hibernate by adjusting the configuration files.
* You can apply the Spring Framework declarative transaction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

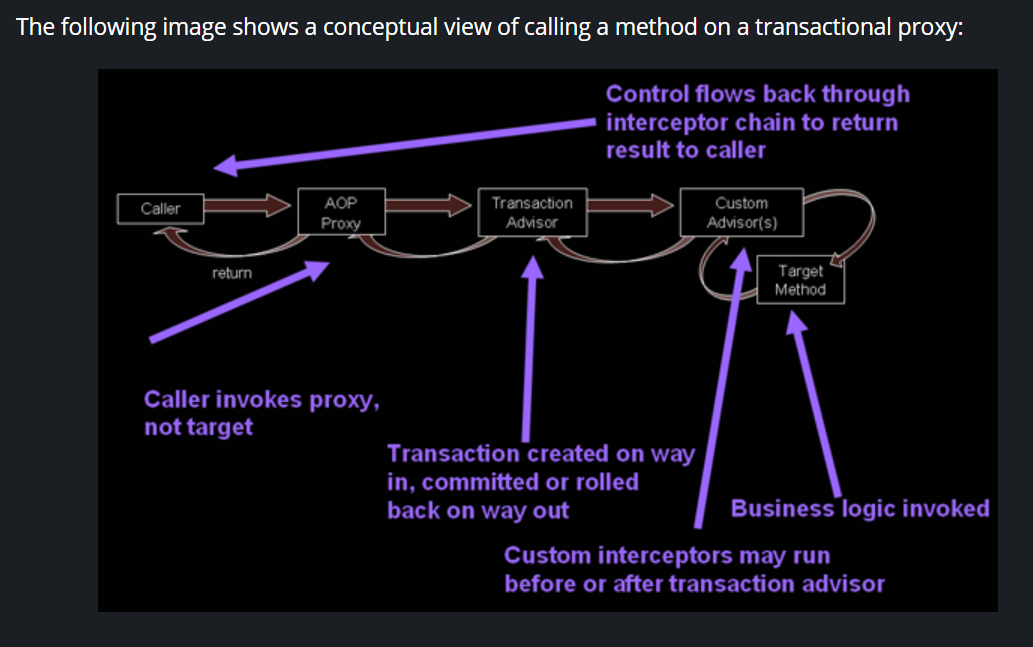
EJB container default behavior automatically rolls back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an application exception (that is, a checked exception other than java.rmi.RemoteException). While the Spring default 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 flavor 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.

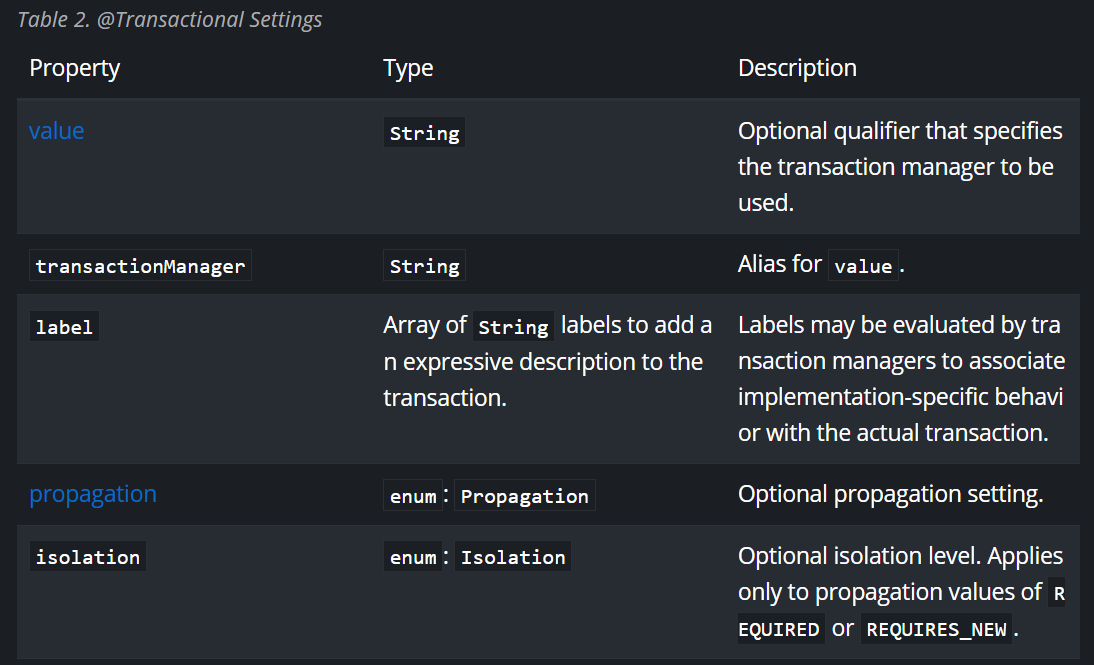




**@Transactional Settings**

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() { ... }

}

Copied!

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](https://docs.spring.io/spring-framework/reference/core/beans/classpath-scanning.html#beans-meta-annotations) 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 {

}

