<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html>

<https://dzone.com/articles/spring-transaction-management>

<http://www.baeldung.com/transaction-configuration-with-jpa-and-spring>

http://www.byteslounge.com/tutorials/spring-transaction-propagation-tutorial

@Target(value=TYPE)

@Retention(value=RUNTIME)

@Documented

[@Import](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Import.html)([value](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Import.html#value--)=[TransactionManagementConfigurationSelector.class](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/TransactionManagementConfigurationSelector.html))

public @interface **EnableTransactionManagement**

**EnableTransactionManagement** annotation enables transaction in spring application.

The possible parameters to this transaction

|  |  |
| --- | --- |
| [**AdviceMode**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html) | [**mode**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#mode--)  Indicate how transactional advice should be applied.  **The default is**[**AdviceMode.PROXY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#PROXY)**.** Please note that proxy mode allows for  Interception of calls through the proxy only. Local calls within the same class cannot  get intercepted that way; an [Transactional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html) annotation on such a method within a  local call will be ignored since Spring's interceptor does not even kick in for such a  Runtime scenario. For a more advanced mode of interception, consider switching  this to [AdviceMode.ASPECTJ](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#ASPECTJ).  **Default:**  org.springframework.context.annotation.AdviceMode.PROXY |
| int | [**order**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#order--)  Indicate the ordering of the execution of the transaction advisor when multiple  Advices are applied at a specific joinpoint. |
| boolean | [**proxyTargetClass**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#proxyTargetClass--)  Indicate whether subclass-based (CGLIB) proxies are to be created (true)  as opposed to standard Java interface-based proxies (false).  public abstract boolean proxyTargetClass  Indicate whether subclass-based (CGLIB) proxies are to be created (true) as opposed to  Standard Java interface-based proxies (false). The default is false. **Applicable only if**[**mode()**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#mode--)  **Is set to** [**AdviceMode.PROXY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#PROXY).  Note that setting this attribute to true will affect *all* Spring-managed beans requiring  Proxying, not just those marked with @Transactional. For example, other beans marked  With Spring's @Asyncannotation will be upgraded to subclass proxying at the same time.  This approach has no negative impact in practice unless one is explicitly expecting one type  Of proxy vs another, e.g. in tests.  **Default:**  false |

**mode**

[**AdviceMode.PROXY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#PROXY)

[**AdviceMode.ASPECTJ**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#ASPECTJ)

Default value [**AdviceMode.PROXY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#PROXY)

[**proxyTargetClass**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#proxyTargetClass--)

This is applicable only if mode is set to [**AdviceMode.PROXY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html#PROXY)

**Its Boolean value true/false**

**Default value is false means** Java interface-based proxies

True means subclass-based (CGLIB) proxies

**Important note**

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with @Transactional annotation as compared to annotating interfaces.

The reason for this is if you put an annotation on the Interface Level and if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure .i.e Transactional behaviour will not be applied.

The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a public method on a class.

If you want some methods in the class (annotated with @Transactional) to have different attributes settings like isolation or propagation level then put annotation at method level which will override class level attribute settings.

In proxy mode (which is the default), only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with @Transactional.

**Spring creates proxies for all the classes annotated with @Transactional** – either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the method being invoked – mainly for **starting and committing the transaction**.

What is important to keep in mind is that, if the transactional bean is implementing an interface, by default **the proxy will be a Java Dynamic Proxy**. This means that only external method calls that come in through the proxy will be intercepted – **any self-invocation calls will not start any transaction** – even if the method is annotated with @Transactional.

Another caveat of using proxies is that **only public methods should be annotated with @Transactional** – methods of any other visibilities will simply ignore the annotation silently as these are not proxied.

***@Transactional* Annotation**

With transactions configured, a bean can now be annotated with @Transactional either at the class or method level:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Service  @Transactional  public class FooService {      //...  } |

The annotation supports **further configuration** as well:

* the **Propagation Type** of the transaction
* the **Isolation Level** of the transaction
* a **Timeout** for the operation wrapped by the transaction
* a ***readOnly* flag** – a hint for the persistence provider that the transaction should be read only
* the **Rollback** rules for the transaction

Note that – by default, rollback happens for runtime, unchecked exceptions only. **The checked exception does not trigger a rollback** of the transaction; the behaviour can, of course, be configured with the ***rollbackFor***and ***noRollbackFor*** annotation parameters.

### **Read-Only Transactions**

The **readOnly flag** usually generates confusion, especially when working with JPA; from the Javadoc:

This just serves as a hint for the actual transaction subsystem; it will not necessarily cause failure of write access attempts. A transaction manager which cannot interpret the read-only hint will not throw an exception when asked for a read-only transaction.

The fact is that it **cannot be guaranteed** that an insert or update will not occur when the readOnly flag is set – its behaviour is **vendor dependent** whereas JPA is vendor agnostic.

It is also important to understand that the readOnly flag is only relevant **inside a transaction**; if an operation occurs outside of a transactional context, the flag is simply ignored. A simple example of that would call a method annotated with:

|  |  |
| --- | --- |
| 1 | @Transactional( propagation = Propagation.SUPPORTS , readOnly = true ) |

From a non-transactional context – a transaction will not be created and the readOnly flag will be ignored.

**MANDATORY**

Support a current transaction, throw an exception if none exists.

**NESTED**

Execute within a nested transaction if a current transaction exists,

Behave like PROPAGATION\_REQUIRED else.

**NEVER**

Execute non-transactionally, throw an exception if a transaction exists.

**NOT\_SUPPORTED**

Execute non-transactionally, suspend the current transaction if one exists.

**REQUIRED**

Support a current transaction, create a new one if none exists.

**REQUIRES\_NEW**

Create a new transaction, and suspend the current transaction if one exists.

**SUPPORTS**

Support a current transaction, execute non-transactionally if none exists.

**Dirty Read:- (rollback scenario)**

Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.

**Dirty Read Example:-**

Transaction A begins.  
UPDATE EMPLOYEE SET SALARY = 10000 WHERE EMP\_ID= ‘123’;

Transaction B begins.  
SELECT \* FROM EMPLOYEE;  
(Transaction B sees data which is updated by transaction A. But, those updates have not yet been committed.)

**Non-Repeatable Read:- (holding old value)**

Non Repeatable Reads happen when in a same transaction same query yields to a different result. This occurs when one transaction repeatedly retrieves the data, while a difference transactions alters the underlying data. This causes the different or non-repeatable results to be read by the first transaction.

**Non-Repeatable Example:-**

Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE EMP\_ID= ‘123’;

Transaction B begins.  
UPDATE EMPLOYEE SET SALARY = 20000 WHERE EMP\_ID= ‘123’;  
(Transaction B updates rows viewed by the transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.

**Phantom Read:- (getting different result everytime)**

Phantom read occurs where in a transaction execute same query more than once, and the second transaction result set includes rows that were not visible in the first result set. This is caused by another transaction inserting new rows between the execution of the two queries. This is similar to a non-repeatable read, except that the number of rows is changed either by insertion or by deletion.

**Phantom Read Example:-**

Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE SALARY > 10000;

Transaction B begins.  
INSERT INTO EMPLOYEE (EMP\_ID, FIRST\_NAME, DEPT\_ID, SALARY) VALUES (‘111’, ‘Jamie’, 10, 35000);  
Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

**An isolation level:** It is about how much a transaction may be impacted by the activities of other concurrent transactions. It a supports consistency leaving the data across many tables in a consistent state. It involves locking rows and/or tables in a database.

**The problem with multiple transaction**

**Scenario 1**.If T1 transaction reads data from table A1 that was written by another concurrent transaction T2.If on the way T2 is rollback, the data obtained by T1 is invalid one.E.g a=2 is original data .If T1 read a=1 that was written by T2.If T2 rollback then a=1 will be rollback to a=2 in DB. But, Now, T1 has a=1 but in DB table it is changed to a=2.

**Scenario2**.If T1 transaction reads data from table A1.If another concurrent transaction(T2) update data on table A1.Then the data that T1 has read is different from table A1.Because T2 has updated the data on table A1.E.g if T1 read a=1 and T2 updated a=2.Then a!=b.

**Scenario 3**.If T1 transaction reads data from table A1 with certain number of rows. If another concurrent transaction (T2) inserts more rows on table A1.The number of rows read by T1 is different from rows on table A1

Scenario 1 is called **Dirty reads.**

Scenario 2 is called **Non-repeatable reads.**

Scenario 3 is called **Phantom reads.**

So, isolation level is the extent to which **Scenario 1, Scenario 2, Scenario 3** can be prevented. You can obtain complete isolation level by implementing locking. That is preventing concurrent reads and writes to the same data from occurring. But it affects performance .The level of isolation depends upon application to application how much isolation is required.

**ISOLATION\_READ\_UNCOMMITTED**: Allows to read changes that haven’t yet been committed. It suffer from Scenario 1, Scenario 2, Scenario 3

**ISOLATION\_READ\_COMMITTED**: Allows reads from concurrent transactions that have been committed. It may suffer from Scenario 2 and Scenario 3. Because other transactions may be updating the data.

**ISOLATION\_REPEATABLE\_READ**: Multiple reads of the same field will yield the same results until it is changed by itself. It may suffer from Scenario 3.Because other transactions may be inserting the data

**ISOLATION\_SERIALIZABLE**: Scenario 1, Scenario 2, Scenario 3 never happens. It is complete isolation. It involves full locking. It affects performance because of locking.