Spring Transaction with Annotation (@Transactional)

The @Transactional annotation is metadata that specifies that an interface, class, or method must have transactional semantics;

Enable the annotation:

```
<tx:annotation-driven proxy-target-class="true">
```

- 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.

The default @Transactional settings are:

- The propagation setting is PROPAGATION_REQUIRED
- The isolation level is ISOLATION_DEFAULT
- The transaction is read/write (readonly)
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any RuntimeException will trigger rollback, and any checked Exception will not

Let us now understand different @Transactional attributes:

- The default is Isolation. DEFAULT
- Most of the times, you will use default unless and until you have specific requirements.
- Informs the transaction (tx) manager that the following isolation level should be used for the current tx. Should be set at the point from where the tx starts because we cannot change the isolation level after starting a tx.

@Transactional(isolation=Isolation.READ_COMMITTED)

A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

@Transactional(isolation=Isolation.READ_UNCOMMITTD)

This isolation level states that a transaction may read data that is still uncommitted by other transactions.

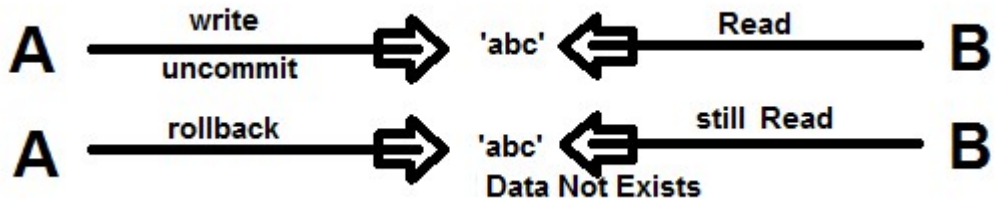
@Transactional(isolation=Isolation.REPEATABLE_READ)

A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

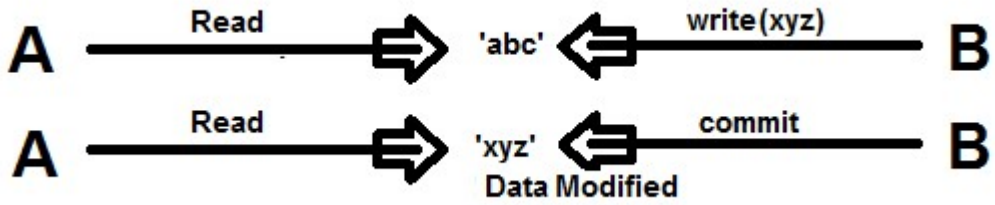
@Transactional(isolation=Isolation. SERIALIZABLE)

A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.

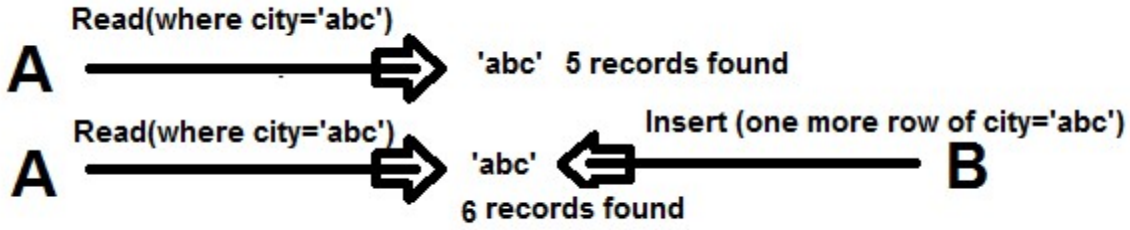
- **Dirty Reads**: Transaction 'A' writes a record. Meanwhile Transaction 'B' reads that same record before Transaction A commits. Later Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.



- **Non-Repeatable Reads**: Transaction 'A' reads some record. Then Transaction 'B' writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.



- **Phantom Reads:** Transaction 'A' reads a range of records. Meanwhile Transaction 'B' inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).



@Transactional(timeout=60)

Informs the tx manager about the time duration to wait for an idle tx before a decision is taken to rollback non-responsive transactions.

@Transactional(propagation=Propagation.REQUIRED)

If not specified, the default propagational behavior is REQUIRED.

Indicates that the target method cannot run without an active tx. If a tx has already been started before the invocation of this method, then it will

continue in the same tx or a new tx would begin soon as this method is called.

@Transactional(propagation=Propagation. REQUIRES_NEW)

Indicates that a new tx has to start every time the target method is called. If already a tx is going on, it will be suspended before starting a new one.

@Transactional(propagation=Propagation. MANDATORY)

Indicates that the target method requires an active tx to be running. If a tx is not going on, it will fail by throwing an exception.

@Transactional(propagation=Propagation. SUPPORTS)

Indicates that the target method can execute irrespective of a tx. If a tx is running, it will participate in the same tx. If executed without a tx it will still execute if no errors.

Methods which fetch data are the best candidates for this option.

@Transactional(propagation=Propagation.NOT_SUPPORTE)

Indicates that the target method doesn't require the transaction context to be propagated.

Mostly those methods which run in a transaction but perform in-memory operations are the best candidates for this option.

@Transactional(propagation=Propagation. NEVER)

Indicates that the target method will raise an exception if executed in a transactional process.

This option is mostly not used in projects.

@Transactional (rollbackFor=Exception.class)

- Default is rollbackFor=RunTimeException.class
- In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction.
- The problem is only with checked exceptions. So this option can be used to declaratively rollback a transaction if Checked Exception occurs.

@Transactional (noRollbackFor=IllegalStateException.class)

- Indicates that a rollback should not be issued if the target method raises this exception.

*Now the last but most important step in transaction management is the **placement of @Transactional annotation**. Most of the times, there is a confusion where should the annotation be placed: at Service layer or DAO layer?*

@Transactional: Service or DAO Layer?

- The Service is the best place for putting @Transactional, service layer should hold the detail-level use case behavior for a user interaction that would logically go in a transaction.
- There are a lot of CRUD applications that don't have any significant business logic for them having a service layer that just passes data through between the controllers and data access objects is not useful. In these cases we can put transaction annotation on Dao.
- Consider another example where your Service layer may call two different DAO methods to perform DB operations. If your first DAO operation failed then other two may be still passed and you will end up inconsistent DB state. Annotating Service layer can save you from such situations.

Program Example:

AccountService Class

```
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import com.swadesibank.transaction.bean.Account;
import com.swadesibank.transaction.dao.IAccountDao;
public class AccountService {

	private IAccountDao accountDao;
	public void setAccountDao(IAccountDao accountDao) {
		this.accountDao = accountDao;
	}
	@Transactional(propagation=Propagation.REQUIRED,readOnly=false)
	public void fundTransfer(final Account fromAccount,final Account toAccount,Double transferAmount){

			System.out.println("tranferring amount:"+transferAmount+" to the accountno:"+toAccount.getAccountno());
			fromAccount.debit(transferAmount);
			toAccount.credit(transferAmount);
			accountDao.update(fromAccount);
			accountDao.update(toAccount);

			System.out.println("Tranfer completed...");
	}
	@Transactional(propagation=Propagation.REQUIRED,readOnly=false,rollbackFor=Exception.class)
	public void fundTransferWithException(final Account fromAccount,final Account toAccount,Double transferAmount) throws Exception{

			System.out.println("tranferring amount:"+transferAmount+" to the accountno:"+toAccount.getAccountno());
			fromAccount.debit(transferAmount);
			toAccount.credit(transferAmount);
			accountDao.update(fromAccount);
			accountDao.update(toAccount);
			System.out.println("Tranfer completed...");//put here debugger point.
			//assume exception occurs at below line [before exception you may see result in table in uncommitted mode]
			throw new Exception("Opps!! some exception occurs!!");

	}
}
```

Bean Definition:

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/tx
 http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
 http://www.springframework.org/schema/context/
 http://www.springframework.org/schema/context/spring-context.xsd">

        <!-- Enable annotation -->
        <tx:annotation-driven transaction-manager="transactionManager"/>
        <!-- Register Data Source Connection with DB -->
        <bean id="myDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/test"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
        </bean>
        <!-- Register Transaction Manager Bean -->
        <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="myDataSource"/>
        </bean>
        <!-- Register Jdbc Template Bean -->
        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="myDataSource"></property>
        </bean>
        <!-- Register DAO class Bean -->
        <bean id="accountDao" class="com.swadesibank.transaction.daoImpl.AccountDaoImpl">
                <property name="jdbcTemplate" ref="jdbcTemplate"/>
        </bean>
        <!-- Register Service class Bean -->
        <bean id="accountService" class="com.swadesibank.transaction.service.AccountService">
                <property name="accountDao" ref="accountDao"/>
        </bean>
</beans>
```