Transactions in Spring

* Transactions in Java, as in general refer to a series of actions that must all complete successfully. Hence, **if one or more action fails, all other actions must back out leaving the state of the application unchanged**. This is necessary to ensure that the integrity of the application state is never compromised.
* At a high level, **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 running method, mainly for starting and committing the transaction.
* What’s 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 has the @Transactional annotation.
* 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(isolation = Isolation.SERIALIZABLE). In serializable isolation mode the database will not allow the data read by transaction to be modified.
* The fact is that **we can’t be sure that an insert or update won’t occur when the *readOnly* flag is set.** This behavior is vendor-dependent, whereas JPA is vendor agnostic.

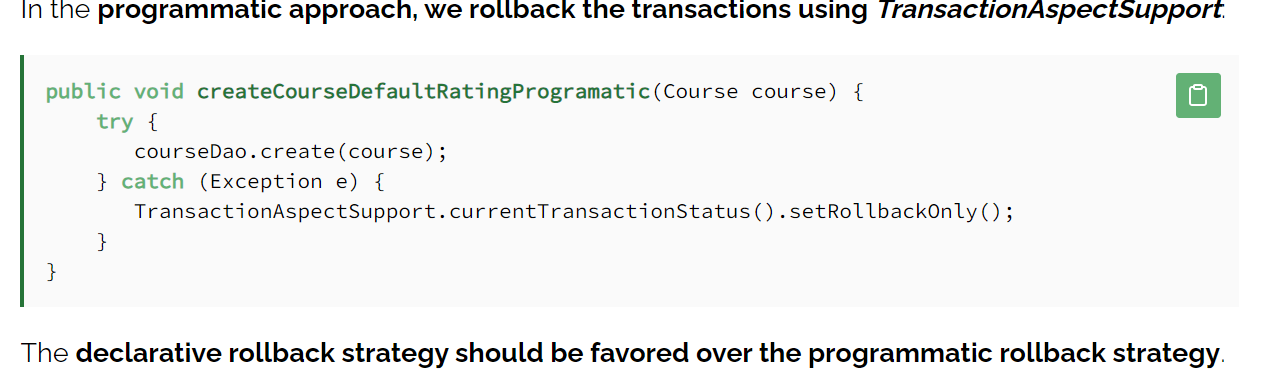
It’s also important to understand that**the *readOnly* flag is only relevant inside a transaction.** If an operation occurs outside of a transactional context, the flag is simply ignored.

Eg: A simple example of that would call a method annotated with:

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

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

* propagation: Specifies how the transactional behavior should propagate. In this case, it supports an existing transaction.
* The @Transactional annotation makes use of the attributes **rollbackFor** or **rollbackForClassName** to rollback the transactions, and the attributes **noRollbackFor** or **noRollbackForClassName** to avoid rollback on listed exceptions.



* In Spring 3.1 and later, you can use the @EnableTransactionManagement annotation in a @Configuration class to enable transactional support.
* If you’re using Spring Boot and have relevant dependencies (such as spring-data-\* or spring-tx), transaction management is enabled by default.

The annotation supports additional configuration:

* Propagation type (how transactions interact with each other)
* Isolation level (transaction isolation behavior)
* Timeout (maximum time for the operation wrapped by the transaction)
* Read-only flag (hint for read-only transactions)
* Rollback rules (conditions for rolling back the transaction)