What does @Transactional do? What is the PlatformTransactionManager?

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:

• Propagation setting is PROPAGATION_REQUIRED.

• Isolation level is ISOLATION_DEFAULT.

• Transaction is read/write.

• Transaction timeout defaults to the default timeout of the underlying transaction system, or to none

if timeouts are not supported.

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

org.springframework.transaction

Interface PlatformTransactionManager

All Known Subinterfaces:

`CallbackPreferringPlatformTransactionManager`, `ResourceTransactionManager`

All Known Implementing Classes:

`AbstractPlatformTransactionManager`, `CciLocalTransactionManager`, `DataSourceTransactionManager`, `HibernateTransactionManager`, `HibernateTransactionManager`, `HibernateTransactionManager`, `JdoTransactionManager`, `JmsTransactionManager`, `JpaTransactionManager`, `JtaTransactionManager`, `WebLogicJtaTransactionManager`, `WebSphereUowTransactionManager`

```
public interface PlatformTransactionManager
```

This is the central interface in Spring's transaction infrastructure. Applications can use this directly, but it is not primarily meant as API: Typically, applications will work with either TransactionTemplate or declarative transaction demarcation through AOP.

For implementors, it is recommended to derive from the provided `AbstractPlatformTransactionManager` class, which pre-implements the defined propagation behavior and takes care of transaction synchronization handling. Subclasses have to implement template methods for specific states of the underlying transaction, for example: begin, suspend, resume, commit.

The default implementations of this strategy interface are `JtaTransactionManager` and `DataSourceTransactionManager`, which can serve as an implementation guide for other transaction strategies.

Since:

16.05.2003

Author:

Rod Johnson, Juergen Hoeller

See Also:

`TransactionTemplate`, `TransactionInterceptor`, `TransactionProxyFactoryBean`

Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	Method and Description
void	`commit(TransactionStatus status)` Commit the given transaction, with regard to its status.
`TransactionStatus`	`getTransaction(TransactionDefinition definition)`

	Return a currently active transaction or create a new one, according to the specified propagation behavior.
void	**rollback**(TransactionStatus status) Perform a rollback of the given transaction.

Method Detail

getTransaction

```
TransactionStatus getTransaction(TransactionDefinition definition)
                          throws TransactionException
```

Return a currently active transaction or create a new one, according to the specified propagation behavior.

Note that parameters like isolation level or timeout will only be applied to new transactions, and thus be ignored when participating in active ones.

Furthermore, not all transaction definition settings will be supported by every transaction manager: A proper transaction manager implementation should throw an exception when unsupported settings are encountered.

An exception to the above rule is the read-only flag, which should be ignored if no explicit read-only mode is supported. Essentially, the read-only flag is just a hint for potential optimization.

Parameters:

`definition - TransactionDefinition instance (can be null for defaults), describing propagation behavior, isolation level, timeout etc.`

Returns:

`transaction status object representing the new or current transaction`

Throws:

`TransactionException - in case of lookup, creation, or system errors`

`IllegalTransactionStateException - if the given transaction definition cannot be executed (for example, if a currently active transaction is in conflict with the specified propagation behavior)`

See Also:

`TransactionDefinition.getPropagationBehavior(), TransactionDefinition.getIsolationLevel(), TransactionDefinition.getTimeout(), TransactionDefinition.isReadOnly()`

commit

```
void commit(TransactionStatus status)
     throws TransactionException
```

Commit the given transaction, with regard to its status. If the transaction has been marked rollback-only programmatically, perform a rollback.

If the transaction wasn't a new one, omit the commit for proper participation in the surrounding transaction. If a previous transaction has been suspended to be able to create a new one, resume the

previous transaction after committing the new one.

Note that when the commit call completes, no matter if normally or throwing an exception, the transaction must be fully completed and cleaned up. No rollback call should be expected in such a case.

If this method throws an exception other than a TransactionException, then some before-commit error caused the commit attempt to fail. For example, an O/R Mapping tool might have tried to flush changes to the database right before commit, with the resulting DataAccessException causing the transaction to fail. The original exception will be propagated to the caller of this commit method in such a case.

Parameters:

`status - object returned by the getTransaction method`

Throws:

`UnexpectedRollbackException - in case of an unexpected rollback that the transaction coordinator initiated`

`HeuristicCompletionException - in case of a transaction failure caused by a heuristic decision on the side of the transaction coordinator`

`TransactionSystemException - in case of commit or system errors (typically caused by fundamental resource failures)`

`IllegalTransactionStateException - if the given transaction is already completed (that is, committed or rolled back)`

`TransactionException`

See Also:

`TransactionStatus.setRollbackOnly()`

rollback

```
void rollback(TransactionStatus status)
       throws TransactionException
```

Perform a rollback of the given transaction.

If the transaction wasn't a new one, just set it rollback-only for proper participation in the surrounding transaction. If a previous transaction has been suspended to be able to create a new one, resume the previous transaction after rolling back the new one.

Do not call rollback on a transaction if commit threw an exception. The transaction will already have been completed and cleaned up when commit returns, even in case of a commit exception. Consequently, a rollback call after commit failure will lead to an IllegalTransactionStateException.

Parameters:

`status - object returned by the getTransaction method`

Throws:

`TransactionSystemException - in case of rollback or system errors (typically caused by fundamental resource failures)`

`IllegalTransactionStateException - if the given transaction is already completed (that is, committed or rolled back)`

```
TransactionException
```