SPRING - TRANSACTION MANAGEMENT

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as **ACID**:

- **Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.
- **Isolation:** There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
- **Durability:** Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all the four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows:

- Begin the transaction using *begin transaction* command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs requires an application server, but Spring transaction management can be implemented without a need of application server.

Local vs. Global Transactions

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic vs. Declarative

Spring supports two types of transaction management:

- Programmatic transaction management: This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
- Declarative transaction management: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by the *org.springframework.transaction.PlatformTransactionManager* interface, which is as follows:

```
public interface PlatformTransactionManager {
   TransactionStatus getTransaction(TransactionDefinition definition);
   throws TransactionException;
   void commit(TransactionStatus status) throws TransactionException;
   void rollback(TransactionStatus status) throws TransactionException;
}
```

S.N.	Method & Description
1	**TransactionStatus getTransaction*TransactionDefinitiondefinition*** This method returns a currently active transaction or create a new one, according to the specified propagation behavior.
2	**void commit*TransactionStatusstatus*** This method commits the given transaction, with regard to its status.
3	**void rollback*TransactionStatusstatus*** This method performs a rollback of the given transaction.

The *TransactionDefinition* is the core interface of the transaction support in Spring and it is defined as below:

```
public interface TransactionDefinition {
   int getPropagationBehavior();
   int getIsolationLevel();
   String getName();
   int getTimeout();
   boolean isReadOnly();
}
```

S.N.	Method & Description
1	**int getPropagationBehavior** This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT.
2	**int getIsolationLevel** This method returns the degree to which this transaction is isolated from the work of

	other transactions.
3	**String getName** This method returns the name of this transaction.
4	**int getTimeout** This method returns the time in seconds in which the transaction must complete.
5	**boolean isReadOnly** This method returns whether the transaction is read-only.

Following are the possible values for isolation level:

S.N.	Isolation & Description
1	**TransactionDefinition.ISOLATION_DEFAULT** This is the default isolation level.
2	**TransactionDefinition.ISOLATION_READ_COMMITTED** Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
3	**TransactionDefinition.ISOLATION_READ_UNCOMMITTED** Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
4	**TransactionDefinition.ISOLATION_REPEATABLE_READ** Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5	**TransactionDefinition.ISOLATION_SERIALIZABLE** Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

Following are the possible values for propagation types:

S.N.	Propagation & Description
1	**TransactionDefinition.PROPAGATION_MANDATORY** Support a current transaction; throw an exception if no current transaction exists.

2	**TransactionDefinition.PROPAGATION_NESTED** Execute within a nested transaction if a current transaction exists.
3	**TransactionDefinition.PROPAGATION_NEVER** Do not support a current transaction; throw an exception if a current transaction exists.
4	**TransactionDefinition.PROPAGATION_NOT_SUPPORTED** Do not support a current transaction; rather always execute non-transactionally.
5	**TransactionDefinition.PROPAGATION_REQUIRED** Support a current transaction; create a new one if none exists.
6	**TransactionDefinition.PROPAGATION_REQUIRES_NEW** Create a new transaction, suspending the current transaction if one exists.
7	**TransactionDefinition.PROPAGATION_SUPPORTS** Support a current transaction; execute non-transactionally if none exists.
8	**TransactionDefinition.TIMEOUT_DEFAULT** Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

The *TransactionStatus* interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
   boolean isNewTransaction();
   boolean hasSavepoint();
   void setRollbackOnly();
   boolean isRollbackOnly();
   boolean isCompleted();
}
```

S.N.	Method & Description
1	**boolean hasSavepoint** This method returns whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint.
2	**boolean isCompleted**

This method returns whether this transaction is completed, that is, whether it has already been committed or rolled back.

3

boolean isNewTransaction

This method returns true in case the present transaction is new.

4

boolean isRollbackOnly

This method returns whether the transaction has been marked as rollback-only.

5

void setRollbackOnly

This method sets the transaction rollback-only.