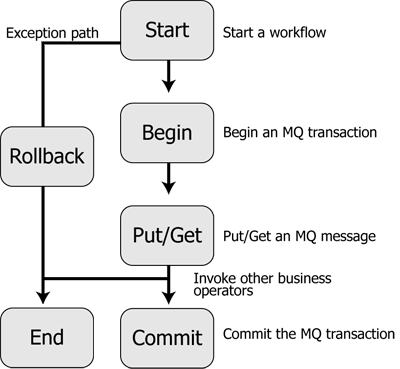
1. [Transaction Propagation and Isolation](#transaction_propagation)

<http://javadecodedquestions.blogspot.com/2013/03/java-investment-bank-question.html>

transaction Management is always favourite interview questions for any senior software [java developer](javascript:void(0);). Transaction management can be related to multi resource application like [web app](javascript:void(0);) with database interaction or [messaging system](javascript:void(0);) with database interaction. Classic question related to transaction management is 'if you have received the message from JMS provider and Database insert fails then how you are going to manage transaction?'. Below diagram shows generally we start the transaction at the start of the processing and rollback / commit changes on basic of processing result. If there are no exception then we commit the transaction and if there were exception then we rollback the transaction.

[](http://docs.oracle.com/cd/E14981-01/wli/docs1031/controls/wwimages/mq_flow2.gif)

  Best way to prepare to confirm how you are managing transaction in your current application. In spring based application we define  transaction requirement at each service. In EJBs we can use BMT [ Bean managed transaction ] or CMT [ Container managed transaction] and in case of BMT mainly we use UserTransaction and We define the scope and execution of transaction in bean.

Any resource if it capable of handling transaction, it will support ACID properties.

ACID Property:

This property needs to be fulfil by all RDBMS.  Any database operation should fulfill these requirements.

**Atomicity**

     Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. Modification on the data in the database either fail or succeed. The beginning of such a modification starts with a transaction and ends when a transaction finishes

**Consistency**

    The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints,cascades, triggers, and any combination thereof.

**Isolation**

    The isolation property ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially, i.e. one after the other. Each transaction has to execute in total isolation i.e. if T1 and T2 are being executed concurrently then both of them should remain unaware of each other's presence.

**Durability**

    Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter).

Transaction Isolation levels

     Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

**Database anomalies**

    Database anomalies are generated results that seem incorrect when looked at from the scope of a single transaction, but are correct when looked at from the scope of all transactions. The different types of database anomalies are described as follows:

• **Dirty reads [Read uncommitted]**occur when:

     ◦Transaction A inserts a row into a table.

     ◦Transaction B reads the new row.

     ◦Transaction A rolls back.

     ◦Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

• **Non-Repeatable reads [Read committed]**occur when:

   ◦ Transaction A reads a row.

   ◦ Transaction B changes the row.

   ◦ Transaction A reads the same row a second time and gets the new results.

• **Phantom reads**occur when:

  ◦ Transaction A reads all rows that satisfy a WHERE clause on an SQL query.

  ◦ Transaction B inserts an additional row that satisfies the WHERE clause.

  ◦ Transaction A re-evaluates the WHERE condition and picks [up the](javascript:void(0);) additional row.

Transaction isolation level expose the application to the allowable database anomolies at the prescribed levels due to its locking strategies.

### JDBC transaction isolation levels

    There are five levels of transaction isolation in the [IBM](javascript:void(0);) Developer Kit for Java JDBC API. Listed from least to most restrictive, they are as follows:

**TRANSACTION\_NONE**

     This is a special constant indicating that the [JDBC driver](javascript:void(0);) does not support transactions.

**TRANSACTION\_READ\_UNCOMMITTED**

    This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

**TRANSACTION\_READ\_COMMITTED**

    This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

**TRANSACTION\_REPEATABLE\_READ**

    This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

**TRANSACTION\_SERIALIZABLE**

    Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

### Transaction Propagation

Propagation means how the service will behave in case of new or no or nested transactions. These are **Transaction Propagation** available in Spring:

[MANDATORY](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  
          Support a current transaction, throw  an exception if none exists.

[NESTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NESTED)  
          Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

[NEVER](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NEVER)  
          Execute non-transactionally, throw an exception if a transaction exists.

[NOT\_SUPPORTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  
          Execute non-transactionally, suspend the current transaction if one exists.

[REQUIRED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  
          Support a current transaction, create a new one if none exists.

[REQUIRES\_NEW](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  
          Create a new transaction, suspend the current transaction if one exists.

[SUPPORTS](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  
          Support a current transaction, execute non-transactionally if none exists.

The interface Connection includes five values that represent the transaction isolation levels you can use in JDBC:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Isolation Level** | **Transactions** | **Dirty Reads** | **Non-Repeatable Reads** | **Phantom Reads** |
| TRANSACTION\_NONE | Not supported | *Not applicable* | *Not applicable* | *Not applicable* |
| TRANSACTION\_READ\_COMMITTED | Supported | Prevented | Allowed | Allowed |
| TRANSACTION\_READ\_UNCOMMITTED | Supported | Allowed | Allowed | Allowed |
| TRANSACTION\_REPEATABLE\_READ | Supported | Prevented | Prevented | Allowed |
| TRANSACTION\_SERIALIZABLE | Supported | Prevented | Prevented | Prevented |
|  |  |  |  |  |

**Spring Transaction Management:**

    In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration in Spring. Declaring transaction semantics directly in the [Java source code](javascript:void(0);) puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the @Transactional annotation is best illustrated with an example, after which all of the details will be explained. Consider the following

*Class definition:*

*// the service class that we want to make transactional*

***@Transactional***

*public class DefaultFooService implements FooService {*

*Foo getFoo(String fooName);*

*Foo getFoo(String fooName, String barName);*

*void insertFoo(Foo foo);*

*void updateFoo(Foo foo);*

*}*

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely one line of XML configuration, like so:

|  |
| --- |
| *<!-- from the file 'context.xml' -->*  **<?xml version="1.0" encoding="UTF-8"?>**  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">  *<!-- this is the service object that we want to make transactional -->*  <bean id="fooService" class="x.y.service.DefaultFooService"/>  *<!-- enable the configuration of transactional behavior based on annotations -->*  <tx:annotation-driven transaction-manager="txManager"/>  *<!-- a PlatformTransactionManager is still required -->*  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  *<!-- (this dependency is defined somewhere else) -->*  <property name="dataSource" ref="dataSource"/>  </bean>  *<!-- other <bean/> definitions here -->*  </beans> |

#### **Use Spring to manage the transaction**

Spring supports two types of transaction management:

1. **Programmatic transaction management**: This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
2. **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.

Transaction isolation level expose the application to the allowable database anomolies at the prescribed levels due to its locking strategies.

### JDBC transaction isolation levels

    There are five levels of transaction isolation in the IBM Developer Kit for Java JDBC API. Listed from least to most restrictive, they are as follows:

**TRANSACTION\_NONE**

     This is a special constant indicating that the JDBC driver does not support transactions.

**TRANSACTION\_READ\_UNCOMMITTED**

    This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

**TRANSACTION\_READ\_COMMITTED**

    This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

**TRANSACTION\_REPEATABLE\_READ**

    This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

**TRANSACTION\_SERIALIZABLE**

    Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

### Transaction Propagation

Propagation means how the service will behave in case of new or no or nested transactions. These are **Transaction Propagation** available in Spring:

[MANDATORY](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  
          Support a current transaction, throw  an exception if none exists.

[NESTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NESTED)  
          Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

[NEVER](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NEVER)  
          Execute non-transactionally, throw an exception if a transaction exists.

[NOT\_SUPPORTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  
          Execute non-transactionally, suspend the current transaction if one exists.

[REQUIRED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  
          Support a current transaction, create a new one if none exists.

[REQUIRES\_NEW](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  
          Create a new transaction, suspend the current transaction if one exists.

[SUPPORTS](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  
          Support a current transaction, execute non-transactionally if none exists.

**Spring Transaction Management:**

    In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration in Spring. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the @Transactional annotation is best illustrated with an example, after which all of the details will be explained. Consider the following