Search Tutorials

# Spring Boot - Transaction Isolation Tutorial



In previous tutorial - Spring Boot Transaction Management Example (/spring/boot-transaction) we saw what are transactions and implemented declarative transaction management. Also in previous tutorial we had implemented the various transaction propagation types. (/spring/boot-transaction-propagation) In this tutorial we will be understanding what is transaction isolation and its different types.

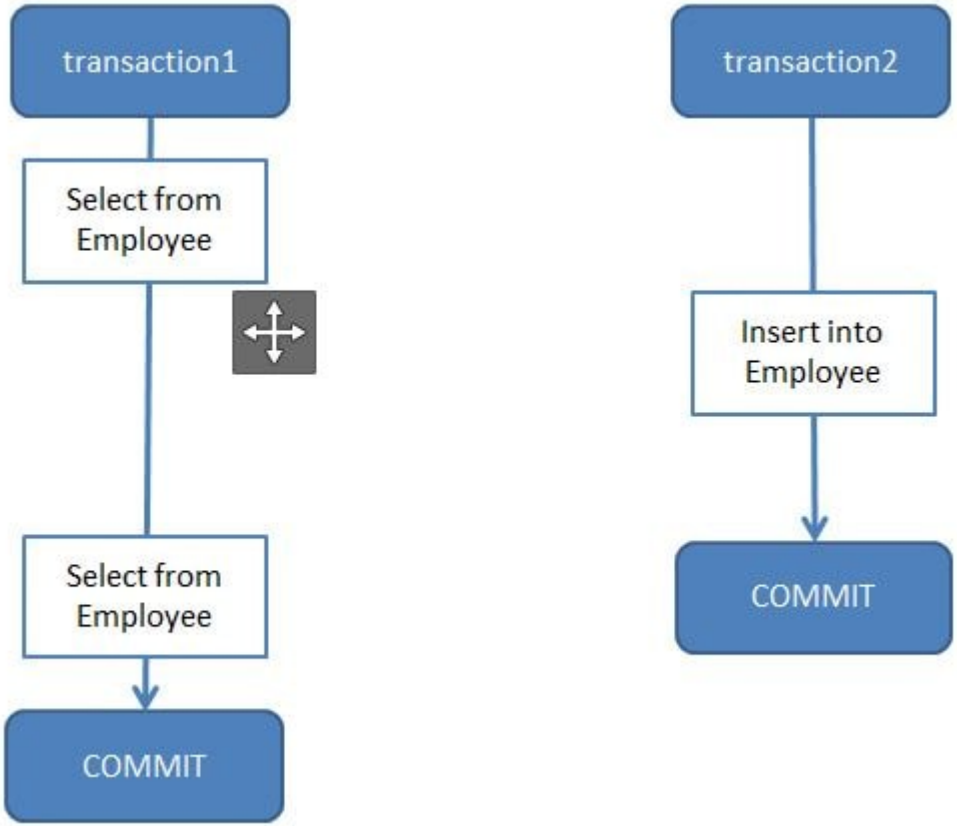
## Lets Begin-

### What is Transaction Isolation?



Transaction isolation defines the database state when two transactions concurrently act on the same database entity. It involves locking of database records. So it describes the **behaviour or state of the database when one transaction is working on database entity and then some other concurrent transaction tries to simultaneously access/edit the same database entity.**

The ANSI/ISO standard defines four isolation levels. **Isolation is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties.** So transaction isolation level is not something specific to Spring Framework. Using Spring we can change the isolation level to suit our business logic.




Before implementing Isolation Level using Spring, let us first understand isolation levels at Database level.

We will be create a table name employee and using this table try understand the isolation levels-

```
CREATE TABLE employee (
  empId VARCHAR(10) NOT NULL,
  empName VARCHAR(100) NOT NULL
);
```

Some of the SQL commands I have used for implementing Isolation Levels are

```
//Show existing transaction isolation level if mysql version >= 8
SELECT @@TRANSACTION_ISOLATION;

//Show existing transaction isolation level if mysql version < 8
SELECT @@TX_ISOLATION;

//Set transaction isolation level to serializable. Using same syntax
//we can set it to other isolation level.
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

//By default auto commit is enabled for mysql transaction. So we will disable it.
SET AUTOCOMMIT=0;

//Start transaction
BEGIN

//Commit transaction
COMMIT
```

The following are the types of Transaction Isolation Levels-

- **SERIALIZABLE**

  If two transactions are executing concurrently then it is as if the transactions get executed serially i.e the first transaction gets committed only then the second transaction gets executed. This is **total isolation**. So a running transaction is never affected by other transactions. However this may cause issues as **performance**

**will be low and deadlock might occur**.

| Transaction1 | Transaction2 |
| --- | --- |
| Begin Transaction1. On select we get 3 records | |

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@TX_ISOLATION;
+----------------+
| @@TX_ISOLATION |
+----------------+
| SERIALIZABLE   |
+----------------+
1 row in set, 1 warning (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
+-------+---------+
3 rows in set (0.00 sec)
```

Begin Transaction2. Try inserting a record. It will not allow to insert

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@TX_ISOLATION;
+----------------+
| @@TX_ISOLATION |
+----------------+
| SERIALIZABLE   |
+----------------+
1 row in set, 1 warning (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into employee(empId,empName)values('emp4','emp4');
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql>
```

On doing a select again we fetch 3 records. The

transaction2 is not allowed to insert a new record till transaction1 completes.

```
mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
+-------+---------+
3 rows in set (0.00 sec)
```

- **REPEATABLE_READ**

  If two transactions are executing concurrently - **till the first transaction is committed the existing records cannot be changed by second transaction but new records can be added.** After the second transaction is committed, the new added records get reflected in first transaction which is still not committed. For MySQL the default isolation level is REPEATABLE_READ.

  However the REPEATABLE READ isolation level behaves differently when using mysql. When using MYSQL we are not able to see the newly added records that are committed by the second transaction.

| Transaction1 | Transaction2 |
|---|---|
| Begin Transaction1. On select we get 2 records | |

```
mysql> SELECT @@TX_ISOLATION;
+-----------------+
| @@TX_ISOLATION  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
```

```
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
+-------+---------+
2 rows in set (0.00 sec)
```

Begin Transaction2. Insert a record and do commit.

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@TX_ISOLATION;
+-----------------+
| @@TX_ISOLATION  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql> begin
    -> ;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into employee(empId,empName)values('emp3','emp3');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

On doing a select again we fetch 2 records. The record inserted in transaction2 is not reflected

```
mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
+-------+---------+
2 rows in set (0.00 sec)
```

- **READ_COMMITTED**

  If two transactions are executing concurrently - **before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction.** After the second transaction is committed, the newly added and also updated records get reflected in first

transaction which is still not committed.

| Transaction1 | Transaction2 |
|---|---|
| Begin Transaction1. On select we get 4 records<br>`mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> set autocommit=0;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> begin;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> select * from employee;`<br>`+-------+---------+`<br>`\| empId \| empName \|`<br>`+-------+---------+`<br>`\| emp1 \| emp1 \|`<br>`\| emp2 \| emp2 \|`<br>`\| emp3 \| emp3 \|`<br>`\| emp4 \| emp4 \|`<br>`+-------+---------+`<br>`4 rows in set (0.00 sec)` | |
| | Begin Transaction2. Insert a record and do commit.<br>`mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> set autocommit=0;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> begin;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> insert into employee(empId,empName)values('emp5','emp5');`<br>`Query OK, 1 row affected (0.00 sec)`<br><br>`mysql> commit;`<br>`Query OK, 0 rows affected (0.01 sec)` |
| On doing a select again we fetch 5 records. The record inserted in transaction2 is reflected in transaction1 only after transaction 2 is committed<br>`mysql> select * from employee;`<br>`+-------+---------+` | |

```
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
| emp4  | emp4    |
| emp5  | emp5    |
+-------+---------+
5 rows in set (0.00 sec)
```

- **READ_UNCOMMITTED**

  If two transactions are executing concurrently - before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction. **Even if the second transaction is not committed the newly added and also updated records get reflected** in first transaction which is still not committed.

| Transaction1 | Transaction2 |
|---|---|
| Begin Transaction1. On select we get 3 records<br>`mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> set autocommit=0;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> begin;`<br>`Query OK, 0 rows affected (0.00 sec)`<br><br>`mysql> select * from employee;`<br>`+-------+---------+`<br>`\| empId \| empName \|`<br>`+-------+---------+`<br>`\| emp1 \| emp1 \|`<br>`\| emp2 \| emp2 \|`<br>`\| emp3 \| emp3 \|`<br>`+-------+---------+`<br>`3 rows in set (0.00 sec)` | |
| | Begin Transaction2. Insert a record and do not do a commit.<br>`mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;`<br>`Query OK, 0 rows affected (0.00 sec)` |

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@TX_ISOLATION;
+-----------------+
| @@TX_ISOLATION  |
+-----------------+
| READ-UNCOMMITTED |
+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into employee(empId,empName)values('emp4','emp4');
Query OK, 1 row affected (0.00 sec)
```

On doing a select again we fetch 4 records. The record inserted in transaction2 is not committed, but still gets reflected in transaction1.

```
mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
| emp4  | emp4    |
+-------+---------+
4 rows in set (0.00 sec)
```

## Summary

- **Dirty Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A modifies a record but not commits it. Transaction B reads this record but then Transaction A again rollbacks the changes for the record and commits it. So Transaction B has a wrong value.
- **Non-Repeatable Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B modifies these records before transaction A has been committed. So if Transaction A again reads these records they will be different. So same select statements result in different existing records.

- **Phantom Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B adds more such records before transaction A has been committed. So if Transaction A again reads there will be more records than the previous select statement. So same select statements result in different number records to be displayed as new records also get added.

| Isolation Level | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|---|---|---|---|
| **SERIALIZABLE** | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially |

| | | | |
|---|---|---|---|
| **REPEATABLE_READ** | This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value. | This scenario is not possible since any record can be changed only after a transaction has been committed. So multiple select statements before transaction commit will always return same existing records. | This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place. |
| **READ_COMMITTED** | This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value. | This scenario is possible as other transactions can modify existing records even if first transaction commit has not taken place. | This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place. |

| | | | |
|---|---|---|---|
| **READ_UNCOMMITTED** | This scenario is possible as any record can be read by other transactions even if the first transaction is not committed. So if first transaction rollbacks the record changes then other transactions will have wrong values | This scenario is possible since any record can be changed even if a transaction is not committed. | This scenario is possible as any record can be inserted even if a transaction is not committed. |

## Implement Transaction Isolation using Spring Boot

When using Transaction Isolation with Spring Boot, the default transaction isolation taken is that of the underlying database. So for our spring boot application the default transaction isolation will be REPEATABLE_READ since we are using MySQL database. In previous tutorial - Spring Boot Transaction Management Example (/spring/boot-transaction) we saw what are transactions and implemented declarative transaction management. We will be modifying this code. We can change the transaction isolation level as follows-

```
package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;

@Service
public class OrganzationServiceImpl implements OrganizationService {

        @Autowired
        EmployeeService employeeService;

        @Autowired
        HealthInsuranceService healthInsuranceService;

        @Override
        // Using Transactional annotation we can define any isolation level supported by
        @Transactional(isolation = Isolation.SERIALIZABLE)
        public void joinOrganization(Employee employee, EmployeeHealthInsurance employee
                employeeService.insertEmployee(employee);
```

```
                healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthIns
        }

        @Override
        @Transactional
        public void leaveOrganization(Employee employee, EmployeeHealthInsurance employe
                employeeService.deleteEmployeeById(employee.getEmpId());
                healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthI
        }
}
```

## Download Source Code

Download it -

Spring Boot Transaction Isolations (/zip/spring/boot/boot-jdbc-iso.rar)

## Popular Posts

- Spring Boot Interview Questions (/spring/SpringBootInterviewQuestions)