http://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial
https://www.tutorialspoint.com/spring/spring_transaction_management.htm

Transcation is an activity or group of activites that are performed as single unit of work,. The characteristic of a transaction is the either all the activities that are part of the transcation is the either all the activites that are that are part of the transcation are performed or none performed,
In other words, even if one of the activity fail then all other activites are cancelled and the system comes back to the sate it was in when the transaction was started.

Characteristics of a transcation(ACID)
The software industry coined the acronym ACID characteristic that a transaction must have.ACID stands for **Atomic**,**Consistent**,**Isolated** and **Durable**

Atomic-Atomic sys that either all activities of the transaction occure or none occure. i.e. even if one of the activity in the group fails that the other activites are cancelled(rolled back)

Consistent-- Once the transaction is complete the system in put back into a state that is propety defined. From a database point of view, consisten also means that all none of the database constraints are violated. So even if you dont commit a transcation, at no point should a foreign key constraint be violated.
This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.

Isolated- the transaction allow multiple people to work on the same data in way that one transcation does not affect the data of the rest of the system. Therefore two transcation can occur simultaneously without dirty reads, this is generally accompished by locking the rows of the database or the database table.
There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.

Durable-- 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.

Itroduction
Transcation is isolation level is a concept that is not exclusive to spring framework . It is applied to transactions in general and is directly with ACID transaction properties, Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and alos how and when that changed data become availabe to other transcation when we define a transcation using the spring framework we are also able to configure in which isolation level that same transcation will be executed.

Usage Example

Using the **@Transactional** annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level

Introduction

Transaction isolation level is a concept that is not exclusive to the Spring framework. It is applied to transactions in general and is directly related with the ACID transaction properties. Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

Usage example

Using the **@Transactional** annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level:

Isolation level in a transactional method

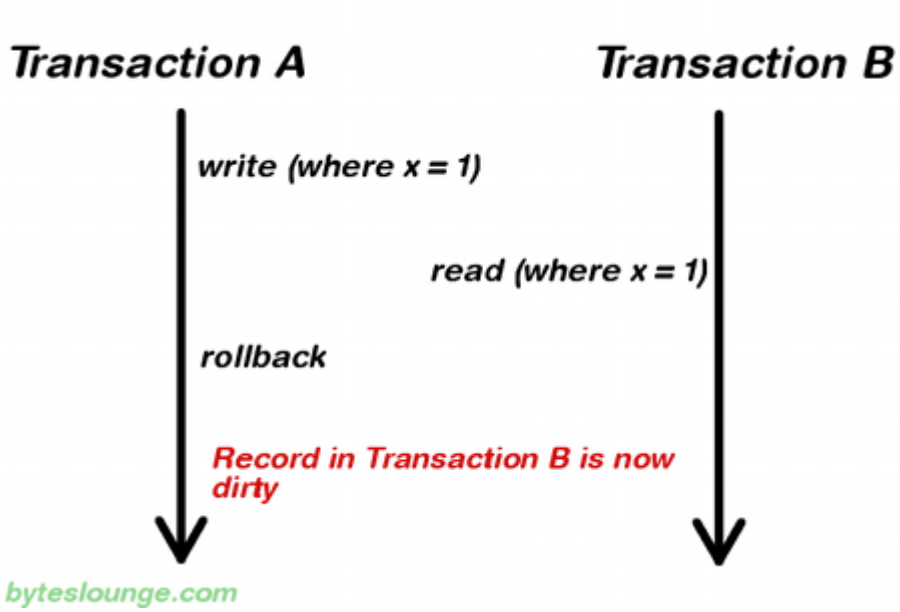
```
@Autowired

private TestDAO testDAO;



@Transactional(isolation=Isolation.READ_COMMITTED)

public void someTransactionalMethod(User user) {



   // Interact with testDAO



}
```

We are defining this method to be executed in a transaction which isolation level is **READ_COMMITTED**. We will see each isolation level in detail in the next sections.

READ_UNCOMMITTED

READ_UNCOMMITTED isolation level states that a transaction **may** read data that is still **uncommitted** by other transactions. This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like **dirty reads**. Let's see the following image:

Dirty read



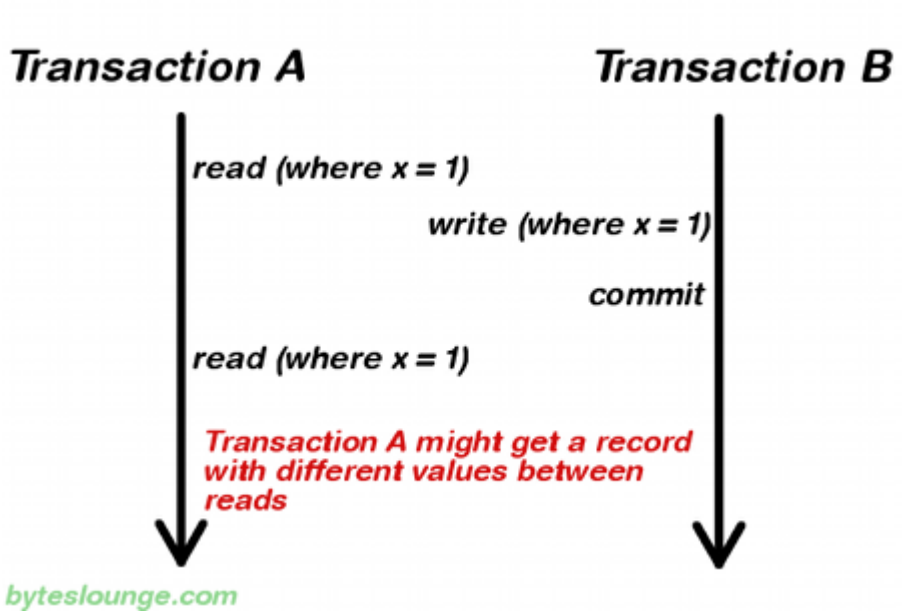
In this example **Transaction A** writes a record. Meanwhile **Transaction B** reads that same record before **Transaction A** commits. Later **Transaction A** decides to rollback and now we have changes in **Transaction B** that are inconsistent. This is a **dirty read**. **Transaction B** was running in **READ_UNCOMMITTED** isolation level so it was able to read **Transaction A** changes before a commit occurred.

Note: READ_UNCOMMITTED is also vulnerable to **non-repeatable reads** and **phantom reads**.

We will also see these cases in detail in the next sections.

READ_COMMITTED isolation level states that a transaction can't read data that is **not** yet committed by other transactions. This means that the **dirty read** is no longer an issue, but even this way other issues may occur. Let's see the following image:

Non-repeatable read



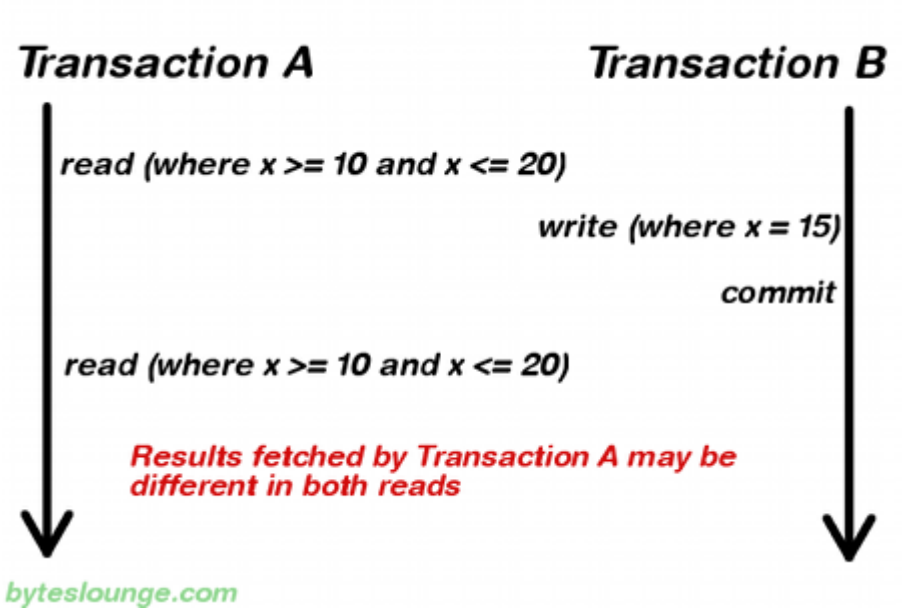
In this example **Transaction A** reads some record. Then **Transaction B** writes that same record and commits. Later **Transaction A**reads that same record again and may get different values because **Transaction B** made changes to that record and committed. This is a **non-repeatable read**.

> **Note:** READ_COMMITTED is also vulnerable to **phantom reads**. We will also see this case in detail in the next section.

REPEATABLE_READ

REPEATABLE_READ isolation level states that if a transaction reads one record from the database multiple times the result of all those reading operations must always be the same. This eliminates both the **dirty read** and the **non-repeatable read** issues, but even this way other issues may occur. Let's see the following image:

Phantom read



In this example **Transaction A** reads a **range** of records. Meanwhile **Transaction B** inserts a new record in the same range that **Transaction A** initially fetched and commits. Later **Transaction A** reads the same range again and will also get the record that **Transaction B** just inserted. This is a **phantom read**: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

SERIALIZABLE

SERIALIZABLE isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (**read**, **range** and **write** locking) so they appear as if they were executed in a serialized way. This leads to a scenario where **none**of the issues mentioned above may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

DEFAULT

DEFAULT isolation level, as the name states, uses the default isolation level of the datastore we are actually connecting from our application.

Summary

To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

ISOLATION_RED_UNCOMMITTED: Allow to read changes that haven't yet been committed. It suffer from Dirty reads, Nonrepeatable read and Phantom read

ISOLATION_RED_COMMITTED: Allows read from concurrent transactions that have been committed. It may suffer from Nonrepeatable read and phantom read Because other transactions may be updating data.

ISOLATION_REPEATABLE_READ: Mutiple read of the same field will yield the same results untill it is changed by itself. It may suffer from phantom read Because other transactions may be inserting the data.

ISOLATION_SERIALIZABLE: SERIALIZABLE isolation level is the most restrictive of all isolation levels. Transaction are executed with locking at all levles (read, range and write locking) so they appear as if they were executed in a serialized way. This leads to a scenario where none of the issues mentioned above may occur , but in the other way we don't allow transaction concurrency and consequentily introduce a performance penalty.

Different propagation behaviors provided by spring

1 **REQUIRED** behavior

Spring **REQUIRED** behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the spring container will create a new one, if multiple methods configured as **REQIRED** behavior are called in a nested way they will be assigned **distrinct logical transactions** but they will all share the **same phsical transaction ,** in short this means that if an inner method causes a transaction to rollback,the outer method will fail to commit and will also rollback the transaction, Let's see an example

Note :

that the inner method throws a **RuntimeException** and is an annotated with **REQUIRED** behavior. This means that it will use the **same** transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

Example

OuterBean

```
@Autowired

private TestDAO testDAO;

	@Autowired
	private InnerBean innerBean;

	@Override
	@Transactional(propagation=Propagation.REQUIRED)
	public void testRequired(User user) {

		testDAO.insertUser(user);
		try {
			innerBean.testRequired();
		}catch(Exception e) {


		}
```

Inner Bean

```
@Override

	@Transactional(propagation=Propagation.REQUIRED)

	public void testRequired() {

		throw RuntimeException("Rollback this transaction");

	}
```

2REQUIRES_NEW behavior

REQUIRES_NEW behavior means that a new physical transaction will always be created by the container. In other words the inner transaction may commit or rollback independently of the outer transaction i.e. the outer transaction will not be affected by the inner transaction result: they will run in **distinct physical transaxtions**

outerBean

```
@Autowired

	private TestDAO testDAO;

	@Autowired
	private InnerBean innerBean;
	@Override
	@Transactional(propagation=Propogation.REQUIRED)
	public void testRequiresNew(User user) {

		testDAO.insertUser(user);

		try {
			innerBean.testRequiresNew();
		}catch(Exception e) {

		}

	}
```

InnerBean

```
@Override

	@Transactional(propagation=Propagation.REQUIRES_NEW)

	public void testRequired() {

		throw RuntimeException("Rollback this transaction");

	}
```

The inner method is annotated with REQUIRES_NEW and throws a RunTimeException so it will set its transaction to rollback but will not affect the outer transaction. The outer transaction is paused inner transaction starts and then resumes after the inner transaction concluded. The run independently of each other so the outer transaction commit successfully.

3 NESTED behavior

The **NESTED** behavior makes nested Spring transactions to user the same physical transaction but sets **savepoints** between nested invocations so inner transactions may also rollback independently of outer transactions. This may be familiar to JDBC aware developers as the savepoints are achieved with JDBC savepoints, so this behavior should only be used with Spring JDBC managed transactions

4 MANDATORY behavior

the MANDATORY behavior states that an existing opened transaction must already exists. If no transaction opend then exception will be thrown by the container.

5 NEVER behavior

the **NEVER** behavior states that an existing opened transaction must be not already exists. if a transaction exists an exception will be thrown by the container.

6 NOT_SUPPORTED behavior

the **NOT_SUPPORTED** behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.

7 SUPPORTED behavior

The **SUPPORTED** behavior will execute in the scope of a transaction if an opened transaction already exists. If there isn't an already opened transaction the method will execute anyway but in a no-transactional way.

Example default transaction management

----------XML----------

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:aop="http://www.springframework.org/schema/aop"

      xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd">


      <context:component-scan base-package="com.infotech.dao" />
      <context:component-scan base-package="com.infotech.service" />
```

```
	<beans:bean id="bankService"
class="com.infotech.service.impl.BankServiceImpl">
		<beans:property name="bankDao" ref="bankDao"></beans:property>
		<beans:property name="transactionTemplate"
ref="transactionTemplate"></beans:property>
	</beans:bean>



	<beans:bean id="bankDao" class="com.infotech.dao.impl.BankDAOImpl">
		<beans:property name="jdbcTemplate"
ref="jdbcTemplate"></beans:property>
	</beans:bean>


	<beans:bean id="dataSource"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<beans:property name="driverClassName" value="${db.driver}" />
		<beans:property name="url" value="${db.url}" />
		<beans:property name="username" value="${db.username}" />
		<beans:property name="password" value="${db.password}" />
	</beans:bean>

	<beans:bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
		<beans:constructor-arg name="dataSource" ref="dataSource" />
	</beans:bean>

	<beans:bean id="transactionTemplate"
		class="org.springframework.transaction.support.TransactionTemplate">
		<beans:constructor-arg name="transactionManager"
			ref="transactionManager"></beans:constructor-arg>
	</beans:bean>


	<beans:bean id="transactionManager"

	class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
		<beans:constructor-arg name="dataSource"
ref="dataSource"></beans:constructor-arg>
	</beans:bean>


	<beans:bean


class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
		<beans:property name="locations">
			<beans:list>
				<beans:value>database.properties</beans:value>
			</beans:list>
		</beans:property>
	</beans:bean>

</beans:beans>
```

----------Main class----------

```
public static void main(String args[]) {


            AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

            BankService bankService =
context.getBean("bankService",BankServiceImpl.class);

            Account fromAccount = new Account();
            fromAccount.setAccountNumber(110l);

            Account toAccount = new Account();
            toAccount.setAccountNumber(220l);

            bankService.transferFund(fromAccount, toAccount, 1000d);

            context.close();



      }
```

---------Service class-----------------

```
public interface BankService {


      public abstract void transferFund(Account fromAccount,Account toAccount,
Double amount);
}
```

```
public class BankServiceImpl implements BankService{


      private BankDao bankDao;

      private TransactionTemplate transactionTemplate;

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
            this.transactionTemplate = transactionTemplate;
      }

      public TransactionTemplate getTransactionTemplate() {
            return transactionTemplate;
      }

      public BankDao getBankDao() {
            return bankDao;
      }

      public void setBankDao(BankDao bankDao) {
            this.bankDao = bankDao;
      }
```

```
@Override
public void transferFund(Account fromAccount, Account toAccount, Double amount)
{

  transactionTemplate.execute(new TransactionCallback<Void>() {

            @Override
            public Void doInTransaction(TransactionStatus arg0) {

      getBankDao().withdraw(fromAccount, toAccount, amount);
      getBankDao().deposite(fromAccount, toAccount, amount);


                                    return null;
                              }
            });

      }


}
```

----------DAO class----------

```
public interface BankDao {


      public abstract void withdraw(Account fromAccount,Account toAccount,
Double amount);
      public abstract void deposite(Account fromAccount,Account toAccount,
Double amount);

}
```

-

```
@Repository

public class BankDAOImpl implements BankDao{

	private JdbcTemplate JdbcTemplate;

	public JdbcTemplate getJdbcTemplate() {
		return JdbcTemplate;
	}

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		JdbcTemplate = jdbcTemplate;
	}

	@Override
public void withdraw(Account fromAccount, Account toAccount, Double amount) {

		Account account =getAccountFromDb(fromAccount.getAccountNumber());
		String SQL="UPDATE icici_bank set account_balance=? where
account_no=?";
		if(account.getAccountBalance()>amount)
		{

			Double balance = account.getAccountBalance()-amount;

			int update =
getJdbcTemplate().update(SQL,balance,fromAccount.getAccountNumber());

			if(update>0) {

				System.out.println("Withdraw  Amount rs----"+balance+"is
transferred from "+fromAccount.getAccountNumber()+" to Account
"+toAccount.getAccountNumber());

			}

		}else {

			try {
				throw new InsufficiantBalance("Insufficiant amout please
try again");
			} catch (InsufficiantBalance e) {
				// TODO Auto-generated catch block
				//e.printStackTrace();
				System.out.println(e.getMessage());
			}

		}



	}
```

```
	@Override
	public void deposite(Account fromAccount, Account toAccount, Double
amount) {


			Account account =getAccountFromDb(toAccount.getAccountNumber());
			String SQL="UPDATE icici_bank set account_balance=? where
account_no=?";

			Double balance = account.getAccountBalance()+amount;

			int update =
getJdbcTemplate().update(SQL,balance,toAccount.getAccountNumber());

			if(update>0) {

				System.out.println(" Deposite  Amount rs----"+balance+"is
transferred from "+toAccount.getAccountNumber()+" to Account
"+toAccount.getAccountNumber());

			}
			//here i generate exception for rollback the transaction because if
any place raised the exception then
			//transaction will be rollback

			//throw new RuntimeException("Generate manullay exception for
transaction rollback");

	}

	private Account getAccountFromDb(Long id) {

			String SQL ="SELECT * FROM icici_bank WHERE account_no=?";

			Account account = getJdbcTemplate().queryForObject(SQL, new
AccountRowMapper(),id);

			return account;
	}

}

public class AccountRowMapper implements RowMapper<Account>{


	@Override
	public Account mapRow(ResultSet rs, int arg1) throws SQLException {
			// TODO Auto-generated method stub

			Account account = new Account();
			account.setAccountNumber(rs.getLong("account_no"));
			account.setAccountHoderName(rs.getString("account_holder_name"));
			account.setAccountBalance(rs.getDouble("account_balance"));
			account.setAccountType(rs.getString("account_type"));
			return account;
	}


}
```

```
--------------------Modal----------------------


public class Account {


	private Long accountNumber;

	private String accountType;

	private Double accountBalance;

	private String accountHoderName;


	public Long getAccountNumber() {
		return accountNumber;
	}
	public void setAccountNumber(Long accountNumber) {
		this.accountNumber = accountNumber;
	}
	public String getAccountType() {
		return accountType;
	}
	public void setAccountType(String accountType) {
		this.accountType = accountType;
	}
	public Double getAccountBalance() {
		return accountBalance;
	}
	public void setAccountBalance(Double accountBalance) {
		this.accountBalance = accountBalance;
	}
	public String getAccountHoderName() {
		return accountHoderName;
	}
	public void setAccountHoderName(String accountHoderName) {
		this.accountHoderName = accountHoderName;
	}




}
```

--------------Annotation based configuration--------------------

need third party aop jar include

aopalliance-1.0.jar

aopalliance-alpha1.jar

aspectj-1.6.5.jar

aspectjrt-1.6.11.jar

aspectjweaver-1.6.10-sources.jar

-----------------xml file-----------------------

```
<tx:annotation-driven  transaction-manager="transactionManager"  proxy-target-
class="true" />


      <beans:bean id="transactionManager"

      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
            <beans:constructor-arg name="dataSource"
ref="dataSource"></beans:constructor-arg>
      </beans:bean>

      <beans:bean id="bankService"
class="com.infotech.service.impl.BankServiceImpl" >
            <beans:property name="bankDao" ref="bankDao"></beans:property>

      </beans:bean>



      <beans:bean id="bankDao" class="com.infotech.dao.impl.BankDAOImpl">
            <beans:property name="jdbcTemplate"
ref="jdbcTemplate"></beans:property>
      </beans:bean>


      <beans:bean id="dataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <beans:property name="driverClassName" value="${db.driver}" />
            <beans:property name="url" value="${db.url}" />
            <beans:property name="username" value="${db.username}" />
            <beans:property name="password" value="${db.password}" />
      </beans:bean>

      <beans:bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
            <beans:constructor-arg name="dataSource" ref="dataSource" />
      </beans:bean>
```

```
	<beans:bean

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
		<beans:property name="locations">
			<beans:list>
				<beans:value>database.properties</beans:value>
			</beans:list>
		</beans:property>

	</beans:bean>



----------------Service class-------------------------

public class BankServiceImpl implements BankService{


	private BankDao bankDao;

	public BankDao getBankDao() {
		return bankDao;
	}

	public void setBankDao(BankDao bankDao) {
		this.bankDao = bankDao;
	}

	@Override

@Transactional(isolation=Isolation.READ_COMMITTED,propagation=Propagation.REQUI
RED,readOnly=false,rollbackFor=Exception.class)
public void transferFund(Account fromAccount, Account toAccount, Double amount)
{
		getBankDao().withdraw(fromAccount, toAccount, amount);
		getBankDao().deposite(fromAccount, toAccount, amount);

	}

}

------------------DAO impl-------------------------



-

@Repository

public class BankDAOImpl implements BankDao{

	private JdbcTemplate JdbcTemplate;

	public JdbcTemplate getJdbcTemplate() {
		return JdbcTemplate;
	}

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		JdbcTemplate = jdbcTemplate;
```

```
	}

	@Override
	public void withdraw(Account fromAccount, Account toAccount, Double
amount) {

		Account account =getAccountFromDb(fromAccount.getAccountNumber());
		String SQL="UPDATE icici_bank set account_balance=? where
account_no=?";
		if(account.getAccountBalance()>amount)
		{

			Double balance = account.getAccountBalance()-amount;

			int update =
getJdbcTemplate().update(SQL,balance,fromAccount.getAccountNumber());

			if(update>0) {

				System.out.println("Withdraw  Amount rs----"+balance+"is
transferred from "+fromAccount.getAccountNumber()+" to Account
"+toAccount.getAccountNumber());

			}

		}else {

			try {
				throw new InsufficiantBalance("Insufficiant amout please
try again");
			} catch (InsufficiantBalance e) {
				// TODO Auto-generated catch block
				//e.printStackTrace();
				System.out.println(e.getMessage());
			}

		}



	}

	@Override
	public void deposite(Account fromAccount, Account toAccount, Double
amount) {


		Account account =getAccountFromDb(toAccount.getAccountNumber());
		String SQL="UPDATE icici_bank set account_balance=? where
account_no=?";

		Double balance = account.getAccountBalance()+amount;

		int update =
getJdbcTemplate().update(SQL,balance,toAccount.getAccountNumber());

		if(update>0) {

			System.out.println(" Deposite  Amount rs----"+balance+"is
transferred from "+toAccount.getAccountNumber()+" to Account
"+toAccount.getAccountNumber());
```

```
            }

      }

      private Account getAccountFromDb(Long id) {

            String SQL ="SELECT * FROM icici_bank WHERE account_no=?";

         Account account = getJdbcTemplate().queryForObject(SQL, new
AccountRowMapper(),id);

            return account;
      }

}
```

---------Explanation about each attribute------------

readOnly=**false** doen't allow to write operation on database it wll give exception if your are going to perform some write operation

rollbackFor=Exception.**class**

specify some rules about excception means we don't allow to rollback some exception cases or allow rollback some exception cases then we can specify the Exception rules

example **Exception.class** is only applicable for checked exception

RunTimeException.class is applicable only for RunTimeException