

1. **Overview**

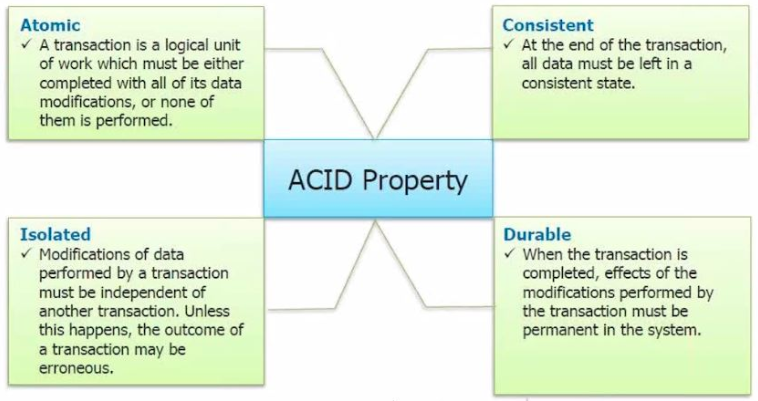
A transaction is a unit of work in which either all operations must execute or none of them. To understand the importance of the transaction, think of an example which applies to all of us. “Transferring Amount from one account to another “ – this operation includes below at least below two steps

1. Deduct the balance from the sender’s account
2. Add the amount to the receiver’s account.

Now think of the situation where the amount is deducted from the sender’s account, but not gets delivered to receiver account due to some errors. Such issues are managed by transaction management in which both the steps are performed in a single unit of work where either both steps are performed successfully or in case anyone gets failed, it should be roll backed.

There are four important terms which are very important to understand.

1. Atomic -  As described above, atomicity makes sure that either all operations within a transaction must be successful or none of them.
2. Consistent- This property makes sure that data should be in consistent state once the transaction is completed.
3. Isolated-  this property allows multiple users to access the same set of data and each user’s processing should be isolated from others.
4. Durable – Result of the transaction should be permanent once the transaction is completed to avoid any loss of data.



## ****Spring Transaction Management Support****

Spring provides support for both programmatic and declarative transactions similar to EJB.

1. Programmatic Transactions – With programmatic transactions , transaction management  code like,  commit when everything is successful or rolling back if anything goes wrong  is clubbed with the business logic.
2. Declarative Transactions-Declarative transactions separates transaction management code from business logic. Spring supports declarative transactions using transaction advice (using AOP).

Choosing declarative or programmatic transactions is convenience  versus fine control. Programmatic approach provides a fine control on transaction boundaries, whereas declarative approach provides a great configurability using configuration files.

If our application is working with one data source only then we can manage the transactions using commit and rollback methods of transaction object, but to have a transaction between multiple data sources we would need to rely on transaction management provides by J2EE servers . Spring also supports distributed (XA) transactions which can be used for later case.

**Declarative Transactions**

**Spring**

**Programmatic Transactions**

Declarative transactions separates transaction management code from business logic. Spring supports declarative transactions using transaction advice (using AOP)

With programmatic transactions , transaction management  code like,  commit when everything is successful or rolling back if anything goes wrong  is clubbed with the business logic

## ****Choosing Transaction Manage****

Instead of managing the transaction, Spring supports several transaction managers which  delegate the transaction management responsibilities to platform specific implementations. Plarform Transaction manager is the parent of all transaction manager implementations.

Some of the transaction managers are-

1. **DataSource Transaction manager** -  We can use DataSourceTransactionManager for simple JDBC persistence mechanism. Sample configuration of DataSourceTransactionManager looks like below

     <bean id=”transactionManager”   
           class=”org.springframework.jdbc.datasource.DataSourceTransactionManager>  
          <property name=”dataSource” ref= “datasource” />  
     </bean>

1. **Hibernate Transaction manager** – Hibernate transaction manager should be used when our application is using Hibernate. Sample configuration of HibernateTransactionManager looks like below

                <bean id=”transactionManager”   
                      class=”org.springframework.orm.hibernate3.HibernateTransactionManager>  
                     <property name=”sessionFactory” ref= “sessionFactory” />  
                </bean>

1. **Jdo Transaction manager** –Use below configuration to use Java data object transaction manager .

              <bean id=”transactionManager”   
                    class=”org.springframework.orm.jdo.JdoTransactionManager>  
                   <property name=”persistanceManagerFactory” ref= “persistanceManagerFactory” />  
              </bean>

1. **Jta Transaction manager** – If our transaction is across multiple data sources than we need to use Java Transactions API transactions . Internally  JTA implementation handles transaction responsibility.

          Use below configuration to configure JTA transaction manager.

          <bean id=”transactionManager”   
                class=”org.springframework.transaction.jta.JtaTransactionManager>  
               <property name=”transactonManagerName” ref= “java:/TransactionManager” />  
          </bean>

## ****Programmatic Transaction Management****

Spring provides platform independent Transaction manager API with the help of an abstract transaction management unit known as **PlatformTransactionManager**interface under *org. springframework. transaction package.*

     This interface defines below methods

* **getTransaction(TransactionDefinition)-**This method returns a transaction (either a active transaction or create a new one)
* **void commit(TransactionStatus)-** This method commits the transaction based on the status.
* **void rollback(TransactionStatus)-**This method rollbacks the transaction based on status.

     TransactionStatus interface defines several methods to get the status of transaction and controls the transaction execution as well.

**a.  Create schema**

***create schema SpringJDBC;***

**b.  Create account table**

               CREATE TABLE 'account' (  
                      'id' int(11) NOT NULL AUTO\_INCREMENT,  
                      'username' varchar(45) DEFAULT NULL,  
                      'amount' int(11) DEFAULT NULL,  
                      PRIMARY KEY ('id')  
                 )

**c.  Create product table**

               CREATE TABLE 'products' (  
                    'id' int(11) NOT NULL,  
                    'name' varchar(45) DEFAULT NULL,  
                    'price' int(11) DEFAULT NULL,  
                    'stock' int(11) DEFAULT NULL,  
                    PRIMARY KEY ('id')  
                );

**d.  Insert data in tables**

                 INSERT INTO 'springjdbc'.'account' ('username', 'amount') VALUES ('user1', 2000);   
                 INSERT INTO 'springjdbc'.'account' ('username', 'amount') VALUES ('user2', 4000);   
                 INSERT INTO 'springjdbc'.'account' ('username', 'amount') VALUES ('user3', 1300);

                 INSERT INTO 'springjdbc'.'products' ('id', 'name', 'price', 'stock') VALUES (101, 'Product A', 100, 20);   
                 INSERT INTO 'springjdbc'.'products' ('id', 'name', 'price', 'stock') VALUES (102, 'Product B', 200, 25);

**e.   Product.java**

package jdbc;

public class Product {

    private int productId;

    private String productName;

    private int price;

    private int  stock;

    public int getProductId() {

        return productId;

    }

    public void setProductId(int productId) {

        this.productId = productId;

    }

    public String getProductName() {

        return productName;

    }

    public void setProductName(String productName) {

        this.productName = productName;

    }

    public int getPrice() {

        return price;

    }

    public void setPrice(int price) {

        this.price = price;

    }

    public int getStock() {

        return stock;

    }

    public void setStock(int stock) {

        this.stock = stock;

    }

    @Override

    public String toString() {

        return "Product [productId=" + productId + ", productName="

                + productName + ", price=" + price + ", stock=" + stock + "]";

    }

}

**f.  Account.java**

package jdbc;

public class Account {

       private String username;

    private int balance;

    public String getUsername() {

        return username;

    }

    public void setUsername(String username) {

        this.username = username;

    }

    public int getBalance() {

        return balance;

    }

    public void setBalance(int balance) {

        this.balance = balance;

    }

    @Override

    public String toString() {

            return "Account [username=" + username + ", balance=" + balance + "]";

    }

}

**g.  BuyProduct.java**

package jdbc;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.DefaultTransactionDefinition;

public class BuyProduct {

    private JdbcTemplate jdbcTemplate;

    private PlatformTransactionManager transactionManager;

    public void setDataSource(DataSource dataSource) {

        this.jdbcTemplate = new JdbcTemplate(dataSource);

    }

    public void setTransactionManager( PlatformTransactionManager transactionManager){

        this.transactionManager = transactionManager;

    }

    public void buyProduct(String username , int productId){

         TransactionDefinition def = new DefaultTransactionDefinition();

         TransactionStatus status = transactionManager.getTransaction(def);

         try

         {

            int price =

jdbcTemplate.queryForI    ( "SELECT PRICE FROM PRODUCTS WHERE Id = ?",

new Object[] { new Integer(productId) });

             jdbcTemplate.update("UPDATE PRODUCTS SET STOCK =

                          STOCK - 1 WHERE ID = ?", new Object[] { new Integer(productId) });

             jdbcTemplate.update( "UPDATE ACCOUNT SET AMOUNT =

                          AMOUNT - ?  WHERE USERNAME = ?", new Object[] { price, username });

             transactionManager.commit(status);

          }

          catch (DataAccessException e){

               transactionManager.rollback(status);

               throw e;

          }

    }

    public void displayState(String username, int productId){

         System.out.println("Database state after purchase ");

         Product product = jdbcTemplate.queryForObject

        ("SELECT \* FROM PRODUCTS WHERE ID = ? ", new Object[]{new Integer(productId)},

            new RowMapper<Product>() {

            @Override

            public Product mapRow(ResultSet rs, int row)

            throws SQLException {

                 Product product = new Product();

                 product.setPrice(rs.getInt("price"));

                 product.setProductId(rs.getInt("id"));

                 product.setProductName(rs.getString("name"));

                product.setStock(rs.getInt("stock"));

                return product;

            }

       } );

       System.out.println(product);

       Account account = jdbcTemplate.queryForObject

           ("SELECT \* FROM ACCOUNT WHERE USERNAME = ? ", new Object[]{username},

             new RowMapper<Account>() {

                @Override

                public Account mapRow(ResultSet rs, int row)

                throws SQLException {

                Account  account = new Account();

                account.setBalance(rs.getInt("amount"));

                account.setUsername(rs.getString("username"));

                return account;

              }

          } );

        System.out.println(account);

   }

}

**h.  TestTransaction.java**

package jdbc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestTransaction {

     public static void main(String args[]){

        ApplicationContext context =

                new ClassPathXmlApplicationContext("beans.xml");

        BuyProduct buy  = (BuyProduct)context.getBean("buyProduct");

        buy.buyProduct("user1", 101);

        buy.displayState("user1", 101);

        buy.buyProduct("user2", 102);

        buy.displayState("user2", 102);

    }

}

**i. beans.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:context="<http://www.springframework.org/schema/context>"

    xsi:schemaLocation="http://www.springframework.org/schema/beans

           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

           http://www.springframework.org/schema/context

           http://www.springframework.org/schema/context/spring-context-3.0.xsd

           http://www.springframework.org/schema/aop

           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

     <bean id="dataSource"

         class="org.springframework.jdbc.datasource.DriverManagerDataSource">

         <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

         <property name="url" value="jdbc:<mysql://localhost:3306/SpringJDBC>"/>

         <property name="username" value="root"/>

         <property name="password" value="password"/>

     </bean>

     <bean id="buyProduct"  class="jdbc.BuyProduct">

          <property name="dataSource"  ref="dataSource" />

          <property name="transactionManager"  ref="transactionManager" />

     </bean>

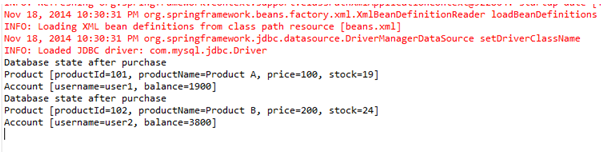
     <bean id="transactionManager"

         class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

         <property name="dataSource"  ref="dataSource" />

     </bean>

</beans>

****

## ****Declarative Transaction Management****

As we saw in section 20.4 programmatic approach has a tight coupling on the transaction code with  business logic. In Declarative transactions, we can separate transaction management code from business logic.

Spring supports declarative transactions using

1. **Transaction Advices (using Aspect Oriented Programming)** - In this approach we need to-
2. declare a transaction advice via the <tx:advice> element defined in the tx namespace.  For example

        <tx:advice id="txAdvice" transaction-manager="transactionManager">  
            <tx:attributes>  
               <tx:method name="buy\*" propagation="REQUIRED"/>  
               <tx:method name="\*" propagation="SUPPORTS" />

           </tx:attributes>  
       </tx:advice>

        Transaction advice defines <tx:attributes> element which can have several <tx:method> tags to define several properties. In above example, any method whose name starts with “buy”  are required or         a part of the transaction, but all other methods will execute in current transaction but will not create a new if the transaction does not exist.

    b.  Now we need to associate transaction advice (<tx:advice>)  with a pointcu using  <aop:config>

            <aop:config>  
                <aop:advisor advice-ref="txAdvice"   
               pointcut =" execution(\* jdbc.BuyProduct.buyProduct(..))"/>  
     
            </aop:config>

### **20.5.2  Transaction Advice Example**

We will be using the same scenario that we have implemented in section 20.4.1 so follow steps a to h of section 20.4.1

**a.  PurchaseProduct.java**

package jdbc;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

Import org. springframework. Dao. DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.DefaultTransactionDefinition;

public class PurchaseProduct {

      private JdbcTemplate jdbcTemplate;

      public void setDataSource(DataSource dataSource) {

           this.jdbcTemplate = new JdbcTemplate(dataSource);

      }

      public void buyProduct(String username , int productId){

         try{

             int price = jdbcTemplate.queryForInt

                  ( "SELECT PRICE FROM PRODUCTS WHERE Id = ?",    new Object[] { new Integer(productId) });

             jdbcTemplate.update("UPDATE PRODUCTS SET STOCK =

                      STOCK - 1 WHERE ID = ?", new Object[] { new Integer(productId) });

             jdbcTemplate.update( "UPDATE ACCOUNT SET AMOUNT =

                           AMOUNT - ?  WHERE USERNAME = ?", new Object[] { price, username });

            }

            catch (DataAccessException e){

               throw e;

            }

     }

     public void displayState(String username, int productId){

           System.out.println("Database state after purchase ");

            Product product = jdbcTemplate.queryForObject

                        ("SELECT \* FROM PRODUCTS WHERE ID = ? ", new Object[]{new Integer(productId)},

            new RowMapper<Product>() {

                @Override

                public Product mapRow(ResultSet rs, int row)

                  throws SQLException {

                    Product product = new Product();

                    product.setPrice(rs.getInt("price"));

                    product.setProductId(rs.getInt("id"));

                    product.setProductName(rs.getString("name"));

                    product.setStock(rs.getInt("stock"));

                    return product;

                 }

           } );

           System.out.println(product);

           Account account = jdbcTemplate.queryForObject

                    ("SELECT \* FROM ACCOUNT WHERE USERNAME = ? ", new Object[]{username},

           new RowMapper<Account>() {

               @Override

               public Account mapRow(ResultSet rs, int row)

               throws SQLException {

                 Account  account = new Account();

                 account.setBalance(rs.getInt("amount"));

                 account.setUsername(rs.getString("username"));

                 return account;

               }

           } );

          System.out.println(account);

    }

}

**b.  beans.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: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-3.0.xsd

         http://www.springframework.org/schema/tx

         http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

         http://www.springframework.org/schema/aop

         http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

        <bean id="dataSource"

              class="org.springframework.jdbc.datasource.DriverManagerDataSource">

              <property name="driverClassName" value="com.mysql.jdbc.Driver" />

              <property name="url" value="jdbc:<mysql://localhost:3306/SpringJDBC>" />

              <property name="username" value="root" />

              <property name="password" value="password" />

        </bean>

        <bean id="purchaseProduct" class="jdbc.PurchaseProduct">

            <property name="dataSource" ref="dataSource" />

        </bean>

        <bean id="transactionManager"

            class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

            <property name="dataSource" ref="dataSource" />

        </bean>

        <tx:advice id="txAdvice" transaction-manager="transactionManager">

           <tx:attributes>

              <tx:method name="buy\*" propagation="REQUIRED" />

              <tx:method name="\*" propagation="SUPPORTS" />

          </tx:attributes>

       </tx:advice>

       <aop:config>

          <aop:advisor advice-ref="txAdvice"

           pointcut=" execution(\* jdbc.PurchaseProduct.buyProduct(..))" />

       </aop:config>

</beans>

1. **TestTransaction.java**

package jdbc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestTransaction {

     public static void main(String args[]){

          ApplicationContext context =

              new ClassPathXmlApplicationContext("beans.xml");

          PurchaseProduct purchase  = (PurchaseProduct)context.getBean("purchaseProduct");

          purchase.buyProduct("user1", 101);

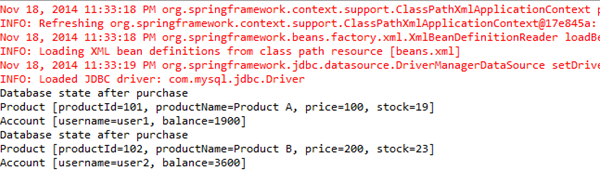
          purchase.displayState("user1", 101);

          purchase.buyProduct("user2", 102);

          purchase.displayState("user2", 102);

    }

}

****

### **5.1 @Transactional Annotation**

Along with the declarative approach with pointcut , advisors configuration in beans configuration file , Spring allows another way of declarative transaction management using @Transactional annotation        on the method which requires transaction and enabling <tx:annotation-driven> element in the beans configuration file.  This approach requires JDK1.5 onwards.

### **5.2 @Transactional Annotation Example**

We will be using the same scenario that we have implemented in section 20.4.1 so follow steps ( a to h )  of section 20.4.1

**a. PurchaseProduct.java**

package jdbc;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.annotation.Transactional;

import org.springframework.transaction.support.DefaultTransactionDefinition;

public class PurchaseProduct {

     private JdbcTemplate jdbcTemplate;

     public void setDataSource(DataSource dataSource) {

              this.jdbcTemplate = new JdbcTemplate(dataSource);

     }

     @Transactional

     public void buyProduct(String username , int productId){

        try

        {

          int price = jdbcTemplate.queryForInt

              ( "SELECT PRICE FROM PRODUCTS WHERE Id = ?",    new Object[] { new Integer(productId) });

          jdbcTemplate.update("UPDATE PRODUCTS SET STOCK =

                 STOCK - 1 WHERE ID = ?", new Object[] { new Integer(productId) });

          jdbcTemplate.update( "UPDATE ACCOUNT SET AMOUNT = AMOUNT - ?  WHERE USERNAME = ?", new Object[] { price, username });

        }

        catch (DataAccessException e)

        {

           throw e;

        }

     }

     public void displayState(String username, int productId){

       System.out.println("Database state after purchase ");

       Product product = jdbcTemplate.queryForObject("SELECT \* FROM PRODUCTS WHERE ID = ? ", new Object[]{new Integer(productId)},

       new RowMapper<Product>() {

          @Override

          public Product mapRow(ResultSet rs, int row)

          throws SQLException {

            Product product = new Product();

            product.setPrice(rs.getInt("price"));

            product.setProductId(rs.getInt("id"));

            product.setProductName(rs.getString("name"));

            product.setStock(rs.getInt("stock"));

            return product;

         }

      } );

      System.out.println(product);

      Account account = jdbcTemplate.queryForObject("SELECT \* FROM ACCOUNT WHERE USERNAME = ? ", new Object[]{username},

        new RowMapper<Account>() {

      @Override

        public Account mapRow(ResultSet rs, int row)

        throws SQLException {

            Account  account = new Account();

            account.setBalance(rs.getInt("amount"));

            account.setUsername(rs.getString("username"));

            return account;

        }

      } );

      System.out.println(account);

   }

}

**b. beans.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: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-3.0.xsd

       http://www.springframework.org/schema/tx

       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

       http://www.springframework.org/schema/aop

       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

       <tx:annotation-driven transaction-manager="transactionManager"/>

       <bean id="dataSource"

          class="org.springframework.jdbc.datasource.DriverManagerDataSource">

          <property name="driverClassName" value="com.mysql.jdbc.Driver" />

          <property name="url" value="jdbc:<mysql://localhost:3306/SpringJDBC>" />

          <property name="username" value="root" />

          <property name="password" value="password" />

       </bean>

       <bean id="purchaseProduct" class="jdbc.PurchaseProduct">

           <property name="dataSource" ref="dataSource" />

       </bean>

       <bean id="transactionManager"

           class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

           <property name="dataSource" ref="dataSource" />

       </bean>

</beans>

**c.  TestTransaction.java**

package jdbc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestTransaction {

        public static void main(String args[]){

            ApplicationContext context =

                new ClassPathXmlApplicationContext("beans.xml");

             PurchaseProduct purchase  = (PurchaseProduct)context.getBean("purchaseProduct");

             purchase.buyProduct("user1", 101);

             purchase.displayState("user1", 101);

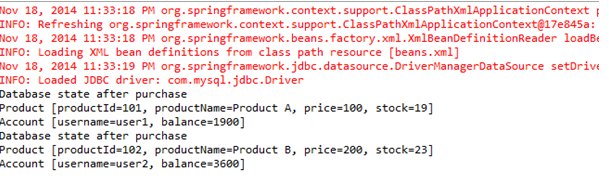
             purchase.buyProduct("user2", 102);

             purchase.displayState("user2", 102);

        }

}

**c. Run TestTransaction.java**



## ****Transaction propagation Behavior****

        Transaction Propagation behavior can be specified in <tx:method> element. Below are some of the most commonly used propagation modes.

* **REQUIRED**-  Current method must run in an existing  transaction  and if there are no existing transactions the start a new transaction and run within it.
* **REQUIRES NEW** -  Current method must start a new transaction and run within it.
* **SUPPORTS** -  Current method can run in existing transaction if exists else it is not necessary to run within a transaction.
* **NOT SUPPORTED** - The current method should not run within a transaction.
* **MANDATORY**- The current method must run within a transaction. If there’s no existing transaction in progress, an exception will be thrown.

## ****Isolation Levels****

     When multiple transactions in the application are operating concurrently on the same data can lead to below problems.

1. Dirty Read – If there are two transactions running concurrently and one thread has read the data which is being updated but not yet committed by another transaction and instead of committing, it rolls back the changes.
2. Nonrepeatable read- This problem occurs when a  transaction gets the different data for the same query when executed multiple times. This usually happens if another transaction is committing the data simultaneously.
3. Phantom Read-  this problem occurs when a transaction is inserting new data while another transaction is reading the data. In this scenario reading transaction will find additional data which was not there earlier.

In ideal scenario, transactions should be completely isolated from each other

Following are the most commonly used Isolation levels and are supported by Spring.

* **DEFAULT**– This isolation level uses the default isolation level of the underlying database.
* **READ UNCOMMITTED**– This isolation level supports transactions to read uncommitted data by other transactions as well. With this isolation, there are  chances of dirty red, nonrepeatable read, and phantom read.
* **READ COMMITTED** - This isolation level supports transactions to read only data committed by other transactions as well. There are chances of nonrepeatable read and phantom read.