* A web service is any piece of code or software that makes itself available over the internet and uses a standardized JSON or XML format data exchange between two systems or applications. Because xml or Json can be understandable by any programming language for example, a client invokes a web service by calling RESTAPI URL , then waits for a corresponding JSON or XML response. All the communication is in webservices are by using JSON or XML, web services provide interoperability it means it is not tied to any one programming language one application developed in one language can communicate with any other application developed in any language Java can talk with PHP or .NET Python etc…
* Use cases of webservices
* **Case 1:** I want to develop a Web application which shows weather report based on location, for this if we want to track weather report we need sensors or some equipment like satellite to get temperature humidity etc.. but I cannot invest huge amount on this hardware. Solution for this is in this case we may have a big company like Google or some service provider where he can setup all the hardware or equipment to get all the weather information and store in his database in real-time like every min. We can pay some amount to the service provider to get weather information, but he will not give direct access to the database he will create a RestAPI(Producer application) which will give you only required information from his data base, he will just give you an **URL** , when we call that URL from our Restful application(Consumer) we will get required information in JSON format.
* **Case 2:** I want to develop a beautiful application with very good UI(user-interface ) front-end screen, but we cannot do this with just JSP or HTMLwe may required some other front end technologies like AngularJS which will be used to develop only UI-frontend screens, but angular is just a frontend application which will show view on screen , it cannot perform any business logics. In this case we will develop a Restful application RestAPI which will give JSON data when we call a URL , so Angular(front-end) application will call that Restapi url and it will get JSON data , that JSON data will be shown in screen with angular as they want.
* **Case 3:** if you want to develop a application like Flipkart which should work on all platforms like web application, Android application, IOS, we cannot write our business logic separately for all three platforms so we will create a RESTAPI application which will expose a URL with this URL all the android or web applications, IOS can communicate and exchange only JSON data.
* **CASE 4:** if you want to book a bus ticket with redbus you can book private travel’s and also different RTC buses like APSRTC or TSRTC, here RedBUS is a different application and RTC is different application. If you want to get all RTC bus information from redbus you cannot access directly RTC bus application, redbus first need to contact RTC, like we need your application services to display and book rtc ticket’s through redbus app. Then RTC people will give their RESTAPI information like call this url to display bus information, use this url to book tickets.

Types of Web Services

There are mainly two types of web services.

1. SOAP web services. (it is Purely XML Based and legacy)
2. RESTful web services. (we can use JSON/XML it is latest)

## Java Web Services API

There are two main API's Specifications(interfaces) defined by Java JavaEE 6.

) **JAX-WS**: for SOAP web services. The are two ways to write JAX-WS application code: by RPC style and Document style. (XML based)

2) **JAX-RS**: for RESTful web services. There are mainly 2 implementations currently in use for creating JAX-RS application: **Jersey and RESTeasy.**

**Spring has its own implementation for Webservices**

**We use Spring-Restful webservices to create RESTful API.**

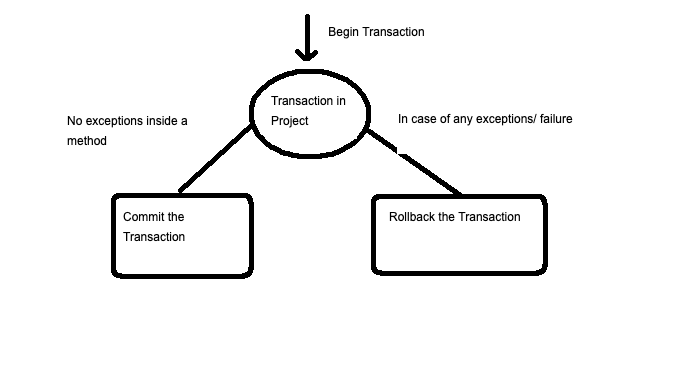
In general to develop a web-application we use spring framework, we can use same process to develop RESTAPI also, in normal Spring MVC application we will be having a VIEW page(JSP pages) but in RESTAPI we will not have any View pages we will just return JSON data, so to convert our JAVA Object’s into Json format we need some library like jackson-databind , and we need to put some spring annotations like @responsebody along with request mapping url so that spring can identify to send the data back in JSON format in response. Later in new versions spring we use @RestContoller instead of @responsebody ,we can just use @RestContoller on top of the class, it indicates all the request will be converted into json.

@RestContoller is Combination of @Contoller+@RespnseBody annotations

# **What is a Transaction?**

It is considered to be a *set of actions* that allows to modify or make changes in a database. Whenever a Transaction happens, and in case of any failure in between the course of an action, then the entire transaction has to be rolled off.

In other words, a transaction must be committed only after the successful execution of the entire actions. Otherwise, the transaction should be rolled off in case of any exceptions. Below image depicts the same.

* 

**What is transaction**

In an application transaction is a sequence of events or a unit of work. That is either all the events(unit of work) participating in the transaction should happen or none.

• 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.

**Other Example**: if you are booking a ticket, for a successful ticket booking, application need to store user details and generate a ticket but while booking a ticket it has saved user details and allocated seat number but during payment transaction got failed in this case without payment we booked a ticket which is not a good one.

To fix this issue we need to apply transaction management so that only for successful payment transaction ticket will be booked and our details will be stored.

In case of any payment failure transaction will get rolled back and seat will get deallocated.

The main goal of a transaction is to provide ACID characteristics to ensure the consistency and validity of your data.

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.

**Advantages of the Spring Framework’s transaction support model**

Traditionally, Java EE developers have had two choices for transaction management: global or local transactions, both of which have profound limitations. Global and local transaction management is reviewed in the next two sections, followed by a discussion of how the Spring Framework’s transaction management support addresses the limitations of the global and local transaction models.

### 16.2.1 Global transactions

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be sourced from JNDI, meaning that you also need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

Previously, the preferred way to use global transactions was via EJB CMT (Container Managed Transaction): CMT is a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups, although of course the use of EJB itself necessitates the use of JNDI. It removes most but not all of the need to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

### 16.2.2 Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

### 16.2.3 Spring Framework’s consistent programming model

Spring resolves the disadvantages of global and local transactions. It enables application developers to use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Most users prefer declarative transaction management, which is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence do not depend on the Spring Framework transaction API, or any other transaction API.

**Types of Transaction management in Spring:**

1. **Programmatic**: In this type, we need to write *extra code* for managing transactions (i.e) creating a transaction instance, begin the transaction, commit or rollback statements. Please refer below.

Transaction transaction = entityManager.getTransaction();  
try{  
transaction.begin();  
transaction.commit();  
}catch(Exception e){transaction.rollback();  
throw e;  
}

This involves a lot of *boiler code* that incurs additional performance issues.

2. **Declarative**: Here, we can manage transactions with the help of an annotation -@Transactional or XML based approach. This would decrease the lines of code. In this article, we will be focusing the usage of -@Transactional annotation.

### Spring Transaction management types

Spring Transaction management has support for both **programmatic transaction management** and **declarative transaction management**.

**Programmatic transaction management**

Spring Framework provides two means of programmatic transaction management-

* Using TransactionTemplate.
* Using a PlatformTransactionManager implementation directly.

Mostly declarative transaction that too using annotation is used so we'll focus on that. If you want to use TransactionTemplate then you can configure it as follows by providing the reference of the Transaction manager.

<propertyname="transactionTemplate">

<bean class="org.springframework.transaction.support.

TransactionTemplate">

<property name="transactionManager"

ref="txManager" />

</bean>

</property>

**Declarative transaction management in Spring**

By using Declarative transaction for Spring transaction management you keep transaction management separate from the business code. You can define declarative transactions using annotations or XML based configuration using AOP.

The annotation used for Declarative transaction management is **@Transactional annotation**. You can place the @Transactional annotation before an interface definition, a method on an interface, a class definition, or a public method on a class.

To make Spring framework aware of the @Transactional annotation you will have to define **<tx:annotation-driven/>** element in your XML configuration.

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

If you using Java configuration then you can enable @Transactional annotation support by adding **@EnableTransactionManagement** to your config class.

@Configuration

@EnableTransactionManagement

public class AppConfig{

...

...

}

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Transactional annotation, as opposed to annotating interfaces.

### @Transactional settings in Spring framework

You can provide transaction properties like propagation behavior, isolation level along with @Transactional annotation. Full list of the properties of the @Transactional annotation are as follows-

* **propagation**- Optional propagation setting.
* **isolation**- Optional isolation level. Only applicable to propagation REQUIRED or REQUIRES\_NEW.
* **timeout**- Optional transaction timeout. Only applicable to propagation REQUIRED or REQUIRES\_NEW. Defined in seconds using an int value.
* **readOnly**- Read/write vs. read-only transaction. Only applicable to REQUIRED or REQUIRES\_NEW.
* **rollbackFor**- Optional array of exception classes that must cause rollback.
* **rollbackForClassName**- Optional array of names of exception classes that must cause rollback.
* **noRollbackFor**- Optional array of exception classes that must not cause rollback.
* **noRollbackForClassName**- Optional array of names of exception classes that must not cause rollback.

The default @Transactional settings are as follows:

* Propagation setting is PROPAGATION\_REQUIRED.
* Isolation level is ISOLATION\_DEFAULT.
* Transaction is read/write.
* Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
* Any RuntimeException triggers rollback, and any checked Exception does not.

### Example using @Transactional annotation

@Transactional(readOnly = true, propagation=Propagation.SUPPORTS)

public class TestService implements FooService {

public Foo getValue(String Id) {

// do something

}

// these settings have precedence for this method

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW, timeout=60, rollbackFor=ValueNotFoundException.class)

public void updateValue(Person person) {

// do something

}

}

Here at the class level TestService class is annotated with @Transactional annotation which is for all the methods in the class that all the methods will support transaction and will be read only. Method updateValue overrides it by having its own @Transactional annotation which requires a new transaction, read only is false and transaction is rolled back if ValueNotFoundException is thrown.

## *Basic elements of a transaction*

Every transaction has properties that allow us to configure it within our application. Before we understand ways to manage transaction operations using Spring Framework, we should understand the important properties associated with each transaction. Some of the transaction properties are - *propagation, isolation, read-only, roll back for exception, timeout*.

## *Propagation*

Using the *propagation* property of transaction, we can define the behaviour of a transaction i.e. we can specify a method of a class which is going to perform the transaction operation and have this method executed as a part of a new transaction each time it is called or part of an existing transaction.  
  
Spring Framework provides us *propagation constants* through *org.springframework.transaction.annotation.Propagation enumeration*, using which we can define the *propagation* property of a transaction.

|  |  |
| --- | --- |
| **Constants** | **Description** |
| **PROPOGATION\_MANDATORY** | Specifies a method which should be executed as a part of a transaction. |
| **PROPOGATION\_REQUIRED** | Searched an existing transaction and executes the associated method within it or creates a new transaction. |
| **PROPAGATION\_REQUIRES\_NEW** | Specifies a method which should be executed in a new transaction. |
| **PROPAGATION\_SUPPORTS** | Specifies a method which can be executed inside or outside a transaction. |
| **PROPAGATION\_NEVER** | Specifies a method should never be executed in a transaction. If an transaction exists, an exception is thrown. |
| **PROPAGATION\_NOT\_SUPPORTED** | Specifies a method should not be executed in a transaction. If an transaction exists, it is suspended until method finishes its execution. |

## *Isolation*

Using the *isolation* property of transaction, we can define different ways of configuring the execution of concurrent transactions within an application which helps us to maintain data consistency and integrity i.e. we can specify whether or not a particular transaction should only be executed in isolation by having the complete access control over the data it needs(*without sharing this data with any other transaction until it completes its execution*).  
  
Spring Framework provides us *isolation constants* through *org.springframework.transaction.annotation.Isolation enumeration*, using which we can define the *isolation* property of a transaction.

|  |  |
| --- | --- |
| **Constants** | **Description** |
| **ISOLATION\_DEFAULT** | Specifies a default isolation level defined in the relational databases. |
| **ISOLATION\_READ\_COMMITTED** | Specifies that a transaction *cannot read or change* data which is still acquired by an existing transaction. |
| **ISOLATION\_READ\_UNCOMMITTED** | Specifies that a transaction *can read or change* the data which is still acquired by an existingtransaction. |
| **ISOLATION\_REPEATABLE\_READ** | This constant helps us to acquire the lock on the data a transaction needs. |
| **ISOLATION\_SERIALIZABLE** | Specifies that a transaction should be executed in a proper isolation, without the intrusion from another transaction. |

## *Read-only*

Using the *read-only* property of transaction, we can specify whether a particular transaction is going to modify the data or not. This property of transaction takes a **boolean** value.

## *Rollback-for*

Using the *rollback-for* property of transaction, we can specify a *single or multiple exception classes*. When any of these exceptions is thrown, the associated transaction is going to rolled back(*without affecting the database consistency and integrity*).

## *Ways to perform transaction management*

*There are two ways to perform transaction management -*

* **Programmatic Transaction Management** - Programmatic Transaction Management involves writing the code performing the transaction management i.e. writing the code to begin the transaction until its end, to control the execution of transaction in specific situations, to handle the exception while the transaction is being performed and to rollback the transaction in specific situations etc.  
    
  To perform the transaction management programmatically, Spring Framework provide us a *org.springframework.transaction.support.TransactionTemplate* class.

* **Declarative Transaction Management** - Unlike the programmatic transaction management approach, the declarative transaction management approach doesn't require us to write or modify the source code to perform transaction management.  
    
  declarative Transaction Management involves performing the transaction management through the configuration ***xml*** file i.e. *you can declared all the transaction elements within the configuration****xml****file of your project. This configuration****xml****file is maintained by the Spring Container in Spring Framework.*  
    
  **Note :***Declarative transaction management is applied to****methods****which are performing the transaction operations.*

## ****20.6  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.

## ****20.7  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.

### 2.2. How to Use @Transactional

We can put the annotation on definitions of interfaces, classes, or directly on methods.  They override each other according to the priority order; from lowest to highest we have: interface, superclass, class, interface method, superclass method, and class method.

**Spring applies the class-level annotation to all public methods of this class that we did not annotate with @Transactional.**

**However, if we put the annotation on a private or protected method, Spring will ignore it without an error.**

Let's start with an interface sample:

@Transactional

**public** **interface** **TransferService** {

**void** **transfer**(String user1, String user2, **double** val);

}

Usually it's not recommended to set @Transactional on the interface; however, it is acceptable for cases like @Repository with Spring Data. We can put the annotation on a class definition to override the transaction setting of the interface/superclass:

@Service

@Transactional

**public** **class** **TransferServiceImpl** **implements** **TransferService** {

@Override

**public** **void** **transfer**(String user1, String user2, **double** val) {

// ...

}

}

Now let's override it by setting the annotation directly on the method:

@Transactional

**public** **void** **transfer**(String user1, String user2, **double** val) {

// ...

}

## 3. Transaction Propagation

Propagation defines our business logic's transaction boundary. Spring manages to start and pause a transaction according to our propagation setting.

Spring calls TransactionManager::getTransaction to get or create a transaction according to the propagation. It supports some of the propagations for all types of TransactionManager, but there are a few of them that are only supported by specific implementations of TransactionManager.

Let's go through the different propagations and how they work.

### 3.1. REQUIRED **Propagation**

REQUIRED is the default propagation. Spring checks if there is an active transaction, and if nothing exists, it creates a new one. Otherwise, the business logic appends to the currently active transaction:

@Transactional(propagation = Propagation.REQUIRED)

**public** **void** **requiredExample**(String user) {

// ...

}

Furthermore, since REQUIRED is the default propagation, we can simplify the code by dropping it:

@Transactional

**public** **void** **requiredExample**(String user) {

// ...

}

Let's see the pseudo-code of how transaction creation works for REQUIRED propagation:

**if** (isExistingTransaction()) {

**if** (isValidateExistingTransaction()) {

validateExisitingAndThrowExceptionIfNotValid();

}

**return** existing;

}

**return** createNewTransaction();

### 3.2. SUPPORTS **Propagation**

For SUPPORTS, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional:

@Transactional(propagation = Propagation.SUPPORTS)

**public** **void** **supportsExample**(String user) {

// ...

}

Let's see the transaction creation's pseudo-code for SUPPORTS:

**if** (isExistingTransaction()) {

**if** (isValidateExistingTransaction()) {

validateExisitingAndThrowExceptionIfNotValid();

}

**return** existing;

}

**return** emptyTransaction;

### 3.3. MANDATORY **Propagation**

When the propagation is MANDATORY, if there is an active transaction, then it will be used. If there isn't an active transaction, then Spring throws an exception:

@Transactional(propagation = Propagation.MANDATORY)

**public** **void** **mandatoryExample**(String user) {

// ...

}

Let's again see the pseudo-code:

**if** (isExistingTransaction()) {

**if** (isValidateExistingTransaction()) {

validateExisitingAndThrowExceptionIfNotValid();

}

**return** existing;

}

**throw** IllegalTransactionStateException;

### 3.4. NEVER **Propagation**

For transactional logic with NEVER propagation, Spring throws an exception if there's an active transaction:

@Transactional(propagation = Propagation.NEVER)

**public** **void** **neverExample**(String user) {

// ...

}

Let's see the pseudo-code of how transaction creation works for NEVER propagation:

**if** (isExistingTransaction()) {

**throw** IllegalTransactionStateException;

}

**return** emptyTransaction;

### 3.5. NOT\_SUPPORTED **Propagation**

If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction:

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public** **void** **notSupportedExample**(String user) {

// ...

}

**The JTATransactionManager supports real transaction suspension out-of-the-box. Others simulate the suspension by holding a reference to the existing one and then clearing it from the thread context**

### 3.6. REQUIRES\_NEW **Propagation**

When the propagation is REQUIRES\_NEW, Spring suspends the current transaction if it exists, and then creates a new one:

@Transactional(propagation = Propagation.REQUIRES\_NEW)

**public** **void** **requiresNewExample**(String user) {

// ...

}

**Similar to NOT\_SUPPORTED, we need the JTATransactionManager for actual transaction suspension.**

The pseudo-code looks like so:

**if** (isExistingTransaction()) {

suspend(existing);

**try** {

**return** createNewTransaction();

} **catch** (exception) {

resumeAfterBeginException();

**throw** exception;

}

}

**return** createNewTransaction();

### 3.7. NESTED **Propagation**

For NESTED propagation, Spring checks if a transaction exists, and if so, it marks a save point. This means that if our business logic execution throws an exception, then the transaction rollbacks to this save point. If there's no active transaction, it works like REQUIRED.

**DataSourceTransactionManager supports this propagation out-of-the-box. Some implementations of JTATransactionManager may also support this.**

[**JpaTransactionManager**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaTransactionManager.html)**supports NESTED only for JDBC connections. However, if we set the nestedTransactionAllowed flag to true, it also works for JDBC access code in JPA transactions if our JDBC driver supports save points.**

Finally, let's set the propagation to NESTED:

@Transactional(propagation = Propagation.NESTED)

**public** **void** **nestedExample**(String user) {

// ...

}

## 4. Transaction Isolation

Isolation is one of the common ACID properties: Atomicity, Consistency, Isolation, and Durability. Isolation describes how changes applied by concurrent transactions are visible to each other.

Each isolation level prevents zero or more concurrency side effects on a transaction:

* **Dirty read:** read the uncommitted change of a concurrent transaction
* **Nonrepeatable read**: get different value on re-read of a row if a concurrent transaction updates the same row and commits
* **Phantom read:** get different rows after re-execution of a range query if another transaction adds or removes some rows in the range and commits

We can set the isolation level of a transaction by @Transactional::isolation. It has these five enumerations in Spring: DEFAULT, READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE.

### 4.1. Isolation Management in Spring

The default isolation level is DEFAULT. As a result, when Spring creates a new transaction, the isolation level will be the default isolation of our RDBMS. Therefore, we should be careful if we change the database.

We should also consider cases when we call a chain of methods with different isolation. In the normal flow, the isolation only applies when a new transaction is created. Thus, if for any reason we don't want to allow a method to execute in different isolation, we have to set TransactionManager::setValidateExistingTransaction to true.

Then the pseudo-code of transaction validation will be:

**if** (isolationLevel != ISOLATION\_DEFAULT) {

**if** (currentTransactionIsolationLevel() != isolationLevel) {

**throw** IllegalTransactionStateException

}

}

Now let's get deep in different isolation levels and their effects.

### 4.2. READ\_UNCOMMITTED Isolation

READ\_UNCOMMITTED is the lowest isolation level and allows for the most concurrent access.

As a result, it suffers from all three mentioned concurrency side effects. A transaction with this isolation reads uncommitted data of other concurrent transactions. Also, both non-repeatable and phantom reads can happen. Thus we can get a different result on re-read of a row or re-execution of a range query.

We can set the isolation level for a method or class:

@Transactional(isolation = Isolation.READ\_UNCOMMITTED)

**public** **void** **log**(String message) {

// ...

}

**Postgres does not support READ\_UNCOMMITTED isolation and falls back to READ\_COMMITED instead. Also, Oracle does not support or allow READ\_UNCOMMITTED.**

### 4.3. READ\_COMMITTED Isolation

The second level of isolation, READ\_COMMITTED, prevents dirty reads.

The rest of the concurrency side effects could still happen. So uncommitted changes in concurrent transactions have no impact on us, but if a transaction commits its changes, our result could change by re-querying.

Here we set the isolation level:

@Transactional(isolation = Isolation.READ\_COMMITTED)

**public** **void** **log**(String message){

// ...

}

**READ\_COMMITTED is the default level with Postgres, SQL Server, and Oracle.**

### 4.4. REPEATABLE\_READ Isolation

The third level of isolation, REPEATABLE\_READ, prevents dirty, and non-repeatable reads. So we are not affected by uncommitted changes in concurrent transactions.

Also, when we re-query for a row, we don't get a different result. However, in the re-execution of range-queries, we may get newly added or removed rows.

Moreover, it is the lowest required level to prevent the lost update. The lost update occurs when two or more concurrent transactions read and update the same row. REPEATABLE\_READ does not allow simultaneous access to a row at all. Hence the lost update can't happen.

Here is how to set the isolation level for a method:

@Transactional(isolation = Isolation.REPEATABLE\_READ)

**public** **void** **log**(String message){

// ...

}

**REPEATABLE\_READ is the default level in Mysql. Oracle does not support REPEATABLE\_READ.**

### 4.5. SERIALIZABLE Isolation

SERIALIZABLE is the highest level of isolation. It prevents all mentioned concurrency side effects, but can lead to the lowest concurrent access rate because it executes concurrent calls sequentially.

In other words, concurrent execution of a group of serializable transactions has the same result as executing them in serial.

Now let's see how to set SERIALIZABLE as the isolation level:

@Transactional(isolation = Isolation.SERIALIZABLE)

**public** **void** **log**(String message){

// ...

}

## 5. Conclusion

In this article, we explored the propagation property of @Transaction in detail. We then learned about concurrency side effects and isolation levels.