# **Transaction Management in your application:**

**We have used @Transcational annotation on top of the service method to manage the transactions**.

**Extra-points**: when ever we add @Transcational annotation spring will take care of transactions, it means while we are performing set of actions (unit of work) like saving user information and also order information, if any one action (saving into db) fails both should rollback, it should not save partial data. Only when both actions are success without any exception then only data should be committed to database, we can achieve this by using transaction management, we use spring provided @transcational annotation, spring will manage these transactions.

**Some More Extra-points :**

If we want we can control the transaction’s by applying some configuration on @transcationl annotation properties like - *propagation, isolation, read-only, roll back for exception, timeout*. This is an optional configuration not mandatory. We didn’t not used this in our project.

Example :

@Transactional(propagation = Propagation.REQUIRED)

@Transactional(isolation = Isolation.***DEFAULT***)

**Using the *propagation* property** of transaction, we can define the behaviour of a transaction, *means transaction boundary’s like* method executed as a part of a new transaction each time it is called or part of an existing transaction.

**Using the *isolation* property** of transaction, we can define the execution of concurrent transactions within an application which helps us to maintain data consistency and integrity i.e. we can specify whether 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*)**.

***Propagation Properties or levels***

|  |  |
| --- | --- |
| **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\_Properties or levels*

|  |  |
| --- | --- |
| **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. |

## *How you have handled Exceptions in your Project:*

## We have used @ControllerAdvice and @ExceptionHandler(Exception.class) annotations to handle the exceptions globally over the application, We have created a class GlobalExceptionHandler class(our own class) and ontop of that class we have added @ControllerAdvice annotation and inside the class we have created exception handing methods annotated with @ExceptionHandler and with method parameter as Exception and inside the method we have created ErrorDetails Object(our own obj) and we have set the some of the data like data-time details of exception and exception message (exception.getMessage()) and http status code and this will be sent as response (in Json format in rest)

## When ever any exception occurs that exception will be delegated to GlobalExceptionHandler(ourclass) class then based on appropriate exception it will it will be handled by specific exception handler method. If there is no specific exception handling method means By default it will go to @ExceptionHandler(Exception.class) as Exception is base class for all exceptions

## *Did you created any Custom Exception in your Project ?*

## Yes, I have created in one case like while updating or creating the diagnostic report of a car, first we need to check VIN number (Vehicle information Number ) in DB is present or not ,if it doesn’t found we will be throwing a custom Exception like VINNotFoundException with proper error message.

## We have created a class with VinNotFoundException that extends a Exception class and a parameterised constructor that which takes String type message, and inside constructor we will call super method by passing our error message. So in application when ever we check if VIN==Null then we will throw VinNotFoundException(“VIN Not found”) by using throw keyword.

## *How did you validation in your project ( your bean data or input data or requestbody data)*

## We have used @valid annotation in the controller method parameters along with @requestbody and also we will be applying some of validation annotations like @Size(min=, max=) @Email, @NotNull etc on top of our bean properties , we can also give our own validation message @NotNull(message = "VIN should not be null").

## If in case of above validation fails it will throw MethodArgumentNotValidException we can handle that in our GlobalException handler class with @ExceptionHandler(MethodArgumentNotValidException.class) and we can send the validation error details in response

## spring-boot-starter-validation dependency used for validations we can use javax.validation package annotations or hibernate validator annotations we have used javax.validation package annotations

## Is there any other way we can validate without using @valid annotation ?

## Yes we can do that, by using Errors errors Object of spring validation by using errors.hasErrors() method if it is true we can read those error details from error object reference errors.getAllErrors(), Note: we need to apply validation annotations like @NotNull or @Email on properties

## *By using Which Client you will interact a webservice or how will you communicate with a webservice ?*

## We have used RestTemplate object to interact from our application to external api or(external application) we have used RestTemplate methods like getforobject to access a data or to consume from external api and to send data from our application to external api we have used postforObject method of RestTemplate class.

**How did you secured your RestAPI or can you explain about security concept used in your project?**

we have used Apigee to secure our RestApi. Apigee is an API Management platform where Apigee acts as a proxy server and authorization server where the request will first come to Apigee then it will validate and authorize the request by using oauth2, then our api will be get accessed. We have Apigee Team they will implement this we will just share our rest-endpoint url and input data (by using restapi information in a document or swagger file)

**How will you document your restapi ?**

**Ans 1:** We have documented normally in a document file, by giving all information like rest-end-point url details and functionality description and sample input data, sample response data etc..

**Ans 2:** We have used swagger to document our rest api, by using this we can easily get all the information about rest-end points information and its details, we can also test with some sample data using swgger-ui. We have added springfox-swagger2 and springfox-swagger-ui dependencies in pom.xml and then created a swagger configuration file and we have used annotations like @configuration and @EnableSwagger2 annotaions on top of our swaggerconfig class then we have created a Docket bean by setting all the information like api info details etc and also we have used some of annotations like @ApiOperation to define the functionality a rest-end-point (method) and @ApiResponse to give response status info in controller class and @ApiModelProperty to define properties in model or dto class

Explain about hibernate Associations?

hibernate Associations means Establishing relationship between the entities by using some of annotations like one-one, one-many, many-one, many-many.

For example like one vehicle will have only one unique VIN(vehicle identification number).

One-many: One vehicle will have many defects, many-many: many-cars can were repaired in many dealer service centres , many service centres repairs many cars.

Just check code once for mapping annotations in hibernate

**Extra-points :** You can map each of them as a unidirectional - or bidirectional association , in case of unidirectional we can access data only one way and also change’s done in one side will not impact other means if we delete one side it will not delete other side, but in bi-directional we can navigate in both ways and impact of changes will be on both sides . it defines in which direction you can use the relationship in your domain model

## how you used logger in your project ? why do we need logs?

We have used log4j in our project, we have used logger in each and every method starting and ending with log.info(“start of method”) and log.info(“end of method) to track whether the particular block got executed or not and also in some places we have used log.debug(“input data”+ ourdata) to check input data and also result to get debug report when we run application with debug level log config and also we have used log.error inside the exception blocks to print all the exception’s details.

**Some extra-points:** we use log4j or any other logging mechanism to track the application status like any issues or any bugs in our application, we can store all the logs information in log files, when ever we get any issue in production first we will check the logs from production is there any exceptions or any errors present in log file. Based on exceptions we will find the solution to fix the issue. Some of recent issues like suddenly dealers are not able to login to application and reported the issue as high-priority, so when we checked the production logs we found that we are getting database authorization exception, when we contacted database team they mentioned user name and password has some expiry, need to reset password , after we reset the password and updated new password in config file and restarted the server application is up and running , due to this issue application was down for 5hrs.

## How will read properties in your application?

## We can read in two ways one way by using @value annotation and other way is by using Environment object reference .getproperty() method

## How will you test your code or how will you UnitTest ?

## We use Junit for UnitTesting, we will mainly test the service method’s to test business logic is working as expected or not, we will create test-methods on top of methods we use @test annotation and we will create some objects and we will pass those mock(dummy data) to service method and we will check the expected value with actual value by using assertEquals method if expected value and actual value are same means the test gets passed. We test in this way and also we manually test each functionality.

## Explain Recent task which you did ?

## Recently we got a new enhancement where we need to few more properties(engine data information) in diagnosis report , we have added new properties in existing dto and entity and then we did some modifications in our service layer based on some business conditions.

## Explain about recent bugfix ?

## Same Data base issue answer

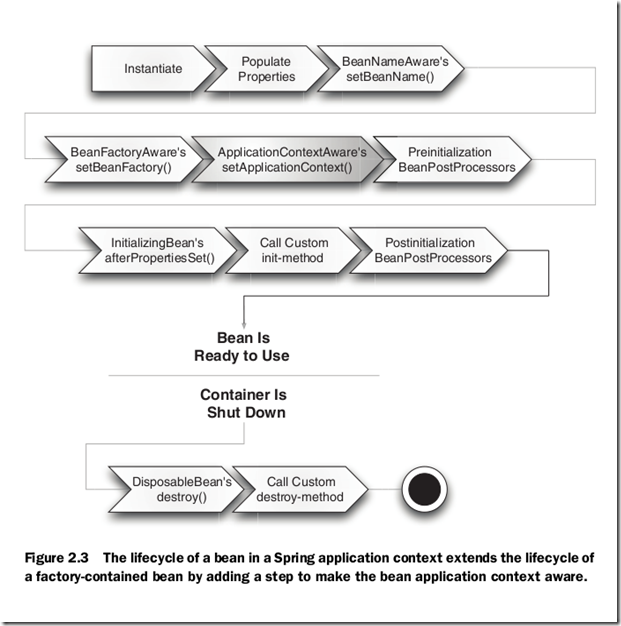
## Explain About Deployment process you have used in your project

## We have used gitlab ci/cd(continues integration and continues deployment) process for deployment ,devops team configured ci/cd in gitlab we are using the same deployment process, if incase we face any issue in deployment with ci/cd devops team will support us.

**Spring Bean life cycle**

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, with help of component scanning spring container will find for annotations like @component, @contoller, @service , @repository ,@bean then the container creates the instance of a bean as per the request and then dependencies are injected with @autowired . And finally, the bean is destroyed when the spring container is closed, with **destroy()** method.

Bean life cycle is as follows:



* The IoC **container instantiates the bean** from the bean’s definition in the XML file or with the help of annotations configuration.
* Spring then **populates all of the properties** using the dependency injection as specified in the bean definition.
* The bean factory container calls setBeanName() which takes the bean ID and the corresponding bean has to implement BeanNameAware interface.
* The factory then calls setBeanFactory() by passing an instance of itself (if BeanFactoryAware interface is implemented in the bean).
* If BeanPostProcessors is associated with a bean, then the preProcessBeforeInitialization() methods are invoked.
* If an init-method is specified, then it will be called.
* Lastly, postProcessAfterInitialization() methods will be called if there are any BeanPostProcessors associated with the bean needs to be run post the creation.

**Thread-local**

ThreadLocal class provides thread-local variables where each thread accesses **its own, independent copy of the variable.**

Even when two or more threads runs the same line of code that has reference to the ThreadLocal variable, the threads cannot see any other Thread's copy of the ThreadLocal variable. Using ThreadLocal we can achieve thread-safety

**Extra-Points:**

way to create a ThreadLocal Variable.

**private** ThreadLocal myThrLocalVariable = **new** ThreadLocal();

**When should I use a ThreadLocal variable?**

One of the common use is when we have a object that is not thread-safe and not want to synchronize on the object, we may use ThreadLocal to have each thread its own instance of the object.

The well known example for ThreadLocal is SimpleDateFormat which is not thread-safe and we may use ThreadLocal to have every thread its own instance of SimpleDateFormat class.

**Difference between ThreadLocal class and volatile keyword in Java**.

Both are completely unrelated.

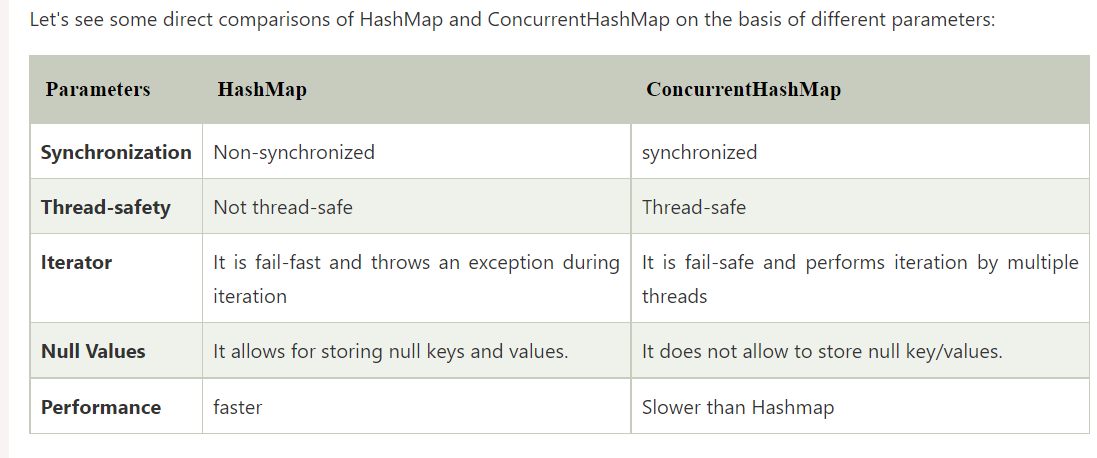
The ThreadLocal class stores a variable in the Thread object of the current running thread, each thread gets its own copy of ThreadLocal instance, the ThreadLocal instances are not shared among the Threads.

Volatile keyword instructs the compiler that this variable be accessed/shared across multiple threads.

How to test a void method with Junit

**Concurrent HashMap:**

* The underlined data structure for ConcurrentHashMap is [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/).
* ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updating the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**.
* Inserting null objects is not possible in ConcurrentHashMap as a key or value.
* In the multi-threaded environment, The ConcurrentHashMap has improved performance than Synchronized HashMap.
* In the single-threaded environment, The HashMap is slightly better than ConcurrentHashMap.
* In HashMap,if one thread is iterating the object and the other thread wants to modify the objects, we will get a ConcurrentModificationException runtime exception. But, in ConcurrentHashMap, one thread can perform modification while the other thread is running.



* **ConcurrentHashMap** is beneficial in a multi-threaded environment and performs better than HashMap. It provides thread-safety, scalability, and synchronization.
* **HashMap** : For the single-threaded environment, the HashMap performs slightly better than ConcurrentHashMap.

**Weak HashMap**

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage-collected when its key is no longer referenced outside of the WeakHashMap.

**HashMap**

In HashMap, we have a key and a value pair. Even though the object is specified as key in hashmap, it does not have any reference and it is **not** eligible for garbage collection if it is associated with HashMap i.e. HashMap dominates over Garbage Collector.

**Spring boot starters**

Starters are a set of dependency descriptors that we can include in our application. Instead of searching for each dependency separately, Starters provide one-stop-shop for all the Spring and other related dependency that we need. Starters helps to manage dependencies jars easily, For example, without starters if you want to create a spring rest application we need to add dependencies like spring-webmvc, Jackson-databind (for data binding), tomcat server where as if we use spring starter web we will get all the dependencies automatically we can easily build application, it provides default embedded container also , if we want database access, just if we include the spring-boot-starter-data-jpa dependency in our project, we will get all other dependencies like hibernate , transaction jars automatically.

## Spring Boot Starter Parent

The spring-boot-starter-parent  is a special starter–

**It also provides default configuration for Maven plugins** (such as *maven-failsafe-plugin*, *maven-jar-plugin*, *maven-surefire-plugin*, *maven-war-plugin*.)

Beyond that, it also inherits dependency management from *spring-boot-dependencies*which is the parent to the s*pring-boot-starter-parent*.

Once, we declare the starter parent in our project**, we don't need to define versions of the dependencies, Maven will download jar files based on the version defined for starter parent in the parent tag.**

For example, if we're building a web project, we can add spring-boot-starter-web directly, and we don't need to specify the version:

**Design patterns** (in your project)

In my project I have used Design patterns like singleton, factory, Template pattern, IOC Pattern, MVC pattern, Front controller pattern, DAO Pattern

**Singleton** means only one instance will be created for a particular class. We use singleton pattern when ever we want to use a single object in entire application for example if we want to access database connection instead of creating connection object for every request, we can use singleton pattern which creates only one instance and same object will be used by all other requests, we can use this pattern in many cases where ever we have shared recourse like reading application properties from a file or creating a database connection etc….

**Factory method pattern** enables to create an object without exposing the creation logic to the client, it is just a method which will create a object and it will return.  
This patterns is used by spring to load beans using BeanFactory and Application context. getBean() is a factory method which returns a bean.

Template Pattern:

Template design pattern is to define an algorithm or a sequence of steps which we use regularly this pattern helps extensively to deal with boilerplate repeated code.

For example for database operations we need to perform few steps repeatedly like

1. Establish a connection
2. Execute query
3. Perform cleanup
4. Close the connection

To avoid such kind of repeated boilerplate code we will define this repeated logic method as a template method so that we can reuse same method ,examples like JdbcTemplate ,HibernateTemplate, RestTemplate

**IOC Pattern** : By using this pattern instead of we create object’s we can delegate responsibility of object creation to some other class, IOC mainly helps us to achieve loose coupling.

**MVC Pattern** MVC Pattern means Model-View-Controller Pattern. This pattern is used to separate application's concerns. so that we can develop each layer separately, so that application can be developed rapidly, code can be easily maintainable.

* **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
* **View** - View represents the visualization of the data that model contains.
* **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

**Front Controller** The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

* **Front Controller** - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
* **Dispatcher** - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.

**DAO Pattern** :The Data Access Object (DAO) pattern is a structural pattern that allows us to **isolate the application/business layer from the persistence layer using an abstract API**.

* The functionality of this API is to hide from the application all the complexities involved in performing database operations in the underlying storage mechanism. This permits both layers to be separately without knowing anything about each other.

**Java 1.7 or 1.8 features**

#### **JavaSE 7 Features**

The important features of JavaSE 7 are try with resource, catching multiple exceptions etc.

* String in switch statement (Java 7)
* The try-with-resources (Java 7)
* Caching Multiple Exceptions by single catch (Java 7)

**Java 8 Features**

The important features of JavaSE 8 are

Lambda-expressions, functional-interface, streams, methods references, default methods classes ,java 8 date/time etc…

**Current Version of java is 16**

**JVM Architecture**

When a java class is compiled .class file will get generated, which contains byte code information, this class file will get loaded by JVM. Inside JVM we have class loader which is responsible for loading class file and verifying class file whether the file is properly formatted and generated by a valid compiler or not and later in Initializationphase, all static variables are assigned with their values defined in the code and static block(if any).JVM contains mains 3 memory areas(stack-area, heap-area, method-area) Stack area where all the local variables will be stored , and in Heap area all the instance variables information like Objects information will be stored  ,and In Method Area, all class level information like static variable’s, class name, static methods and variables information etc. There is only one method area per JVM, and it is a shared resource. Then in Execution Engine Just in Time (JIT) complier read the code line by line and it will execute the code.

How component scan will take base package in spring boot

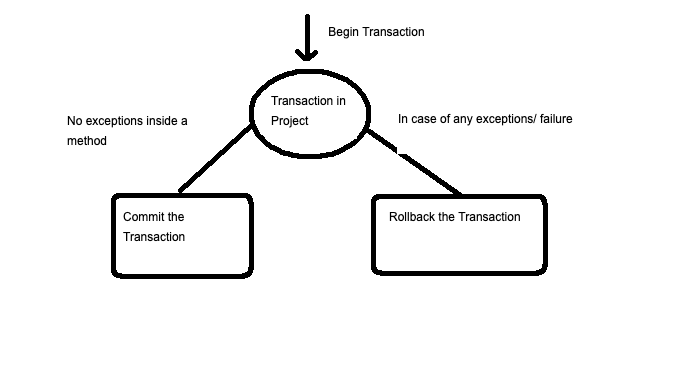
=================================================================================

**Below Theory is just for simple understanding purpose only not required to learn**

**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.

**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();**

**Save(obj);**  
**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**

Transaction transaction = entityManager.getTransaction();  
try{  
**transaction.begin();**

**Save(obj);**  
**transaction.commit();**  
}**catch(Exception e){transaction.rollback();**

**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.

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.

Transaction Propagation indicates if any component or service will or will not participate in transaction and how will it **behave if the calling component/service already has or does not have a transaction created already.**  
  
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*).

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.  
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.

https://www.netjstech.com/2018/08/spring-transaction-attributes-propagation-isolation-settings.html#Propagation