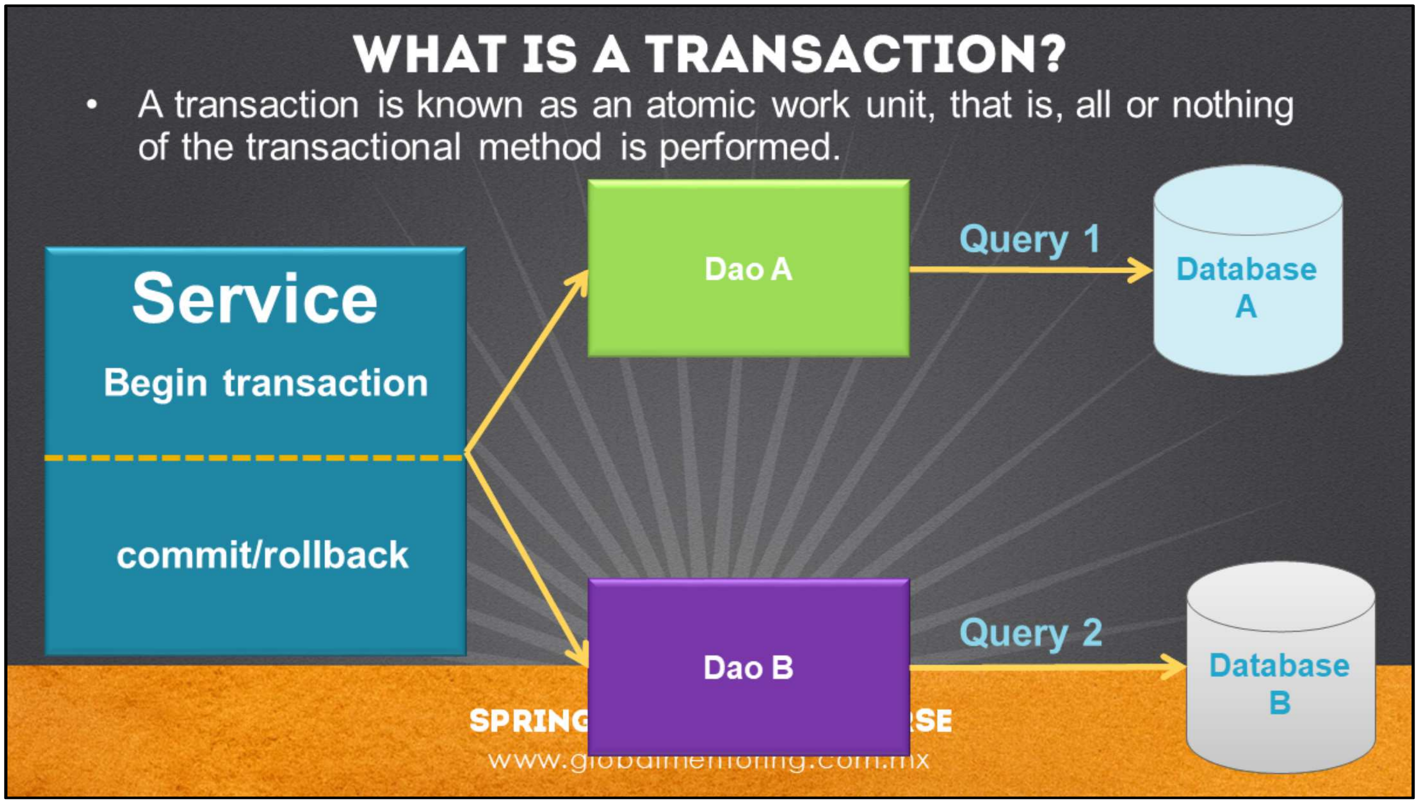
Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson ..

We are going to study the topic of Transaction Management in the Spring Framework.

Are you ready? Come on!



## WHAT IS A TRANSACTION?

- A transaction is known as an atomic work unit, that is, all or nothing of the transactional method is performed.

Transactional Management is one of the crucial issues in terms of requirements for business applications. This motivation arises because in any business system we are interested in maintaining the integrity of our information, with this in mind is that the issue of transactions arises.

In the figure we can see a service method that executes calls to more than one DAO, and in turn each DAO modifies the state of the database when writing and / or modifying its information.

The objective of a transaction is to execute all the lines of code of our method and finally save the information in a repository, for example in our case, a database. This is known as commit of our transaction.

If for some reason something were to fail in our Service method, the changes made to the database would be reversed. This is known as rollback.

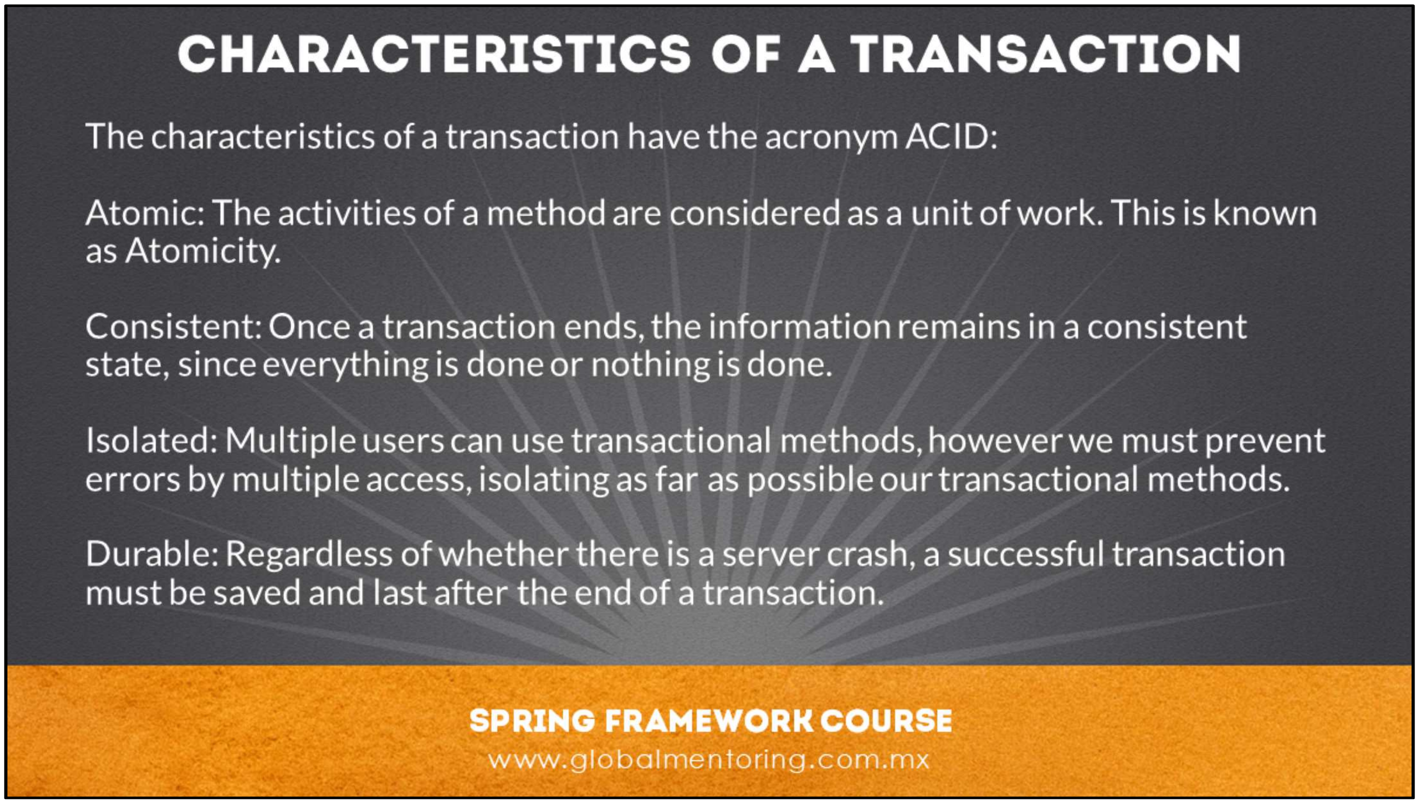
This allows our information, whether a single database or not, is complete, and there is no possibility of corrupted data due to errors or failures in the execution of our Java methods.

For example, imagine a system for selling airline tickets online. In this case, the necessary steps to reserve a ticket are:

1. Verify available tickets
2. Book a ticket
3. Make the payment
4. Receive the ticket information

If for any reason any of the steps fail, then the ticket should not be considered as a sold ticket, but as an available ticket. In case everything has been done successfully, then the ticket is no longer available and the state of the database is updated to reflect one less ticket for new users interested in purchasing a plane ticket.

So the methods that we make part of a transaction will be executed as a single action, which guarantees that they are executed completely, or if they fail, none of the information is persisted. The framework of Spring helps us through AOP to apply transaction management in a very simple way.



## CHARACTERISTICS OF A TRANSACTION

The characteristics of a transaction have the acronym ACID:

Atomic: The activities of a method are considered as a unit of work. This is known as Atomicity.

Consistent: Once a transaction ends, the information remains in a consistent state, since everything is done or nothing is done.

Isolated: Multiple users can use transactional methods, however we must prevent errors by multiple access, isolating as far as possible our transactional methods.

Durable: Regardless of whether there is a server crash, a successful transaction must be saved and last after the end of a transaction.

SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

The characteristics of a transaction have the acronym ACID:

Atomicity: The activities of a method are considered as a unit of work. This is known as Atomicity. This concept ensures that all transactions in a transaction run all or nothing.

If all the instructions or lines of code of a transactional method are executed successfully, then at the end a commit is made, that is, saved from the information.

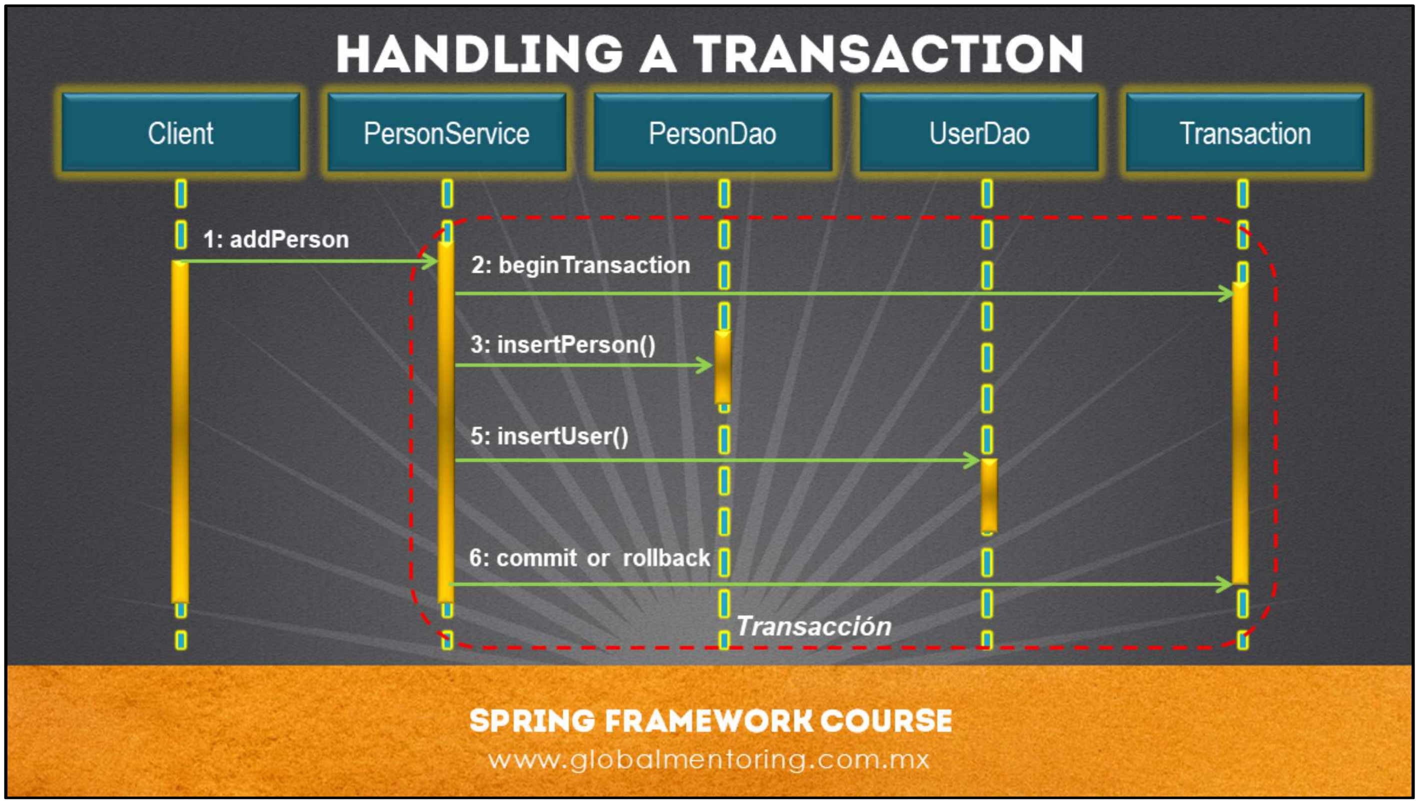
If any of the instructions fails, a rollback is performed, that is, none of the information is stored in the database or the repository where said information persists.

Consistent: Once a transaction ends (regardless of whether it has been successful or not) the information remains in a consistent state, since everything was done or nothing, and therefore the data should not be corrupt in any way.

Isolated: Multiple users can use transactional methods, without affecting the access of other users. However, we must prevent errors by multiple access, isolating as far as possible our transactional methods. Isolation normally involves blocking registers or database tables, this is known as locking.

Durable: Regardless of whether there is a server crash, a successful transaction must be saved and last after the end of a transaction.

In our example of buying tickets, a transaction can ensure atomicity because you can undo all changes made if any of the steps fails. The consistency of data would also be applied to ensure that nothing is left in a partial way, executing all or nothing. Isolation foresees that another user takes the reserved ticket while the transaction has not yet finished. And finally at the time of committing or rollback of the transaction the information is stored without inconsistencies, allowing it to be durable even after the completion of the transaction.

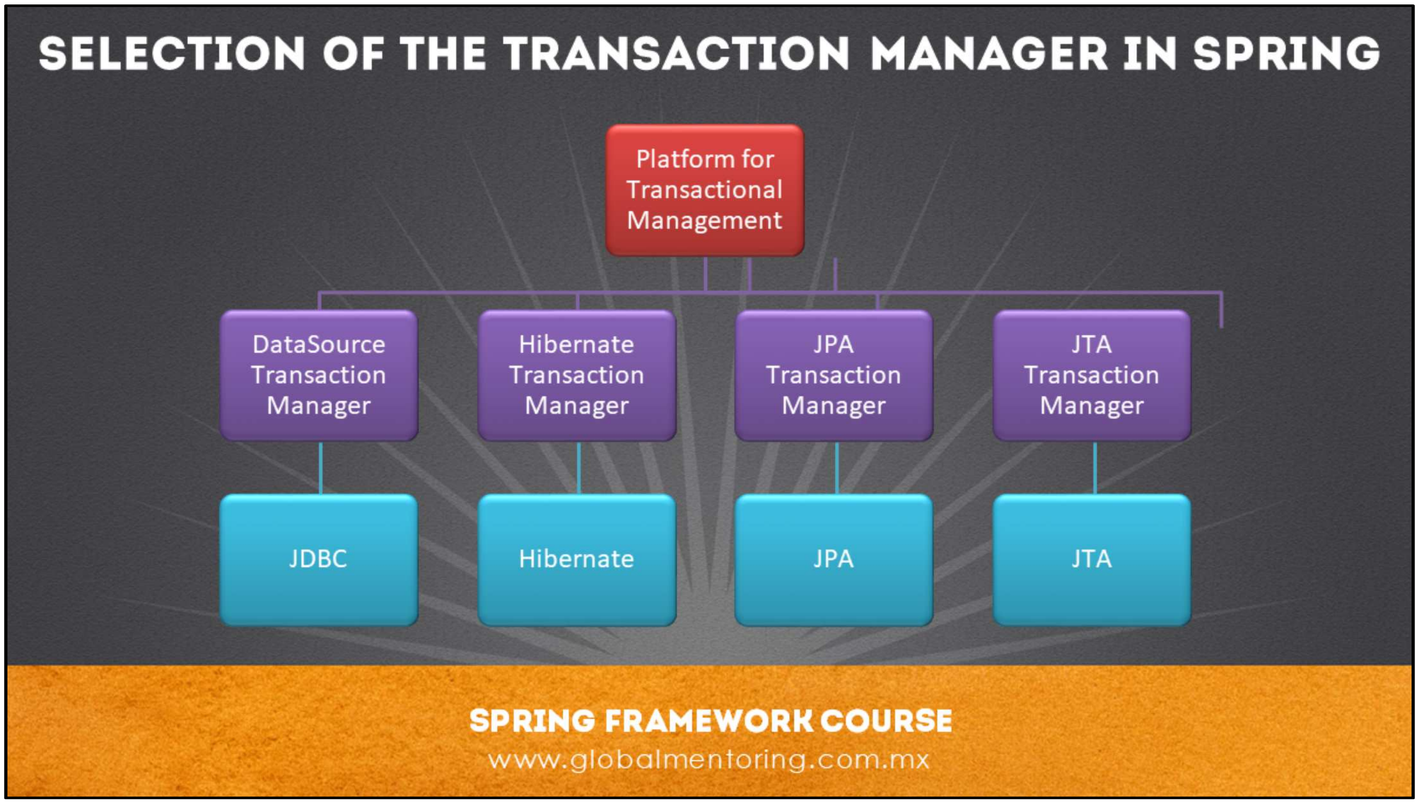


When creating business applications we must pay special attention to persistence. In addition, a business application, as we have studied, is divided into different layers, which have well-defined responsibilities.

In the figure we can see at what moment the Transaction object enters into participation. We observe the 3 layers of a business architecture (Client - Service - Data Access). In this architecture, the transaction begins at the service layer, which is in charge of establishing communication with the Data Layer (DAO).

The data layer is responsible for establishing communication with the database through the Entity objects. The transaction starts from the service layer, and propagates to the data layer. This is because the service layer can communicate with many DAO classes, and therefore the transaction ends until the business method has inserted, modified, deleted and selected all the required data from the database, applying the characteristics of the transaction management that we have commented previously.

The methods of the Service layer are those that contain much of the business logic of the application, and therefore it is at this level that we define transactional management, since if we apply it at the level of the data layer, the commit / rollback handling for each operation of a DAO, would affect the remaining operations that a business method has.


SELECTION OF THE TRANSACTION MANAGER IN SPRING
Platform for Transactional Management
DataSource Transaction Manager
Hibernate Transaction Manager
JPA Transaction Manager
JTA Transaction Manager
JDBC
Hibernate
JPA
JTA
SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

Spring, like the EJB's (Enterprise JavaBeans), provides support for handling transactions both programmatically and declaratively. However Spring adds more features that the EJB's do not provide.

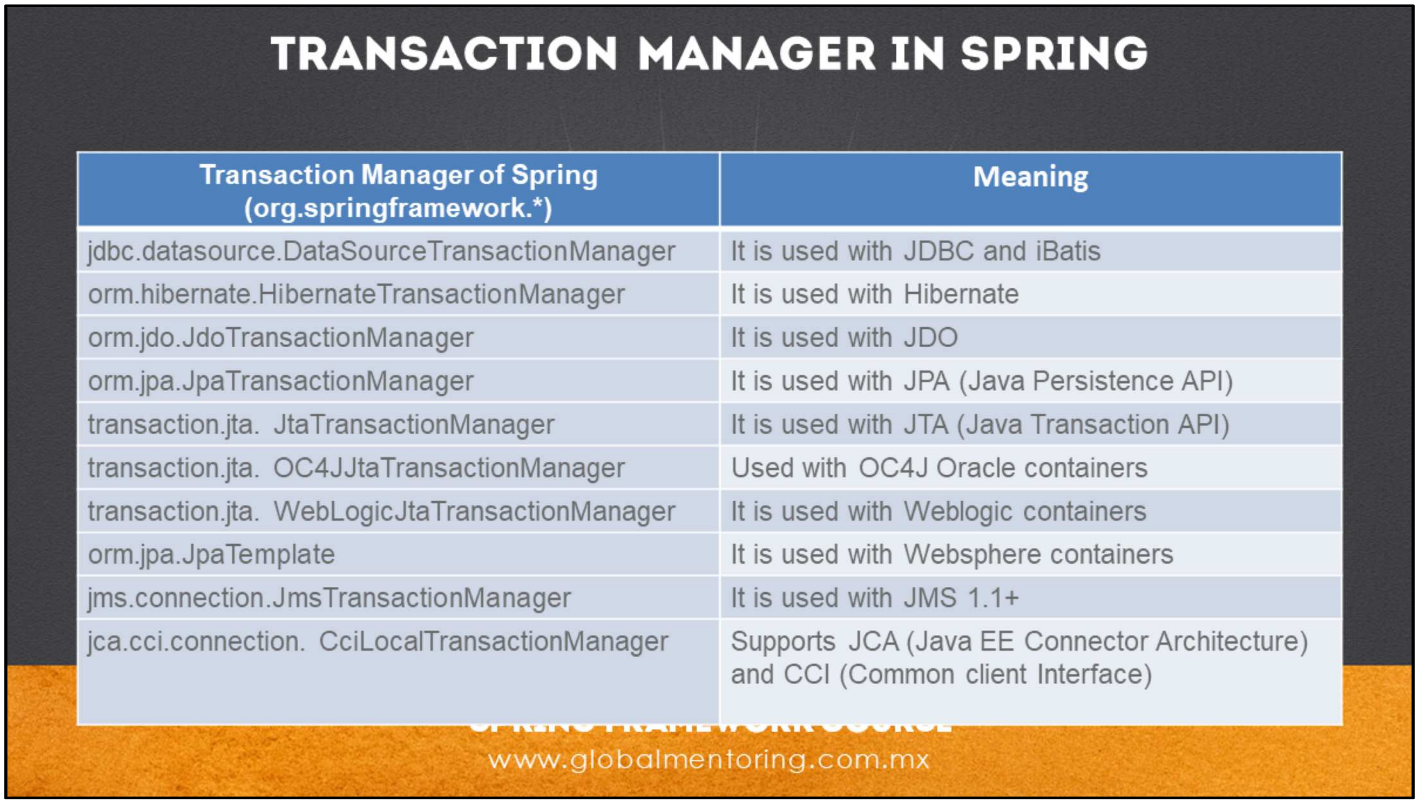
The EJB's are coupled with the JTA API to perform their transaction management work. Spring uses a callback mechanism that abstracts the implementation of the transactional code, so it does not need a JTA implementation to perform the transactional handling.

Spring uses AOP to handle transactions, allowing to define in a very flexible and precise way the limits of a transaction and the way in which they involve the call of a method that we will define as a transaction.

In the figure we can see the options that Spring offers for transactional management, depending on the technology to be used, we can support the support offered by each technology for transactional management.

If our application uses only one database (instead of several), Spring can use the transactional support offered by the selected technology. This includes JDBC, Hibernate and JPA, so a Java business server is NOT required.

On the other hand, if our application uses several databases or JMS messages (Java Message Service), support for distributed transactions (XA) must be used using some JTA implementation and therefore a Java Enterprise Server is required.



## TRANSACTION MANAGER IN SPRING

| Transaction Manager of Spring (org.springframework.*) | Meaning |
|---|---|
| jdbc.datasource.DataSourceTransactionManager | It is used with JDBC and iBatis |
| orm.hibernate.HibernateTransactionManager | It is used with Hibernate |
| orm.jdo.JdoTransactionManager | It is used with JDO |
| orm.jpa.JpaTransactionManager | It is used with JPA (Java Persistence API) |
| transaction.jta. JtaTransactionManager | It is used with JTA (Java Transaction API) |
| transaction.jta. OC4JJtaTransactionManager | Used with OC4J Oracle containers |
| transaction.jta. WebLogicJtaTransactionManager | It is used with Weblogic containers |
| orm.jpa.JpaTemplate | It is used with Websphere containers |
| jms.connection.JmsTransactionManager | It is used with JMS 1.1+ |
| jca.cci.connection. CciLocalTransactionManager | Supports JCA (Java EE Connector Architecture) and CCI (Common client Interface) |

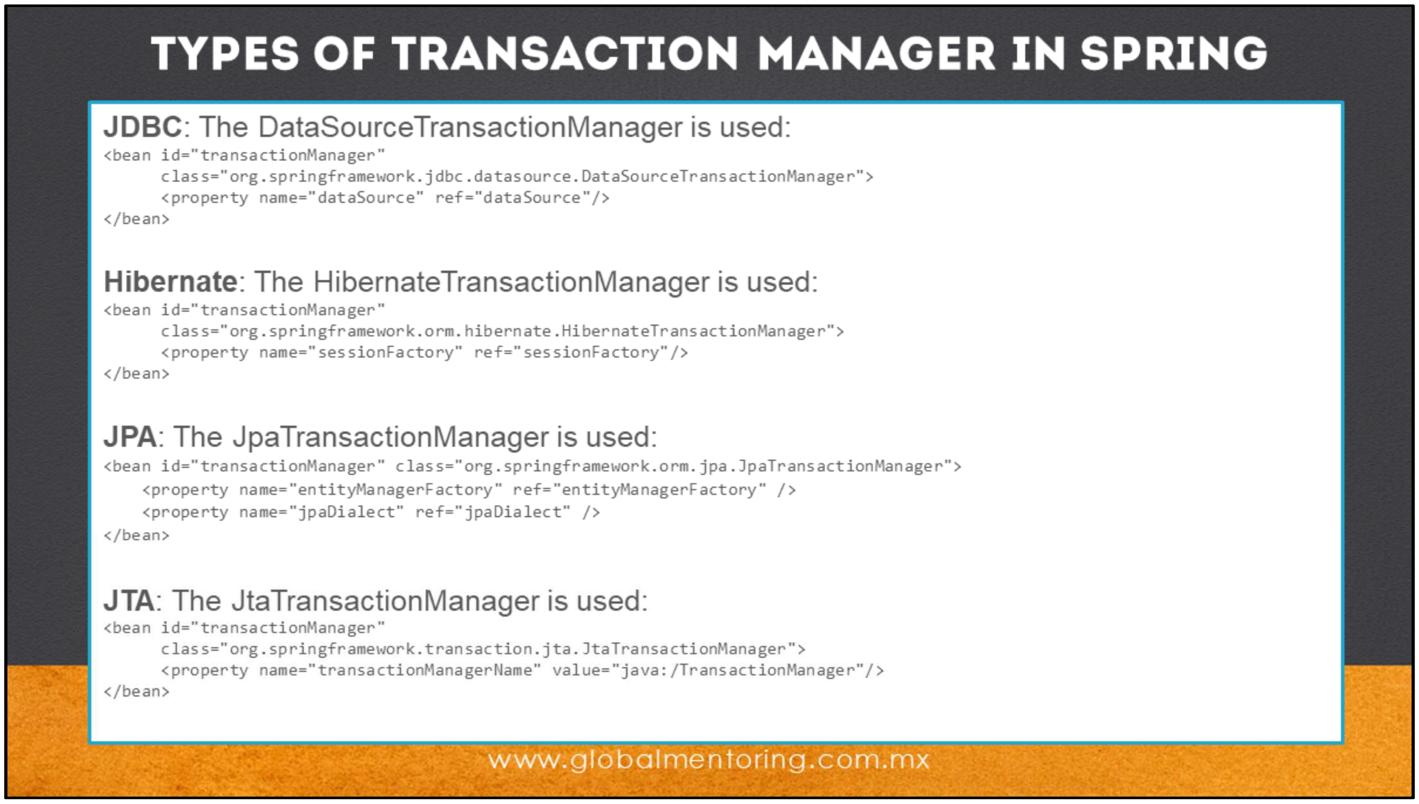
www.globalmentoring.com.mx

Spring support for transactional management is based on the original definition of the EJB's known as CMT (Container-Managed Transactions), although Spring offers greater flexibility compared to the technology of the EJBs.

The decision to use transactional management in a programmatic or declarative manner depends on how much detail in the transaction management we need, however, the declarative way covers most of the requirements, being simpler to configure and add to our projects.

As we can see in the figure, Spring offers a wide range of options for transactional management, and even they are not only support for repositories of Databases, but also messaging using JMS.

Our transactional method can include SQL statements, but also send messages via JMS, and in both cases are considered part of the transaction. In case something fails with any of the instructions to execute, the entire transaction is rolled back. However, to do this we depend on a Java Enterprise Server.



## TYPES OF TRANSACTION MANAGER IN SPRING

**JDBC**: The DataSourceTransactionManager is used:

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

**Hibernate**: The HibernateTransactionManager is used:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

**JPA**: The JpaTransactionManager is used:

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
    <property name="jpaDialect" ref="jpaDialect" />
</bean>
```

**JTA**: The JtaTransactionManager is used:

```
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager">
      <property name="transactionManagerName" value="java:/TransactionManager"/>
</bean>
```

www.globalmentoring.com.mx

To use a transaction manager object, you need to declare it in the application context file of Spring.

In the sheet we can observe the most common configurations to declare our Transaction Manager objects depending on the selected technology.

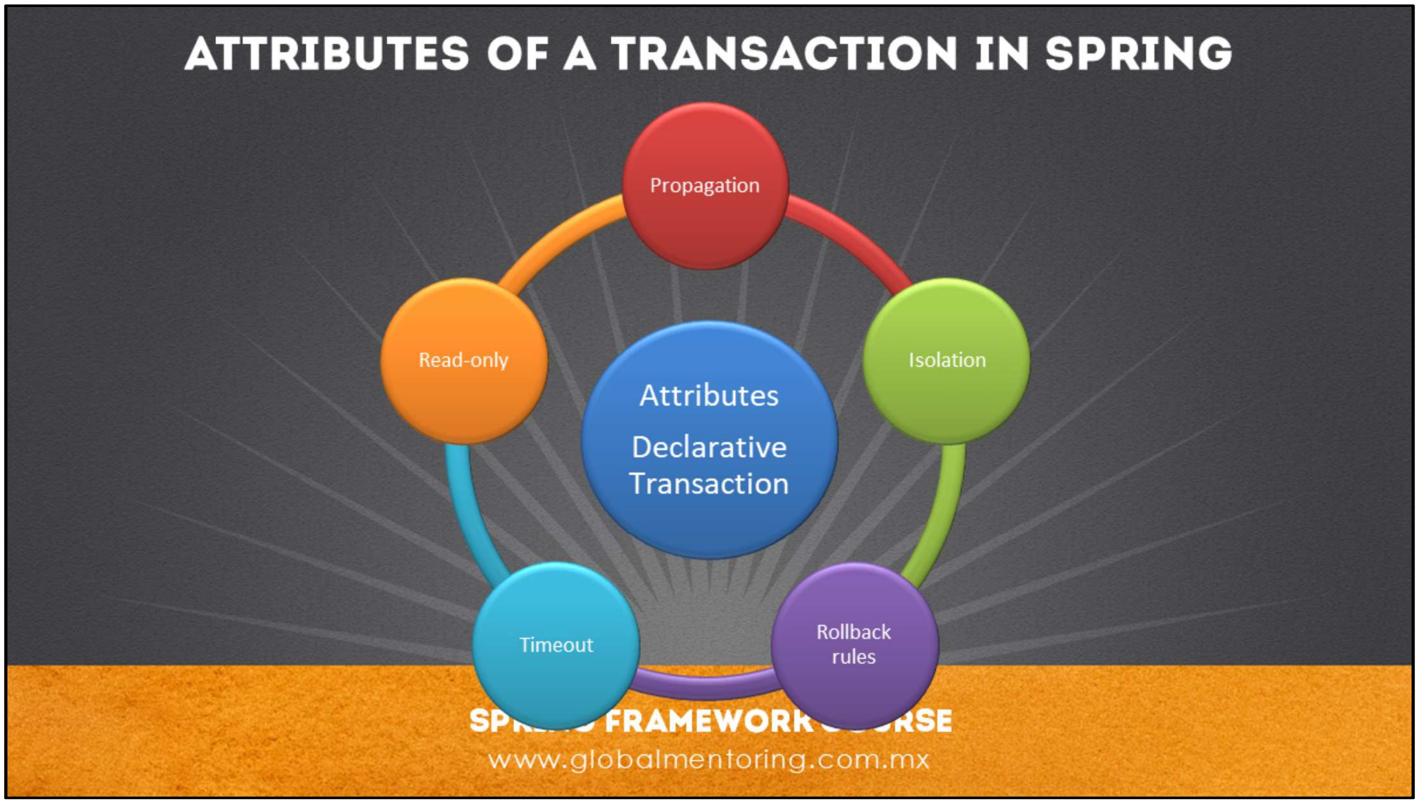
JDBC: If JDBC is used as persistence technology, the DataSourceTransactionManager object allows handling transactionality. This bean needs a reference of the DataSource object, which can be any of those seen in the Spring lesson with JDBC.

Behind, the DataSourceTransactionManager uses the java.sql.Connection object to start (beginTransaction) and conclude the transaction (commit / rollback).

Hibernate: If Hibernate is used for persistence, then the HibernateTransactionManager object can be used for transactional handling. This bean requires a sessionFactory object, which is very similar to the DataSource object. In the topic of Integration with Frameworks, this configuration will be reviewed in detail, as well as the transaction handling with Spring and Hibernate.

JPA: If JPA is used then the JpaTransactionManager object can be used, which needs the entityManagerFactory and JpaDialect objects. JPA supports behind several technologies such as Hibernate, OpenJPA, TopLink, EclipseLink, among others. We will review this topic in the Frameworks Integration chapter.

JTA: If none of the above options covers our needs, or if more than one database is going to be used, then that is when we can use JTA. For this, we rely on the JtaTransactionManager object. There are several implementations of JTA depending on the Java application server that we use, however the search for a JTA resource is done via JNDI in our Java enterprise application server.



In this course we will focus on studying the handling of declarative transactions, because they solve most of the problems.

In Spring, declarative transactions are defined by means of Attributes. These attributes allow to describe the way in which the transaction will behave. There are 5 attributes which we will describe below:

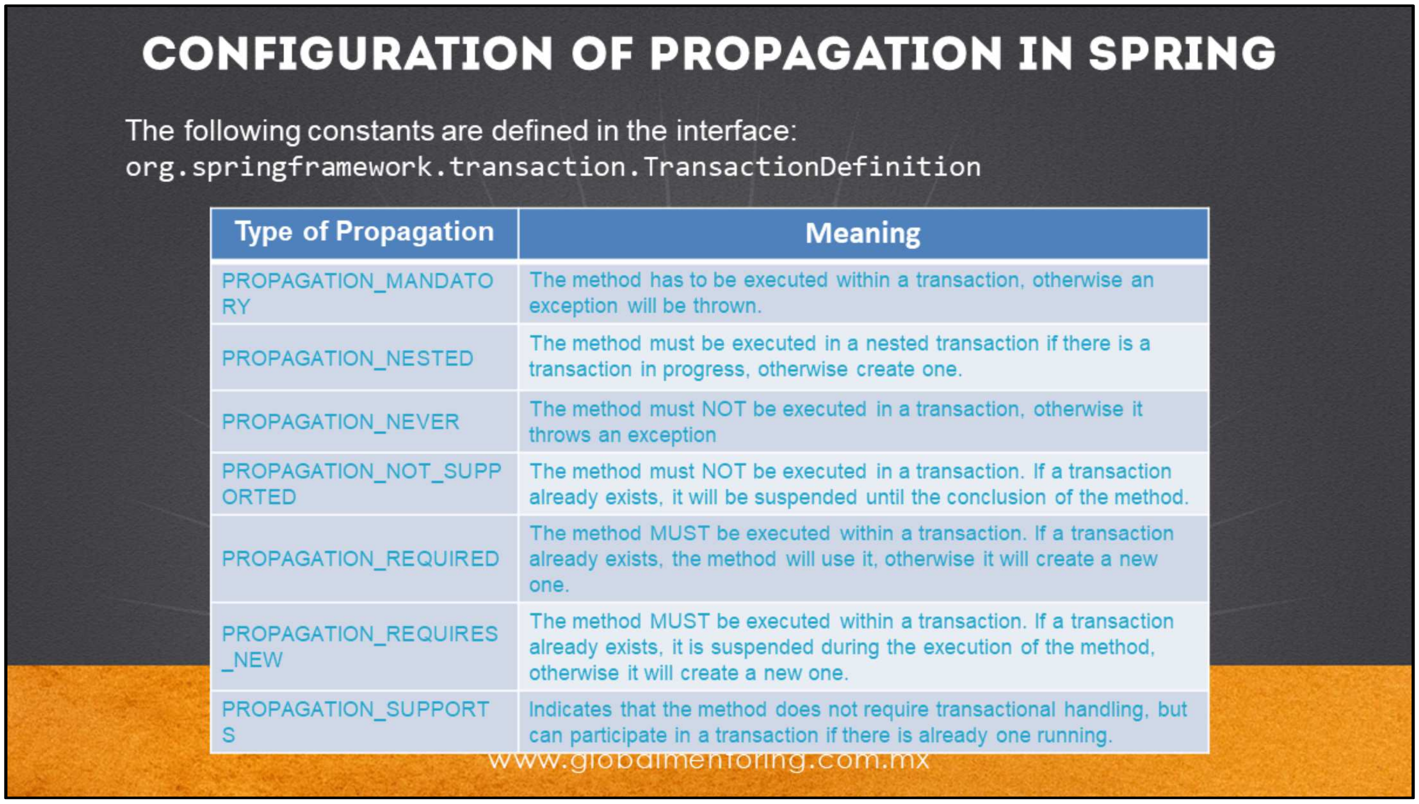
- Propagation: Propagation indicates how a method will behave before a transaction that has been previously initiated in another method, that is, how a transaction will propagate between different transactional methods. Spring defines several constants to specify the type of propagation that a transaction can have.

- Isolation: This attribute defines the level of isolation of the current transaction with respect to other active and concurrent transactions, that is, how much it is going to block access to information (records or basic tables of complete data) of the requested resource, in case that more than one user wants to use the same transactional method concurrently.

- Read-only: This feature allows to indicate if a transactional method will only perform query operations in a database. If the methods are marked with this attribute, the database in question can perform some extra optimizations, because it will only read data. Its default value is readOnly = false.

- Timeout: If a transaction takes too long to execute, it needs to be stopped in some way. The timeout attribute allows to indicate the time elapsed before the transaction expires.

- Rollback rules: By default, transactions apply rollback if the thrown exception is a runtime type. If they are of type checked exceptions, rollback is not applied. If we want to modify this behavior by default, we can use these rollback rules and indicate in which cases, depending on the exception that is thrown, the rollback must be applied or omitted. The advantage is that we can indicate any type of exception, either exceptions of type checked or unchecked (runtime).



## CONFIGURATION OF PROPAGATION IN SPRING

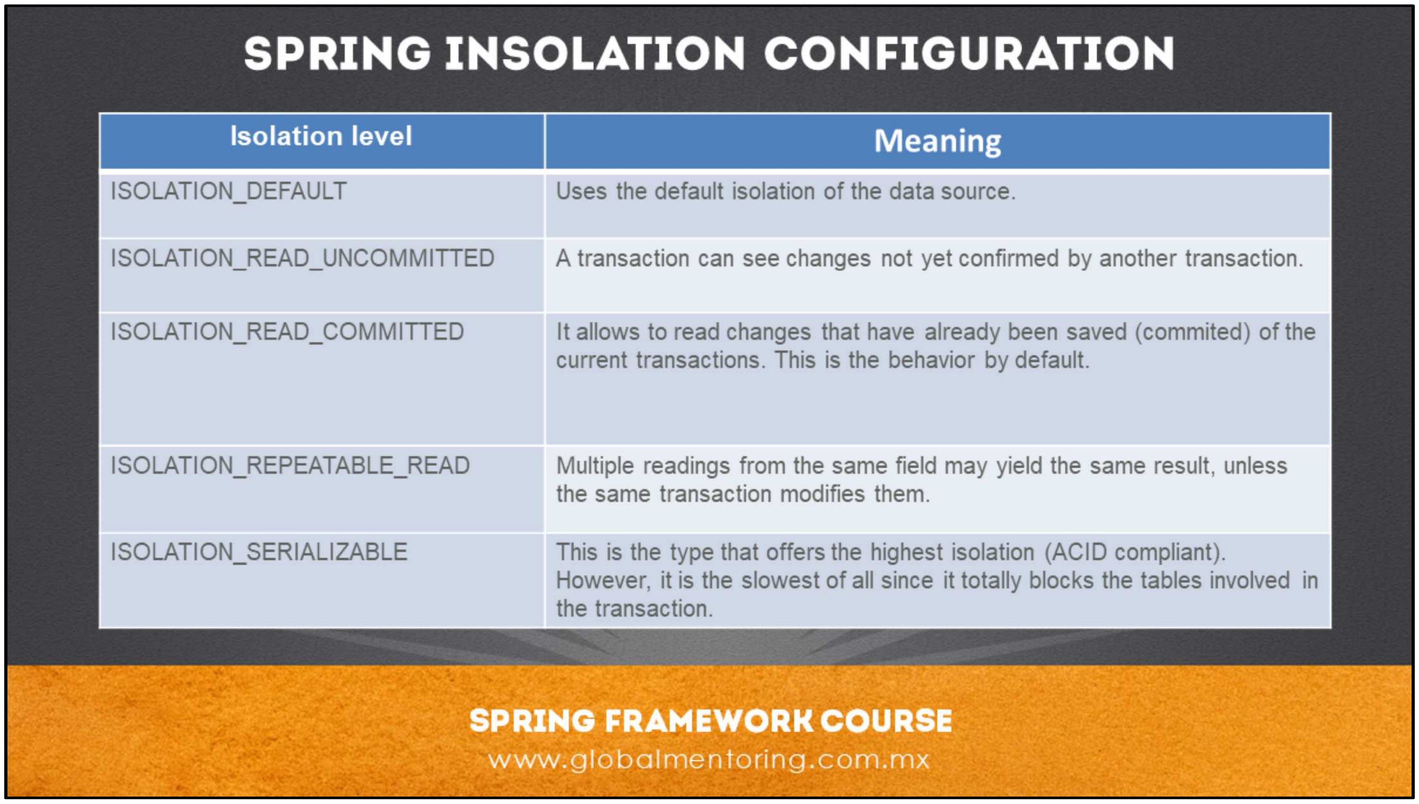
The following constants are defined in the interface:
`org.springframework.transaction.TransactionDefinition`

| Type of Propagation | Meaning |
|---|---|
| PROPAGATION_MANDATORY | The method has to be executed within a transaction, otherwise an exception will be thrown. |
| PROPAGATION_NESTED | The method must be executed in a nested transaction if there is a transaction in progress, otherwise create one. |
| PROPAGATION_NEVER | The method must NOT be executed in a transaction, otherwise it throws an exception |
| PROPAGATION_NOT_SUPPORTED | The method must NOT be executed in a transaction. If a transaction already exists, it will be suspended until the conclusion of the method. |
| PROPAGATION_REQUIRED | The method MUST be executed within a transaction. If a transaction already exists, the method will use it, otherwise it will create a new one. |
| PROPAGATION_REQUIRES_NEW | The method MUST be executed within a transaction. If a transaction already exists, it is suspended during the execution of the method, otherwise it will create a new one. |
| PROPAGATION_SUPPORTS | Indicates that the method does not require transactional handling, but can participate in a transaction if there is already one running. |

Propagation indicates how a method will behave before a transaction that has been previously initiated in another method, that is, how a transaction propagates between transactional methods. The default value of the propagation is REQUIRED.

The types of propagation in a transaction are the following:

- PROPAGATION_MANDATORY: If there is no transaction, an exception is thrown.
- 
- PROPAGATION_NESTED: For this attribute to work as described, the data source must support nested transactions, that is, create one transaction within another.
- PROPAGATION_NEVER: Throws an exception if a transaction is running.
- PROPAGATION_NOT_SUPPORTED: Because it indicates that it does not support transactions, if any, it suspends until the execution of the method marked with this attribute is completed.
- PROPAGATION_REQUIRED: This is the default behavior. If a transaction already exists, it spreads and uses it, otherwise it creates a new one.
- PROPAGATION_REQUIRES_NEW: Should only be used if the action on the database needs to be confirmed (commit) regardless of the result of the transaction in progress, for example, a log in an activity log, for each persistence of information, regardless of whether persists successfully or not.
- PROPAGATION_SUPPORTS: It does not need a transaction, however if one exists it uses one.



## SPRING INSOLATION CONFIGURATION

| Isolation level | Meaning |
|---|---|
| ISOLATION_DEFAULT | Uses the default isolation of the data source. |
| ISOLATION_READ_UNCOMMITTED | A transaction can see changes not yet confirmed by another transaction. |
| ISOLATION_READ_COMMITTED | It allows to read changes that have already been saved (commited) of the current transactions. This is the behavior by default. |
| ISOLATION_REPEATABLE_READ | Multiple readings from the same field may yield the same result, unless the same transaction modifies them. |
| ISOLATION_SERIALIZABLE | This is the type that offers the highest isolation (ACID compliant). However, it is the slowest of all since it totally blocks the tables involved in the transaction. |

SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

The isolation level defines how much a transaction will impact the tasks of another concurrent transaction. Another way to explain it is how much a transaction will allow sharing your data.

In a business application, usually multiple transactions are executed concurrently, often using the same data in order to perform their task. However, concurrent access leads to the following possible problems.

- Dirty data reading (dirty or invalid): This happens when a transaction has written information in the database, but has not committed. If another transaction reads this information, then it is known as dirty information, since it possibly ends in a rollback, considering this reading as invalid.

- Nonrepeatable reading: This happens when you obtain different values in case you want to read the value on different occasions. This happens because another transaction updated the data between read attempts.

- Phantom reading (ghost): It is similar to nonrepeatable reading. It occurs when one transaction (T1) reads the requested records, and another transaction (T2) inserts several records. Therefore, in subsequent queries, the first transaction (T1) will find records that were not initially.

An ideal scheme would be to completely isolate the data while a transaction is running, however this scheme would not be practical since it would block information that other users want to obtain, impacting on performance and blocking data.

Spring provides the following levels of isolation. The default value is ISOLATION_READ_COMMITTED.

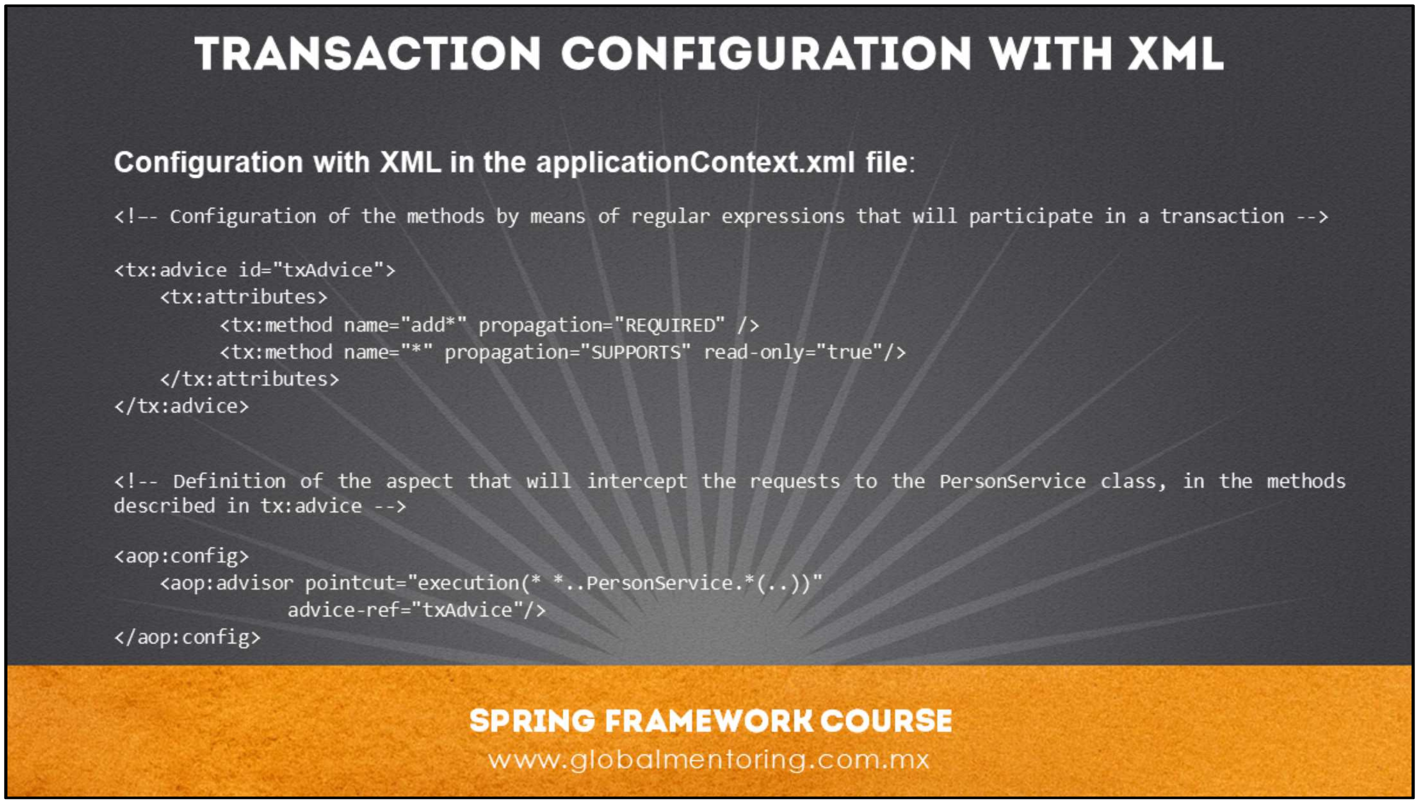
- ISOLATION_DEFAULT: Uses the default isolation of the data source.

- ISOLATION_READ_UNCOMMITTED: A transaction can see changes not yet confirmed by another transaction. There can be dirty, phantom and nonrepeatable readings.

- ISOLATION_READ_COMMITTED: Allows you to read changes that have already been saved (commited) of the current transactions. Dirty type readings are prevented, however phantom and nonrepeatable readings may still be presented. This is the behavior by default.

- ISOLATION_REPEATABLE_READ: Multiple readings in the same field may yield the same result, unless the same transaction modifies them. Dirty and nonrepeatable readings are prevented, however phantom type may still occur.

- ISOLATION_SERIALIZABLE: This is the type that offers the highest isolation (ACID compliant), since it provides readings of type dirty, phantom and nonrepeatable. However, it is the slowest of all since it totally blocks the tables involved in the transaction.



## TRANSACTION CONFIGURATION WITH XML

**Configuration with XML in the applicationContext.xml file:**

```
<!-- Configuration of the methods by means of regular expressions that will participate in a transaction -->

<tx:advice id="txAdvice">
    <tx:attributes>
         <tx:method name="add*" propagation="REQUIRED" />
         <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
</tx:advice>


<!-- Definition of the aspect that will intercept the requests to the PersonService class, in the methods
described in tx:advice -->

<aop:config>
    <aop:advisor pointcut="execution(* *..PersonService.*(..))"
               advice-ref="txAdvice"/>
</aop:config>
```

SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

In the first versions of Spring the Bean TransactionProxyFactoryBean was used, however this resulted in configurations that were more complex than what is really necessary (verbose). However, with the tx namespace, this configuration has been greatly simplified.

In the figure we can see an example of configuration using the namespace tx and its use using Spring AOP.

Basically what we do is define a bean called "txAdvice". This bean adds by means of regular expressions, the methods that must be intercepted and participate in a transaction. They also add the attributes of a transaction in the definition of each method.

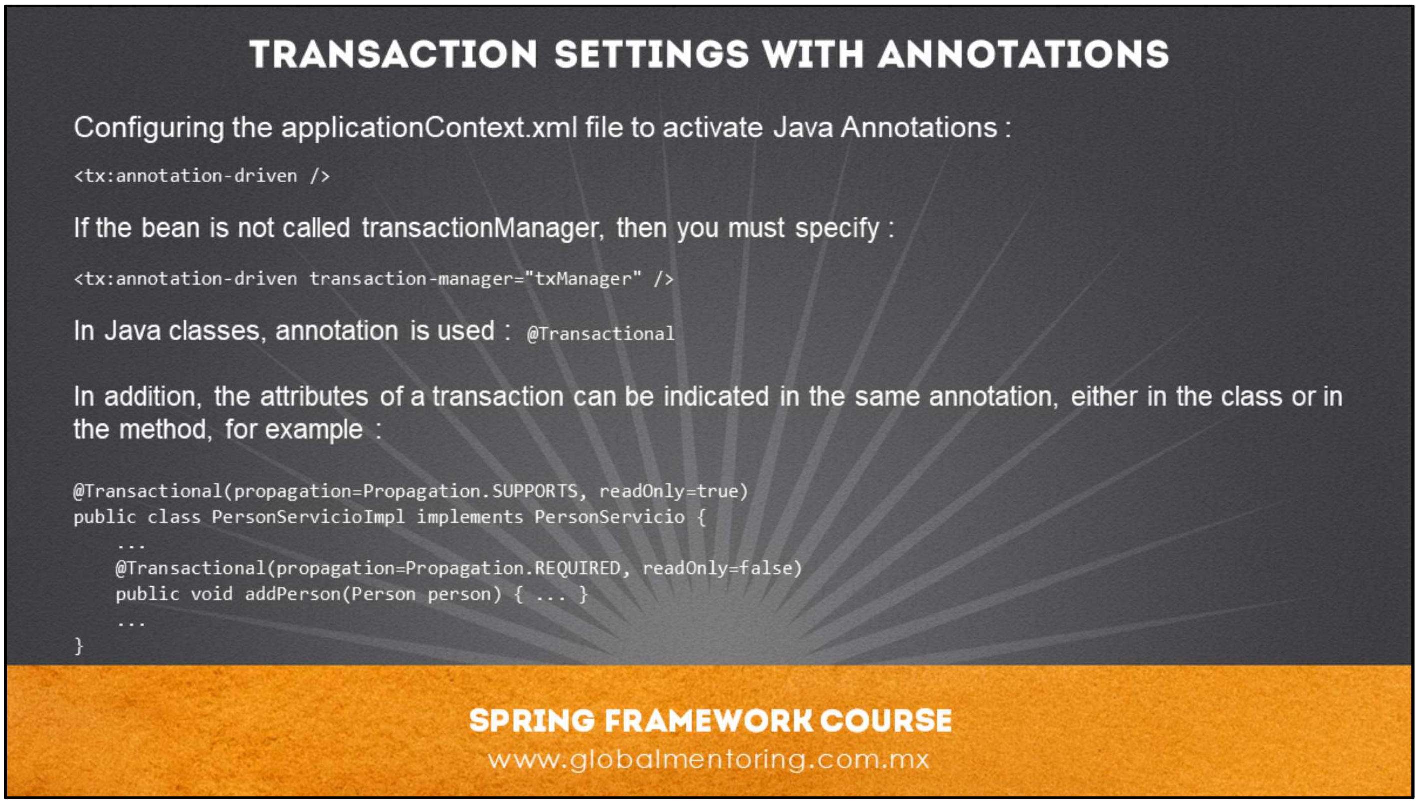
In element tx: advice you must specify the transactionManager bean to be used, however if the name of the bean is the same as the one described, it can be omitted. Otherwise it must be specified as follows:

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

Once we have the configuration of the methods to be intercepted, we create a transactional aspect, as shown by the element aop: config. In this aspect we indicate which is the class or classes that we want to intercept and add the transactional concept.

If we want to intercept all the classes of a particular package we can use the following pointcut:

execution (* com.service. *. * (..))



## TRANSACTION SETTINGS WITH ANNOTATIONS

Configuring the applicationContext.xml file to activate Java Annotations :

```
<tx:annotation-driven />
```

If the bean is not called transactionManager, then you must specify :

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

In Java classes, annotation is used : `@Transactional`

In addition, the attributes of a transaction can be indicated in the same annotation, either in the class or in the method, for example :

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class PersonServicioImpl implements PersonServicio {
    ...
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void addPerson(Person person) { ... }
    ...
}
```

SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

The <tx: annotation-driven> element enables transactional handling with Java Annotations.

In case the transactionManager bean does not have the same name, we must specify it:

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

When the @Transactional annotation is used without any parameters, the propagation mode is REQUIRED, the read-only flag is false, the isolation level of the transaction is READ_COMMITTED, and the transaction is not going to perform a rollback on an exception not -checked (that is, not from java.lang.RuntimeException).

If we want to add that the exceptions of type checked also rollback we can add rollbackFor = Exception.class as an attribute of our annotation. You can also use noRollbackFor to indicate for which exception classes you should NOT apply rollback.

When configuring our classes with the @Transactional annotation, it is common to note the class as shown in the figure, read only, this will apply for all methods, however when re-using it in our methods, it overwrites the values defined at the level of the class, so we can overwrite the value of readOnly = false for the methods that write to the database and not only read information.

ONLINE COURSE

SPRING FRAMEWORK

By: Eng. Ubaldo Acosta

SPRING FRAMEWORK COURSE

www.globalmentoring.com.mx