****Transactions :**** Basically, there are two distinct ways to configure Transactions, annotations and AOP, each with their own advantages.

****Configure Transactions :**Spring 3.1 introduces the **@EnableTransactionManagement** annotation that we can use in a @Configuration class to enable transactional support:**

****@Configuration****

****@EnableTransactionManagement****

**public class **PersistenceJPAConfig**{**

****@Bean****

**public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(){**

**//...**

**}**

****@Bean****

**public PlatformTransactionManager transactionManager(){**

**JpaTransactionManager transactionManager = new JpaTransactionManager();**

**transactionManager.setEntityManagerFactory(entityManagerFactoryBean().getObject() );**

**return transactionManager;**

**}}**

**However, if we're using a Spring Boot project and have a spring-data-\* or spring-tx dependencies on the classpath, then transaction management will be enabled by default.**

**Configure Transactions With XML : For versions before 3.1, or if Java is not an option, here is the XML configuration using annotation-driven and the namespace support:**

**<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">**

**<property name="entityManagerFactory" ref="myEmf" />**

**</bean>**

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

**The @Transactional Annotation : With transactions configured, we can now annotate a bean with @Transactional either at the class or method level.**

****@Service****

****@Transactional****

**public class FooService {**

**//...**

**}**

****The annotation supports further configuration as well:****

* **the Propagation Type of the transaction**
* **the Isolation Level of the transaction**
* **a Timeout for the operation wrapped by the transaction**
* **a readOnly flag – a hint for the persistence provider that the transaction should be read only**
* **the Rollback rules for the transaction**

**Note that by default, rollback happens for runtime, unchecked exceptions only. The checked exception does not trigger a rollback of the transaction. We can, of course, configure this behavior with the **rollbackFor**and **noRollbackFor**annotation parameters.**

**Potential Pitfalls**

**Transactions and Proxies : At a high level, Spring creates proxies for all the classes annotated with **@Transactional,** either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.**

**if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted. Any self-invocation calls will not start any transaction, even if the method has the @Transactional annotation.**

**Another caveat (warning) of using proxies is that only public methods should be annotated with @Transactional. Methods of any other visibilities will simply ignore the annotation silently as these are not proxied.**

### ****Changing the Isolation Level :** @Transactional(isolation = Isolation.SERIALIZABLE)**

****Read-Only Transactions :**It's also important to understand that the readOnly flag is only relevant inside a transaction. If an operation occurs outside of a transactional context, the flag is simply ignored. A simple example of that would call a method annotated with:**

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

**From a non-transactional context, a transaction will not be created and the readOnly flag will be ignored.**

****Transaction Logging :** A helpful method to understand transactional related issues is fine-tuning logging in the transactional packages. The relevant package in Spring is “org.springframework.transaction”, which should be configured with a logging level of TRACE.**

****Transaction Rollback :** We have two ways to rollback a transaction: declarative and programmatic.**

****In the declarative approach**, we annotate the methods with the @Transactional annotation. The @Transactional annotation makes use of the attributes rollbackFor or rollbackForClassName to rollback the transactions, and the attributes noRollbackFor or noRollbackForClassName to avoid rollback on listed exceptions.**

**The default rollback behavior in the declarative approach will rollback on runtime exceptions.**

**Let's see a simple **example** using the declarative approach to rollback a transaction for runtime exceptions or errors:**

****@Transactional****

**public void createCourseDeclarativeWithRuntimeException(Course course) {**

**courseDao.create(course);**

**throw new DataIntegrityViolationException("Throwing exception for demoing Rollback!!!");**

**}**

**Next we'll use the declarative approach to rollback a transaction for the listed checked exceptions. The rollback in our example is on SQLException:**

****@Transactional(rollbackFor = { SQLException.class })****

**public void createCourseDeclarativeWithCheckedException(Course course) throws SQLException {**

**courseDao.create(course);**

**throw new SQLException("Throwing exception for demoing rollback");**

**}**

**Let's see a simple use of attribute noRollbackFor in the declarative approach to prevent rollback of the transaction for the listed exception:**

****@Transactional(noRollbackFor = { SQLException.class })****

**public void createCourseDeclarativeWithNoRollBack(Course course) throws SQLException {**

**courseDao.create(course);**

**throw new SQLException("Throwing exception for demoing rollback");**

**}**

**In the **programmatic** approach, we rollback the transactions using **TransactionAspectSupport**:**

**public void createCourseDefaultRatingProgramatic(Course course) {**

**try {**

**courseDao.create(course);**

**} catch (Exception e) {**

**TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();**

**}**

**}**

**The declarative rollback strategy should be favored over the programmatic rollback strategy.**

****Spring Transaction Management interview questions :****

### ****Have you worked on transaction management in your project :**** YES. I have worked on application level transactions

### **What is a transaction? :** Transaction is a logical unit of work which either must be executed completely or must be rolled back completely. A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency.

### **Could you give me an example of a transaction? :** Sure. The most common example of transaction is “Fund Transfer”.If we are transferring money from our account to someone else’s, then debit from our account and credit of other person’s account, BOTH must happen as part of one transaction.If not, there will be a mismatch in the account books and worst possible user experience.Either, both tasks should get completed or none.

### **What are different characteristics of a transaction? :Note**: Remember the acronym **ACID**.

1. **Atomicity** – Either everything happens or none.A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
2. **Consistency** – All the application state changes should be exactly same as what is desired following all the rules and constraints. This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
3. **Isolation** – Within a transaction, different tasks can execute concurrently or in parallel, but to others, they are visible as a single unit.There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
4. **Durable** – Once application changes are done in the database, it should survive any crashes or failure. Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows −

* Begin the transaction using begin transaction command.
* Perform various deleted, update or insert operations using SQL queries.
* If all the operation are successful then perform commit otherwise rollback all the operations.

**Spring framework** provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both **programmatic and declarative transaction management.** EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

## **Programmatic vs. Declarative :**Spring supports two types of transaction management −

**[Programmatic transaction management](https://www.tutorialspoint.com/spring/programmatic_management.htm) −** This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

**[Declarative transaction management](https://www.tutorialspoint.com/spring/declarative_management.htm) −** This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

**Declarative transaction management** is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

### **What are different components of Spring transaction management?**

1. First thing to know is PlatformTransactionManager
2. TransactionDefinition is other which defines:

* Propagation method
* Time-out
* Read-only

1. TransactionException

### **What are different isolation levels in Spring transaction?**

1. DEFAULT
2. READ\_COMMITTED
3. READ\_UNCOMMITTED
4. REPEATABLE\_READ
5. SERIALIZABLE

### **What are different transaction propagation levels?**

1. ****PROPAGATION\_MANDATORY****: If current transaction exists, supports it otherwise throws exception.
2. ****PROPAGATION\_NESTED****: If current transaction exists, executes within that.
3. ****PROPAGATION\_NEVER****: Throws exception if current transaction exists.
4. ****PROPAGATION\_NOT\_SUPPORTED****: Does not support transaction whether it exists or not.
5. ****PROPAGATION\_REQUIRED****: Supports transaction, if current transaction doesn’t exists it creates a new one.
6. ****PROPAGATION\_REQUIRED\_NEW****: Supports transaction. Always creates a new transaction even if there is already a current transaction in existence.
7. ****PROPAGATION\_SUPPORTS****: If transaction exists, supports it. If not, doesn’t becomes part of the transaction.

### **What is the difference between different Spring transaction isolation levels?**

Isolation levels differs in terms of: **Dirty Read, Phantom Read and non-repeatable Read.**

| **Isolation Level** | **Dirty Read?** | **Phantom Read?** | **Non-Repeatable Read?** |
| --- | --- | --- | --- |
| ****DEFAULT**** | Same as datasource | Same as datasource | Same as datasource |
| ****READ\_COMMITTED**** | No | Might | Might |
| ****READ\_UNCOMMITTED**** | Might | Might | Might |
| ****REPEATABLE\_READ**** | No | Might | No |
| ****SERIALIZABLE**** | No | No | No |

### **transaction management :** An important aspect of transaction management is defining the right transaction boundary, when should a transaction start, when should it end, when data should be committed in DB and when it should be rolled back (in the case of exception).

The most important aspect for developers is to understand how to implement transaction management in an application, in the best way. So now let's explore different ways.

### **Ways of Managing Transactions :** A transaction can be managed in the following ways:

#### **1. Programmatically manage by writing custom code**

This is the legacy way of managing transaction.

EntityManagerFactory factory = Persistence.createEntityManagerFactory("PERSISTENCE\_UNIT\_NAME");

EntityManager entityManager = entityManagerFactory.createEntityManager();

Transaction transaction = entityManager.getTransaction()

try {

transaction.begin();

someBusinessCode();

transaction.commit();

} catch(Exception ex) {

transaction.rollback();

throw ex;

}

**Pros:** The scope of the transaction is very clear in the code.

**Cons:**  It's repetitive and error-prone. Any error can have a very high impact.

A lot of boilerplate needs to be written and if you want to call another method from this method then again you need to manage it in the code.

#### **2. Use Spring to manage the transaction**

Spring supports two types of transaction management:

1. **Programmatic transaction management:** This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
2. **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions. Declarative transactions are highly recommended.

**Now, let us discuss each approach in detail.**

**2.1 Programmatic transaction management:**  The Spring Framework provides two means of programmatic transaction management.

**a. Using the TransactionTemplate (Recommended by Spring Team):**

**Let's see how to implement this type with the help of below code :**

**Context XML file:**

<!-- Initialization for data source -->

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"> <property name="driverClassName" value="com.mysql.jdbc.Driver"/>

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

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

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

</bean>

**<!-- Initialization for TransactionManager -->**

**<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">**

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

**</bean>**

<!-- Definition for ServiceImpl bean -->

<bean id="serviceImpl" class="com.service.ServiceImpl">

<constructor-arg ref="transactionManager"/>

</bean>

**Service Class:**

public class ServiceImpl implements Service{

private final TransactionTemplate transactionTemplate;

// use constructor-injection to supply the PlatformTransactionManager

public ServiceImpl(PlatformTransactionManager transactionManager) {

this.transactionTemplate = new TransactionTemplate(transactionManager);

}

// the transaction settings can be set here explicitly if so desired hence better control

//This can also be done in xml file

this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED); this.transactionTemplate.setTimeout(30); // 30 seconds

// and so forth...

public Object someServiceMethod()

{

return transactionTemplate.execute(new TransactionCallback()

{

// the code in this method executes in a transactional context

public Object doInTransaction(TransactionStatus status)

{

updateOperation1();

return resultOfUpdateOperation2();

}

});

}}

If there is no return value, use the convenient TransactionCallbackWithoutResult class with an anonymous class

**b. Using a PlatformTransactionManager implementation directly:**

Let's see this option again with the help of code.

<!-- Initialization for data source -->

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

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

</bean>

<!-- Initialization for TransactionManager -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

public class ServiceImpl implements Service{

private PlatformTransactionManager transactionManager;

public void setTransactionManager( PlatformTransactionManager transactionManager) {

this.transactionManager = transactionManager;

}

DefaultTransactionDefinition def = new DefaultTransactionDefinition();

// explicitly setting the transaction name is something that can only be done programmatically

def.setName("SomeTxName");

def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);

try {

// execute your business logic here

}catch (Exception ex){

txManager.rollback(status);

throw ex;

}

txManager.commit(status);

}

Now, before going to the next way of managing transactions, lets see how to choose which type of transaction management to go for.

**Choosing between Programmatic and Declarative Transaction Management:**

* Programmatic transaction management is good only if you have a small number of transactional operations. (Most of the times, this is not the case.)
* Transaction name can be explicitly set only using Programmatic transaction management.
* Programmatic transaction management should be used when you want explicit control over managing transactions.
* On the other hand, if your application has numerous transactional operations, declarative transaction management is worthwhile.
* Declarative Transaction management keeps transaction management out of businesslogic,and is not difficult to configure.

**2.2. Declarative Transaction (Usually used almost in all scenarios of any web application) :**

**Step 1:** Define a transaction manager in your Spring application context XML file.

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"/>

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

**Step 2:** Turn on support for transaction annotations by adding below entry to your Spring application context XML file. OR add @EnableTransactionManagement to your configuration class as below:

**@Configuration**

**@EnableTransactionManagement**

**public class AppConfig{**

**...**

**}**

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

The reason for this is if you put an annotation on the Interface Level and if you are using class-based proxies ( proxy-target-class="true" ) or the weaving-based aspect ( mode="aspectj" ), then the transaction settings are not recognized by the proxying and weaving infrastructure .i.e Transactional behavior will not be applied.

**Step 3:**Add the @Transactional annotation to the Class (or method in a class) or Interface (or method in an interface).

<tx:annotation-driven proxy-target-class="true">

Default configuration: **proxy-target-class="false"**

* The @Transactional   annotation may be placed before an interface definition, a method on an interface, a class definition, or a public method on a class.
* If you want some methods in the class (annotated with  @Transactional) to have different attributes settings like isolation or propagation level then put annotation at method level which will override class level attribute settings.
* In proxy mode (which is the default), only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with @Transactional.

**Let us now understand different @Transactional   attributes.**

**@Transactional (isolation=Isolation.READ\_COMMITTED)**

* The default is Isolation.DEFAULT
* Most of the times, you will use default unless and until you have specific requirements.
* Informs the transaction (tx) manager that the following isolation level should be used for the current tx. Should be set at the point from where the tx starts because we cannot change the isolation level after starting a tx.

**DEFAULT**:****Use the default isolation level of the underlying database.

**READ\_COMMITTED:**A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

**READ\_UNCOMMITTED:**This isolation level states that a transaction may read data that is still uncommitted by other transactions.

**REPEATABLE\_READ:**A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

**SERIALIZABLE:**A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.

**What do these Jargons dirty reads, phantom reads, or repeatable reads mean?**

* **Dirty Reads:** Transaction "A" writes a record. Meanwhile, Transaction "B" reads that same record before Transaction A commits. Later, Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.
* ****Non-Repeatable Reads****: Transaction "A" reads some record. Then Transaction "B" writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.
* ****Phantom Reads:****Transaction "A" reads a range of records. Meanwhile, Transaction "B" inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

****@Transactional(timeout=60) :****Defaults to the default timeout of the underlying transaction system. Informs the tx manager about the time duration to wait for an idle tx before a decision is taken to rollback non-responsive transactions.

****@Transactional(propagation=Propagation.REQUIRED) :****If not specified, the default propagational behavior is REQUIRED. Other options are  REQUIRES\_NEW , MANDATORY  , SUPPORTS  , NOT\_SUPPORTED  , NEVER  , and  NESTED .

****REQUIRED:**** Indicates that the target method cannot run without an active tx. If atxhas already been started before the invocation of this method, then it will continue in the same tx or a newtxwould begin soon as this method is called.

****REQUIRES\_NEW :**** Indicates that a newtxhas to start every time the target method is called. If already atxis going on, it will be suspended before starting a new one.

****MANDATORY :**** Indicates that the target method requires an active tx to be running. If atxis not going on, it will fail by throwing an exception.

****SUPPORTS :****

* Indicates that the target method can execute irrespective of atx. If atxis running, it will participate in the same tx. If executed without a tx it will still execute if no errors.
* Methods which fetch data are the best candidates for this option.

****NOT\_SUPPORTED****

* Indicates that the target method doesn’t require the transaction context to be propagated.
* Mostly those methods which run in a transaction but perform in-memory operations are the best candidates for this option.

****NEVER****

* Indicates that the target method will raise an exception if executed in a transactional process.
* This option is mostly not used in projects.

****@Transactional (rollbackFor=Exception.class)****

* Default is rollbackFor=RunTimeException.class
* In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction.
* The problem is only with checked exceptions. So this option can be used to declaratively rollback a transaction if Checked Exception occurs.

****@Transactional (noRollbackFor=IllegalStateException.class)****

* Indicates that a rollback should not be issued if the target method raises this exception.

Now the last but most important step in transaction management is the ****placement of @Transactional annotation****. Most of the times, there is a confusion where should the annotation be placed: at Service layer or DAO layer?

**@Transactional: Service or DAO Layer? :**The Service is the best place for putting @Transactional, service layer should hold the detail-level use case behavior for a user interaction that would logically go in a transaction.

There are a lot of CRUD applications that don't have any significant business logic for them having a service layer that just passes data through between the controllers and data access objects is not useful. In these cases we can put transaction annotation on Dao.

So in practice, you can put them in either place, it's up to you.

Also if you put @Transactional   in DAO layer and if your DAO layer is getting reused by different services then it will be difficult to put it on DAO layer as different services may have different requirements.

If your service layer is retrieving objects using Hibernate and let's say you have lazy initializations in your domain object definition then you need to have a transaction open in service layer else you will face LazyInitializationException  thrown by the ORM.

Consider another example where your Service layer may call two different DAO methods to perform DB operations. If your first DAO operation failed, then the other two may be still passed and you will end up inconsistent DB state. Annotating a Service layer can save you from such situations.

****How Does Spring @Transactional Really Work?****

Spring @Transactional explained :

**Usage and use cases**

@Transactional(value = &quot;myTransactionManager&quot;, propagation = Propagation.REQUIRED)

public void myMethod()

{

...

}

The value attribute of the @Transactional annotation is not mandatory. If not mentionned Spring will look by default for any bean declared in the context with the name “transactionManager” (defaultConvention).

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

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

</bean>

**Registration in Spring context**

<tx:annotation-driven/>

To make the @Transactional annotation work, you should declare the <tx:annotation-driven> tag (tx being the shortcut of the namespace for “http://www.springframework.org/schema/tx“)

**how the <tx:annotation-driven> tag declaration is handled in the Spring context:**

The transaction interceptor class is created.

Then the declared transaction manager in <tx:annotation-driven> is searched in the Spring context and attached to this transaction interceptor

Finally the bean is registered in the Spring context

Spring declares an TransactionAttributeSourceAdvisor bean and registers it into the context.

**@Transactional parsing :** how the @Transactional annotation is parsed during runtime by Spring to retrieve transaction-related properties

The parser will retrieve all attributes of the ****@Transactional**** annotation, among which:

* propagation behavior
* isolation level
* timeout value for the transaction
* ****readOnly**** flag
* and the most important attribute of all: ****value****, which corresponds to the bean name of the transactionManager declared in the Spring context and responsible for the current transaction.

If omitted, the ****value**** attribute defaults to ****“transactionManager”****. When dealing with multiple databases or multiple datasources applications, more than one transactionManager are defined in the Spring context so the ****value**** is important to help Spring choosing the right one.

* First Spring retrieves the transaction attributes
* Then it gets the transaction manager from the Spring context and transaction attributes
* A transaction is created by the underlying entity manager
* The target method is invoked
* After returning from the method invocation, the transaction is committed

##### public abstract class org.springframework.transaction.interceptor.****TransactionAspectSupport****

This class is doing 2 main tasks:

* determine the transaction manager to manage the current transaction, either using the value attribute of the @Transactional annotation or using the transaction-manager attribute of the the tx:annotation-driven tag
* delegates the creation of the transaction to the AbstractPlatFormTransactionManager class

##### abstract class org.springframework.transaction.support.****AbstractPlatformTransactionManager****

The getTransaction() delegates the creation and the start of the transaction itself to the underlying JpaTransactionManager.

We can see here how Spring manages different types of Propagation behavior.

org.springframework.orm.jpa.****JpaTransactionManager****

Most of the important jobs are done in this class.

****Point J : doGetTransaction()****

* first Spring tries to look in the TransactionSynchronizationManager ThreadLocal map to see if there is an existing entity manager using the entity manager factory as search key
* The entity manager factory was injected into the Jpa transaction manager in the Spring XML definition.  
  If this is not done explicitely, Spring will do the job for you during initialization of the transaction manager by looking for a bean named “entityManagerFactory” (default name by convention) in the context.
  + - If an entity manager is found in the ThreadLocal map, Spring wraps it around an ****EntityManagerHolder**** object with a boolean flag *isNew = false* since this entity manager has been created before hand somewhere in the code. (****line 26****)
    - Otherwise the ****EntityManagerHolder**** of the ****JpaTransactionObject**** will be ****null****
    - Spring also retrieves the ****dataSource**** declared for this transaction manager and stores it in the ****JpaTransactionObject**** (****line 46****)

****Point K : doBegin()****

* Spring checks the ****JpaTransactionObject**** to look for an ****EntityManagerHolder****.
  + If not found, Spring delegates the creation of the entity manager to the attached entity manager factory (****line 65****). Then Spring wraps an ****EntityManagerHolder**** object around this entity manager with the flag *isNew = true* to indicate that this entity manager was created in the current transaction and not before (****line 73****)
* Then Spring delegates the creation of a new JDBC transaction to the underlying JPA Dialect (****line 91 to 97****). This dialect is defined in the ****META-INF/persistence.xml**** file for each *persistenceUnit*
* Spring registers the current JDBC connection to the ****TransactionSynchronizationManager**** ThreadLocal map using the dataSource as key (****line 125****)
* If the flag *isNew = true* is set on the ****JpaTransactionObject****, Spring will also register the newly created entity manager to the ****TransactionSynchronizationManager**** ThreadLocal map using the entity manager factory as key.

##### org.springframework.orm.jpa.****HibernateJpaDialect**** extends ****DefaultJpaDialect****

Let’s consider the ****HibernateJpaDialect**** as default Jpa dialect. We can see that this class is calling the superclass ****DefaultJpaDialect**** to start the transaction (****line 9****)

Then it calls the internal method *prepareTransaction()* (****line 12****)

The transaction is started by the entity manager. We can clearly see that only the default ISOLATION level is supported by vanilla ****HibernateJpaDialect****. Any attempt to set the isolation level to something other that ISOLATION\_DEFAULT will trigger an Exception.

The *prepareTransaction()* method is setting and saving previous flush mode, nothing more that that…

This method is just delegating the transaction commit to the transaction manager

Again, apart from calling some trigger code to prepare the commit, the real job of committing is delegated to the method *doCommit()*  
After the commit is done, *cleanupAfterCompletion()* is called to clean up the ****TransactionSynchronizationManager**** ThreadLocal map if necessary

Again, the commit is a plain call to *getEntityManager().getTransaction().commit()*, no magic in it.

Lots of interesting pieces of code here:

* ****Line 6 & 7**** : if the ****JpaTransactionObject**** has its flag *isNew = true* then Spring remove its from the ****TransactionSynchronizationManager**** ThreadLocal map. Indeed *isNew = true* means that the entity manager was created from scratch for this current transaction and now since the transaction is committed there is no reason to keep it in the ThreadLocal map

* Similarly, at ****line 32**** Spring will close gracefully the entity manager if flag *isNew = true*

* If the flag *isNew = false* meaning that the entity manager used in the current transaction has been registered in the ****TransactionSynchronizationManager**** ThreadLocal map before hand, nothing happens. It is not closed and still exists in the ThreadLocal map (****line 35****).

**Summary :**

After digging into the Spring code for ****@Transactional****,

* There is many levels of indirection in the code. A single task like commit requires 3 method calls. I suppose it is due to the open & flexible architecture of Spring which allows end-users to plug their custom implementation of each component. It can also be explained by the fact that the transactional code should be as generic as possible so it can apply not only to JDBC transaction but also to other type of transactions (JMS, Web Services …)
* The ****TransactionSynchronizationManager**** plays the key role in the transaction management. It is acting as a thread-level cache to carry the current entity manager along all layers for the current transaction
* The ****TransactionSynchronizationManager**** public methods may suggest that it can be used programmatically to gain finer control on the lifecycle of the entity manager.

**Pseudo-code for **@Transactional** management:**

* If ****TransactionSynchronizationManager.getResource(emf)**** exists, use it
* Else, retrieve the ****EntityManagerFactory**** and create a new ****EntityManager**** instance from scratch and register it to the ThreadLocal map with ****TransactionSynchronizationManager.bindResource(emf,em)****
* Start a new DB transaction by calling the underlying JPADialect implementation *getJpaDialect().beginTransaction(…)*
* Commit the transaction by calling ****entityManager.getTransaction().commit()****
* If the current entity manager was created from scratch, remove it from the ThreadLocal map and close it
* Else do nothing

**Spring Security :**

**<https://docs.spring.io/spring-security/site/docs/current/reference/html5/#samples>**