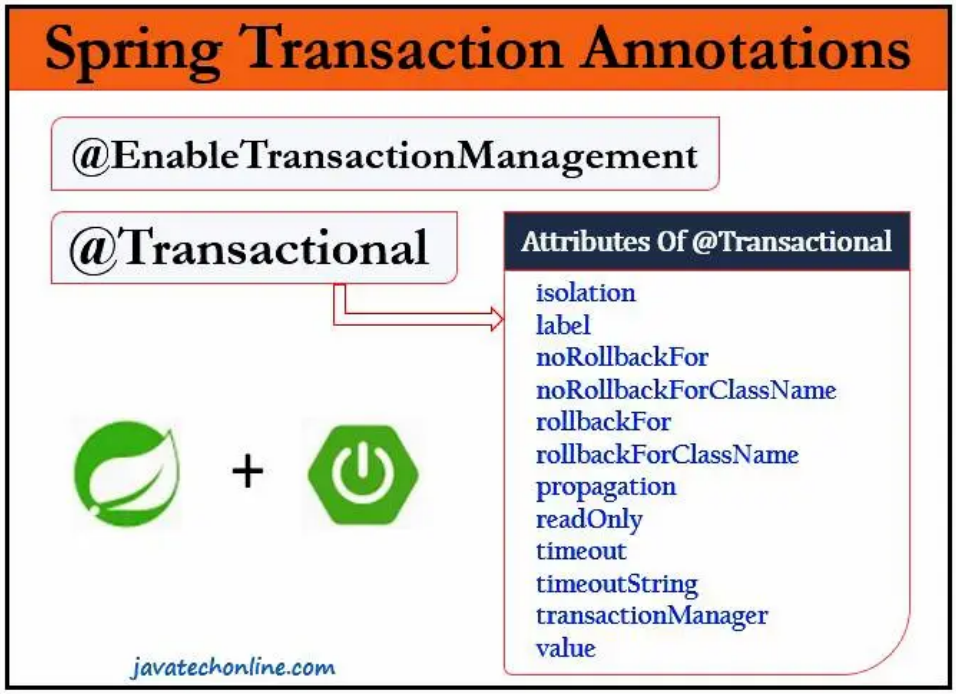
Making Java easy to learn
Java Technology and Beyond

Spring Transaction Annotations With Examples

java Spring Spring Boot *by devs5003 - August 11, 2022* 0



In this article, we will discuss on 'Spring Transaction Annotations With Examples'. Obviously, all Spring transactions related annotations will also work with a Spring Boot Application. These annotations play a very important role while creating a web application in Spring Boot. If you want to learn all annotations which are generally used in a Spring Boot Project, kindly visit our article 'Spring Boot Annotations with Examples'. Let's discuss about 'Spring Transaction Annotations With Examples' here only.

First of all, let's discuss some fundamentals on the term Transaction, and then move ahead to 'Spring Transaction Annotations With Examples'.

Table of Contents

What is a Transaction?

Transactions regulate the modifications that we execute in one or more software systems. These software systems can be databases, message brokers etc. The primary intention of a transaction is to offer ACID characteristics to allow the consistency and validity of our data. ACID is an acronym that stands for atomicity, consistency, isolation, and durability.

Atomicity

Atomicity works on 'all or nothing' principle. Either all operations carried out within the transaction get executed or none of them. In simple words, if your transaction committed successfully, you can be sure that all operations got performed. It also allows you to terminate a transaction and roll back all operations if an error occurs in any one of the operation.

Consistency

The consistency characteristic ensures that your transaction takes a system from one consistent state to another consistent state. That means that either all operations were rolled back and the data was set back to the state you started with or the changed data passed all consistency checks. In a relational

database, that means that the modified data needs to pass all constraint checks, like foreign key or unique constraints, defined in your database.

Isolation

Isolation means that changes that you perform within a transaction are not visible to any other transactions until you commit them successfully.

Durability

Durability ensures that your committed changes get persisted.

Relational databases support ACID transactions, and the JDBC specification allows you to control them. Spring provides annotations and different transaction managers to make it easier to use.

There are two ways to manage transactions: Programmatic and Declarative.

Programmatic vs. Declarative Transaction Management

Spring supports two types of transaction management –

Programmatic Transaction Management

In Programmatic transaction management, we have to manage the transaction with the help of programming. It gives us extreme flexibility, but it is tough to maintain.

Declarative Transaction Management

In declarative transaction management, we separate transaction management from the business code. We only use annotations or XML-based configuration to manage the transactions.

Generally, declarative transaction management is preferred over programmatic transaction management. However, declarative transaction management is less flexible than programmatic transaction management. But as a kind of crosscutting concern, declarative transaction management can be modularized with the help of AOP approach. Spring framework also supports declarative transaction management through its Spring AOP module.

@EnableTransactionManagement

Spring Transaction support is not enabled by default. Therefore, we use the *@EnableTransactionManagement* annotation in a *@Configuration* annotated class to enable Transaction related support. It is similar to the support found in Spring's <tx:*> XML namespace. However, if we are on a Spring Boot Project and already have Spring-data-* or Spring-transaction related dependencies on the classpath, then all the features of this annotation will be available by default. For

example, below code demonstrates the feature:

```
@Configuration
@EnableTransactionManagement
public class AppConfig {
   @Bean
   public DataSource dataSource() {
      // configure and return the necessary JDBC DataSource
   }
   @Bean
   public PlatformTransactionManager txManager() {
      return new DataSourceTransactionManager(dataSource());
   }
}
```

In the above code, *@EnableTransactionManagement* is responsible for registering the necessary Spring components that enable annotation-driven transaction management.

@Transactional

We can annotate a class or method with *@Transactional*. We use the *@Transactional* annotation to separate transaction management code from the code that incorporates business logic. Let's understand it with the help of an example. To illustrate, let's assume that we have to develop a money transfer business logic in a Banking application. Then we will write code something like below:

```
BankTransaction btx = entityManager.getTransaction();
  try {
       btx.begin();
       transferMoney();
       btx.commit();
  }
  catch(Exception ex) {
       btx.rollback();
       throw ex;
  }
```

Now, If we use *@Transactional* annotation, our code will look like below:

```
@Transactional
```

```
public void transferMoney() {
    ...//logic to transfer money
}
```

This is the magic of *@Transactional*. You saw how *@Transactional* has separated the transaction management code from the actual business logic code.

Guidelines to Use @Transactional

1) Make sure to import correct package while applying @Transactional annotation. It comes under package:

```
org.springframework.transaction.annotation //Spring
```

On the other hand, Spring also provides support for @Transaction in the context of JPA. The @Transaction for JPA comes under package:

```
javax.transaction     //JPA
```

Both of them is used to tell container that the method is transactional. Hence, take extra care before importing the package that which one is applicable in your context.

2) We can apply @Transactional at class, interface or method level, but it is not recommended to use it at an interface. However, it is acceptable in cases such as @Repository with Spring Data.

3) When @Transactional annotation is declared at class level, it applies to all public methods of the declaring class by default. However, if we put the annotation on a private or protected method, Spring will ignore it without an error.

4) The @Transactional annotation does not apply to inherited methods of its ancestor classes. To make them transactional, we need to locally redeclare those methods in this class or its subclasses.

5) While applying @Transactional at the method level, make sure that the method is not already a transactional method. For example, if methods in your service class are internally using JpaRepository methods, then it is not required to apply @Transactional at your service class methods. This is because methods provided by JpaRepository are transactional by default.

Attributes of @Transactional

The @Transactional annotation supports various attributes in order to customize the behavior. Let's discuss them one by one.

1) isolation

Isolation represents that how changes applied by concurrent transactions are visible to each other. Each isolation level prevents zero or more concurrency side effects on a transaction. Its type is Isolation. It is an enum with five values:
DEFAULT, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE. The default isolation value is Isolation.DEFAULT. For example:

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
```

2) label

Label is used to describe the transaction. Its type is String [ ]. We can define zero(0) or more transaction descriptions. For example:

```
@Transactional(label = {"Write the Transaction description for the method."})
```

3) noRollbackFor

As the name suggests, it indicates something where no rollback should be applied to the transaction. Actually, it is talking about the exception classes for which there must not be a transaction rollback. It takes zero (0) or more exception Classes, which must be subclasses of Throwable. Remember that the Throwable class is the superclass of all errors and exceptions in the Java language. Therefore, its type is Class<? extends Throwable>[ ]. For example:

```
@Transactional(noRollbackFor = {NullPointerException.class, ClassCastException.class})
```

4) noRollbackForClassName

The purpose of this attribute is the same as for noRollbackFor, but it accepts class names in string form. Hence, its type is String[ ]. In simple words, it takes zero (0) or more exception name patterns, which must be subclasses of Throwable. For example:

```
@Transactional(noRollbackForClassName = {"NullPointerException", "ClassCastException"}
```

5) rollbackFor

This attribute works as the opposite of the noRollbackFor. Actually, it is talking about the exception classes for which there must be a transaction rollback. It takes zero (0) or more exception Classes, which must be subclasses of Throwable. For example:

```
@Transactional(rollbackFor = {NullPointerException.class, ClassCastException.class})
```

6) rollbackForClassName

Similarly, this attribute works as the opposite of the noRollbackForClassName. It accepts zero (0) or more exception name patterns (for exceptions which must be a subclass of Throwable), indicating which exception types must cause a transaction rollback. Its return type is String[ ]. For example:

```
@Transactional(rollbackForClassName = {"NullPointerException", "ClassCastException"})
```

7) propagation

Propagation defines our business logic's transaction boundary. Spring manages to start and pause a transaction according to our propagation setting. The Propagation is an enum type and it has values as MANDATORY, NESTED, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW and SUPPORTS. For example:

```
@Transactional(propagation = Propagation.REQUIRED)
```

8) readOnly

It represents that the transaction is read-only. The default value is false. If it is true, it means that the transaction is read-only. On the other hand, the default value(false) indicates that the transaction is read-write. Obviously, its type is boolean. For example:

```
@Transactional(readOnly = true)
```

9) timeout

The timeout attribute indicates the timeout for the transaction in seconds. Its type is int. For example:

```
@Transactional(timeout = 40)
```

10) timeoutString

The timeoutString attribute indicates the timeout for the transaction in seconds, but its type is String. For example:

```
@Transactional(timeoutString = "40")
```

11) transactionManager

If transaction manager bean name is transactionManager, then @Transactional annotation picks this name by default. If not, it will match qualifier value or bean name of a specific TransactionManager bean definition. For example:

```
@Transactional(transactionManager = "myTransactionManager")
```

12) value

The 'value' attribute is the alias for transactionManager attribute.

That's all about Spring Transaction Annotations With Examples that every developer needs to know.

Tagged @EnableTransactionManagement @enabletransactionmanagement in spring @enabletransactionmanagement vs @transactional @Transactional @transactional annotation @transactional annotation in spring @transactional in spring boot @transactional in spring example spring boot @transactional spring boot annotations interview questions spring boot transaction annotations spring boot transaction management spring boot transactional annotation spring boot transactions spring transaction annotation spring transaction management example spring transactional spring transactional annotation spring transactional example transaction annotation transaction management in spring boot transaction spring boot transactional annotation in spring boot transactional annotation in spring boot example transactional spring boot transactions in spring boot