DZone (/) > Java Zone (/java-jdk-development-tutorials-tools-news) > Effective Spring Transaction Management

Effective Spring Transaction Management

(/users/2756624/nitinprabhu.html) by Nitin Prabhu (/users/2756624/nitinprabhu.html) · Mar. 17, 16 · Java Zone (/java-jdk-development-tutorials-tools-news) · Analysis

Like (48) Comment (8) Save Tweet

Introduction:

Most of the time developers give little importance to transaction management. As a result, lots of code has to be reworked later or a developer implements transaction management without knowing how it actually works or what aspect needs to be used in their scenario.

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");
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. If you want to know the reason for this then read below else jump directly to Declarative transaction management section if you want to implement this option.

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 (taken from Spring docs with some changes).

Please note that the code snippets are referred from Spring Docs.

Context XML file:

```
<!-- Initialization for data source -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">      <property name="driverClassName
<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.transact

  // 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 as follows:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult()
{
protected void doInTransactionWithoutResult(TransactionStatus status)
{
  updateOperation1();
   updateOperation2();
}
});
```

- Instances ofthe `TransactionTemplate` classare thread safe, in these instances do not maintain any conversational state.
- `TransactionTemplate` instances do however maintain configuration state, so while a number of classes may share a single instance ofa `TransactionTemplate` ,if a class needs to use a `TransactionTemplate` with different settings (for example, a different isolation level), then you need to create two distinct TransactionTemplate instances.

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

- 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.

Hope this article helped you