Transactions with Spring and JPA

Last updated: March 17, 2024

Written by: Eugen Paraschiv (https://www.baeldung.com/author/eugen)

Reviewed by: Grzegorz Piwowarek (https://www.baeldung.com/editor/grzegorz-author)

JPA (https://www.baeldung.com/category/persistence/jpa)

Spring Persistence (https://www.baeldung.com/category/persistence/spring-persistence)

JPA Basics (https://www.baeldung.com/tag/jpa-basics)

Transactions (https://www.baeldung.com/tag/transactions)

To learn more about Java features on Azure Container Apps, visit the documentation page (/microsoft-NPI-EA-5-ejfj2).

You can also ask questions and leave feedback on the Azure Container Apps GitHub page (/microsoft-NPI-EA-5-dsak1).

1. Overview

This tutorial will discuss **the right way to configure Spring Transactions**, how to use the *@Transactional* annotation and common pitfalls.

For a more in-depth discussion on the core persistence configuration, check out the Spring with JPA tutorial (/the-persistence-layer-with-spring-and-jpa).

Basically, there are two distinct ways to configure Transactions, annotations and AOP, each with its own advantages. We're going to discuss the more common annotation config here.

Further reading:

Configuring Separate Spring DataSource for Tests (/?post_type=post&p=18989)

A quick, practical tutorial on how to configure a separate data source for testing in a Spring application.

Read more (/?post_type=post&p=18989) →

Quick Guide on Loading Initial Data with Spring Boot (/?post_type=post&p=26851)

A quick and practical example of using data.sql and schema.sql files in Spring Boot.

Read more (/?post_type=post&p=26851) →

Show Hibernate/JPA SQL Statements from Spring Boot (/?post_type=post&p=31899)

Learn how you can configure logging of the generated SQL statements in your Spring Boot application.

Read more (/?post_type=post&p=31899) →

2. 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 entityManagerFactory() {
       //...
   }

   @Bean
   public PlatformTransactionManager transactionManager() {
      JpaTransactionManager transactionManager = new
JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory().getObje
ct());
      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**.

3. 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 namespace support:

```
<bean id="txManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory" ref="myEmf" />
</bean>
<tx:annotation-driven transaction-manager="txManager" />
```

4. 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.

5. Potential Pitfalls (/)

5.1. 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.

What's important to keep in mind is that, 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 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.

5.2. Changing the Isolation Level

```
courseDao.createWithRuntimeException(course);
```

We can also change the transaction isolation level:

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

Note that this has actually been introduced (https://jira.spring.io/browse/SPR-5012) in Spring 4.1; if we run the above example before Spring 4.1, it will result in:

```
org.springframework.transaction.InvalidIsolationLevelException: Standard
JPA does not support custom isolation levels - use a special JpaDialect
for your JPA implementation
```

5.3. Read-Only Transactions

The *readOnly* flag usually generates confusion, especially when working with JPA. From the Javadoc:

```
This just serves as a hint for the actual transaction subsystem; it will
not necessarily cause failure of write access attempts. A transaction
manager which cannot interpret the read-only hint will not throw an
exception when asked for a read-only transaction.
```

The fact is that **we can't be sure that an insert or update won't occur when the *readOnly* flag is set.** This behavior is vendor-dependent, whereas JPA is vendor agnostic.

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.

5.4. 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*.

5.5. Transaction Rollback

The *@Transactional* annotation is the metadata that specifies the semantics of the transactions on a method. 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**.

6. Conclusion

In this article, we covered the basic configuration of transactional semantics using both Java and XML. We also learned how to use *@Transactional*, and the best practices of a Transactional Strategy.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

(/)

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)

PRIVACY MANAGER

(/)