ABOUT ALEKSEY NOVIK

Software developer, Likes to learn new technologies, hang out on stackoverflow and blog on tips and tricks on Java/Javascript polyglot enterprise development.

How does Spring @Transactional Really Work?

Posted by: Aleksey Novik in Enterprise Java June 5th, 2014 0 320 Views

In this post we will do a deep dive into Spring transaction management. We will go over on how does

```
@Transactional
```

really work under the hood. Other upcoming posts will include:

- how to use features like propagation and isolation
- what are the main pitfalls and how to avoid them

JPA and Transaction Management

It's important to notice that JPA on itself does not provide any type of declarative transaction management. When using JPA outside of a dependency injection container, transactions need to be handled programatically by the developer:

```
UserTransaction utx = entityManager.getTransaction();

    try {
        utx.begin();

        businessLogic();

        utx.commit();
    } catch(Exception ex) {
        utx.rollback();
        throw ex;
    }
```

This way of managing transactions makes the scope of the transaction very clear in the code, but it has several disavantages:

- it's repetitive and error prone
- any error can have a very high impact
- errors are hard to debug and reproduce
- this decreases the readability of the code base
- What if this method calls another transactional method?

Using Spring @Transactional

With Spring

```
@Transactional
```

, the above code gets reduced to simply this:

```
@Transactional
    public void businessLogic() {
        ... use entity manager inside a transaction ...
    }
```

This is much more convenient and readable, and is currently the recommended way to handle transactions in Spring.

By using

```
@Transactional
```

, many important aspects such as transaction propagation are handled automatically. In this case if another transactional method is called by

```
businessLogic()
```

, that method will have the option of joining the ongoing transaction.

One potential downside is that this powerful mechanism hides what is going on under the hood, making it hard to debug when things don't work.

What does

```
@Transactional
```

mean?

One of the key points about

```
@Transactional
```

is that there are two separate concepts to consider, each with it's own scope and life cycle:

- the persistence context
- the database transaction

The transactional annotation itself defines the scope of a single database transaction. The database transaction happens inside the scope of a *persistence context*.

The persistence context is in JPA the

```
EntityManager
```

, implemented internally using an Hibernate

```
Session
```

(when using Hibernate as the persistence provider).

The persistence context is just a synchronizer object that tracks the state of a limited set of Java objects and makes sure that changes on those objects are eventually persisted back into the database.

This is a very different notion than the one of a database transaction. One Entity Manager **can be used across several database transactions**, and it actually often is.

When does an EntityManager span multiple database transactions?

The most frequent case is when the application is using the Open Session In View pattern to deal with lazy initialization exceptions, see this previous blog post for it's pros and cons.

In such case the queries that run in the view layer are in separate database transactions than the one used for the business logic, but they are made via the same entity manager.

Another case is when the persistence context is marked by the developer as

```
PersistenceContextType.EXTENDED
```

, which means that it can survive multiple requests.

What defines the EntityManager vs Transaction relation?

This is actually a choice of the application developer, but the most frequent way to use the JPA Entity Manager is with the "Entity Manager per application transaction" pattern. This is the most common way to inject an entity manager:

```
@PersistenceContext
    private EntityManager em;
```

Here we are by default in "Entity Manager per transaction" mode. In this mode, if we use this Entity Manager inside a

```
@Transactional
```

method, then the method will run in a single database transaction.

How does @PersistenceContext work?

One question that comes to mind is, how can

```
@PersistenceContext
```

inject an entity manager only once at container startup time, given that entity managers are so short lived, and that there are usually multiple per request.

The answer is that it can't:

```
EntityManager
```

is an interface, and what gets injected in the spring bean is not the entity manager itself but *a context aware proxy* that will delegate to a concrete entity manager at runtime.

Usually the concrete class used for the proxy is

```
SharedEntityManagerInvocationHandler
```

, this can be confirmed with the help a debugger.

How does @Transactional work then?

The persistence context proxy that implements

```
EntityManager
```

is not the only component needed for making declarative transaction management work. Actually three separate components are needed:

- The EntityManager Proxy itself
- The Transactional Aspect
- The Transaction Manager

Let's go over each one and see how they interact.

The Transactional Aspect

The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method. The concrete class for implementing the aspect is

```
TransactionInterceptor
```

.

The Transactional Aspect has two main responsibilities:

- At the 'before' moment, the aspect provides a hook point for determining if the business method about to be called should run in the scope of an ongoing database transaction, or if a new separate transaction should be started.
- At the 'after' moment, the aspect needs to decide if the transaction should be committed, rolled back or left running.

At the 'before' moment the Transactional Aspect itself does not contain any decision logic, the decision to start a new transaction if needed is delegated to the Transaction Manager.

The Transaction Manager

The transaction manager needs to provide an answer to two questions:

- should a new Entity Manager be created?
- should a new database transaction be started?

This needs to be decided at the moment the Transactional Aspect 'before' logic is called. The transaction manager will decide based on:

- the fact that one transaction is already ongoing or not
- the propagation attribute of the transactional method (for example

  ```
  REQUIRES_NEW
  ```

  always starts a new transaction)

If the transaction manager decides to create a new transaction, then it will:

- create a new entity manager
- bind the entity manager to the current thread
- grab a connection from the DB connection pool
- bind the connection to the current thread

The entity manager and the connection are both bound to the current thread using ThreadLocal variables.

They are stored in the thread while the transaction is running, and it's up to the Transaction Manager to clean them up when no longer needed.

Any parts of the program that need the current entity manager or connection can retrieve them from the thread. One program component that does exactly that is the EntityManager proxy.

The EntityManager proxy

The EntityManager proxy (that we have introduced before) is the last piece of the puzzle. When the business method calls for example

```
entityManager.persist()
```

, this call is not invoking the entity manager directly.

Instead the business method calls the proxy, which retrieves the current entity manager from the thread, where the Transaction Manager put it.

Knowing now what are the moving parts of the

```
@Transactional
```

mechanism, let's go over the usual Spring configuration needed to make this work.

Putting It All Together

Let's go over how to setup the three components needed to make the transactional annotation work correctly. We start by defining the entity manager factory.

This will allow the injection of Entity Manager proxies via the persistence context annotation:

```
@Configuration
    public class EntityManagerFactoriesConfiguration {
        @Autowired
        private DataSource dataSource;

        @Bean(name = "entityManagerFactory")
        public LocalContainerEntityManagerFactoryBean emf() {
            LocalContainerEntityManagerFactoryBean emf = ...
            emf.setDataSource(dataSource);
            emf.setPackagesToScan(
                new String[] {"your.package"});
            emf.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
            return emf;
        }
    }
```

The next step is to configure the Transaction Manager and to apply the Transactional Aspect in

```
@Transactional
```

annotated classes:

```
@Configuration
    @EnableTransactionManagement
    public class TransactionManagersConfig {
        @Autowired
        EntityManagerFactory emf;
        @Autowired
        private DataSource dataSource;

        @Bean(name = "transactionManager")
        public PlatformTransactionManager transactionManager() {
            JpaTransactionManager tm =
                new JpaTransactionManager();
                tm.setEntityManagerFactory(emf);
                tm.setDataSource(dataSource);
            return tm;
        }
    }
```

The annotation

```
@EnableTransactionManagement
```

tells Spring that classes with the

```
@Transactional
```

annotation should be wrapped with the Transactional Aspect. With this the

```
@Transactional
```

is now ready to be used.

Conclusion

The Spring declarative transaction management mechanism is very powerful, but it can be misused or wrongly configured easily.

Understanding how it works internally is helpful when troubleshooting situations when the mechanism is not at all working or is working in an unexpected way.

The most important thing to bear in mind is that there are really two concepts to take into account: the database transaction and the persistence context, each with it's own not readily apparent life cycle.

A future post will go over the most frequent pitfalls of the transactional annotation and how to avoid them.

Reference: How does Spring @Transactional Really Work? from our JCG partner Aleksey Novik at the The JHades Blog blog.

Tagged with: SPRING

(*0* *rating,* *0* *votes*)

You need to be a registered member to rate this. Start the discussion 320 Views Tweet it!

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

Leave a Reply

Start the discussion...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe

"Android Full Application Tutorial" series

KNOWLEDGE BASE

Courses
Examples
Minibooks
Resources

HALL OF FAME

11 Online Learning websites that you should check out
Advantages and Disadvantages of Cloud Computing – Cloud computing pros and

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on crea ultimate Java to Java developers resource center; targeted at the technical ar technical team lead (senior developer), project manager and junior developer JCGs serve the Java, SOA, Agile and Telecom communities with daily news wi domain experts, articles, tutorials, reviews, announcements, code snippets an source projects.

DISCLAIMER