How does LockModeType.OPTIMISTIC work in JPA and Hibernate

JANUARY 26, 2015 / VLADMIHALCEA

(Last Updated On: January 29, 2018)

Explicit optimistic locking

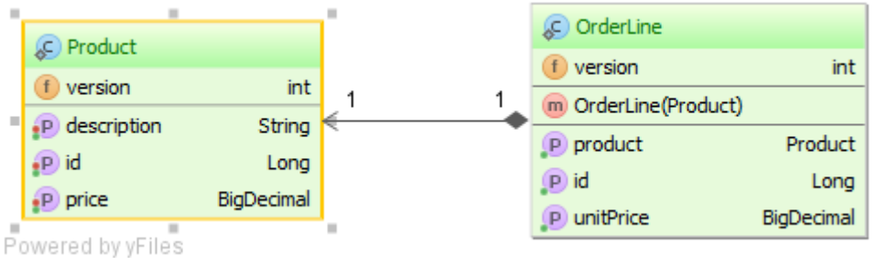
In my previous post, I introduced the basic concepts of Java Persistence locking.

The implicit locking mechanism prevents lost updates and it's suitable for entities that we can actively modify. While implicit optimistic locking is a widespread technique, few happen to understand the inner workings of explicit optimistic lock mode.

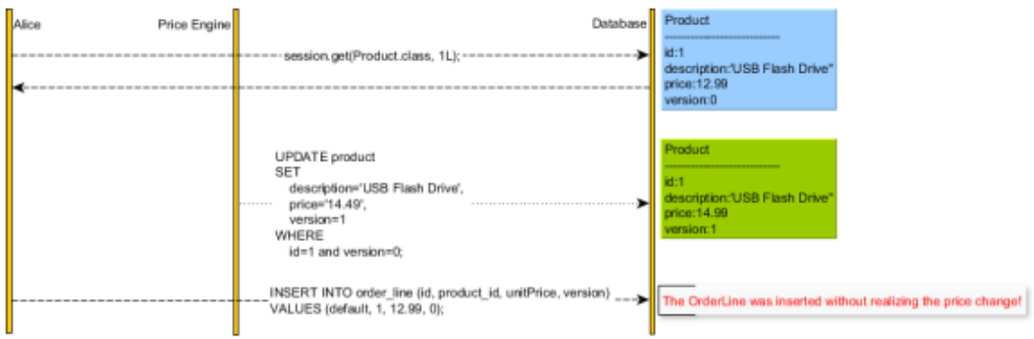
Explicit optimistic locking may prevent data integrity anomalies when the locked entities are always modified by some external mechanism.

The product ordering use case

Let's say we have the following domain model:



Our user, Alice, wants to order a product. The purchase goes through the following steps:



- Alice loads a Product entity
- Because the price is convenient, she decides to order the Product
- the price Engine batch job changes the Product price (taking into consideration currency changes, tax changes and marketing campaigns)
- Alice issues the Order without noticing the price change

Implicit locking shortcomings

First, we are going to test if the implicit locking mechanism can prevent such anomalies. Our test case looks like this:

```
doInTransaction(session -> {
    final Product product = (Product) session.get(Product.class, 1L);
    try {
        executeSync(() -> doInTransaction(_session -> {
            Product _product = (Product) _session.get(Product.class, 1L);
            assertNotSame(product, _product);
            _product.setPrice(BigDecimal.valueOf(14.49));
        }));
    } catch (Exception e) {
        fail(e.getMessage());
    }
    OrderLine orderLine = new OrderLine(product);
    session.persist(orderLine);
});
```

The test generates the following output:

```
#Alice selects a Product
Query:{[select abstractlo0_.id as id1_1_0_, abstractlo0_.description as descript2_1_0_, abstrac

#The price engine selects the Product as well
Query:{[select abstractlo0_.id as id1_1_0_, abstractlo0_.description as descript2_1_0_, abstrac
#The price engine changes the Product price
Query:{[update product set description=?, price=?, version=? where id=? and version=?][USB Flas
#The price engine transaction is committed
DEBUG [pool-2-thread-1]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Alice inserts an OrderLine without realizing the Product price change
Query:{[insert into order_line (id, product_id, unitPrice, version) values (default, ?, ?, ?)][
#Alice transaction is committed unaware of the Product state change
DEBUG [main]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

The implicit optimistic locking mechanism cannot detect external changes unless the entities are also changed by the current Persistence Context. To protect against issuing an Order for a stale Product state, we need to apply an explicit lock on the Product entity.

Explicit locking to the rescue

The Java Persistence LockModeType.OPTIMISTIC is a suitable candidate for such scenarios, so we are going to put it to a test.

Hibernate comes with a LockModeConverter utility, that's able to map any Java Persistence LockModeType to its associated Hibernate LockMode.

For simplicity sake, we are going to use the Hibernate specific LockMode.OPTIMISTIC, which is effectively identical to its Java persistence counterpart.

According to Hibernate documentation, the explicit OPTIMISTIC Lock Mode will:

> assume that transaction(s) will not experience contention for entities. The entity version will be verified near the transaction end.

I will adjust our test case to use explicit OPTIMISTIC locking instead:

```
try {
    doInTransaction(session -> {
        final Product product =
            (Product) session.get(Product.class, 1L, new LockOptions(LockMode.OPTIMISTIC));

        executeSync(() -> {
            doInTransaction(_session -> {
                Product _product = (Product) _session.get(Product.class, 1L);
                assertNotSame(product, _product);
                _product.setPrice(BigDecimal.valueOf(14.49));
            });
        });

        OrderLine orderLine = new OrderLine(product);
        session.persist(orderLine);
    });
```

```
        fail("It should have thrown OptimisticEntityLockException!");
    } catch (OptimisticEntityLockException expected) {
        LOGGER.info("Failure: ", expected);
    }
```

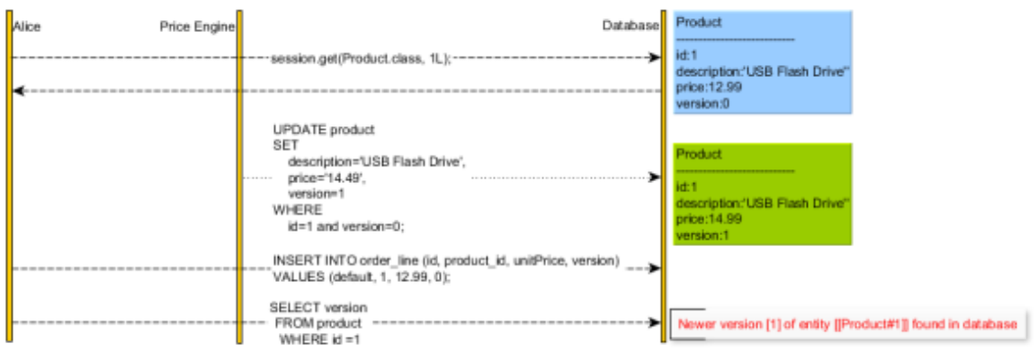
The new test version generates the following output:

```
#Alice selects a Product
Query:{[select abstractlo0_.id as id1_1_0_, abstractlo0_.description as descript2_1_0_, abstrac

#The price engine selects the Product as well
Query:{[select abstractlo0_.id as id1_1_0_, abstractlo0_.description as descript2_1_0_, abstrac
#The price engine changes the Product price
Query:{[update product set description=?, price=?, version=? where id=? and version=?][USB Flas
#The price engine transaction is committed
DEBUG [pool-1-thread-1]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Alice inserts an OrderLine
Query:{[insert into order_line (id, product_id, unitPrice, version) values (default, ?, ?, ?)][
#Alice transaction verifies the Product version
Query:{[select version from product where id =?][1]}
#Alice transaction is rolled back due to Product version mismatch
INFO  [main]: c.v.h.m.l.c.LockModeOptimisticTest - Failure:
org.hibernate.OptimisticLockException: Newer version [1] of entity [[com.vladmihalcea.hibernate
```

The operation flow goes like this:



The Product version is checked towards transaction end. Any version mismatch triggers an exception and a transaction rollback.

Race condition risk

Unfortunately, the application-level version check and the transaction commit are not an atomic operation. The check happens in EntityVerifyVersionProcess, during the *before-transaction-commit* stage:

```
1  public class EntityVerifyVersionProcess implements BeforeTransactionCompletionProcess {
2      private final Object object;
3      private final EntityEntry entry;
4
5      /**
6       * Constructs an EntityVerifyVersionProcess
7       *
8       * @param object The entity instance
9       * @param entry The entity's referenced EntityEntry
10      */
11     public EntityVerifyVersionProcess(Object object, EntityEntry entry) {
12         this.object = object;
13         this.entry = entry;
14     }
15
16     @Override
17     public void doBeforeTransactionCompletion(SessionImplementor session) {
18         final EntityPersister persister = entry.getPersister();
19
20         final Object latestVersion = persister.getCurrentVersion( entry.getId(), session );
21         if ( !entry.getVersion().equals( latestVersion ) ) {
22             throw new OptimisticLockException(
23                     object,
24                     "Newer version [" + latestVersion +
25                             "] of entity [" + MessageHelper.infoString( entry.getEntityName(),
26                             "] found in database"
27             );
28         }
29     }
30 }
```

The AbstractTransactionImpl.commit() method call, will execute the *before-transaction-commit* stage and then commit the actual transaction:

```
@Override
public void commit() throws HibernateException {
    if ( localStatus != LocalStatus.ACTIVE ) {
        throw new TransactionException( "Transaction not successfully started" );
    }

    LOG.debug( "committing" );

    beforeTransactionCommit();

    try {
        doCommit();
        localStatus = LocalStatus.COMMITTED;
        afterTransactionCompletion( Status.STATUS_COMMITTED );
    }
    catch (Exception e) {
        localStatus = LocalStatus.FAILED_COMMIT;
        afterTransactionCompletion( Status.STATUS_UNKNOWN );
        throw new TransactionException( "commit failed", e );
    }
    finally {
        invalidate();
        afterAfterCompletion();
    }
}
```

Between the check and the actual transaction commit, there is a very short time window for some other transaction to silently commit a Product price change.

Conclusion

The explicit OPTIMISTIC locking strategy offers a limited protection against stale state anomalies. This race condition is a typical case of Time of check to time of use data integrity anomaly.

In my next article, I will explain how we can save this example using an *optimistic-to-pessimistic lock upgrade* technique.

Code available on GitHub.

Related

How do LockModeType.PESSIMISTIC_READ and LockModeType.PESSIMISTIC_WRITE work in JPA and Hibernate
February 24, 2015
In "Hibernate"

How to fix optimistic locking race conditions with pessimistic locking
February 3, 2015
In "Hibernate"

A beginner's guide to Java Persistence locking
January 12, 2015
In "Hibernate"

Categories: Hibernate, Java Tags: concurrency control, explicit locking, hibernate, LockModeType.OPTIMISTIC, optimistic locking, Training, Tutorial

← How to get a 10,000 points StackOverflow reputation

How to fix optimistic locking race conditions with pessimistic locking →

One thought on “How does LockModeType.OPTIMISTIC work in JPA and Hibernate”

vladmihalcea says:
MARCH 6, 2018 AT 9:10 PM

You’re welcome.

★ Loading...

REPLY

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Sign me up for the newsletter!

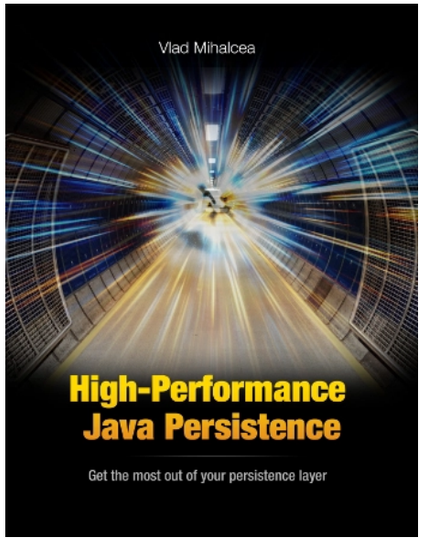
Post Comment

Notify me of follow-up comments by email.

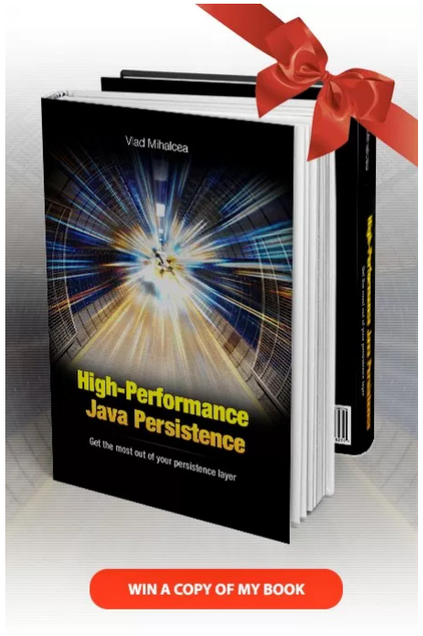
This site uses Akismet to reduce spam. Learn how your comment data is processed.

VIDEO COURSE

HIGH-PERFORMANCE JAVA PERSISTENCE



WIN A COPY OF MY BOOK!

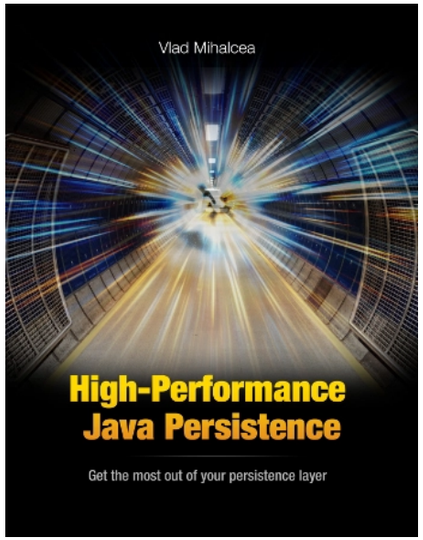


HIBERNATE PERFORMANCE TUNING TIPS

HYPERSISTENCE

WIN A COPY OF MY BOOK!

HIGH-PERFORMANCE JAVA PERSISTENCE



VIDEO COURSE

JDK.IO WORKSHOP – COPENHAGEN

JDKIO
0xCAFEBABE
CONFERENCE
2018

WORKSHOP

High-Performance Java Persistence

RSS - Posts

RSS - Comments

ABOUT

About

Privacy Policy

Terms of Service