Optimistic Locking in JPA

Last modified: March 21, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

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

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

1. Introduction

When it comes to enterprise applications, it's crucial to manage concurrent access to a database properly. This means we should be able to handle multiple transactions in an effective and most importantly, error-proof way.

What's more, we need to ensure that data stays consistent between concurrent reads and updates.

To achieve that we can use optimistic locking mechanism provided by Java Persistence API. It leads that multiple updates made on the same data at the same time do not interfere with each other.

2. Understanding Optimistic Locking

In order to use optimistic locking, **we need to have an entity including a property with *@Version* annotation**. While using it, each transaction that reads data holds the value of the version property.

Before the transaction wants to make an update, it checks the version property again.

If the value has changed in the meantime an *OptimisticLockException* is thrown. Otherwise, the transaction commits the update and increments a value version property.

3. Pessimistic Locking vs. Optimistic Locking

It's good to know that in contrast to optimistic locking JPA gives us pessimistic locking. It's another mechanism for handling concurrent access for data.

We cover pessimistic locking in one of our previous articles – Pessimistic Locking in JPA (/jpa-pessimistic-locking). Let's find out what's the difference and how we can benefit from each type of locking.

As we've said before, **optimistic locking is based on detecting changes on entities by checking their version attribute**. If any concurrent update takes place, *OptmisticLockException* occurs. After that, we can retry updating the data.

We can imagine that this mechanism is suitable for applications which do much more reads than updates or deletes. What is more, it's useful in situations where entities must be detached for some time and locks cannot be held.

On the contrary, pessimistic locking mechanism involves locking entities on the database level.

Each transaction can acquire a lock on data. As long as it holds the lock, no transaction can read, delete or make any updates on the locked data. We can presume that using pessimistic locking may result in deadlocks. However, it ensures greater integrity of data than optimistic locking.

4. Version Attributes

Version attributes are properties with *@Version* annotation. **They are necessary for enabling optimistic locking.** Let's see a sample entity class:

```
@Entity
public class Student {

    @Id
    private Long id;

    private String name;

    private String lastName;

    @Version
    private Integer version;

    // getters and setters

}
```

There are several rules which we should follow while declaring version attributes:

- each entity class must have only one version attribute
- it must be placed in the primary table for an entity mapped to several tables
- type of a version attribute must be one of the following: *int*, *Integer*, *long*, *Long*, *short*, *Short*, *java.sql.Timestamp*

We should know that we can retrieve a value of the version attribute via entity, but we mustn't update or increment it. Only the persistence provider can do that, so data stays consistent.

It's worth noticing that persistence providers are able to support optimistic locking for entities which don't have version attributes. Yet, it's a good idea to always include version attributes when working with optimistic locking.

If we try to lock an entity which does not contain such attribute and the persistence provider does not support it, it will result in a *PersitenceException*.

5. Lock Modes

JPA provides us with two different optimistic lock modes (and two aliases):

- *OPTIMISTIC* – it obtains an optimistic read lock for all entities containing a version attribute
- *OPTIMISTIC_FORCE_INCREMENT* – it obtains an optimistic lock the same as *OPTIMISTIC* and additionally increments the version attribute value
- *READ* – it's a synonym for *OPTIMISTIC*
- *WRITE* – it's a synonym for *OPTIMISTIC_FORCE_INCREMENT*

We can find all types listed above in the *LockModeType* class.

5.1. *OPTIMISTIC* (*READ*)

As we already know, *OPTIMISTIC* and *READ* lock modes are synonyms. However, JPA specification recommends us to use *OPTIMISTIC* in new applications.

Whenever we request the *OPTIMISTIC* lock mode, a persistence provider will prevent our data from dirty reads as well as non-repeatable reads.

Put simply, it should ensure any transaction fails to commit any modification on data that another transaction:

- has updated or deleted but not committed
- has updated or deleted successfully in the meantime

5.2. *OPTIMISTIC_INCREMENT* (*WRITE*)

The same as previously, *OPTIMISTIC_INCREMENT* and *WRITE* are synonyms, but the former is preferable.

OPTIMISTIC_INCREMENT must meet the same conditions as *OPTIMISTIC* lock mode. **Additionally, it increments the value of a version attribute.** However, it's not specified whether it should be done immediately or may be put off until commit or flush.

It's worth to know that a persistence provider is allowed to provide *OPTIMISTIC_INCREMENT* functionality when *OPTIMISTIC* lock mode is requested.

6. Using Optimistic Locking

We should remember that for versioned entities optimistic locking is available by default. Yet there are several ways of requesting it explicitly.

6.1. Find

To request optimistic locking we can pass the proper *LockModeType* as an argument to find method of *EntityManager*:

```
entityManager.find(Student.class, studentId, LockModeType.OPTIMISTIC);
```

6.2. Query

Another way to enable locking is using the *setLockMode* method of *Query* object:

```
Query query = entityManager.createQuery("from Student where id = :id");
query.setParameter("id", studentId);
query.setLockMode(LockModeType.OPTIMISTIC_INCREMENT);
query.getResultList()
```

6.3. Explicit Locking

We can set a lock by calling EnitityManager's *lock* method:

```
Student student = entityManager.find(Student.class, id);
entityManager.lock(student, LockModeType.OPTIMISTIC);
```

6.4. Refresh

We can call the *refresh* method the same way as the previous method:

```
Student student = entityManager.find(Student.class, id);
entityManager.refresh(student, LockModeType.READ);
```

6.5. NamedQuery

The last option is to use @NamedQuery with the *lockMode* property:

```
@NamedQuery(name="optimisticLock",
  query="SELECT s FROM Student s WHERE s.id LIKE :id",
  lockMode = WRITE)
```

7. *OptimisticLockException*

When the persistence provider discovers optimistic locking conflicts on entities, it throws *OptimisticLockException*. We should be aware that due to the exception the active transaction is always marked for rollback.

It's good to know how we can react to *OptimisticLockException*. Conveniently, this exception contains a reference to the conflicting entity. **However, it's not mandatory for the persistence provider to supply it in every situation**. There is no guarantee that the object will be available.

There is a recommended way of handling the described exception, though. We should retrieve the entity again by reloading or refreshing. Preferably in a new transaction. After that, we can try to update it once more.

8. Conclusion

In this tutorial, we got familiar with a tool which can help us orchestrate concurrent transactions. **Optimistic locking uses version attributes included in entities to control concurrent modifications on them.**

Therefore, it ensures that any updates or deletes won't be overwritten or lost silently. Opposite to pessimistic locking, it doesn't lock entities on the database level and consequently, it isn't vulnerable to DB deadlocks.

We've learned that optimistic locking is enabled for versioned entities by default. However, there are several ways of requesting it explicitly by using various lock mode types.

Another fact we should remember is that each time there are conflicting updates on entities, we should expect an *OptimisticLockException*.

Lastly, the source code of this tutorial is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/hibernate-jpa).