JPA, Asynchronous Processing, and "Leaky Abstractions"

JPA excludes the possibility of an easy solution for the concurrent modification problem. JPA is a perfect example of "leaky abstraction" described by Joel Spolsky.

by Jakub Kubrynski MVB · Nov. 09, 15 · Java Zone · Tutorial

A few years ago in the Java world it was almost obvious that every "enterprise" class project needed JPA to communicate with a database. JPA is a perfect example of "leaky abstraction" described by Joel Spolsky. Great and easy at the beginning but hard to tune and limiting at the end. Hacking and working directly with caches, flushes, and native queries is a daily routine for many backend developers involved in the data access layer. There are enough problems and workarounds to write a dedicated book "JPA for hackers", but in this article I'll focus only on concurrent entity processing.

Let's assume this situation: we have ***Person*** entity in which some business process is updated by some service.

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String uuid = UUID.randomUUID().toString();

    private String firstName;

    private String lastName;

    // getters and setters

}
```

To ignore any domain complexity, we're talking about updating the first and last name of the person. Of course it's just a trivial use case but it allows us to focus on the real issues instead of discussing domain modeling. We can imagine that the code looks like the snippet below:

```
firstNameUpdater.update(personUuid, "Jerry");
lastNameUpdater.update(personUuid, "Newman");
```

After some time, business managers decide it's taking too long to update both elements, so reducing duration becomes top priority. Of course, there are a lot of different ways of doing it, but let's assume that in this particular case, going concurrent will solve our pain.

This seems to be trivially easy. We just need to annotate our service methods with ***@Async*** from Spring and *voilà*—problem solved. Really? We have two possible issues here depending on the use of the optimistic locking mechanism.

- With optimistic locking it's almost certain that we'll get ***OptimisticLockException*** from one of the ***update*** methods - the one that finishes second. And that's a better situation compared to not using optimistic locking at all.
- Without versioning, all updates will finish without any exceptions, but after loading updated entities from database, we'll discover only one change. Why does this happen? Both methods were updating different fields! Why has the second transaction has overwritten other update? Because of the leaky abstraction :)

We know that Hibernate is tracking changes (it's called dirty checking) made on our entities. But to reduce the time needed to compile the query, by default it's including all the fields in the update query instead of only those changed. Does it look strange? Fortunately we can configure Hibernate to work in a different way and generate update queries based on truly changed values.

It can be enabled with***@DynamicUpdate*** annotation. This can be considered a workaround for the partial-updates problem, but you have to remember it's a trade-off. Now every update of this entity is more time-consuming than it was before.

Now let's get back to the situation with optimistic locking. To be honest, what we want to do is generally the opposite of this kind of locking, which assumes that there probably won't be any concurrent modification of the entity, and when such a situation occurs, it raises an exception. Now we definitely want concurrent modification! As an express workaround we can exclude those two fields (***firstName*** and ***lastName***) from locking mechanism. It can be achieved with ***@OptimisticLock(excluded = true)*** added on to each field. Now updating names won't trigger version increments - it'll stay unmodified, which of course can be a source of many nasty and hard to find consistency issues.

Another solutuion is to spin change. To use it we have to wrap update logic with a loop, which renews while transactions with OptimisticLock occur. That works better the less threads

Another solution is to spin change. To use it we have to wrap update logic with a loop, which retries while transactions with OptimisticLock occur. That works better the less threads that are involved in the process. Source code with all those solutions can be found on my GitHub in jpa-async-examples repository. Just explore the commits.

Wait - still no proper solution? In fact no. By using JPA, we've excluded any easy solutions for the concurrent modification problem. Of course we can remodel our application to introduce some event based approaches, but we still have JPA above. If we use Domain Driven Design, we try to close the whole aggregate by using ***OPTIMISTIC_FORCE_INCREMENT*** locking, just to be sure that changing composite entities, or adding elements to a collection will update the whole aggregate, as it should protect invariants. So why not just use any direct access tool? JOOQ or ***JdbcTemplate***, for example? The idea is great, but unfortunately it won't work concurrently with JPA. Any modification done by JOOQ won't propagate to JPA automatically, which means sessions or caches can contain outdated values.

To solve this situation properly, we should extract this context into separate elements - for example a new table, which would be handled directly with JOOQ. As you probably noticed, doing this kind of concurrent update in SQL is extremely easy:

```
update person set first_name = "Jerry" where uuid = ?;
```

With JPA abstraction, it becomes a really complex task, which requires a really deep understanding of Hibernate behavior as well as implementation internals. To sum up, in my opinion JPA is not following the "reactive" approach. It was built to solve some problems, but currently we force it to solve different problems, and in many applications persistence is not one of them.

Topics: JPA , JAVA , JAVAEE