JPA Implementation Patterns: Saving (Detached) Entities

by Vincent Partington MVB · **Jul. 24, 09 · Java Zone · Not set**

We kicked off our hunt for JPA implementation patterns with the Data Access Object pattern and continued with the discussion of how to manage bidirectional associations. This week we touch upon a subject that may seem trivial at first: how to save an entity.

Saving an entity in JPA is simple, right? We just pass the object we want to persist to EntityManager.persist. It all seems to work quite well until we run into the dreaded "detached entity passed to persist" message. Or a similar message when we use a different JPA provider than the Hibernate EntityManager.

A:link {color: blue;} A:visited {color: #A000A0;} html body p { line-height: 120%; } ul li, ul.menu li, .item-list ul li, li.leaf { list-style-type: disc; list-style-position:outside;} ol li, ol.menu li, .item-list ol li, li.leaf { list-style-type: decimal; list-style-position:outside;}

So what is that detached entity the message talks about? A detached entity (a.k.a. a detached object) is an object that has the same ID as an entity in the persistence store but that is no longer part of a persistence context (the scope of an EntityManager session). The two most common causes for this are:

- The EntityManager from which the object was retrieved has been closed.
- The object was received from outside of our application, e.g. as part of a form submission, a remoting protocol such as Hessian, or through a BlazeDS AMF Channel from a Flex client.

The contract for persist (see section 3.2.1 of the JPA 1.0 spec) explicitly states that an EntityExistsException is thrown by the persist method when the object passed in is a detached entity. Or any other PersistenceException when the persistence context is flushed or the transaction is committed. Note that it is *not* a problem to persist the same object twice within one transaction. The second invocation will just be ignored, although the persist operation might be cascaded to any associations of the entity that were added since the first invocation. Apart from that latter consideration there is no need to invoke EntityManager.persist on an already persisted entity because any changes will automatically be saved at flush or commit time.

saveOrUpdate vs. merge

Those of you that have worked with plain Hibernate will probably have grown quite accustomed to using the Session.saveOrUpdate method to save entities. The saveOrUpdate method figures out whether the object is new or has already been saved before. In the first case the entity is saved, in the latter case it is updated.

When switching from Hibernate to JPA a lot of peopleare dismayed to findthat method missing. The closest alternative seems to be the EntityManager.merge method, but there is a big difference that has important implications. The Session.saveOrUpdate method, and its cousin Session.update, attach the passed entity to the persistence context while EntityManager.merge method copies the state of the passed object to the persistent entity with the same identifier and then return a reference to that persistent entity. The object passed is *not* attached to the persistence context.

That means that after invoking EntityManager.merge, we have to use the entity reference returned from that method in place of the original object passed in. This is unlike the the way one can simply invoke EntityManager.persist on an object (even multiple times as mentioned above!) to save it and continue to use the original object. Hibernate's Session.saveOrUpdate *does* share that nice behaviour with EntityManager.persist (or rather Session.save) even when updating, but it has one big drawback; if an entity with the same ID as the one we are trying to update, i.e. reattach, is already part of the persistence context, a NonUniqueObjectException is thrown. And figuring out what piece of code persisted (or merged or retrieved) that other entity is harder than figuring out why we get a "detached entity passed to persist" message.

Putting it all together

So let's examine the three possible cases and what the different methods do:

Scenario	EntityManager.persist	EntityManager.merge	SessionManager.saveOrUpdate
Object passed was never persisted	1. Object added to persistence context as new entity 2. New entity inserted into database at flush/commit	1. State copied to new entity. 2. New entity added to persistence context 3. New entity inserted into database at flush/commit 4. New entity returned	1. Object added to persistence context as new entity 2. New entity inserted into database at flush/commit
Object was previously persisted, but not loaded in this persistence context	1. EntityExistsException thrown (or a PersistenceException at flush/commit)	2. Existing entity loaded. 2. State copied from object to loaded entity 3. Loaded entity updated in database at flush/commit 4. Loaded entity returned	1. Object added to persistence context 2. Loaded entity updated in database at flush/commit
Object was previously persisted and already loaded in this persistence context	1. EntityExistsException thrown (or a PersistenceException at flush or commit time)	1. State from object copied to loaded entity	1. NonUniqueObjectException thrown

this persistence context	PersistenceException at flush or commit time)	entity 2. Loaded entity updated in database at flush/commit 3. Loaded entity returned	thrown

Looking at that table one may begin to understand why the saveOrUpdate method never became a part of the JPA specification and why the JSR members instead choose to go with the merge method. BTW, you can find a different angle on the saveOrUpdate vs. merge problem in Stevi Deter's blog about the subject.

The problem with merge

Before we continue, we need to discuss one disadvantage of the way EntityManager.merge works; it can easily break bidirectional associations. Consider the example with the Order and OrderLine classes from the previous blog in this series. If an updated OrderLine object is received from a web front end (or from a Hessian client, or a Flex application, etc.) the order field might be set to null. If that object is then merged with an already loaded entity, the order field of that entity is set to null. But it won't be removed from the orderLines set of the Order it used to refer to, thereby breaking the invariant that every element in an Order's orderLines set has its order field set to point back at that Order.

In this case, or other cases where the simplistic way EntityManager.merge copies the object state into the loaded entity causes problems, we can fall back to the *DIY merge pattern*. Instead of invoking EntityManager.merge we invoke EntityManager.find to find the existing entity and copy over the state ourselves. If EntityManager.find returns null we can decide whether to persist the received object or throw an exception. Applied to the Order class this pattern could be implemented like this:

```
Order existingOrder = dao.findById(receivedOrder.getId());if(existingOrder == null) {dao.persist(receivedOrder);} else {existingOrder.setCustomerName(receivedOrder
```

The pattern

So where does all this leave us? The rule of thumb I stick to is this:

- When and only when (and preferably where) we create a new entity, invoke EntityManager.persist to save it. This makes perfect sense when we view our domain access objects as collections. I call this the *persist-on-new pattern*.
- When updating an existing entity, we do not invoke any EntityManager method; the JPA provider will automatically update the database at flush or commit time.
- When we receive an updated version of an existing simple entity (an entity with no references to other entities) from outside of our application and want to save the new state, we invoke EntityManager.merge to copy that state into the persistence context. Because of the way merging works, we can also do this if we are unsure whether the object has been already persisted.
- When we need more control over the merging process, we use the *DIY merge pattern*.

I hope this blog gives you some pointers on how to save entities and how to work with detached entities. We'll get back to detached entities when we discuss Data Transfer Objects

in a later blog. But next week we'll handle a number of common entity retrieval pattern first. In the meantime your feedback is welcome. What are your JPA patterns?

From http://blog.xebia.com

Like This Article? Read More From DZone

JPA, Asynchronous Processing, and "Leaky Abstractions"

Domain Object Persistence

Spring Data JPA With an Embedded Database and Spring Boot

Free DZone Refcard
Getting Started With Vaadin 10

Topics: JAVA , JPA