Avoid Lazy JPA Collections

by Bozhidar Bozhanov MVB · Oct. 28, 11 · Java Zone · Not set

Hibernate (and actually JPA) has collection mappings: @OneToMany, @ManyToMany, @ElementCollection. All of these are by default lazy. This means the collections are specific implementations of the List or Set interface that hold a reference to the persistent session and the values are loaded from the database only if the collection is accessed. That saves unnecessary database queries if you only occasionally use the collection.

However, there's a problem with that. The problem that manifests itself through the exception that in my observations is 2nd most commonly asked exception (after NullPointerException) – the LazyInitializationException. The problem is that the session is usually open for your service layer and is closed as soon as you return the entity to the view layer. And when you try to iterate the uninitialized collection in your view (jsp for example), the collection throws LazyInitializationException, because the session that they hold a reference to is already closed and they can't fetch the items.

How is this solved? The so called OpenSessionInView / OpenEntityManagerInView "patterns". In short: you make a filter that opens the session when the request starts and closes it after the view has been rendered (and not after the service layer finishes). Some people call that an anti-pattern, because it leaks persistence handling into the view layer, and complicates the setup. I wouldn't say it's that bad: generally it solves the problem without introducing other problems. But in all recent project I've been involved, we aren't using OpenSessionInView, and it works fine.

It works fine because we aren't using lazy collections. But then, you'll rightly point, you will be fetching "the whole world" when you load a single entity. Well, no. There are two types of *ToMany mappings:

- value-type mappings where the collection logically does not hold more than a dozen elements. This is in most cases @ElementCollection, and also @*ToMany with items like "Category" or "Price" that are just more complex value objects, but that do not hold any other mappings themselves. Another common feature of these types of collections is that they are usually displayed in the UI together with their owning entity. It is most likely that you want to display the categories of an article, for example. For this type of collections EAGER is the better option. You'll have to fetch them anyway, why not let hibernate (or any jpa implementation) think of some clever join? As I said – the collections are logically not bigger than a dozen or two, so fetching them won't be a performance hit. And, logically, they won't fetch a big object graph with them.

- mappings across the big, core entities. This can be "all orders made by the user" or "all users for the organization", "all items of the supplier", etc. You certainly don't want to fetch them eagerly. Because if you fetch 2000 users for an organization, which in turn have 1000 orders each, and an order has 3 items on average which in turn have a collection of all people who have purchased it.. you'll end up with your entire database in memory. Obviously you need lazy collections, right? Well, no. In that case you should not be using collection mappings at all. These types of relations are, in 99% of the cases, displayed in paged lists in the UI. Or in search results. They are never (and should never) be displayed all on one screen (or should rarely be returned in one API call, if your application provides something like a REST API). You have to make queries for them, and use query.setMaxResults and query.setFirstResult() (or limit them with some restrictive criteria). Furthermore having the collections mapped means someone will try to use them at some point, which may fail. And if the object is serialized (xml, json, etc.) the collection contents will be fetched. Something you almost certainly don't want to happen. (A draft idea here: JPA could have a PagedList collection that would allow paged lazy fetching, thus eliminating the need for a query)

So what did I just say – that you should never use lazy collections. Use eager collections for very simple, shallow mappings, and use paged queries for the bigger ones.

Well, not exactly. Lazy collections are there and they have application, though it is rather limited. Or at least they are way less applicable than they are used. Here's an example scenario where I found it applicable. In my side-project I have a Message entity, and it holds a collection of Picture entities. When a user uploads a picture, it is stored in that collection. A message can have no more than 10 pictures, so the collection could very well be eager. But then, Message is the most commonly used entity – it's fetched virtually on every request. But only some messages have pictures (how many of the tweets on your stream have a a picture upload?). So I don't want hibernate to make queries just to find out there are no pictures for a given message. Hence I store the number of pictures in a separate field, make the pictures collection lazy, and Hibernate.initialize(..) it manually only if the number of pictures is > 0.

So there are scenarios, when the entity has *optional* collections that fall into the first category above ("small, shallow collections"). So if it is small, shallow and optional (say, used in less than 20% of the cases), then you should go with Lazy to save unnecessary queries.

For everything else – having lazy collections will make your life harder.

From http://techblog.bozho.net/?p=645
