JPA Implementation Patterns: Lazy Loading

Model your complete database with all its relations with this JPA pattern for lazy loading.

by Vincent Partington MVB · Aug. 19, 09 · Database Zone · Tutorial

Anybody that has been working with Hibernate for a while has probably seen a LazyInitializationException or two, usually followed by a message such as *"failed to lazily initialize a collection of role: com.xebia.jpaip.order.Order.orderLines, no session or session was closed"* or *"could not initialize proxy - no Session"*. Even though these message may baffle new users of Hibernate, they are a lot better than the NullPointerExceptions OpenJPA gives you in these cases (at least when using runtime bytecode enhancement).

To use JPA to its full potential it is imperative to understand how lazy loading works, as it allows you to model your complete database with all its relations *without* loading that whole database as soon as you access just one entity.

Because of this it is unfortunate that the JPA 1.0 specification doesn't cover this subject in more depth than a few sentences along the lines of:

> **The EAGER strategy is a requirement on the persistence provider runtime that data must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed. The implementation is permitted to eagerly fetch data for which the LAZY strategy hint has been specified.**

As of this writing, the proposed final draft of the JPA 2.0 specification does not add anything in this department. The best we can do for now is read the documentationof ourJPA provider and do some experiments.

When Lazy Loading Can Occur

All @Basic, @OneToMany, @ManyToOne, @OneToOne, and @ManyToMany annotations have an optional parameter called fetch. If this parameter is set to FetchType.LAZY it is interpreted as a hint to the JPA provider that the loading of that field *may* be delayed until it is accessed for the first time:

- the property value in case of a @Basic annotation,
- the reference in case of a @ManyToOne or a @OneToOne annotation, or
- the collection in case of a OneToMany or a @ManyToMany annotation.

The default is to load property values eagerly and to load collections lazily. Contrary to what you might expect if you have used plain Hibernate before, references are loaded eagerly by default.

Before discussing how to use lazy loading, let's have a look into how lazy loading may be implemented by your JPA provider.

Build-Time bytecode Instrumentation, Run-Time bytecode Instrumentation and Run-Time Proxies

To make lazy loading work, the JPA provider has to do some magic to make the objects that are not there yet appear as if they are there. There are a number of different ways JPA providers can accomplish this. The most popular methods are:

- **Build-time bytecode instrumentation** - The entity classes are instrumented just after they have been compiled and before they are packaged to be run. The advantage of this approach is that bytecode instrumentation can provide the best performance and the least leaky abstraction. A disadvantage is that it requires you to change your build procedure and is (therefore) not always compatible with IDE's. the instrumented classes may be binary incompatible with their uninstrumented versions which could cause Java serialization problems and the like, but this is not something I have heard anybody mention as a problem yet.
- **Run-time bytecode instrumentation** - Instead of instrumenting the entity classes at build-time, they can also be instrumented at run-time. This requires installing a Java agent using the -javaagent option from JDK 1.5 and upwards, using class retransformation when running under JDK 1.6 or a later version, or some proprietary method if you are using an older JDK. So while this method does not require you to modify your build procedure, it is very specific to the JDK you are using.
- **Run-time proxies** - In this case the classes are not instrumented but the objects returned by the JPA provider are proxies to the actual entities. These proxies can be dynamic proxy classes, proxies that have been created by CGLIB, or proxy collections classes. While requiring the least setup, this method is the least transparent of the ones available to JPA implementors and therefore requires you to know most about them.

Run-Time Proxy Based Lazy Loading with Hibernate

While Hibernate supports build-time bytecode instrumentation to enable lazy loading of individual properties, most users of Hibernate will be using run-time proxies; it is the default and works well for most cases. So let's explore Hibernate's run-time proxies.

Two kind of proxies are created by Hibernate:

1. When lazily loading an entity through a lazy many-to-one or one-to-one association or by invoking EntityManager.getReference, Hibernate uses CGLIB to create a subclass of the entity class that acts a proxy to the real entity. The first time any method on that proxy is invoked, the entity is loaded from the database and the method call is passed on to the loaded entity. My colleague Maarten Winkels has blogged about the pitfalls of these Hibernate proxies last year.
2. When lazily loading a collection of entities though a one-to-many or a many-to-many association, Hibernate returns an instance of a class that implements the PersistentCollection interface such as PersistentSet or PersistentMap. The first time that collection is accessed, its members are loaded. The member entities are loaded as regular classes, so the Hibernate proxy pitfalls mentioned above don't apply here.

To get a feel for what happens here, you might want to step through some simple JPA code in the debugger and see the objects that Hibernate creates. It will increase your 🙂 understanding if the mechanism a lot.

Run-Time bytecode Instrumentation with OpenJPA

OpenJPA offers a number of *enhancement methods*, as the documentation calls it, of which I found *run-time bytecode instrumentation* the easiest to set up.

Stepping through the debugger you can see that OpenJPA does not create proxies. Instead a few extra fields have appeared in each entity class, with names like pcStateManager or pcDetachedState. More importantly you can see that a lazily loaded entity has all its fields set to 0 or null and that its state is only loaded when a method is invoked on it. More precisely, a property that is configured to be loaded lazily is only loaded when *its* getter is invoked.

It is very important to know that direct access to the fields of a lazily loaded entity (or the field behind a lazily loaded property) does not trigger loading of that entity (or field). Also, when the session is no longer available OpenJPA does not throw an exception as Hibernate does but just leaves the values in their uninitialized state, later causing the NullPointerExceptions I mentioned above.

OpenJPA vs. Hibernate

The first difference between these two approach you might notice is the objects that get proxied/instrumented:

- OpenJPA instruments *all entities* which means it can detect when you access a lazy reference or collection *from the referring entity* and it will then return an actual entity or a collection of actual entities. Only when you lazily load an entity using EntityManager.getReference or when you have configured a property to be lazily loaded, will you get a (partially) empty entity.
- In the case of a lazy reference (or an entity that has been lazily loaded with EntityManager.getReference) Hibernate proxies *the lazy object itself* using CGLIB, which causes the proxy pitfalls mentioned before. When using a lazy collection Hibernate is just as transparent as OpenJPA. Finally, Hibernate does not support lazily loaded properties using proxies.

If you compare OpenJPA's instrumentation to the run-time proxies created by Hibernate you can see that the approach taken by OpenJPA is more transparent. Unfortunately it is let down somewhat by OpenJPA's less than robust error handling.

The Pattern

So now that we know how to configure lazy loading and how it works, how can we use it properly?

Step 1 would be to examine all your associations and see which should be lazily loaded and which should be eagerly loaded. As a rule of thumb I start out leaving all *-to-one associations eager (the default). They usually don't add up to a large number of queries anyway and if they do, I can change them. Then I examine all *-to-many associations. If any of them are to entities that are always accessed and therefore always loaded, I configure them to be loaded eagerly. And sometimes I use the Hibernate specific @CollectionOfElements annotation to map such "value type" entities.

Step 2 is the most important. To prevent any LazyInitializationExceptions or NullPointerExceptions you need to make sure that all access to your domain objects occurs *within one transaction*. When domain objects are accessed *after* a transaction has finished, the persistence context can no longer be accessed to load the lazy objects, and that causes these problems. There are two ways to solve this:

1. The most pure way is to place a Service Facade (or Remote Facade if you will) in front of your services and only communicate with clients of your service facade through Transfer Objects (a.k.a. Data Transfer Objects a.k.a. DTO's). The facade's responsibility is to copy all appropriate values from your domain objects to the data transfer objects, including making deep copies of references and collections. The transaction scope of your application should include the service facade for this pattern to work, i.e. set your facade to be @Transactional or give it a proper @TransactionAttribute.
2. If you are writing a *Model 2* web application with an MVC framework, a widely used alternative is to use the *open EntityManager in view* pattern. In Spring you can configure a Servlet filter or a Web MVC interceptor that will open the entity manager when a request comes in and will keep it open until the request has been handled. The means the same transaction is active in your controller *and* in your view (JSP or otherwise). While purists may argue that this makes your presentation layer depends on your domain objects, it is a compelling approach for simple web applications.

Step 3 is to enable SQL logging of your JPA provider and exercise some of the use cases of your applications. It is enlightening to see what queries are performed when entities are accessed. The SQL log can also provide you with input for performance optimizations so you can revisit the decisions you made in step 1 and tune your database. In the end lazy loading is all about performance, so don't forget this step!

I hope this blog has given you some insight into how lazy loading works and how you can use it in your application. In the next blog I will delve deeper into the topic of the DTO and Service Facade patterns. But before I leave you I would like to thank everybody that came to my J-Spring 2009 talk on this subject. I had a lot of fun! It really seems a lot of people are wondering how to effectively use JPA because I got a lot of questions. Unfortunately the questions made me run out of time. Next time I will pay more attention to the 😉 *girl with the time card. And bring my own water. Thanks again for being there!*

From http://blog.xebia.com

From http://blog.xebia.com

Running out of memory? Never run out of memory with Redis Enterprise database. Start your free trial today.

Like This Article? Read More From DZone

Optimizing JPA Performance: An EclipseLink, Hibernate, and OpenJPA Comparison

Is Hibernate the best choice?

Persisting Entity Classes using XML in JPA

Free DZone Refcard

Graph-Powered Search: Neo4j & Elasticsearch

Topics: JAVA , JPA , OPENJPA , HIBERNATE

Opinions expressed by DZone contributors are their own.

Database Partner Resources

SQL support, smart cache, and amp; speed of 1 million ACID transactions on a single CPU core with Tarantool
Tarantool

Why a NoSQL Database is the Best Solution for a Startup
Hibernating Rhinos

New whitepaper: 6 tips for continuous delivery & Database DevOps. Read now.
Redgate

Example schemas and queries for hybrid data models based on relational + JSON
MariaDB