JPA Implementation Patterns: Field Access vs. Property Access

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

I will continue the JPA implementation patterns series by discussing the relative merits of field access vs. property access.

The JPA specification allows two ways for the persistence provider to access the persistent state of an entity. The persistence provider can either invoke JavaBeans style property accessors (getters and setters) or access the instance fields of the entity directly. Which method is used depends on whether you have annotated the properties or the fields of an entity.

The JPA 1.0 specification does not allow you to mix access types within an entity or even an entity hierarchy. If you have annotated both fields and properties, the behaviour is undefined. The JPA 2.0 specification has the @Access annotation that makes it possible mix access types within an entity or entity hierarchy.

But the interesting question remains; which access type to use? A question that has been discussed before, but one I couldn't resist commenting on too. 😉

- **Encapsulation** - Property access is said to provide better encapsulation, because directly accessing fields is bad, right? Well actually, using property access obliges you to write getters and setters for all your persistent properties. These methods not only allow the JPA provider to set the fields, they also allow any other user of your entity class to do this! Using field access allows you to write only the getters and setters you want to (they're evil, remember?) and also write them as you want them, for example by doing validation in your setters or some calculation in your getters. In contrast, making these methods smarter when using property access is just asking for trouble.
- **Performance** - Some people prefer field access because it supposedly offers better performance than property access. But that is a very bad reason to choose field access. Modern optimizing JVMs will make property access perform just as fast as field access and in any case database calls are orders of magnitude slower than either field access or method invocations.
- **Lazy loading in Hibernate** - Hibernate's lazy loading implementation always initializes a lazy proxy when any method on that proxy is invoked. The only exception to this is the method annotated with the @Id annotation when you use property access. But when you use field access there is no such method and Hibernate initializes the proxy even

when invoking the method that returns the identity of the entity. While some propose to use property access until this bug is fixed, I am not in favour of basing design decisions on framework bugs. If this bug really hurts your performance you might want to try and get the id of entity with the following code:

```
Serializable id = ((HibernateProxy) entity).getHibernateLazyInitializer().getIdentifier()
```

It's nasty, but at least this code will be localized to where you really need it.

- **Field access in Hibernate** - It is good to know that while field access is OK for Hibernate to populate your entities, your code should still access those values through methods. Otherwise you will fall into the first of the Hibernate proxy pitfalls mentioned by my colleague Maarten Winkels.

To summarize I think field access is the way to go because it offers better encapsulation (without it properly managing bidirectional associations is impossible) and the performance impact is negligible (#1 on the performance problems top 10 is still the interplay between the database and your Java app). The only downside are some snafu's in Hibernate's lazy loading implementation that require you to take extra care when using field access.

What access type do you prefer? Do you see any difference in the way field access and property access are implemented in JPA providers other than Hibernate? Please let me know by leaving a comment below. See you at the next JPA implementation patterns blog in which I will talk about mapping inheritance hierarchies in JPA.

From http://blog.xebia.com

Like This Article? Read More From DZone

Proposed Jakarta EE Design Principles

Advanced Research Initiatives: Focus on the Future

How to Configure NGINX High Availability Cluster Using Pacemaker on Ubuntu 16.04

Free DZone Refcard

Getting Started With Kotlin

Topics:

Opinions expressed by DZone contributors are their own.

Java Partner Resources