The Wayback Machine - https://web.archive.org/web/20090317113716/http://java.sys-con.com:80/node/43552

Java Authors: Maureen O'Gara, Tom Deneau, Jeremy Geelan, Bruno Ropu Rovagnati, Alan Williamson
Related Stories: Java

Java: Article

The Delegation-Managed Persistence Entity Bean

A composite entity bean for a new generation

BY **TAL COHEN**

ARTICLE RATING:

FEBRUARY 5, 2004 12:00 AM EST

READS: 17,232

RELATED PRINT EMAIL FEEDBACK ADD THIS BLOG THIS

With the introduction of the EJB 2.0 specification, the classic composite entity bean design pattern became outdated overnight. In this article, I present a new pattern that can serve as a proper replacement. This pattern, called Delegation-Managed Persistence bean (DMP bean), allows developers to represent objects that span multiple database tables. DMP beans provide a better solution than the original Composite EJB pattern without making you roll your own persistence mechanism.

Let's start with the basics. Why are composite entity beans required, anyway? The problem is that in many cases (depending on your application server and database), beans with container-managed persistence (CMP entity beans) can span only a single database table. However, in many enterprise databases, a single conceptual object is stored in numerous independent tables. The standard practice for solving this, in the days of EJB 1.x, was using bean-managed persistence (BMP entity beans). With BMP beans, the developer provides his or her own implementation for storing the bean to, and loading it from, the persistent storage (namely the database). This is a tiresome, repetitive, and error-prone job, and it often sacrifices portability for the sake of performance. Portability, for example, could be lost if nonstandard SQL statements (or even stored procedures) are used. The old composite entity bean pattern was basically just a bean that knew how to load itself from several tables using as many JDBC queries as needed, and likewise knew how to store itself to these tables, again using JDBC. This allowed developers to bypass the CMP limit of one table per bean, while providing a proper object-oriented representation of the notion represented by the bean.

A simple example would be the notion of client in an enterprise application, where information about each client is stored in several distinct tables - one table contains contact information, another contains the client's credit status, and so on. From an object-oriented point of view, we're interested in a single Client class that provides access to all the information stored in all the individual tables.

With the introduction of the EJB 2.0 standard, the common solution to such problems became using container-managed relationships (CMRs). A CMR allows a CMP entity bean to maintain a "relationship" to other CMP beans as long as these relationships are represented in the underlying database as foreign keys. This allows you to define fine-grained entity objects, rather than coarse-grained composite beans. The main problem with fine-grained objects in EJB 1.x was the price of remote method invocations. But now that entity beans are strongly encouraged to use the new local interface feature, this becomes a nonissue. Application clients access session beans, following the Session Façade design pattern, and these session beans access the entity beans using their local home and component interfaces.

This sounds like a good solution, but it suffers from two serious drawbacks. First, as noted earlier, the use of CMRs limits the usability of this solution to those composite objects that, in their database representation, use foreign keys. While this is indeed common, it is not always the case. The second problem is more bothersome: the fine-grained entities provide an accurate depiction of the database tables, rather than a high-level, object-oriented view of the concepts with which we deal.

True, the application clients do not deal with these low-level objects, but rather with high-level services offered by the session beans; but this is simply a shift of focus. The session beans now serve as clients to the entity beans. These session beans often contain key parts of the application logic - and it's expressed using a low-level view of the object model. This is unsatisfactory, and in fact contradictory to the original notion of entity EJBs as a high-level object model for the application data.

The original Composite Entity bean pattern solved this problem by providing a high-level view of the data, but at a great price, namely the tiresome and error-prone work required to create these beans. So allow me to present the new Delegation-Managed Persistence (DMP) bean pattern, which provides the same functionality and high-level view as composite entity beans do, while being easy to create and maintain, and taking full advantage of container-managed persistence.

The new pattern will be described here using the simple case of a composite bean Item, which is stored across two database tables: ITEM_DATA_1 and ITEM_DATA_2. We'll assume that each Item occupies one row in each of these two tables. Naturally, the pattern is applicable to a much wider range of cases.

We begin by defining two CMP entity beans, ItemData1 and ItemData2, over the two database tables. These are the DMP bean's underlying fine-grained entities, and we'll refer to them as Item's component beans.

Next we define the Item class as a BMP entity bean. Don't worry, while defined as a BMP bean, our DMP bean will not really have to manage its own persistence issues.

In the bean class, we define a field for each of the component beans: these fields are the component bean references and Item has two such fields. We also define the component key references as one additional field per component bean; the type of these fields is the type of each component bean's primary key class.

Again for the sake of simplicity, we will assume that both ItemData1 and ItemData2 use java.lang.Integer for their primary key classes. So Item's bean class definition begins like this:

```
public class ItemBean implements EntityBean {
// Component bean references:
private ItemData1Local itemData1;
private ItemData2Local itemData2;
```

Related Stories

- Flashback: The End of Middleware – Exclusive 2004 Perspective by Sun President, Jonathan Schwartz
- "Open Source Is In Our Blood," Says Sun's Jonathan Schwartz
- Sun CEO Jonathan Schwartz Scopes Out Future for Sun's Cloud
- AJAX and RIA Market Is Heating Up: Sun CEO

Comments

Optimize Flex Applications By Breaking Them Into Modules

L B wrote: Hi, very good article... i was interisting in your previous article about a new way to look at portal. can i ask you some question? When you talk about portlet window you mean that portlet window is a mx panel and it contains a module, is it true? how to achieve direct module to module communication? Where download the source? thanks in advance Regards Lord

Mar. 16, 2009 11:29 AM EDT

```
// Component key references:
private Integer itemData1Key;
private Integer itemData2Key;
```

Note that, true to the spirit of EJB 2.0, we access the underlying fine-grained entities via their local component interfaces. As you can probably guess, the component key references are maintained so we can lazily load the beans on a per-need basis. Two private methods, getItemData1() and getItemData2(), will be used internally to access the component bean references. The pseudo-code for the first of these methods would be:

```
private ItemData1Local getItemData1() {
if (itemData1 == null) {
// find local home, probably using a home factory
ItemData1LocalHome home = ...;

itemData1 = home.findByPrimaryKey(itemData1Key);
}

return itemData1;
}
```

In itself, Item's bean class does not contain any fields for representing bean attributes. Any getter or setter method for loading or changing attribute values is delegated to the underlying component beans via the component references, like this:

```
public getSomeAttribute() {
return getItemData1().getSomeAttribute();
}
```

Like attributes, any business operations defined in the component beans can also be made available in the higher-level DMP bean using delegation. Yet we can also provide new, more complex features that involve accessing several attributes or several business methods from one or more of the component beans. We are, in effect, providing a high-level view of the single business notion stored across multiple database tables.

The Primary Key Class

The primary key for a DMP should be a composite key: a simple Java class that includes (as fields) the primary keys for every component bean class. In our example, the ItemKey class would have two fields of type Integer, one for ItemData1's key and the other for ItemData2's. All normal primary key rules apply here - make sure the class is serializable, has proper equals() and hashCode() methods, and so forth. Getters and setters for the internal keys are also in order.

Loading and Storing DMPs

In most BMPs, the key life-cycle methods - ejbLoad() and ejbStore() - are complex beasts, accessing one or more tables in the database directly by means of JDBC. Surprisingly, in DMP beans, ejbStore() is completely empty, while ejbLoad() is extremely simple and involves no manual database access.

No implementation is required for ejbStore() since any update is delegated to the component CMPs, which by their very nature allow the container to manage their persistence needs.

As for ejbLoad(), in this method the bean obtains its composite primary key from its entity context object and checks if any of the internal keys is different from the privately maintained component key references. If a component's key was changed, we must store the new value, and we can no longer assume that the local reference to that object is valid. To invalidate the maintained reference, we simply nullify it, and it will be loaded again when needed due to the lazy evaluation mechanism detailed earlier. So in our example, ejbLoad() would look like Listing 1.

While technically a BMP, the composite bean does not manage its own persistence: it indirectly delegates it to the container. This is why it was named "Delegation-Managed Persistence bean" or "DMP bean" in the first place.

Passivation and Activation of DMP Beans

No special actions are required when DMP beans are passivated or activated. Still, it could help the container better manage its resources if the component references and component key references are all nullified in ejbPassivate().

Creating, Finding, and Removing DMP Beans

Perhaps the most sensitive part of this pattern is the implementation of the ejbCreate...() and ejbFind...() life-cycle methods. The last remaining life-cycle method, ejbRemove(), is rather simple to implement: just remove each of the component CMPs in turn. Make sure ejbRemove() has the REQUIRES transaction attribute, so if the removal of any of the component beans fails, no removal will take place.

Each ejbFind...() method should return the composite primary key type. This is basically done by finding each of the relevant component CMPs (via whatever finder methods they provide in their own local home interfaces, and possibly using CMRs between these CMPs), and then composing the required primary key from its components (the primary keys of the component CMPs).

Slightly more complex is the case of finders that return collections. While it is possible to retrieve the relevant collections of composing keys, and then iterate over them in order to create a new collection of composite keys, this could be highly ineffective. One solution is to create a lazy collection mechanism that keeps the collections of composing keys, and delves into them only when an iterator is used (a lazy collection). Bear in mind, however, that (depending on your application server software) collections returned by local home interfaces of CMP beans are often lazy collections themselves, and are invalidated as soon as the transaction that created them is over. In such cases, there is no option other than to create the whole collection immediately inside the DMP bean's finder method.

But DMP beans can do more than rely on the finder methods provided by their composing beans. The finder methods are one place where it does make sense to use JDBC directly in DMP beans. Using raw SQL, you can create finders that are too complex, ineffective, or downright impossible to implement using EJB QL. Thus, if your high-level business notion, which spans multiple database tables, suggests high-level search criteria that cannot be expressed by simple searches on individual tables, you can make these searches available without manually iterating over collections of fine-grained objects.

Finally, as for ejbCreate...() methods, these should create the relevant component CMPs (using their own create...() methods from the local home interface), and maintain the resulting local references in the DMP bean's fields. Again, as with ejbRemove(), the entire creation process should normally be enclosed within a single database transaction.

Conclusion

The pattern presented here allows developers to easily create a high-level object-oriented view of complex business objects, which cannot be represented using CMP entity beans due to mapping limitations. These high-level objects can then be used by their clients (normally session beans that would access them via a local interface), simplifying the client code since it no longer has to be aware of the internal structure of these potentially complex objects. While overcoming the limitations of CMP entities, Delegation-Managed Persistence Beans do not necessitate the creation of complex persistence code, since they take full (if indirect) advantage of the automated persistence services offered by the container.

Published February 5, 2004 – Reads 17,232
Copyright © 2004 SYS-CON Media, Inc. All Rights Reserved.

RELATED PRINT EMAIL FEEDBACK ADD THIS BLOG THIS

Related Links

- Source Code

About Tal Cohen

Tal Cohen is a consultant specializing in J2EE and related technologies. Until recently, he had worked as a researcher in IBM's Haifa Research Labs. He can be contacted at tal@forum2.org.