The cascading attribute

The cascading attribute

Til now users of castor where able to automaticalli store/edit or delete objects which are related by a mapping file by using the autostore(boolean) method of a Database implementation. This was quite useful but on a second look you'll see, that it's not exactly what you want to have espacially if you have a big project. You might want to have autostore activated for one object, but not for another. Or even more tricky, you might like to automatically track changes in one relation of an object but not in the other?

Or what about one relation with autostore and one without?

The solution to that is the cascading-attribute in the sql tag of the mapping file!

Options which are going to be implemented so far:

- create: when you are persisting an object, all related objects will be persisted too (this only works if the specified object is not already persisted)

- update: ths is the attribute type if you want to support cascading combined with long transactions. After a long transaction, all related objects are merged with the database like the basic one

- delete: deleting an object from the database will delete all related objects for which this attribute value is set

- commit: load an object, change it and store it back. If xou want to store related objects too, use this attribute

- all: combines all possible cascading attributes

But it's not only possible to use one or all. You can define the attributes for every relation seperately and you can specify one to all attributes as you wish.

Relations

This section will show some examples of mapping files describing different types of relations. The Mapping files are taken from the official castor sitehere.

1:1 relation

The following code fragment shows the mapping file for the Product class. Apart from the simple field declarations, this includes a simple 1:1 relation between `Product` and `ProductGroup`, where every product instance is associated with a `ProductGroup`:

<class name="myapp.Product" identity="id">

<map-to table="prod" />

<field name="id" type="integer"><sql name="id" type="integer" /></field>

<field name="name" type="string"><sql name="name" type="char" /></field>

<field name="price" type="float"><sql name="price" type="numeric" /></field>

<field name="group" type="myapp.ProductGroup" ><sql name="group_id" cascading="create"/></field>

<field name="details" type="myapp.ProductDetail"collection="vector"><sql many-key="prod_id"/></field>

<field name="categories" type="myapp.Category" collection="vector"><sql name="category_id"many-table="category_prod" many-key="prod_id" /></field>

</class>

1:M relation

As you can see, this is a bidirectional relationship. Therefor we can now define different cascading behaviour depending on the direction of the relation.

The following code fragment shows (again) the mapping file for the `Product` class. The field definition highlighted shows how to declare a 1:M relation between `Product` and `ProductDetail`, where every product instance is made up of many `ProductDetails`:

Cascading with the "product" class as operation class only works if we use long transaction [see: update]

<class name="myapp.Product" identity="id">

<map-to table="prod" />

```
<field name="id" type="integer">
<sql name="id" type="integer" />
</field>

<field name="name" type="string">
<sql name="name" type="char" />
</field>

<field name="price" type="float">
<sql name="price" type="numeric" />
</field>

<field name="group" type="myapp.ProductGroup" >
<sql name="group_id" />
</field>

<field name="details" type="myapp.ProductDetail" collection="vector">
<sql many-key="prod_id" cascading="update"/>
</field>

<field name="categories" type="myapp.Category" collection="vector">
<sql name="category_id"
many-table="category_prod" many-key="prod_id" />
</field>
</class>
```

The following code fragment shows the corresponding mapping entry for the `ProductDetail` class that defines the second leg of the 1:M relation between `Product` and `ProductDetail`.

In the opposite direction all changes have effects on the product objects related!

```
<class name="myapp.ProductDetail" identity="id" depends="myapp.Product" >

<description>Product detail</description>

<map-to table="prod_detail" xml="detail" />

<field name="id" type="integer"><sql name="id" type="integer"/></field>

<field name="name" type="string"><sql name="name" type="char"/></field>

<field name="product" type="myapp.Product"><sql name="prod_id" cascading="all"/></field>

</class>
```

M:N relation **still to implement!**

The following code fragment shows (again) the mapping file for the `Product` class. The field definition highlighted shows how to declare a M:N relation between `Product` and `ProductCategory`, where many products can be mapped to many product categories:

```
<class name="myapp.Product" identity="id">

<map-to table="prod" />

<field name="id" type="integer"><sql name="id" type="integer" /></field>

<field name="name" type="string"><sql name="name" type="char" /></field>

<field name="price" type="float"><sql name="price" type="numeric" /></field>

<field name="group" type="myapp.ProductGroup" ><sql name="group_id" /></field>

<field name="details" type="myapp.ProductDetail" collection="vector"><sql many-key="prod_id"/></field>

<field name="categories" type="myapp.Category" collection="vector"><sql name="category_id"many-table="category_prod" many-key="prod_id"
/></field>

</class>
```

The following code fragment shows the corresponding mapping entry for the `ProductCategory` class that defines the second leg of the M:N relation between `Product` and `Category`.

```
<class name="myapp.Category" identity="id">

<description>A product category, any number of products can belong tothe same category, a product can belong to any number ofcategories.</de
scription>
```

<map-to table="category" xml="category" />

<field name="id" type="integer"><sql name="id" type="integer"/></field>

<field name="name" type="string"><sql name="name" type="char"/></field>

<field name="products" type="myapp.Product" collection="vector"><sql name="prod_id"many-table="category_prod" many-key="category_id" /></field>

</class> |

Technical view

from a technical view we changed the following:

cpa/src/main/java/org/exolab/castor/persist/FieldMolder.java

Here we added a public enum CascadingType and a private attribute _cascading which can be accessed by the public getCascading() Method. The return value is an EnumSet<CascadingType> containing all defined cascading attributes!

cpa/src/main/java/org/castor/persist/resolver/BaseRelationResolver.java

with this class we inserted a new abstraction level between ResolverStrategy and actual implementations of resolvers. It enables us to ensure, that everyone can check whether a specific relation (represented by an implementation of a BaseRelationResolver) is defined to enable cascading.

This can be done by calling isCascadingUpdate() to check whether update is specified, or by calling isCascadingCreate() ... and so on.

This isCascading... Methods now also check whether autoStore is true or not, in order to ensure compatibility with older versions. Later on this only has to be removed in the baseRelationResolver as all other occurences of autoStore should then be replaced by the needed cascading calls!

rest

The rest of the changes will later on be published by a list of all changed occurences of autoStore in the castor code!