Database - Feature #4307

DMO conversion changes

09/09/2019 03:10 AM - Eric Faulhaber

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Ovidiu Maxiniuc	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

History

#1 - 09/09/2019 03:22 AM - Eric Faulhaber

- File Foo.java added

As part of our persistence performance overhaul, we are no longer going to emit source code for DMO implementation classes. Only the interface will be emitted. Please turn off (but don't yet remove) the DMO implementation class conversion in the m0 conversion step by default. This default should be able to be overridden with a flag in the p2j.cfg.xml file, which is not used by default.

We will need the TableMapper to still work as it does today, and therefore we will need all information currently emitted as annotations in the implementation class to be moved to the DMO interface source file. Field-level annotations will move to the corresponding getter methods of the interface. The annotation definitions have changed. Please see the new persist.annotation sub-package in task branch 4011a.

Note that the annotation class names have changed (e.g., LegacyField -> Property), as have some of the annotations' property names (e.g., fieldId -> id). Some new annotation properties have been added. Hopefully, the changes are obvious, but please ask any questions you have.

An example (hand-written) DMO interface file is attached.

#2 - 09/11/2019 02:25 PM - Ovidiu Maxiniuc

- Status changed from New to WIP

At a closer inspection, I see that the extends DataModelObject is not present any more in the new interface. Is this intended? Do we use the Table annotation to identify DMOs? What about the Temporary interface? De we add a temporary field in the Table annotation?

#3 - 09/11/2019 03:08 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> At a closer inspection, I see that the extends DataModelObject is not present any more in the new interface. Is this intended?

It was intended, but I'm not sure it is correct yet. The changes are still pretty fluid. We may find we need some marker interface here as we integrate the new code with the existing framework, but I'm not sure if it will be DataModelObject ultimately. Please leave the conversion code in place, but

comment it out with a "TODO: need to determine if marker interface is needed here".

> Do we use the Table annotation to identify DMOs?

Not necessarily meant as an identifier. The Table annotation was just meant to replace LegacyTable with some changs and some additions.

> What about the Temporary interface? De we add a temporary field in the Table annotation?

Same approach as for DataModelObject: comment out the conversion code, add the TODO.

#4 - 09/11/2019 06:44 PM - Ovidiu Maxiniuc

- File Foo.java added

Now the DMO Impl classes are not generated any more and DMO interfaces are constructed with DMO annotations.
To see the new code in action, add the following line in p2j.cfg.xml:

```
      <!-- rule-sets marked with any of these exclude-conditions will not be executed -->
      <parameter name="exclude-conditions"      value="no-dmo-impl" />
```

We might refine this condition, however.
With this change, the test class is generated as the one attached.

Committed to 4011a as r11332.

#5 - 09/13/2019 11:59 PM - Eric Faulhaber

Well done.

As noted originally, however, the DMO implementation classes should *not* be emitted by default. The default is still to emit them and this has to be overridden with the new flag. The flag needs to be absent by default, otherwise we have to add this to every project configuration going forward, and I just want to be required to override the default behavior for projects which already have dependencies on the DMO implementation classes.

As you note, we may want to refine this further, because really we probably only will want to emit a certain subset of DMO classes, which have dependencies in hand-written code.

For now, let's not worry about this further. We still need to think more about how to handle this "backward compatible" dependency.

#6 - 09/18/2019 03:31 PM - Eric Faulhaber

Ovidiu, I am trying to rationalize the DMO class/interface hierarchy to eliminate casting and differential logic to handle temp-table DMOs and permanent table DMOs. The existing hierarchy has grown quite confusing over the years. However, I need some help understanding how TempTableRecord fits into the current structure.

ATM, TempTableRecord extends Persistable, apparently to deal with some new fields needed for temp-tables, which you discovered while implementing data sets. Do these fields need to be exposed for business logic to access them directly? Are they accessed only indirectly via 4GL attributes and/or methods?

I have moved the primary key id into persist.orm.BaseRecord, which is the ultimate ancestor of all DMO implementation classes. This class is used directly by the new ORM layer to manipulate the data and state of individual DMO instances. persist.Record extends BaseRecord. It is used by classes in the persist package, and serves as an abstraction layer between business logic DMOs and the ORM layer. DMO implementation classes will extend Record directly, for permanent table DMOs.

I would like to define the existing TempTableRecord API within Record, but have these methods be no-ops at this level. Then, I would like to define a thin, abstract class which extends Record (I guess we could name this TempTableRecord or just TempRecord). This class would implement these temp-table specific methods, for temp-table DMO use. I would also define _multiplex at this level, instead of having this be a field of every ancestor temp-table DMO implementation class. All temp-table DMO implementation classes would extend this new class, instead of Record.

This means we can define all persist package APIs which currently accept DataModelObject or Persistable to instead accept Record, and we will not have to do any special casting or type checks to use such objects in TemporaryBuffer.

Does this make sense?

#7 - 09/18/2019 04:15 PM - Ovidiu Maxiniuc

Yes, you deduce correctly. TempTableRecord adds 5 new fields to standard record. You need to look at 4124a to see all of them generated. These new fields are specific to before-tables. There are functions and attributes that access them and also, they can be accessed in 4GL, directly, see #3809#note-220.
As I said, they seem to be present only in before-tables, not in after-tables. However, I've chosen to unify TempTableRecord for before tables, too, for multiple reasons, the two most important being:

- the special functions and attributes return the same values for both before/after image. This was done with performance in mind. Once a record is accessed the new attributes are available. And they can be used in queries, directly. Otherwise, when accessing an attribute of an after-record, the pair before-table needs to be loaded to get the attribute value.
- to add the 'reverse' link from after-to-before. Again, performance-wise. I could not find the linkage to before-image in P4GL. Most likely, to obtain the BEFORE_ROWID(after-buffer) they execute something like:
  FIND before-buffer WHERE before-buffer.__after-rowid__ EQ ROWID(after-buffer).

So, for real before-tables, the fields are also real, generated as normal fields, but with hidden attribute which will exclude them from expanding/copy/compare operations.
For the after-tables, the fields are surrogate, in the same meaning like _multiplex is for all temp-tables, but writeable.
The simple temp-tables (without before/after attribute) ignore those fields and the dedicated buffer functions/attribute return default values (like 0).
Of course, the permanent tables do not extend the TempTableRecord and the dedicated buffer functions/attribute return unknown values.

Note that in the pending 3809e (rebased from 4124a) the methods of TempTableRecord do not have default implementations, the generated DMOs are mandatory to implement them.

#8 - 09/18/2019 04:33 PM - Eric Faulhaber

Let me ask what I'm really trying to understand... Do the temp-table DMO proxies used by business logic ever need to access the TempTableRecord methods, or are these purely for FWD-internal implementation purposes? AFAICT, the TempTableRecord API is not exposed for business logic use, as this interface is not extended by any generated DMO interfaces, unless I am missing something newer from another branch (4124a or 3809e).

If this understanding is correct, it is my intention to roll this API into the Record abstract class (w/ no-op default implementations) and provide real implementations of these methods in a subclass, which actual temp-table DMO implementation classes will extend.

The DMO implementation classes will be generated at runtime only by ASM, so I want them to be as simple and non-duplicative as possible (a good design principle anyway). Also, moving all the common implementation into the superclass will reduce memory requirements slightly.

#9 - 09/19/2019 10:44 AM - Ovidiu Maxiniuc

The TempTableRecord methods are not directly accessed by the business logic. Instead they are accessed only internally, by FWD, through the implementation of the functions and attributes (like ROW-STATE, ERROR-STRING, AFTER-ROWID, etc).
The TempTableRecord is not *extended* by any DMO interface but it is *implemented* by all temporary DMOs. Note the difference in the implementation:

- the TempTableRecord methods from BEFORE-TABLES DMO implementations call the actual field setter/getters;
- the TempTableRecord methods from AFTER-TABLES DMO implementations change the surrogate column.

Hope this helps.

#10 - 02/01/2021 09:22 AM - Eric Faulhaber

- % Done changed from 0 to 100

- Status changed from WIP to Closed

Branch 4011e was merged to trunk rev 11348.

Files

Foo.java	4.8 KB	09/09/2019	Eric Faulhaber
Foo.java	7.72 KB	09/11/2019	Ovidiu Maxiniuc