Development-Audit Trails

Pierce T. Wetter III

This gets discussed a lot, so there's plenty of examples around. For instance, here's an implementation by someone else I found via Google: http://homepage.mac.com/i_love_my/code.html

However, unless you're insane enough to need to log every change to every object, the concept of an "audit trail" ends up being application specific in many cases. If you can simplify the requirements at all, do so. For instance, at www.marketocracy.com, we never "delete" a fund, we just mark it deleted. Similarly in forums, we don't "delete" forum posts we don't want to show, we "hide" them.

Funds we can't delete because they are cross linked to too many objects, but forum posts I purposely setup to not be deleted because I knew the day would come when someone went "oops" (and they did). So those tables have a flag, and the relationship is defined such that it ignores the "deleted" objects.

For "forecasting and recall" on www.marketocracy.com we record every trade already, so it was fairly easy in the business logic to add methods so that you could "time travel" to any point in time. That is, a trade adds shares to your account while subtracting from your cash (or the reverse). So shares on any position at any point in time is sum(trade.shares where date < desired time). So we've been able to avoid the need for an audit trail, because we already have an "accounting model" in that every operation is recorded as a transaction. In fact if I had to do Marketocracy over again, I might implement it as a double-entry bookkeeping system instead.

If I were going to log every change to every object, some typical hints are that if you subclass EOEditingContext, then prior to any saveChanges() operation you can get a list of the inserted/deleted/changed objects (you have to call processRecentChanges() first). If I were designing this, I'd make all my EOF objects have a common superclass/interface that implemented something like:

```
auditTrailChanged()
auditTrailInsert()
auditTrailDelete()
```

Then on a class-by-class basis I could decide what to do. That could include inserting additional objects, etc. Like an object could compare itself to its snapshot if it had changed and it could record the old and new values to a generic object that stored AuditTrail(class name, primary key of object, username, timestamp, attribute, old value (string), new value(string)).

In psuedo code:

```
override saveChanges()
  self.processRecentChanges()
  foreach obj (self.changedObjects)
    obj.auditTrailChanged()
  foreach obj (self.insertedObjects)
    obj.auditTrailChanged()
  foreach obj (self.deletedObjects)
    obj.auditTrailChanged()
  super.saveChanges()
```

Houdah Frameworks:

(scooped from the mailinglist)http://www.mail-archive.com/webobjects-dev@lists.apple.com/msg25686.html

Take a look at Houdah Frameworks. I think the Audit Trail WO:1 WO:2 solution provided by this framework could be helpful.

The Wonder Bug Tracker application also includes support for audit trail. Maybe you can learn something from there.

#1 http://code.google.com/p/houdah-webobjects-frameworks/wiki/HoudahAuditTrail
#2 http://code.google.com/p/houdah-webobjects-frameworks/wiki/AuditTrail

Dov Rosenberg

(scooped from the mailinglist)http://www.mail-archive.com/webobjects-dev@lists.apple.com/msg25683.html

Most of the projects I worked on with this requirement used the built in audit tracking functions of the database. Most DBA's didn't leave the requirement to the developers to enforce. That way everything is tracked in a consistent fashion across applications.

Ken Anderson

(scooped from the mailinglist)http://www.mail-archive.com/webobjects-dev@lists.apple.com/msg25691.html

I use a hybrid approach...

I use the database to copy every row modified or deleted to an audit table. Every object has a trans_id field, which is a foreign key relationship to a transaction table. The primary key of that table increases like regular unique integer primary keys.

In EOF, I have sub-entities of all my EOs that have a prefix (like Aud...). These EOs are also subclasses of their main counterparts, then have an imported text file that represents the code I want all audit EOs to share (here's a good case for multiple inheritance!).

The Aud.. EOs have an additional real column called resp_trans_id (the transaction RESPonsible for causing the row to move to the audit table), plus an additional 'fictitious' attribute called 'asof_trans_id'. This is the trans_id that you want the entire object structure to be 'as of'.

The primary key of the audit EOs is the oid AND asof_trans_id so you can have multiple historical audit EOs in the EC.

The Aud EOs then have store procedures for fetching single objects (faulting), that respects the asof_trans_id. The stored procedure finds the right object for that asof_trans_id. For instance, if I have a fault:

AudOrder oid = 72, asof_trans_id = 155

the stored proc first checks to see if the trans_id of the primary Order table is less than 155. If it is, then this object hasn't changed since trans_id 155, and the primary row is returned (but an AudOrder object is still the object created). If not, we find the audit row who's trans_id is less than or equal to 155. If none exists, the fault fails.

In the AudOrder entity, you can decide whether to override existing relationships (like items) to be from audits, or for reference data, you could just keep the original relationship. For to-many audit relationships, you need another stored proc that will build the unique set, and the asof_trans_id value has to travel along (part of the relationship keys). So, the items relationship would be replaced with an items relationship to AudItem. The stored procedure would build a result set that includes all the items ASOF trans_id 155 (a union between the primary table and the audit table).

Primary entities have a method called 'auditObjects' that goes out and gets all the historical versions of an object.

Whew!

OK - NOW, you have the ability to say:

I have this order EO. Give me the top 10 historical versions...

You'll get an array of 10 AudOrder objects, which you can present to the user.

You can display the date/time of the transaction record that the audit is related to, so the user can pick the version of history they want.

Now that the user has selected an AudOrder object, when you fire the items to-many fault, it runs the stored proc that builds the union of unique objects that existed asof trans_id 155.

You can keep going and going, faulting more historical objects over time.

Cool, huh?

Ken

One important bit that I left out....

Depending on the database you're using, it might be difficult for the delete triggers to easily know the current transaction to update the responsible transaction ID (since on deletes, we're obviously not sending data). In Oracle, I use a trigger on the transaction table to update a temporary table that has a lifetime of just the current transaction. The insert trigger on the transaction table inserts a row into the temporary table with the OID of the new transaction, and the delete triggers read from that table to know the responsible trans ID.

Ken