Relation Centered Development in the Experience of LedgerSMB

Chris Travers (chris@metatrontech.com)

July 20, 2007

Contents

1 A Brief History of the Project

LedgerSMB forked from SQL-Ledger in late august of 2006. The issue at stake was the alleged unresponsiveness of the SQL-Ledger developer to a serious security hole which allowed any attacker to manufacture arbitrary authentication credentials.

Several people on the core team had customers on SQL-Ledger and it was felt that the best way forward was to fork from the existing codebase. Had we known the full extent of the problems we might have just decided started from scratch.

The SQL-Ledger codebase is reasonably full-featured, and was at its time one of the better open source accounting applications for Linux. However, the code itself is extremely difficult to maintain, and even harder to fully understand. In short, the code is a mess.

Partially as a result of these problems, both SQL-Ledger 2.8 and LedgerSMB 1.2 have been plagued with sometimes severe bugs in core functionality. We decided to formalize a new architecture in order to avoid these issues and move all code to it.

1.1 Release 1.0

This version was simply SQL-Ledger 2.6.16 with a couple of security fixes applied.

1.2 Release 1.1

The major work in this release was to address a number of known data integrity issues, tighten security, and merge in useful patches that were done for other customers. Development was relatively smooth, and additional security fixes were available in 1.1.5.

1.3 Release 1.2

This release included centralization of sales tax and price matrix logic, solutions to more data integrity issues, removing all possible vectors for SQL injection, and more merging of patches.

Development of 1.2 did not go smoothly. After a bit over a month of development, feature freeze was announced. Six more months of testing were required before release. After release, major bugs forced the removal of releases 1.2.0 and 1.2.1. The worst issues were caused by difficulty of sanitizing the SQL queries (since many of them were generated in pieces and then assembled together) and difficulty understanding the program flow.

2 Innate challenges for this application

LedgerSMB suffers from a couple of issues that are fairly common among line of business tools. In particular, the structure of the information is quite complex. SQL-Ledger maintains all of the constraints in the application (and there only marginally), and much of the code in LedgerSMB still inherits these problems.

Furthermore, SQL-Ledger's solution to the information complexity ("data"-driven programming around a God-object via an accumulate-and-fire approach) was considered by the LedgerSMB to be somewhat limiting.

Additionally, there is no application which has a greater need for good (and transparent) security than accounting software. If someone can break into a web server, deface it, and execute arbitrary code that is bad, but if someone can break into an accounting app and write themselves fraudulant checks against fraudulant invoices, that is an order of magnitude worse. Simply put, accounting systems have some of the strongest security needs of any application for a business.

3 Relation Centered Development Defined

Relation centered development is the practice of putting the algebraic capabilities of a relational database management system at the heart of the application development.

Relation centered programming is nothing new— it has been around as long as relational databases. Very few applications fully use these capabilities for a number of reasons, however, including the desire to remain portable, and the fear that pushing back data validation will further complicate troubleshooting and debugging.

Relation-centered development is similar in many ways to aspect oriented programming, and in our case, is wrapped inside an object oriented layer.

3.1 Advantages of Relation Centered Development

A relational database schema is nothing more nor less than a mathematical representation of the modelled structure of the information that an application expects to handle. In short one may think of database design as a matter of "defining the axioms which will define the information and relational

algebra behind the application." Getting this structure right is not always easy, and it can be time consuming but it can prevent a number of classes of bugs including:

- Information completeness issues where the application produces incorrect output or operations because it has access to insufficient information
- Invalid information issues, where the application is given information which cannot be valid, and hence the application produces incorrect output or performs improper tasks.

Of course, application developers still need to validate input to the extent that it is dangerous in their programming environment (i.e. buffer overruns and the like still need to be prevented), and applications which require special restrictions on data will still need to enforce those. However, the extent of that validation, and problems that stem from missing an application-level validation routine can be minimized if the relevant rules are defined in the database.

Furthermore, the database has internal access to more data than the application, so it is easier and less complex than worrying about these things in the application. In fact, aside from cases where applications need constraints independant of the database, it is easier to tell the database how to spot invalid data than it is to do the validation in application code.

Finally, this separates data validation from program execution flow, ensuring that the appropriate constraints are checked every appropriate time.

4 Our Approach to MVC

One of the key challenges of developing a LedgerSMB as a large-scale application has been the question of how to effectively handle proper database design practices inside an existing MVC framework. The object-¿relational mapping issue is one of the largest hurdles in doing proper relation-centered development and wrapping the relations inside an object-oriented appropach.

In short, LedgerSMB has developed its own light-weight MVC framework which allows a relation-centered design.

4.1 The Model

The model in LedgerSMB's new architecture has the following layers (from the logical bottom):

1. Normalized relational schema and constraints
2. Stored procedures to access this schema.
3. A light-weight object wrapper for Perl which pulls stored procedures as methods dynamically.

The stored procedures basically act as atomic database API's. They are not designed to do any unnecessary processing of data, just things like generate reports, saving sales invoices, and the like.

The stored procedures follow a naming convention of lc(class)_method, so if we wanted to have a method to the Report class named "inventory" the mapping stored procedure would be called report_inventory. Arguments are either prefaced with "in_" in which case they are mapped back to object properties or they are taken from the argument list passed to the function.

The Perl wrapper automatically picks up on the stored procedures and maps them back to methods of their own class.

Note that we also have some Perl utility classes that we use for things that are better not processed in the database. These classes neither affect storage or retrieval, however.

The model also enforces security, since in 1.3, all application accounts will be native database accounts. Thus we can effectively declare a set of rules not only for data integrity and access but also security and have these enforced orthogonally to program flow.

4.2 The View

The legacy code we inherited from SQL-Ledger generates HTML pages by issuing a series of print commands buried in different functions. This makes the user interface difficult for non-programmers to customize and complicates troubleshooting.

LedgerSMB will introduce the use of Template Toolkit for the generation of any portions of the user interface we have been able to redesign. Eventually, XML/XSLT will be an option as well.

The view actually handles more than just the user interface. Views can be generated and sent as PDFs via email just as they can be displayed to the screen. They can even be sent directly to a printer attached to the server. In essence, the view layer is used to prepare any specific information from the application for human consumption, whether in the application or not.

The view is security-aware in that there are mechanisms for altering the user interface based on available permissions, but it does not strictly enforce security. This is left for the model.

4.3 The Controller

The rest of the logic is held in lightweight workflow scripts that allow for automation of the data model and view components.

4.4 Differences from the "Classical MVC" model

Although the above architecture looks very much like a typical MVC design, there are a number of differences including:

- All stored data objects are defined in the RDBMS schema in this model with only light automation above this layer.
- Security is strictly enforced in the model rather than the controller.
- As many valid and necessary constraints as possible are created declaratively in the RDBMS rather than procedurally in the application.

5 Advantages in Our Experience

So far, the LedgerSMB 1.3 codebase has realized a number of benefits from this approach. The most obvious is that code written in this architecture is quite easy to read and maintain (though it does require fluency in Perl, PL/PGSQL, HTML, Javascript, Template Toolkit, and/or LaTeX. The flow of the program is easy to follow, and it is not difficult to debug.

A second advantage we have already seen has been added security from a delarative security structure. Just prior to the release of LedgerSMB 1.2.7, a major security hole was discovered that allowed authentication bypass. However, by that point, our development version was not vulnerable because the logic relating to the handling of the login request was much more streamlined and restricted.

A third advantage we have seen for LedgerSMB 1.3 has been in the ability to enforce many more data constraints and thus avoid many of the sorts of problems that have plagued SQL-Ledger 2.8 when we have implemented similar features.

6 Roadmap for Future Versions

1.3.0 will introduce the new architecture for many parts of the application including login screens, database management, menus, contact management, and reconciliation. Additionally, the contact

management portions of the application have been entirely re-engineered, and a real framework for permissions enforcement is under development.

New features will also include debit/credit notes and invoices, easier workflows for reversing invoices and transactions, and

1.4.0 will provide a number of additional feature enhancements, and all financial documents and objects will be moved to the new architecture.