

Case Study: Rearchitecting a Web and B2B Financial Mortgage Sourcing Application

Background

The project's goal was to develop a Web-based application to allow home buyers to close a home without having to sign physical paper, and to thereby expedite the loan application process and all associated processes and services that result in documents that must be exchanged and signed at closing as well as the preparation of the mortgage contract and property deed as electronic signed documents.

The technologies used included J2EE Web applications, Javascript, J2EE Enterprise JavaBeans, Java, XML messaging and B2B integration technology (WebMethods), PKI, secure Java applets, relational databases, a document repository, XML ("MISMO") documents and XML-based document signatures.

The application was designed to be primarily Web-based, but it soon became clear that the demand for a B2B usage mode was greater and so it evolved a B2B messaging interface.

The expectation was that the processing volume would reach hundreds of database transactions per second, handling sensitive information and legal/financial transactions (mortgages). The application would be extremely high profile and represent a strategic asset. (Market forces eventually changed expectations due to competition and other factors.)

All these factors contributed to a situation of very high risk, if the endeavor was not executed well.

The application was built initially as a prototype. Actual development was then undertaken based on the prototype – a classic huge mistake. In December 2001 the management of the development effort was taken back from the primary vendor and brought in-house, in order to reduce risk. A quality review was then performed to assess the application's technical soundness.

The findings were typical for Web-based systems grown from a prototype: the application was found to be (1) not scalable, (2) not transactionally sound, and (3) not secure. Further, these deficiencies were found to be application-wide and structural and could not be fixed except by re-architecting the application.

It is important to recognize that these characteristics are not unusual for first-release Web-applications. Most organizations that have not built and hosted a high-reliability high-volume application do not know what it takes to do so, and grossly underestimate the task and skills required.

An effort was undertaken to address the issues. It was extremely successful, and the approach used is described here.

A year later, business conditions changed and the company made a

decision to sell the project to a product company. The product company retained an outside security firm to audit the application to assess its quality from a security perspective. The application failed miserably and the sale almost fell through, except that the company was able to show that *each and every issue uncovered by the security firm was being addressed by architectural changes that were already underway*, as a result of the re-architecting work.

Architecture Team

An effort was begun to address the architectural issues. First, an architectural team was established, to plan improvements and guide future development. Next, the architectural team was enlarged and divided into two sub-teams, to address transactional integrity and security. Scalability was addressed by the main development teams and by a performance measurement group.

All design changes were planned while development work continued on the application under the original design. The architecture team had to implement its changes in a prototype mode in a separate branch of the main code base, and refresh its branch regularly in order to stay in sync with the changes being made in the various feature development efforts. When the architectural changes had been implemented, we therefore had a codebase that implemented the new architecture, but also contained all of the bug fixes and feature enhancements of the main codebase.

Approach

Address Transactions First

Early on, a B2B effort was begun. The initial approach by that team was to force B2B interactions onto the Web front end, because that was the only obvious interface to the system. The B2B team planned to build a B2B client that simulated a Web client. This approach was abandoned when the Architecture team promoted the planned transactional "facade" layer as an alternative and preferable interface.

Transactional layer has these benefits:

1. **Reusable business logic.** The business logic within the transaction layer can be re-used for multiple kinds of clients.
2. **Primary security policy enforcement point.** Can be used as a central point for authorization logic, and security rules defined at a transactional level can be reused across multiple clients without redundancy.
3. **Primary testing point.** Can be used as a primary testing point, for verifying the core business logic of the application.

We reviewed all business transactions, and determined which were composed of multiple underlying database or resource transactions. This application made use of several external resource in addition to a database, including an external document repository, and LDAP server (for dynamic updates), a messaging service, and additional non-transactional resources

such as an email server. We wanted to make sure that if any business function did not complete, that there would not be “artifacts” left behind that would confuse the application or indicate the status of transactions incorrectly or incompletely. This required redesigning many business functions to make them transactionally sound or multi-step, and moving most business logic out of the Web application and into the transactional facade layer.

Identify a Clear Set Of Business Authorization Rules

In the application's original design, authorization rules were dispersed haphazardly across the Web and transactional tiers. The ramification of this was that no one could say for sure that all authorization rules were actually implemented, or what they were. In short, the security rules were not auditable.

A new design pattern was devised that treated the transactional façade as a secure resource layer. All authorization rules were placed at the entry point to that layer, and enforced by a security manager component that aggregated all security policy rules. Further, the rules were decomposed into tests for a set of conditions, and these conditions themselves were tabulated, given names, and reused. This resulted in a highly flexible and understandable framework for managing the security policy.

It turned out that once the authorization rules were tabulated and traced to the existing code base, many rules that had been assumed by the business to have been implemented were *not in fact implemented*, or were *implemented incorrectly*. We would not have discovered this if we had not made

the rules more transparent and auditable the way that we did.

Rejection Of Band- Aid Application Security Products

When the security audit was performed as part of the due-diligence review in preparation for the sale of the project, revealing an array of weaknesses in the original design, band-aid products were considered to quickly remedy the weaknesses. This was because the re-architecting of the application was still underway and management was anxious to be able to say that all problems had actually been fixed. In particular, so-called “application protection proxies” (APP) were considered.

In-depth evaluation of these products proved that they added almost no value to securing our application. The reason was that our application had been built in such a way that an APP cannot be effective, due to the heavy reliance on client-side scripting that effectively made it impossible for server-side products such as an APP to know what was actually being passed to clients. So, work continued on the architectural improvements.

Securing the Inputs

Once all authorization rules had been factored out into a security manager, it was a simple matter to tabulate the inputs used in each authorization rule. Having done this, we could trace the source of each such input and verify that it was set securely and managed securely. For example, if an authorization rule used an LDAP database value specifying the roles that a user belonged to, we could trace what part of the application set that role. (Roles were

set dynamically by other transactions when organizations managed their accounts within the application, rather than all being configured by a single administrator.) This allowed us to be sure that rules could not be bypassed through inappropriate modification of the data upon which they rely.

We also made use of single sign-on using infrastructure products, in order to ensure that critical aspects of authorization context such as user identity and role were managed securely as a request propagated from one tier of the application to another.

Rolled Out the Improvements

The new codebase was merged into the main branch over a weekend – and on Monday the development teams had an entirely re-architected codebase. All tests passed and feature-oriented work continued without interruption. Programmers had to learn the new package structure and design patterns, but lost essentially no time as a result of the implementation of the new architecture.

The new architecture had these

characteristics:

1. Its *security rules were transparent and auditable*, and it addressed all of the issues uncovered in the security audit.
2. Its transactions were *re-usable*, across both B2B and Web application clients.
3. It was *transactionally sound and robust*.

Summary

The project made the common mistake of adding features to a prototype in order to attempt to develop a critical application. The resulting system had serious architectural flaws that had to be addressed through re-design of the entire codebase of the application. The redesign work was performed in parallel with mainstream development, by a separate architecture team. The team tested and implemented improved design patterns in parallel with mainstream development, without impacting that development. The changes focused on securing and fortifying the core business functionality as transactions. The changes were incorporated into the mainstream codebase over a weekend, with no disruption in development.