From Reference to Reality: Going Live with uPortal at The University of British Columbia

Paul Zablosky, Dave Frazer

ITServices
The University of British Columbia

Vancouver
British Columbia
Canada

Abstract: Going from a reference implementation to a production service is a revealing exercise. The robustness, reliability, and controlled behaviour required in a production environment do not come for free. The University of British Columbia was the first site to go into production with uPortal, a community-developed, open source, reference implementation of a university portal from the JA-SIG. UBC’s experience illustrates risks and pitfalls, and various measures of success.
In common with other public institutions of higher education, the University of British Columbia is very interested in delivering information access to its students, using the utility of the Web. The popular approach is the institutional portal. UBC’s choice for a portal implementation was uPortal from the JA-SIG, for a variety of reasons:

**Technology:** The technologies employed promise a sound and comprehensive basis for development work. Specifically, the Java component model, XML, and the J2EE service set attracted us. We believe that investing in competence in these areas will return value to our institution beyond the delivered functionality of a single project.

**uPortal is Open Source:** It is vital for a University to control its Information Technology future. With the right set of skilled people, Open Source initiatives are an excellent means to achieving that control. We were confident that we had the right set of people with the right abilities, and we saw the opportunity to develop our technical competence as an institutional resource.

**Size:** In its original release, the uPortal code is small; small enough to be read and understood in an evening. It is built around a clean, flexible, processing model and a clear simple architecture. We were confident that we could understand it and manage it.

**The Portal Promise:** A portal to us was a means of integrating access to information resources for all members of the institution – in a manner that could be highly personalized. We wanted to address the relationship of the individual to the institution. This is what portals promise to do.

**The JA-SIG:** We liked the idea of working in collaboration with the other member universities of the JA-SIG. The University of British Columbia has a long history of participating in this sort of community as a member of the MTS Project with University of Michigan and other sites. We welcomed the opportunity to engage in another such project.

**Challenges**

**New Technologies:** We hadn’t done anything large scale with the base uPortal technologies, so we had a very steep learning curve in becoming familiar with Java, XML, and the Servlet environment. We felt, however, that they would be valuable additions to our repertoire with returns beyond the portal project.

**An Untested Product:** uPortal was a developers’ reference implementation. The code had never been run in production or gone through any formal quality testing. We had no way to predict its performance characteristics, its reliability, or its robustness.

**Scope of Functionality:** With limited development resources available to us, we were forced to choose an application area in which to focus our efforts. It was important to
pick an application that would attract a large number of users to our initial service offering.

**Reference Implementations and Prototypes**

The product we had chosen to work with was a *reference implementation*. While many reference implantations work well in production, they often share some of the characteristics of prototypes. A prototype is designed to demonstrate *correct* behaviour in a limited environment. The environment is usually predefined, arbitrarily excluding problem areas. The environment is expected to be clean and well behaved. Users can be assumed to act reasonably and conform to preset rules. Databases and other external resources are assumed to be available, consistent, and responsive. The code of a prototype can be limited to core functionality, ignoring all abnormal conditions.

In any implementation of a complex service, less than 20% of the code handles normal cases, and the remainder handles situations where things aren’t totally well-behaved.

**Production Services**

A production service must demonstrate reasonable behaviour in an *existing* environment: your site with all its resources, processes, and users. It is exposed to all the diverse behaviour that a user community can contrive. It must behave reasonably *all the time*. Generally we expect Web applications to run 24 hours a day, seven days a week. Databases must be available, consistent, and maintained.

In brief, a production product must include the 80% of the code that deals with abnormal situations, and which may not be found in a reference implementation.

**A Portal in Production:** In production, a portal framework must ensure that users see reasonable behaviour, no matter what happens. This is not to say that the service must always be totally correct or responsive – this cannot be guaranteed in services delivered through the Web. We have to consider the qualitative difference here from what was possible in the past. Since portal channels will often make use of resources outside the control of the portal managers, there is no way to ensure delivery of all services, as might be expected in a traditional heavy-client/central-server application. Instead, we seek to minimize the inconvenience to the user when things aren’t working perfectly, and try to make clear what is going on. Users of the Web are used to some erratic behaviour. If something doesn’t work, they will try again, without experiencing too much frustration.

**Things to Avoid:** If an interactive service is to be accepted by a user population, certain things must be avoided:

- The service must never be totally unresponsive.
- The user must never be misled about the state of the service.
- The actions one user must not affect the general performance of the service.
• The behaviour of one resource must not affect the general performance of the service.
• The service must not experience meltdown through something going out of control, such as a memory leak, the log filling up, or simple overload.

**Scalability:** The potential for meltdown through overload is much to be feared with products that have not been through load and volume testing. In fact, it inspired the most-asked question from other sites, which was, “Does it scale?” This is a question worth asking about any reference implementation, because it is generally not a requirement.

We found it difficult to come to grips with this question in our first phase of implementation. There were no mechanisms in uPortal to determine the number of active sessions, or the resources they were consuming. If you are worried about scaling, and don’t have a good way to measure load, it’s a good idea to implement on the most powerful hardware available to you, and then grow your customer base gradually.

What worried us more than simple overload was whether there were any pathological behaviours lurking in the code, where without warning, some resource would suddenly get consumed without limit and everything would come crashing down.

**What Gave Us the Confidence to Proceed**

Given our uncertainties, what gave us the confidence to proceed with the project was a shared understanding of what we were undertaking, and the associated risks. We were delivering *useful*, but not *critical* function to our users. While we expected our service to be useful enough to attract users, it would not be critical to any of the University’s core business processes such as registering students, or hiring faculty.

Our release date was important but not critical. We aimed for a release that coincided with the arrival of a new set of students, but the actual date was not significant. This allowed us to employ a gradual release strategy, spreading our invitations to enroll in the portal over a number of days so that we could monitor its performance.

We received a great deal of encouragement from senior management, in particular from our Associate Vice President and our Director. We were encouraged to take risks and we felt comfortable in doing so.

We had available to us a number of people with many years of development experience (even though it was not with the specific technologies we planned to employ).

We had in place proven and well-understood infrastructure which we could use as a foundation for our implementation. This included well-managed hardware platforms, enterprise-level database support, and a substantial application server.
The Major Problem that We Faced

It was clear to us that the major problem that we faced was: “Can we turn a reference implementation into a production service?” What was most important, we decided, was to recognize which portions of the uPortal reference code were of solid production quality, and which had a “prototype” character to them. We then had to decide what strategies we could use to deal with any weaknesses.

What We Did to Address the Problem

Our first step was to retrieve the uPortal code from the shared repository, and freeze a local copy of it. This put us in control of all changes to our code base.

As delivered, uPortal uses a light-weight database. We ported it to Oracle to put the database on solid footing.

We adapted the authentication and authorization components to use an existing service that was already in place at our site. We did not want to invent and manage a whole new set of accounts, and we saved ourselves a lot of work.

We deployed the BEA WebLogic Server as our servlet/JSP engine. UBC already had some experience with this product, and we were already supporting it for another application.

We assembled a team of people skilled and experienced in Operating Systems development and support. We felt that this was the most appropriate type of person to work on a portal framework. They have the right mind set and understanding of the necessary concepts.

In general, our approach was to carve out those parts of the reference implementation that we could replace with proven solutions. This exercise also served to demonstrate how platform-independent and database-independent the code was, and how easy it was to adapt it.

Other Challenges

Our other challenges still remained: the new technologies, and the question of what functionality would be most attractive to users. We embarked on an energetic training and learning program and engaged a firm of consultants (Interactive Business Solutions) to assist us in acquiring the necessary skills and knowledge. IBS has been a major contributor to the uPortal project since its beginning.

For our flagship application we chose to develop a Web Mail service. We had already identified a need for a Web-based mail client that could use our existing IMAP service. The portal was the obvious place to deliver it.
First Experiences – Phase 1 Deployment

We launched our first public portal service – myUBC – in September of 2000. Initially, we didn’t announce it formally or advertise it widely. This gave us the opportunity to respond to some problems:

Secure Connections: In an attempt to make things more secure, we decided to allow only HTTPS connections to the service. Internet Explorer browsers handled this fairly well, but Netscape browsers complained about the mixing of secure and insecure content, and refused to display any channels that obtained their content from “insecure” hosts. We had to rethink and redo our approach.

Scaling Problems: Our first scaling problem had to do with error logging. Too many exceptions were being reported in detail, and the logs were being flooded.

Database Access Recovery: We discovered that if the portal ever failed to get a connection to its database (which happens all too often in the real world of networks), it would detect and log the problem, but would then proceed as if nothing had gone wrong, making no attempt to recover. The result was a flood of “null pointer exception” messages in the log files. Recovery code is a major part of any production implementation.

Database Out of Service: If the portal could not connect to the authentication database, the message to the user simply indicated that the authentication had failed. The same message was issued if the user had entered an incorrect password. The framework failed to distinguish between the two cases. This is misleading behaviour because the user is encouraged to keep trying to log in, even when authentication is impossible.

All of these problems are of the sort that one should expect from new code that hasn’t gone through production trials.

What We Should Have Done According to the Rules

The rules for system development say that we should have done thorough testing. Given our time frame and resources, we managed to do only a small fraction of what is recommended.

- System Testing: We did a quite a lot of testing to verify that the portal performed correctly.
- Volume Testing: We had no practical way to generate scenarios of large numbers of users connecting to the portal.
- User Environment Testing: We tested a small number of browsers, using fixed configurations.
- Stress Testing: We should have run tests with large numbers of users exercising complex layouts. We did none of these.
- Load Testing: Nor did we test with a large number of concurrent requests.
Remarks and Reflections

When you’re not sure how well your service will scale, and you don’t know what the loads will be, investing in powerful hardware can be a cost-effective option. Modern platforms can deliver enormous computing power at a modest price. The main worry then becomes not simple load management, but the potential that some resource or process will go out of control and no amount of hardware will be sufficient.

It’s also worth noting that users’ expectations have changed over that last decade. Users are much more tolerant of Web-based services than they were with time-sharing or heavy-client services. We don’t get many complaints when we have minor outages. We mustn’t be complacent about this. It is necessary to consider the purpose and patterns of usage. If we’re not getting complaints over minor outages, it’s likely because few of the users rely on the portal to respond to the pace and rhythm of their jobs. If UBC staff start using our portal Web Mail to perform their daily tasks, we will begin to see dissatisfaction over service interruptions, because any outage or lack of responsiveness will result in someone not getting their work done. This sets us a challenge for the future. The important thing to understand is that users expectations for availability and reliability are not uniform.

How Did We Achieve Control?

Given the limitations of our approach, with minimal testing, we devised strategies to keep control of the service:

**Few, Well-Behaved Channels:** We restricted our offering to a small number of application channels which we could monitor and repair if there were failures.

**Minimal Dependence on External Resources:** As much as possible, we avoided having the portal and its channels depend on resources that we don’t control. We do use a service from another department (the UBC Registrar) to authenticate student numbers, and we have made sure that our code responds appropriately when that service is not available.

**Recovery Code:** We added extensive recovery code to the database access modules. We trap errors from channels and prevent them from flooding the logs with superfluous messages.

**Substantial Hardware Platform:** We deployed on the most powerful hardware that we had, initially on a Sun E450, and later a Sun Fire 280R.

**Controlled Staging and Promotion:** We have a tightly controlled process for promoting code changes from development, through verification, and into production.
Gradual Release Strategy: We did a gradual release to our users, rather than a big-bang launch.

Local Non-Portable Changes: We found ourselves making some UBC-specific changes without much concern for portability to other sites.

Avoided Over-Control: We didn’t make rules about who was allowed access, but simply let anyone who had credentials for our central services subscribe to the portal, even if they were outside of our target user group. We took the chance of having too many subscribers, but avoided the overhead of a policy of restriction.

How Successful Were We?

We received quite a lot of positive comment with our initial launch, and the few negative comments were all constructive. Enrollment has increased steadily.

A portal lives by its content. In our Phase 1 release we were limited in our ability to bring in new content, mostly because of our concerns with keeping the service stable and reliable.

To meet our local objectives, we had to make changes to the code that forked somewhat from the central reference code. This cut us off from improvements contributed by other sites.

We learned a great deal about developing and deploying Java applications on the Web, enhancing our value as an institutional resource. This was one of our goals.

We were lucky. No wildly chaotic behaviours or meltdowns occurred. Credit is due to the original developers for writing clean code.

We did succeed in addressing our main problem of turning a reference implementation into a production service. People are using the service, and finding it useful.

Would We Do It Differently?

Given the opportunity, would we have proceeded differently? Given our goals and requirements, probably not. Of course if we had had a different immediate objective such as supporting a critical core business process, we would have used a different approach. Our strategy now is to advance to that same level of reliability by bringing some traditional testing, monitoring, and control mechanisms into the project.

Phase 2 Deployment – 8 Months Later

Up to this point this document describes our portal implementation as it was at the end of its Phase 1 deployment in February of 2001. In the eight months following, we have proceeded to work on making uPortal a production-quality product.
Session Management: A “session manager” component was created to keep track of sessions and display metrics in a control panel. Until we had this, we couldn’t assess the instantaneous number of active sessions, or what resources they were using.

Scaling: Our first measurements showed us that a “guest” session – someone just connecting to our splash page without logging in – was consuming 600KB of memory. Clearly this wasn’t going to scale well, as we could imagine hundreds or thousands of such connections if our service became very popular. An extensive retooling of the layout code was undertaken to make the guest content shareable across all sessions.

Dynamic Delivery of Static Content: In our first implementation, all our content was delivered by JSPs (Java Server Pages). This is an unnecessarily CPU-intensive way to deliver content that doesn’t change between requests. We changed the logic to deliver static content through a front-end Web Server whenever possible.

Reconciliation With Central Repository: We retrieved the latest code from the central repository and did a complete re-implementation, reconciling all of our changes with the central source. We contributed all our enhancements and became synchronized in our versions with the other developer sites.

Control of Channels: We established rules for channel behaviour, and added wrapper code to enforce it. We added code to impose timeouts on channel rendering so that a misbehaving channel could not hang the display. We filtered out superfluous error messages from channels to minimize logging overhead.

Testing: We established a practice of extensive end-user-function testing and regression testing. We performed a series of stress tests with artificial loads. We have not yet done volume testing with synthesized loads.

New Content Easier: We have made it much easier for other units on the campus to create and publish their own channels. They can choose from a list of channels types, some of which are simple and easy to build, and some more complex with advanced functionality.

Where Does uPortal Go From Here?

Release 1.6: The UBC implementation of uPortal is based on release 1.6 of the repository code. This is derived from the release 1.0 code, which was all that was available when we began our project. The 1.x releases of uPortal have a very simple rendering model, and use simple XML constructs to store layout information. Unfortunately, this has led to a storage structure which is far from normalized. Redundant descriptions of channel characteristics are replicated within users’ layout records, making the database inflexible and hard to update. We knew of this problem when we undertook our implementation, assuming that it would be addressed in a future version.
**Release 2.0 Beta**: A complete redevelopment of uPortal under the name “Release 2.0” has been underway since 2000. A beta version is now available. Two major architectural improvements distinguish Release 2.0 from its predecessor:

- **XML Based Rendering**: All output from channels is expressed as XML. The channel content is then passed through XSLT transforms to prepare it for the target display device. This extra layer of abstraction allows the same channel code to be used with a variety of display hardware, including hand-held devices.

- **Normalized Database**: The framework database, which keeps track of channels, users’ channel selections (layouts), and user profiles, is now a set of normalized relational tables.iii

**UBC’s Plans**

While we will continue in our efforts to make the portal framework more robust and reliable, it is our intention for the immediate future to concentrate on the channels. A portal can only succeed by the accessibility and usefulness of the content it delivers through its channels. We plan to increase accessibility through a “single signon” for all channel content. To make content more useful, we are embarking on an E-business strategy that focuses on the relationship between the individual and the institution. We expect that future applications will transcend the existing “customer to department” relationships on which many of our services are currently modeled. An institutional portal does not by itself address this objective, but it does provide a focus for identifying what is needed and a foundation for building solutions.

We intend to continue with our involvement in the uPortal project, and our contributions to Release 2.0 and future releases. Having started with 1.0, we will have to go through a substantial conversion process to migrate to Release 2.0. We expect to undertake this when we observe that Release 2.0 exhibits the same level of production quality as we have achieved with Release 1.6.

---

i uPortal is a free, sharable portal under development by institutions of higher education. It is a collaborative development project with the effort shared among several of the JA-SIG member institutions. (From the uPortal web page at mis105.mis.udel.edu/ja-sig/uportal)

ii The Java in Administration Special Interest Group (JA-SIG) is an independent organization designed to increase the flow of information between educational institutions and companies involved in the development of administrative applications using Java technology. (From the JA-SIG mission statement at www.ja-sig.org)

iii The re-implementation of the database service was done at UBC.