Redeveloping the Loughborough online reading list system

Redeveloping the Loughborough Online Reading List System

Jon Knight, Jason Cooper and Gary Brewerton describe the redevelopment of Loughborough University's open source reading list system.

The Loughborough Online Reading Lists System (LORLS) [1] has been developed at Loughborough University since the late 1990s. LORLS was originally implemented at the request of the University's Learning and Teaching Committee simply to make reading lists available online to students. The Library staff immediately saw the benefit of such a system in not only allowing students ready access to academics' reading lists but also in having such access themselves. This was because a significant number of academics were bypassing the library when generating and distributing lists to their students who were then in turn surprised when the library did not have the recommended books either in stock or in sufficient numbers to meet demand.

The first version of the system produced by the Library Systems Team was part of a project that also had a 'reading lists amnesty' in which academics were encouraged to provide their reading lists to the library which then employed some temporary staff over the summer to enter them into the new system. This meant that the first version of LORLS went live in July 2000 with a reasonable percentage of lists already in place. Subsequently the creation and editing of reading lists was made the responsibility of the academics or departmental admin staff, with some assistance from library staff.

LORLS was written in Perl, with a MySQL database back-end. Most user interfaces were delivered via the web, with a limited number of back-end scripts that helped the systems staff maintain the system and alert library staff to changes that had been made to reading lists.

Soon after the first version of LORLS went live at Loughborough, a number of other universities expressed an interest in using or modifying the system. Permission was granted by the University to release it as open source under the General Public Licence (GPL)[2]. New versions were released as the system was developed and bugs were fixed. The last version of the original LORLS code base/data design was version 5, which was downloaded by sites worldwide.

Redesign

By early 2007 it was decided to take a step back and see if there were things that could be done better in LORLS. Some design decisions made in 1999 no longer made sense eight years later. Indeed some of the database design was predicated on how teaching modules were supposed to work at Loughborough and it had already become clear that the reality of how they were deployed was often quite different. For example, during the original design, the principle was that each module would have a single reading list associated with it. Within a few years

several modules had been found that were being taught by two (or more!) academics, all wanting their own independent reading list.

Some of the structuring of the data in the MySQL database began to limit how the system could be developed. The University began to plan an organisational restructuring shortly after the redesign of LORLS was commenced, and it was clear that the simple departmental structure was likely to be replaced by a more fluid school and department mix.

Library staff were also beginning to request new features that were thus increasingly awkward to implement. Rather than leap through hoops to satisfy them within the framework of the existing system, it made sense to add them into the design process for a full redesign.

It was also felt that the pure CGI-driven user interface could do with a revamp. The earlier LORLS user interfaces used only basic HTML forms, with little in the way of client-side scripting. Whilst that meant that they tended to work on any web browser and were pretty accessible, they were also a bit clunky compared to some of the newer dynamic web sites.

A distinct separation of the user interface from the back-end database was decided upon to improve localization and portability of the system as earlier versions of LORLS had already shown that many sites took the base code and then customised the user interface parts of the CGI scripts to their own look and feel. The older CGI scripts were a mix of user interaction elements and database access and processing, which made this task a bit more difficult than it really needed to be.

Separating the database code from the user interface code would let people easily tinker with one without unduly affecting the other. It would also allow local experimentation with multiple user-interface designs for different user communities or devices.

This implied that a set of application programming interfaces (APIs) would need to be defined. As asynchronous JavaScript and XML (AJAX)[3] interactions had been successful applied in a number of recent projects the team had worked on, XML was chosen as the format to be used. At first simple object access protocol (SOAP) style XML requests was experimented with, as well as XML responses, but it was soon realised that SOAP was far too heavy-weight for most of the API calls, so a lighter ‘RESTful’ API was selected. The API was formed of CGI scripts that took normal parameters as input and returned XML documents for the client to parse and display.

Database Design

However, the major change for the redesign for LORLS version 6 though was the need to overhaul the MySQL database schema. The original LORLS design included a generic ‘item’ database table that contained enough fields to handle books, book chapters, journals and journal articles.

However as the item type was not explicit, many academics had added data that made it difficult to programmatically distinguish the item type for each entry in the table, especially if the ISBN and ISSN fields were empty.

Therefore in LORLSv6 a move was made to strong typing of item records, so that each item could easily be identified as a book, journal article, etc. It was also desirable for the typing system to be dynamically extendible, so having separate tables for each item type was quickly rejected. The database design that achieved the easily extendible item types was named 'Loughborough's Universal Metadata Platform' (LUMP).

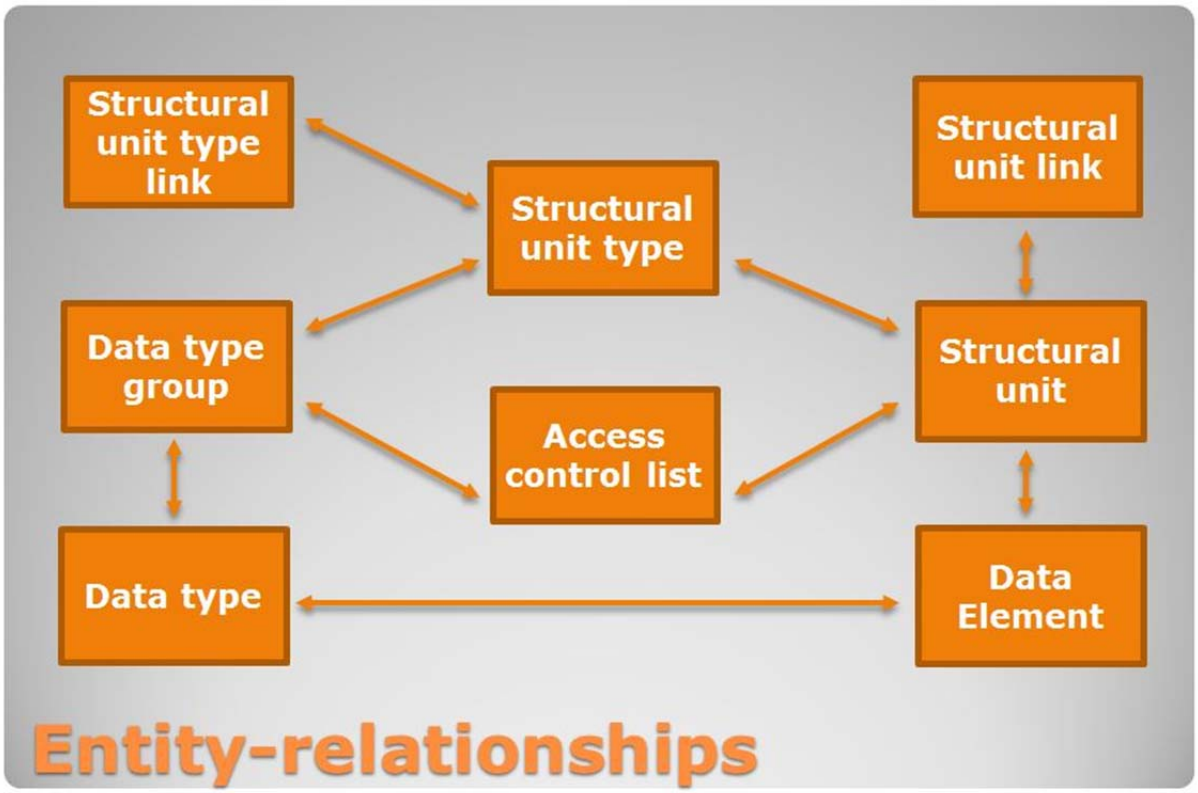

Figure 1: Simplified entity relationship diagram

The fundamental element in LUMP is a table containing rows of 'Structural Units' (SUs). An SU could be a reading list, an item on the reading list (book, journal article, etc), a department, a module or even a whole organisation. Each SU has a single Structural Unit Type (SUT) that supplies this typing. The actual SU table entries only really contain links to the SUT table entry and a bit of administrative metadata (for example whether the SU has been published or deleted, when it was created or last modified and by whom).

The data for each SU is held in one or more rows in a separate Data Element (DE) table. Each DE has a link to a defined data type, stored in the Data Type (DT) table, its value and some administration metadata. Each DT entry contains a name (author, title, ISBN, module code, etc), default value, validation pattern (in the form of a

regular expression) and a suggestion as to how the type should be rendered by the client (text input, radio button, check boxes, etc).

The DTs are themselves grouped into Data Type Groups (DTGs). DTGs serve two purposes. Firstly they are used as a suggestion to the client how the DTs (and thus DEs) should be grouped together. For example a book's bibliographic DTs such as author, title, ISBN and public notes are grouped together, whilst private notes and holding suggestions from academics to library staff are grouped into another DTG while internal library information used by library assistants is grouped into a third DTG. The client may render the DTGs in a way that visually separates them, such as using tabbed forms or an accordion.

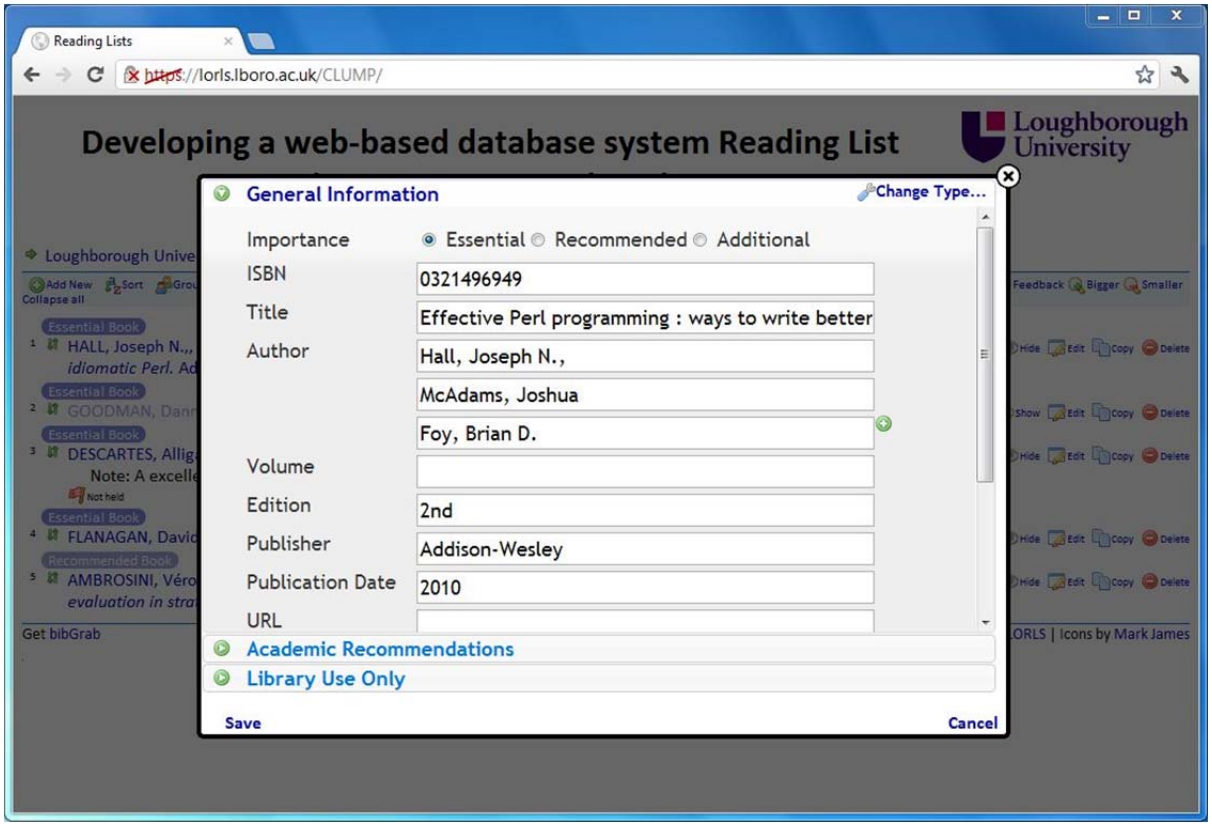

Figure 2: Screenshot of editing form showing different data type editing representations and accordion rendering of data type groups

Secondly the DTGs are used as part of the permissions system in LORLSv6, allowing some DTGs to be hidden from some classes of user. This allows internal library information to be visible to systems staff, librarians and library assistants, but not to academics, departmental admin staff or students.

The DTGs are tied back to the SUT of which they are all part. This results in a loop of tables: an SU has a type given by an SUT, which in turn has a set of DTGs associated with it, the DTGs contain collections of DTs that then provide metadata for handling the data in DEs and, lastly, each DE instance is tied to that particular SU.

Beyond this, each SU can be related to one or more other SUs in the system by entries in a Structural Unit Link (SUL) table. Parent and child relationships are defined, but a strict hierarchy is not enforced.

This provides a lot of flexibility: for example, an SU representing a reading list will have a collection of SUs for books, articles, and similar as its children, but it might itself be a child of several different modules or, indeed, other reading lists. This allows academics to have one reading list that they can use for several modules (which is handy as courses run for different departments are often taught together). As meshes can be generated there is an extra complication in some of the code to ensure that it does not become trapped in infinite recursive loops whilst traversing the mesh.

There are also relationships maintained between the different SUTs in the system. They are held in a Structural Unit Type Link (SUTL) table. This dictates what sort of SUTs can appear as parents or children of each SUT. A book can be a child of a reading list, and a reading list can be a child of a module for example, but it would not make sense to allow a module to be a child of a book.

SUTs, DTGs, DTs, SULs, and SUTLs are all extensible and editable via an admin web interface. New types, groups and links are just rows in these tables, so do not need changes to the database schema or the Perl code that forms the API.

The system needs to consider the permissions that people hold when interacting with these data structures. Previous version of LORLS had groups based on academic staff, departmental admin staff, library assistants, librarians and systems admin staff as classes of users. This had worked well, so it was decided to keep them, but also to design an authorisation system that would also allow other classes of user to be easily added. For authentication LORLS would use existing systems such as Active Directory or Shibboleth.

The permissions are maintained using Access Control Lists (ACLs) and user groups. Individual users are members of one or more user groups and the ACLs indicate what SUs and DTGs the members of each user group can view and/or edit. ACLs allow user groups to be given default rights which can then be overridden for specific SUs and DTGs. Whilst this complicates the ACL checking code somewhat and made development and debugging a slow process, it does mean that the system has far fewer rows in the ACL table and therefore quick authorisation look-ups.

To ease the move from earlier versions of LORLS to LORLSv6, time was spent in producing code to export data from the old database and into the new schema. This was not as simple as it might have first appeared.

The size of the reading list database meant that a simplistic set of single row inserts into each table was not feasible; trying to perform such a migration resulted in run times measured in days. Instead, the migration code had to be able to generate files

of batched SQL queries that allowed thousands of rows to be inserted at a time. As the tables are all inter-related, the Perl migration program had to keep track of items that had previously been added to the SQL files but not yet inserted into the MySQL database.

Rendering Output and Performance

The user interface to LORLSv6 has gone through several iterations. Initially some time was spent on devising a templating system that was accessible through the API. Administrators could define templates that contained fragments of HTML that would then be populated with the results from database searches using other API calls. The templates were held in the database so that LORLS administrators did not need to have file system access to the server that hosted the system (not an issue for the developers at Loughborough, but some other sites had presented difficulties where library systems staff did not have command line access to the servers that hosted their systems).

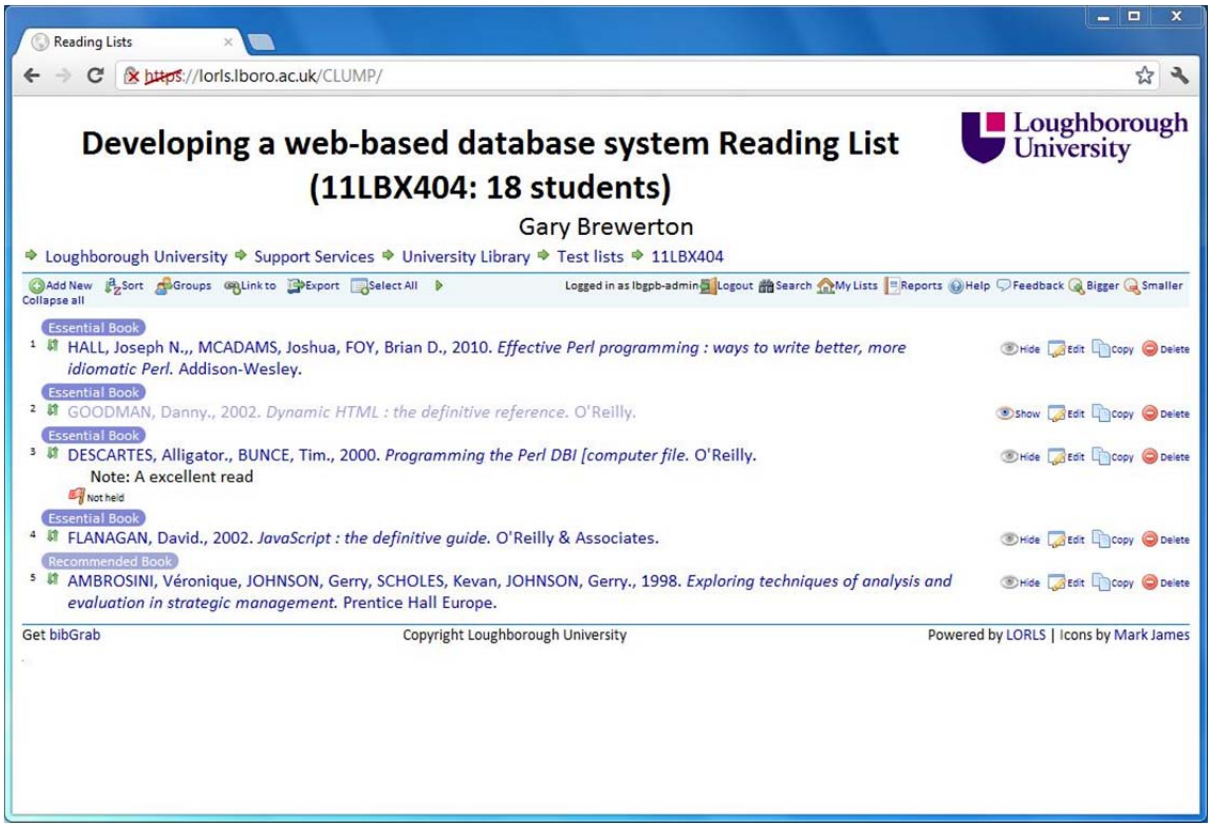

Figure 3: Screenshot of reading list

Whilst this worked it was rather slow, so it was abandoned in favour of the next iteration in which a JavaScript client was written using the JQuery library [4]. This client used the XML output from the APIs and made the rendering decisions locally. This became known as the "Client for LUMP" or CLUMP. The use of JQuery allowed a rich and attractive user interface to be generated, and the client-side scripting reduced some of the load on the servers. As the API was used it was

possible (using a small CGI API proxy) to allow the HTML and JavaScript of the user interface to be hosted on a different server to the API and database.

The performance was improved, but it still was not as fast as users would like. Several techniques were tried at the back-end to speed things up:

- re-coding some of the low-level database interactions to use more streamlined SQL queries
- carefully deciding where indexes would help in the database
- compressing API results on the fly
- building in the ability to cache some common query results in the Perl API so that the SQL database did not need to be hit so often.

One of the biggest performance improvements was made by replacing the general Perl XML::Mini::Document module [5] with a more specific home-grown XMLout() sub-routine. This saved up to half a second on each XML API call, which really improved rendering performance in situations where the front-end code needed to make multiple API calls to render a display. Employment of client-side profiling using Google Chrome's developer tools and server-side profiling of Perl code using NytProf [6], made spotting performance bottlenecks such as this much easier.

The next round of performance improvements involved switching from XML output firstly to JavaScript object notation (JSON) and then JSON with padding (JSONP). The use of JSON for encoding returned data structures required far less client processing than the XML tree traversals that it previously had to perform. It also resulted in smaller, cleaner, more easily understandable code. Switching the APIs to outputting in JSON, instead of XML, was made easy by the use of Perl's JSON module [7], which accepted the same parameters as the XMLout() sub-routine.

JSONP streamlined things further by allowing a callback routine to be specified in the outgoing request which helps to sidestep some of the limitations of straight XML and JSON interactions and, again, to speed up the overall system. One slight downside of JSONP is the lack of synchronous calls; when needed this has to be worked round using blocking whilst waiting for callbacks to be invoked.

The client makes use of multiple threads where possible, rendering the items that the user will see first even it is going to take a while to completely finish all the API requests required. This allows CLUMP to give a response that most users perceive as fast even on really large reading lists with more than 1000 items on them.

JavaScript and JQuery also allow some extra tricks for improving perceived performance. For example light-boxes are used for rendering the details of 'terminal nodes' in the SU mesh (things like books and journal articles that cannot have children SUs). This looks good to the user and has the big advantage that the reading list in the underlying page does not need to be reloaded when the light-box is closed.

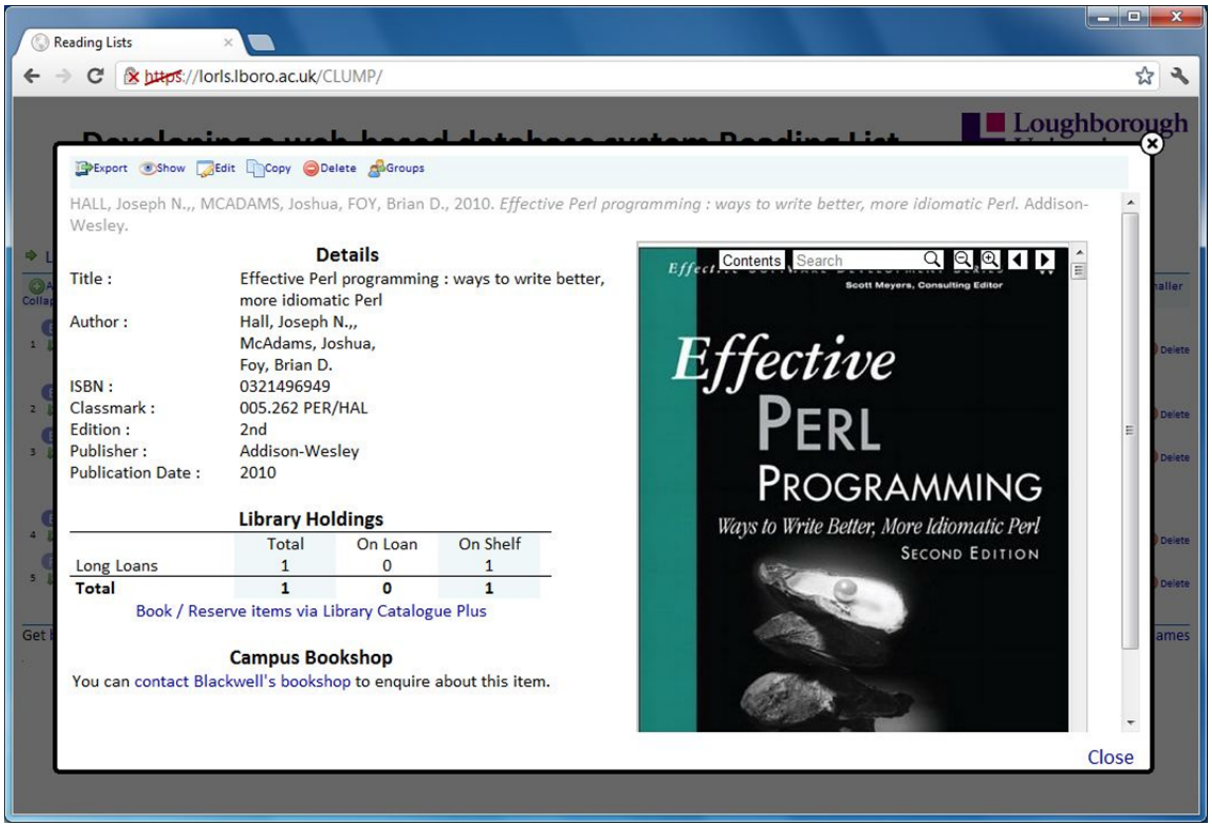

Figure 4: Screenshot of light-box rendering of terminal node

The CLUMP front-end also integrates with a number of other systems. Bibliographic information can be retrieved from the Library Management System (Aleph at Loughborough) via Z39.50, which can be used to fill in work details for academic and library staff after they have provided just the ISSN or ISBN. For the students, links are made where possible to Google Books, allowing the display of either cover art or even the full text of the book in some cases.

The later integration of Shibboleth [8] to allow Single Sign On (SSO) posed a number of problems. Shibboleth does not seem to be really designed to authenticate an AJAX-style API as it wants to redirect the web client to an identity provider (IDP) site if the machine does not have suitable cookies. These cookies are stored by the user's web browser and are not necessarily easily available to JavaScript code, especially if it is being redirected to another site. To work round this, CLUMP makes use of iframes and the back-end web server is configured to allow an API call that can check if Shibboleth authentication has worked or not.

Another client-side development to help academics in generating their reading lists is BibGrab. This is a small ‘bookmarklet’ code that allows users to find interesting works in a web page, highlight them and then import them into one or more of their reading lists in LORLS. Academics can thus build and update their reading lists directly from other material that they or colleagues have placed online.

Development Process, Testing and Deployment

As the redevelopment of LORLS was likely to take a relatively long time, it was decided fairly early in the redesign process that a development diary would be kept on the blog site [9]. This provided both communication with stakeholders and gave the development team a way to record why certain design decisions were made.

After a significant level of development had taken place, the development diary was made visible off-site, so other sites using LORLS could follow the redesign and submit suggestions. Early in the beta testing process, a demo system was made available to allow them to see how the new system worked from the point of view of an end-user. More time was spent on making the installation process easier than earlier versions as well to help other sites install the new code, which made far heavier use of external modules and tool sets than previous versions of LORLS had.

The release of beta test versions of LORLSv6 allowed us to demonstrate the system to stakeholders on campus and elicit feedback on what they did and did not like. This also alerted library staff, academics and also staff in other support service departments that the LORLS system would be changing at some point, allowing them to prepare support materials and ensure that it would integrate with other systems on campus (such as the Moodle-based virtual learning environment).

LORLSv6 was launched into production use on 14 February 2011. The new system was promoted both before and after production use at a number of events (e.g. local e-Learning showcase, presentations at departmental meetings) to help ensure that academics were ready to use the new system and gather more feedback on the system from them. Overall, feedback from staff and students has been both positive and constructive.

Further Development

After the initial production launch had proven successful at Loughborough, a copy of the code was made available online as well for other sites to download. However development did not halt at this point, as a number of performance improvements were made and new features added. Suggestions for new features have come from a variety of users including both those internal to Loughborough University (e.g. Library staff, academics, students) and external (e.g. attendees at the Summer 2011 ‘Meeting the Reading Lists Challenge’ workshop [10]).

New features that are currently in the internally released beta version include a new API call that allows a SU to be converted from one SUT to another, including mapping different DTs onto one another. This allows staff to change a book into a book chapter for example, whilst still maintaining the strong typing in the system and preserving as much data as possible.

Currently code is being developed that can import references in BibTeX, RefWorks or RIS formats into reading lists. This will allow academics to create lists quickly

from their existing bibliographic management tools, and complements similar code that allows students to download reading lists references in the same formats.

A start has been made on coding of the next version of LORLS. Major version numbers in LORLS tend to be increased when the database schema is altered, in the case of LORLSv7 the main changes made so far are to permit the implementation of item ratings. This is intended to allow students to indicate whether they consider particular items good or bad. This information may be of use to other students in choosing which works they will use from large reading lists, as well as providing feedback to academics on what their students think about the works on the lists.

Another development is some code to parse Microsoft Word files in "DOCX" format and extract Harvard style citations. DOCX is effectively a ZIP compressed archive of XML files, so standard XML parsing tools can extract information from them fairly easily. The extracted XML is combined with a set of Perl regular expressions that attempt to extract paragraphs that appear to be in Harvard format.

This is more of an art than a science as many students (and some academics!) have rather liberal views on what constitutes a Harvard style citation. The initial code has been developed against a number of Word documents used by academics on campus, and, if generally successful, will fulfil an often repeated request from academics to be able to submit their reading lists as part of a larger Word document that they give to their students.

Conclusion

Redesigning a successful, decade-old system that forms a core of the Library's online offering was a large task. However it has produced a far more flexible system that is more user-friendly and has provided a platform for on-going developments.

The new database schema provides the dual benefits of strongly typed items held within the system whilst allowing new types to easily be deployed. Strong typing allows the front end code to provide type specific user interactions and also eases subsidiary tasks such as import and exporting to bibliographic reference maintenance software.

Having a separated client using an API to talk to the back-end has allowed a division of development labour and permitted the use of a richer set of user interface designs than previous versions of LORLS have had. These have been popular with both student and staff users of the system.

Performance of such a system can easily be a stumbling block, but it has been found that the combination of profiling tools and multiple performance enhancements in both the back- and front-end code can pay dividends. The use of JSON and JSONP provides significant performance improvements over XML for web-based APIs,

which may be something to consider when designing other asynchronous client-server web systems.

The integration of Shibboleth with an XML or JSON/JSONP based API has proved possible, but only by some rather inelegant hacking with iframes. It is not clear if this is a limitation in the design of Shibboleth or the particular implementation in use.

One thing is clear though: reading list systems are still a fertile place for developments and trying out new features. They are heavily used by students, allow libraries to manage their stock more effectively and can help academics support their teaching. LORLS has thrived for a decade and looks set to continue development well into the future.

References

1. Brewerton, G. and Knight, J., 2003. From local project to open source: a brief history of the Loughborough Online Reading List System (LORLS). VINE, 33(4), pp. 189-195 https://dspace.lboro.ac.uk/handle/2134/441
2. GNU General Public License, version 2 http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
3. Powers, S., 2007. Adding Ajax: Making Existing Sites More Interactive. O'Reilly Media.
4. jQuery: The Write Less, Do More, JavaScript Library http://jquery.com/
5. XML-Mini-1.2.7: XML::Mini::Document http://search.cpan.org/~pdeegan/XML-Mini-1.2.7/lib/XML/Mini/Document.pm
6. Devel-NYTProf-4.06: Devel::NYTProf http://search.cpan.org/~timb/Devel-NYTProf-4.06/lib/Devel/NYTProf.pm
7. JSON-2.53: JSON http://search.cpan.org/~makamaka/JSON-2.53/lib/JSON.pm
8. Shibboleth: What's Shibboleth? http://shibboleth.net/
9. Loughborough Online Reading List System (LORLS) development diary http://blog.lboro.ac.uk/lorls/development-diary
10. Loughborough University: Meeting the reading list challenge http://blog.lboro.ac.uk/mtrlc/