Development of a lightweight library catalogue

Submitted by Jon Knight on 14 June 2017 - 3:51pm

Jason Cooper describes how a lightweight temporary library catalogue system was constructed when Loughborough University opened their second campus in London.

In 2014 Loughborough University launched Loughborough University London, a postgraduate campus located in London. Until this point Loughborough University's Library [1] had been single site, so there were a number of factors to take into account to decide on the best way cater for the second site, including:

- *Size* - the London campus library was significantly smaller than the Loughborough campus library
- *Shared online resources* - the majority of online Library resources were accessible to users at either campus
- *Complexity of switching the Library Management System (LMS) to a multi-site configuration* - Moving the LMS from a single-site configuration to a multi-site configuration would require a lot of time and resources
- *Upcoming replacement of LMS* - The Library's LMS (Ex Libris's Aleph [2]) had recently been reviewed and a project was being prepared to tender for a replacement

After weighing up the factors, it was decided that the best time to move to a multi-site setup of the LMS would be as part of the upcoming replacement of the LMS. Most of the Library services would work at an acceptable level of service by placing London campus specific resources in their own collection, the main exception being the Library's Resource Discovery Service (Primo [3]).

Adapting the Library's existing instance of Primo to have a second interface for the London campus would have required extensive configuration changes, which could have impacted severely on Loughborough students. It was decided that while access to Primo would be available to patrons based at the London campus for activities like checking and renewing current loans, it would also be beneficial to provide them with a lightweight online catalogue for finding London based and online resources (Databases, E-Journals, etc).

Catalogue requirements

The following were drawn up as the key requirements of the system:

- It should allow the user to search for items either physically located at the London campus or available as an e-books collection
- It should list the results of a search with the core details of the items, but allow the user to click on an item to get a popup with more details
- Both the search results and the popup should include thumbnail images of the items
- It should have a form that can be used by the London campus users to request items from the larger Loughborough campus library
- It should provide access to a number of key Library websites
- It should work on multiple types of devices, from the smaller smart-phone size screens all the way up to the large screen sizes available on desktop PCs
- It should be able to detect that it's running on a dedicated OPAC machine and disable

functionality to suit

Design decisions

The team responsible for developing the system would traditionally have used a relational database (usually MySQL [4]), but a quick design of a schema showed that not only would the schema required be complex, but that it would also require complex SQL queries for searching items. It was decided that instead of using a relational database engine that the system would be built upon a Document Store NoSQL database engine [5], as this was what the relational database schema being designed was effectively turning into.

The NoSQL database engine chosen was MongoDB [6], which uses a Document Store model and has full text indexes that can cover multiple fields and a natural search syntax, that would vastly simplify the searching routines. While the team had played with MongoDB in the past this would be their first real project using it.

The next decision the development team made was how to structure the web interface part of the system. The two choices available were

- Server side page generation (e.g. CGI scripts)
- Client side AJAX style, where a JavaScript front-end makes requests to a back-end API

The decision to use a Client side AJAX style structure was chosen as this would reduce the load on the server and also provide a more fluid interface to the user. The development team had successfully used this structure in a number of projects [7, 8, 9] and found it to be very comfortable to work with.

The two options available to the development team to tackle the requirement that it should work on a range of devices from smart-phones to desktops were:

- Device detection
- Responsive web design

The team chose to use responsive web design [10] as it was felt to be easier to maintain going forward due to it separating the display logic from the application logic.

Development

Schema structure

Designing the initial schema for the MongoDB database turned out to be very simple requiring a single collection for items to which indexes were applied on the following fields:

- System number

- Title
- ISBN
- ISSN
- Author
- Collection

In addition to those indexes, a text index was applied that covered the title and author fields.

Harvesting items from the LMS

Having designed the initial schema the next step was to develop the script that would update the MongoDB collection from the LMS. After investigating both Aleph's REST API [11] and X-Server [12] it was decided that the REST API was simpler to use and returned the results faster. Unfortunately neither the REST API or X-Server provided an easy way to extract all items in a specific collection. As the development team already have a lot of experience with Aleph's underlying database it was decided that the simplest way to proceed would be to create an API, in the form of a CGI script, which given a collection code would return all the system numbers of items in those collections.

A script was then developed that, for each collection to be harvested, calls the custom API to retrieve the system numbers of items in that collection. Then for each system number it calls the LMS's REST API to pull back the full item's metadata, which it would create a document object from and “upsert” it into the MongoDB item collection. An upsert in MongoDB is where you specify the document should replace an existing one if it matches a field or expression (in our case the item's system number), if the specified field or expression doesn't match any existing items then it will be added.

The harvesting script was scheduled to run each night to update the items collection in the MongoDB database.

Creating the search API

Once the harvesting of item records from the LMS was complete, attention was turned to developing an API that the front-end would use to find relevant items. The API was developed in Perl[13] and consists of two stages:

1. Parsing the search query to produce a suitable MongoDB query filter

2. Using the query filter to extract the suitable results from the database

Producing the MongoDB query filter

In Perl a MongoDB query filter is a hash reference with nested hash and array references. An example of a query filter produced for a search for "track" in the E-Book collection (collection code EB) would be:

```
{
  '$and' => [
    { 'collection' => 'EB' },
    {
      '$text' => {
        '$search' => 'track',
        '$language' => 'en'
      },
    },
  ],
}
```

The keys starting with a `$` are query operators, so the `$and` in the example states that we're looking for items that have a collection of 'EB' and that match our `$text` search.

The `$text` operator specifies a text search against the collection's text index. In the example we're using the `$search` and `$language` parameters, where the `$search` parameter is the search terms being looked for and `$language` is used to control the stopwords and stemming properties of the search.

In addition to the `$search` and `$language` parameters the `$text` operator can also take the `$caseSensitive` and `$diacriticSensitive` parameters, where `$caseSensitive` indicates whether the search should be case sensitive and `$diacriticSensitive` indicates whether the search should consider characters with diacritical marks as being the same as their non-marked counterpart.

Constructing a suitable query filter in the API is quite simple as the only two values that differ between searches are the collection value and the `$search` value.

Developing the front-end.

Once the search API was complete development of the front-end could start. The design was kept simple and the page was split into three parts, a header, a footer and a list of results (Figure 1 shows a screenshot of the final front-end design).

Figure 1: Final front end design

The header

The header contains the following

- a logo
- search options
    - which category to search
    - the search box that the user can enter their search terms into
    - the search button, which initiates a search
- a help button that shows a popup box containing help on using the system including detailing the more advanced search terms available.
- links to the following additional Library resources:
    - Library website
    - Reading Lists (LORLS [14])

    - E-Journals A-Z (SFX [15])
    - Item Request form
    - Library Account (a deep link into Primo)
- a dropdown list that lets users order their results by relevance, title, order or date

The results list

Once the user has done a search the results list contains all the results found. For each result the following metadata is displayed:

- Title
- Author
- Date
- ISBN
- Imprint
- Shelf Mark
- Notes
- URL (rendered as an active link to the resource)

Where available a thumbnail of the item's cover from Google Books [16] is displayed alongside its metadata. The thumbnail is linked to the item's entry on Google Books.

The footer

The footer simply contains a mailto link that users can use to email the Library.

Responsive web design

The responsive nature of the front-end interface is handled via the CSS [17] stylesheet using `@media` rules. Each `@media` rule used is tied to the width of the screen, so as the screen width drops below certain widths how elements are displayed changes. The actual changes required to keep the system usable on smaller screens are:

- below a width of 880 pixels the logo in the header is hidden
- below a width of 600 pixels some font sizes and padding are reduced to make the most use of the limited screen space
- below a width of 520 pixels the font sizes in the header are reduced again and the size of the search box and related options are altered to fit better on a smaller screen (Figure 2 shows an

example of the screen at this width)

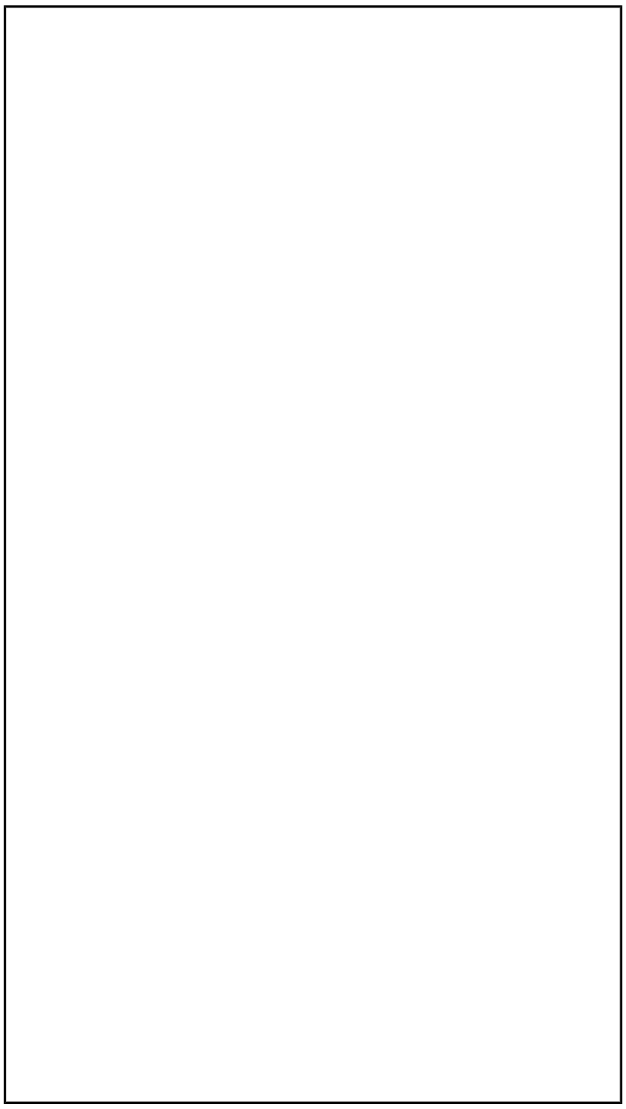

Figure 2: Example of screen at 520 pixels wide.

Detecting when running in a dedicated Catalogue PC and disabling specific options.

The final feature needed to meet the initial requirements was that it should be able to detect when it was running on a dedicated catalogue machine and disable certain features. As the Library control the dedicated catalogue machines they are able to set its browser's useragent string to specific value. When the front-end javaScript detects that the page is being viewed in a browser using this useragent string it makes two changes to the Document Object Model [18].

- it removes the target attribute for the reading list menu entry (which means it will load the reading list in the same window)
- it adds a class of onOPAC to the pages body tag

All other changes required are handled by the page's CSS, specifically:

- all links in the header that would open in a new window are hidden

- the mailto link in the footer is hidden
- all links in results list are disabled by setting their `pointer-events` property to `none`

Going live

The lightweight library catalogue went live at the same time that the London campus opened and required very little maintenance. The system met the majority of the expectations of students and staff. The only significant feature missing was the ability to display the current availability of print items in the London campus library.

Phase 2

After the system had been in use by the London campus library for a year it had become apparent that the project to tender for a replacement LMS was going to take longer than originally thought, so it was decided to extend the lightweight catalogue by adding two new features:

- a new collection of career resources
- displaying current availability of items

Adding a new collection

Adding a new collection was really a couple of minor changes. The first was to extend the Custom API on the LMS to include the system numbers of new items in its results. The second was to add the new collection to the front-end by adding another `option` element to the `select` element of the search form.

Displaying current availability of items

Adding the current availability of items in the result list was a bit more in depth than adding the new collection. The back-end would require a new API adding for retrieving the current availability of items and the front-end would need extending to make use of the new availability API and display the item availability alongside their entry in the search results.

The new backend API developed takes a list of system numbers and then for each one call the LMS's REST API to pull back a list of all the items for that system number along with their collection, loan type (e.g. week loan, month loan) and current availability status. Items statuses that weren't in a suitable collection for the London Campus were discarded. This left only relevant items from which the total number of items and the total number of available items could be calculated for each loan type.

The extension of the front-end to make use of the new API required additional code to be added to the javaScript routine which displayed the results list. As each result is displayed this new code checks to see if the result is from a collection that it should show the current availability status for (i.e. not the E-Books collection) and if so then it adds an empty field to be populated with the item’s availability later, it also adds the results system number to an array.

Once all results have been displayed, the populate availability function is called with the array of system numbers. If the array has contents then the array is concatenated together and used to make an AJAX call to the back-end's availability API and then uses the response to populate the empty fields added earlier with the current availability information for each result.

Conclusion

The lightweight online catalogue has been in active use by the London Campus since it opened in 2014. During this period there have been very few bugs reported, all of which have been minor.

Even though it was the first time the team had used MongoDB in a production service they found that using a NoSQL document store at the heart of the back-end reduced the development times for this style of system and simplifying the logic required in the search API.

The team also found that the use of `@media` rules to provide a responsive site helped keep the layout logic separate from the content being displayed.

References

1. (2017). Loughborough University Library [online] Available at: http://www.lboro.ac.uk/library/ [7th March 2017]

2. (2017) Ex Libris the bridge to knowledge, Aleph. [online] Available at: http://www.exlibrisgroup.com/category/Aleph [7th March 2017]

3. (2017). *Primo Discovery and Delivery* [online] Available at: http://www.exlibrisgroup.com/category/PrimoOverview [7th March 2017]

4. (2017). MySQL. [online] Available at: https://www.mysql.com/ [7th March 2017]

5. (2017). Document-oriented database. [online] Available at: https://en.wikipedia.org/wiki/Document-oriented_database [7th March 2017]

6. (2017). MongoDB for GIANT Ideas [online] Available at: https://www.mongodb.com/ [7th March 2017]

7. COOPER, J. and BREWERTON, G., 2013. Developing a prototype library WebApp for mobile devices. *Ariadne*, (71).

8. BREWERTON, G. and COOPER, J., 2015. Visualising building access data. *Ariadne*, (73).

9. KNIGHT, J., COOPER, J. and BREWERTON, G., 2012. Redeveloping the Loughborough online reading list system. *Ariadne*, (69).

10. (2017). Responsive Web Design - Introduction. [online] Available at: http://www.w3schools.com/css/css_rwd_intro.asp [7th March 2017]

11. (2017) Aleph RESTful APIs. [online] Available at: https://developers.exlibrisgroup.com/aleph/apis/Aleph-RESTful-APIs [7th March 2017]

12. (2017) Aleph X-Services. [online] Available at: https://developers.exlibrisgroup.com/aleph/apis/Aleph-X-Services [7th March 2017]

13. (2017) The Perl Programming Language - www.perl.org. [online] Available at: https://www.perl.org/ [6th June 2017]

14. (2010). LORLS - Loughborough Online Reading List System. [online] Available at: http://blog.lboro.ac.uk/lorls/ [7th March 2017]

15. (2017). SFX Link Resolver [online] Available at: http://www.exlibrisgroup.com/default.asp?catid=%7B7F2829F7-42E6-425C-A9A7-0BBFBD0D6959%7D [7th March 2017](2017)

16. Google Books APIs. [online] Available at: https://developers.google.com/books/ [7th March 2017]

17. (2017) Cascading Style Sheets. [online] Available at: https://www.w3.org/Style/CSS/ [6th June 2017]

18. (2000) What is the Document Object Model? [online] Available at: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html

Date published:
Wednesday, 14 June 2017

- Send to friend