Chapter 4

Patron-Facing Services

Overview

The scripts in chapters 2 and 3 focused on back-of-the-house functions: data quality and technical services workflows. In chapter 4, we'll talk about services patrons interact with directly. One of the wonderful things about coding in libraries is that it inherently breaks down silos: while librarian coders often emerge from technical services, they can be found anywhere, and their work can affect—and link—numerous library functions. A large fraction of library coders are working on library website user experience or other patron-facing services.

Many of these coders are driven by a motivation cited at least as far back as the 1998 version of Eric S. Raymond's landmark essay on open-source software, "The Cathedral and the Bazaar"—"scratching your own itch."[1] Librarians use their library's website, catalog, or other web services frequently, and thus encounter user experience (UX) frustrations that directly affect patrons. Rather than live with inadequate layout or features, they created functionality their system lacked by going outside that system.

Any web product that lets you add some JavaScript—even if you can only add it to the `<head>` and cannot edit the rest of the page—gives you an opportunity to make this sort of tweak. Therefore, JavaScript is the language of choice for most code in this chapter (sometimes augmented and simplified with the wonderful jQuery library).

Examples

The examples in this chapter fall into four categories: UX improvements, presentation of information in new contexts, LibGuides tweaks, and patron services outside of the website.

UX Improvements

Three librarians surveyed wanted to improve the search experience. Amy Wharton wrote a jQuery script to add autocompletion of database names to her search box, allowing users to more quickly find (and correctly spell) what they needed. Joel Marchesoni found that CONTENTdm allowed for Boolean search operators but only through hacking the URL; his ASP script allowed users to enter Boolean queries through conventional search and built the URLs accordingly. Chris Fitzpatrick wrote a CoffeeScript that harvests ISBNs from a page of search results from the Blacklight discovery interface, grabs the corresponding cover image from Google, and enriches the results page with the images—all in a mere twelve lines.

Chris Fitzpatrick's script
https://gist.github.com/cfitz/5265810

Blacklight
http://projectblacklight.org

Rachel Donohue wanted to simplify the process of creating accessible content. As her employer, the National Agricultural Library, a US government agency, has a section 508 compliance mandate, so accessibility is a must; however, Neatline, the Omeka plug-in it uses to create time lines for digital exhibits, doesn't generate compliant content. Her Ruby script makes it fast and simple to provide an accessible alternative.

Rachel Donohue's script
https://gist.github.com/sheepeeh/10417852

Two librarians, Matthew Reidsma and Jason Bengtson, wrote scripts to display the library's hours, including whether it is open right now. It's intriguing to look at these side by side because—while they're both JavaScript programs that display open/closed status on a library website—they're written very differently. Bengston's includes special-case handling for holidays, while Reidsma's covers only standard hours. However, Reidsma's encodes schedule information in a significantly simpler format, which enables him to write much more concise, readable code.

Matthew Reidsma's script
https://github.com/gvsulib/Today-s-Hours/blob/master/todayshours.js

Jason Bengtson's script
https://github.com/techbrarian/openchecker/blob/master/openchecker.js

This comparison illustrates how much of software engineering is driven by brainstorming all the special cases code might need to handle and making choices as to which ones are worth handling in your context. It also underscores the role that aesthetic sensibilities play. There's more than one right way to write programs—in a sense, anything that consistently produces correct results is right. However, authors have different stylistic preferences, and their intuitions about elegance can enormously impact the final result.

Repurposing Information in New Contexts

Just as technical librarians' jobs often break institutional silos, technical librarians' work can break content silos. One of the common frustrations of both library software architecture and library user experience is that information is held in separate systems even when it may be used in shared contexts; the three scripts in this section take information from where it's found to where it's needed.

Indeed, Michael Schofield phrased his JavaScript in terms of breaking silos: "I basically wrote a small API that [took] content that was previously silo'd on whatever platform—LibGuides, WordPress, etc.—and let it syndicate itself around our web presence without having to be duplicated." For instance, "if a patron is looking for business resources and we happened to have a scholarly speaker presenting about business, the API would suggest the event to the user."

Coral Sheldon-Hess also wrote something to syndicate content: in her case, a social media aggregator that pulled content from her library's various social media presences into a single home page display area. While they could have used existing WordPress modules, RSScache offered better performance. She modified this tool to be consistent with her library's branding. (Modifying existing code is often easier and less buggy than writing your own from scratch, as well as being a great starting point if writing your own sounds daunting.) This new website feature "helped us kind of institutionalize social media" by featuring it prominently and unifying previously disparate content. Seeing their new items pushing down older entries also gave the social media team an incentive to write more.

RSScache
www.rsscache.com

Jason Simon wanted to offer subject access to research databases. He had alphabetical lists hardcoded into HTML, but these were time-consuming and error-prone to update; since any given database could appear on multiple subject pages, changes had to be made in each page for every update. Instead, he stored information about which research databases served which subjects in a separate database and wrote a PHP script to pull information from this database and write it automatically into their subject guides.

Subsequently, this project grew organically to encompass new functions, ultimately becoming a "larger back-end homemade ERMS which made it a lot easier to manage subscriptions, statistics, pricing, holdings, etc., for both databases and periodical subscriptions." This points to something that's important to keep in mind if you're a new coder feeling overwhelmed by the size of open-source projects you've looked at—they didn't start out big! In fact, they may not have been started to tackle large problems at all. You can start by writing something small, and along the way you'll learn how you might want to expand it and the skills you need to do so. Conversely, if you know the thing you need to write is large, break it down into the smallest useful pieces and write them one at a time. Other people's software projects didn't spring fully formed like Athena, and yours needn't either.

Springshare Customization

Four survey respondents made changes to their institution's handling of Springshare products, particularly LibGuides. These changes spanned the use cases above—user experience changes and information reuse—but three of them are grouped here to show the variety of possibilities for augmenting this widely deployed product line. The fourth will be the subject of this chapter's deep dive.

Eric Phetteplace's library, Chesapeake College (a hybrid community college/public library), used a wide range of Springshare products for recording library statistics. However, the input forms didn't limit librarians' choices to the recording schema used at Chesapeake; as a result, invalid entries made it hard to analyze the data. Phetteplace installed Tampermonkey on staff computers, which allows for installing additional userscripts—snippets of JavaScript that function as browser plug-ins. He then wrote a userscript that validated form entries before submission, requiring staff to enter valid data. This made it much easier to analyze the data, which in turn helped the library make better choices about how to staff their desks; for example, it could see that it got more computer support questions early in the term and more reference questions near finals and assign desk coverage accordingly. It also helped the library to communicate more effectively with other departments about library usage and impact (see chapter 5).

Tampermonkey
http://tampermonkey.net

At Ohio University Libraries, staff had an array of subject-specific LibGuides and wanted to make sure students looking at the Course Reserves page knew about this option. While the OPAC allowed the library to insert links to LibGuides into those pages, doing so manually would have been prohibitively time-consuming at this school of over 20,000 students. Staff were also concerned that placing links to LibGuides in line with assigned course readings might irk faculty by lessening their control over the content of the course readings area. Instead, Carrie Preston wrote JavaScript (building on the extremely useful jQuery library) that automatically inserted a link to the LibGuides page, including the name of the relevant subject category, in a special block toward the top of the page. You can see it in action on Ohio University's ALICE catalog.

ALICE catalog
http://alice.library.ohiou.edu/search~S7?/rcoms/rcoms/1%2C26%2C29%2CB/frameset&FF=rcoms+4060&1%2C%2C2

Bohyun Kim had the inverse problem. Rather than needing to add LibGuides to course resources, she needed to add course resources to a LibGuide—in her case, hundreds of e-textbooks that were hard to find in the catalog. She had a student worker who was comfortable finding and organizing them but was not comfortable writing HTML, and she wanted to ensure that the end product was compatible with the library's custom LibGuides styling without onerous proofreading on her part. She wrote a web page in HTML and JavaScript (again, taking advantage of jQuery) where her student worker could enter metadata in a human-friendly format. The script then produced appropriately formatted HTML that the student could copy and paste into the LibGuide. You can see the end result, with dozens of e-textbooks alphabetized and properly formatted, on the school's Course E-Books web page. Kim also wrote an ACRL *TechConnect* post that walks through the code.[2] (There are only eleven lines of JavaScript!)

Course E-Books
http://LibGuides.medlib.fiu.edu/courseebooks

Bohyun Kim's script
https://github.com/bohyunkim/examples/blob/master/link.html

Services outside the Web

While most respondents were using code to affect either metadata or the website—that is, strictly computational objects—a few used code to improve services in other domains.

Matt Weaver (whose author name preprocessing script we saw in chapter 2) says, "We purchased a digital signage system from a reseller that didn't really do what we wanted it to do in the first place, and to get it to do something close would have meant a workflow would not have been manageable for one employee." The library wanted the system to display its meeting room schedule, but it could not queue up information to show at set times; staff had to manually change the sign message throughout the day. Weaver's Python script allowed staff to deposit files with event data into a particular folder whenever it was convenient for them to do so. It then pushed the information to the signage software at the appropriate times. Ultimately, "the code rescued a rather expensive, and unpopular project."

In addition, Mike Drake (Deputy Director, Tulare County Library, Tulare, CA) wrote a script to help his children's librarians give better, faster answers to questions like "Do you have any princess books?" In his words, "Some of the most popular books in the children's area are dispersed all over, in different collections. And, most of them are checked out. For example: Disney Princess can be in easy readers, picture books, or juvenile fiction; and filed under several different authors. Our OPAC will only allow us to check the location and status of each title one at a time; and it can be very tedious. I wrote a program that will search the OPAC over the web and return results only for titles that are available, in a single list,

sorted by collection/author. This list can be printed by the librarian, and then the hunt begins!"

While this can be read alongside the UX improvements scripts earlier as another example of transcending the limitations of the OPAC, from a patron perspective it's entirely different. The patrons probably never know that their librarian wasn't using standard library software or that anyone wrote code; most of them probably don't know what code is. They just know that the librarian was able to get them princess books without delay.

Deep Dive: LibGuides Organizer

Jeremy Darrington (Princeton University Library) wrote JavaScript to organize LibGuides to handle an information overload problem. He had some topics for which the library could offer lots of relevant resources, and he wanted to make sure the students could access them all. However, with so many boxes on a page, it was hard for students to navigate the options or get an overall sense of the holdings. He didn't want to clutter the page with too many tabs, either. Instead, he wanted to provide a sidebar table of contents listing all available sections and display only the currently active section so that users didn't feel overwhelmed. Users then could click on the table of contents to selectively reveal sections of interest.

You can see a screencast of the result of the page organizing script on the Princeton website. The page also provides clear, comprehensive instructions on incorporating the script into your own site. (It's based on the older version of LibGuides; LibGuides 2.0 natively incorporates a similar side nav option.)

Page-organizing script screencast
http://LibGuides.princeton.edu/content.php?pid=254621&sid=2824241

The code itself is available on the Princeton site. It's beautifully commented, making clear what each section of the code is doing, as well as specifying the CC BY-NC-SA license. Like a lot of JavaScript, it relies on some understanding of HTML and CSS; if you're rusty on those, pull up a tutorial or reference for them as well. Now, let's dive in!

Jeremy Darrington's script
https://thatandromeda.github.io/ltr/Chapter4.html

Lines 6–8 tell the browser that this function should operate on the document, once it's been fully loaded. They then initialize two variables (that is, create them and assign initial content). These are the lists where we will be storing IDs and titles of the various content boxes (made clear by the excellent variable names `$boxID` and `$boxTitle`). Right now they're empty, but we'll add content over the next few lines.

Lines 10–18 get the titles of the boxes. The CSS selector in **line 10** specifies the header elements of our content boxes. **Line 11** gets the actual text of the header; **lines 12 and 13** test whether it matches a given regular expression. (Regular expressions are ways of specifying patterns of characters; this one means "a number followed by a right parenthesis." This will match the beginning of each line in a list—`1)`, `2)`, and so on.) If the text and the regular expression match, the code strips off the matching part (the `1)`, `2)`, etc.) and adds the remaining text of the header to our list. If there's no match, it adds the entire text. We now have the text of the entries in our table of contents, with unimportant item numbers removed.

Was the regular expression strictly necessary? No; we could have simply added the entire text. Had I been writing this script, the first version would have done just that—and then, after testing it on some LibGuides, I'd have discovered that some table of contents entries had `1)` or `2)` in front of them and some did not. I would have decided that looked weird and added the regular expression to normalize the formatting. This is not necessarily how Darrington proceeded, but this sort of iterative code-test-debug-code process underlies many programs.

Lines 20–21 get the ID attributes of those same boxes and store them in the `$boxID` variable defined earlier.

Lines 24–27 find the box we're going to put the table of contents in. This is a box that was set up during LibGuides configuration, as detailed in the screencast. The HTML IDs that we'll use to find this box (via the selector in **line 24**) are defined by the LibGuides template. It starts out empty, but in **lines 25–27** we loop through `$boxTitle` and `$boxID`—the lists we defined earlier, containing the IDs and titles of our content divs—and add entries to our table of contents. Each time through the loop, we add one line that has a box title as text and that uses the box ID to construct a link. At the end, we have a table of contents. The remaining lines will make it work in the desired manner, hiding and showing page content depending on the currently active link.

Note that these lines assume that the first entry in the `$boxTitle` list corresponds to the first entry in the `$boxID` list —that is, they assume we harvested both from the same place. This happens to be true because the `div[id^="wrapbox"]` selector we use in **lines 10 and 20** to find those boxes always returns them in the same order, and (in those same lines) `.each()` loops through them in the same order each time, *and*

we can count on the lists `$boxTitle` and `$boxID` storing them in the same order we added them. Therefore, we can safely ignore order in this code. However, there are programs where that isn't the case (for example, in Python, lists are stored in order, but dicts are not; when you read a dict, you are not guaranteed to see information in the same order that you wrote it).

Lines 31–51 check the URL the browser is currently pointed at (this is the `location` in **line 31**) to see if it has an anchor at the end (like `#foo`; this is the `.hash` part of **line 31**). We'll do slightly different things in the scenario where it has an anchor and the scenario where it doesn't, using different code blocks (respectively, **lines 32–45** and **lines 47–51**). The curly braces signal to the computer where these code blocks begin and end; the indentation is optional but makes it much easier for humans to keep track.

If there is an anchor in the URL, we'll assume that the user has just clicked on one of the table of contents links (all of which have anchors) and hide or show content accordingly, using the code in **lines 32–39**. **Line 32** gets the ID of the desired box from the anchor link text (ignoring the `#` character at the beginning, which is used by the browser to interpret the URL but is not part of the HTML ID attribute). **Lines 33–34** can be ignored—they're commented out and thus presumably represent failed experiments from the process of writing the code. **Lines 35–39** loop through all the box ID numbers on the page. When they get to our desired `$boxNum`, they show the box, scroll the window to it, and add highlighting to its line in the table of contents to make it clear to the user what the currently active content is. For all other IDs, we hide the div to avoid cluttering up the interface. We're now done processing the scenario where there's an anchor in the URL, and the program will skip down to **line 53**.

If there isn't an anchor in the URL, we'll skip **lines 32–39** and instead process **lines 47–51**. In this scenario, we assume the user has just loaded the page (using its base URL) and should be shown the first content box with the first line of the table of contents highlighted. **Lines 47–49** hide all content divs except the first (LibGuides displayed them all by default). **Lines 50–51** highlight the first line in the table of contents.

To summarize, at this point we've done the following:

- collected information from our page that we'll need to build the table of contents and connect its entries to content areas on the page
- checked the URL to see what the user's currently selected content area is
- made sure the corresponding line in the table of contents is highlighted so users know where they are
- made sure the corresponding content block is shown and the rest are hidden to keep the screen from being cluttered with irrelevant content

Scripts in This Chapter

Chris Fitzpatrick's script
https://gist.github.com/cfitz/5265810

Rachel Donohue's script
https://gist.github.com/sheepeeh/10417852

Matthew Reidsma's script
https://github.com/gvsulib/Today-s-Hours/blob/master/todayshours.js

Jason Bengtson's script
https://github.com/techbrarian/openchecker/blob/master/openchecker.js

Bohyun Kim's script
https://github.com/bohyunkim/examples/blob/master/link.html

Jeremy Darrington's script
https://thatandromeda.github.io/ltr/Chapter4.html

Matthew Reidsma and Kyle Felker's 360Link Reset
https://github.com/gvsulib/360Link-Reset

Matthew Reidsma's snippets on GitHub
https://gist.github.com/mreidsma

Grand Valley State University Libraries scripts on GitHub
https://github.com/gvsulib

Matthew Reidsma's repositories on GitHub
https://github.com/mreidsma

All we have to do now is ensure that, if the user selects a new line in the table of contents, the highlighting shifts to that line, the old content box is hidden, and the new one is revealed; we accomplish this in **lines 53–69**. **Line 53** specifies that this code block is a function that is triggered whenever the user clicks an element whose class is `boxNav`. (This is the class name that Darrington applied to his table of contents entries in **line 26**.) In **lines 54–55**, we find the currently highlighted entry and remove the `currentNav` class (thereby removing the highlight styling). **Lines 56–57** find `this`—a special JavaScript keyword that here represents the element the user clicked—and add the styling that indicates it's the currently active nav entry. **Lines 58–69** then loop through the content areas in the LibGuide, showing (and scrolling to the top) the one that corresponds to the active table of contents entry and hiding the remainder.

And now we're done! When users load the LibGuide, it will show only the first (or active) content area, with the table of contents highlighted accordingly; when they click on the table of contents, the corresponding content will be displayed and the rest hidden. Now Darrington can add quite a lot of content to a LibGuide without overwhelming or confusing the user, as long as he organizes it into logical chunks.

What are some key takeaways from the code? First, clear comments are a great service. Because this code is organized into logical sections and each has an explanatory comment, it's clear what each section of the code is doing even if you don't speak JavaScript. This also makes it much easier to figure out where to look if you'd like to write similar code or change one aspect while keeping the remaining functionality.

Several lines of this code (e.g., 10, 20, 53) also illustrate that JavaScript is often very tightly bound to the HTML of the page it operates on. Changing a single class name or displaying content inside a different element breaks many JavaScripts, as they can no longer find the content they were meant to operate on. On the other hand, if you can control the HTML of a page, or at least have high confidence it won't change, JavaScript gives you a great deal of power. Once you know where to find the information you need (using CSS selectors), you can hide, show, move, and reformat it on the fly. By writing a custom stylesheet and using JavaScript to add or remove classes from that stylesheet as needed, you can (re)define a page's layout, appearance, and usability.

Better yet, you can do this even if you're working with a product that doesn't let you edit the `<body>` of the HTML but does let you insert CSS and JavaScript into the `<head>`. If you read the HTML thoroughly, you can generally construct CSS selectors that uniquely identify parts of the page you'd like to change; you can then write JavaScript to target those parts. Using this technique, Matthew Reidsma and Kyle Felker entirely redesigned Grand Valley State University's 360Link implementation. This let them not only improve design but also address concerns that had arisen during usability testing. For this and other examples of improving user experience through JavaScript, explore the GVSU Libraries' and Matthew Reidsma's personal GitHub repositories.

Matthew Reidsma and Kyle Felker's 360Link Reset
https://github.com/gvsulib/360Link-Reset

Want to modify Darrington's program for use locally and practice your JavaScript (and jQuery and CSS) skills while you're at it? Here are some things you might try:

- **Lines 32–39** assume that any anchor in the URL actually corresponds to an element on the page; they don't defend against the possibility that a user has edited the URL. What happens if the URL has an invalid anchor? If the outcome is bad, can you check for validity before deciding whether to run the code?
- Change the appearance of the highlighting applied to table of contents entries. This actually isn't a JavaScript question at all; the styling comes from the CSS rules defined for the `currentNav` class (in a separate file). Merely changing the CSS, without touching the JavaScript, can give you very different results.
- Write something inspired by this script that works with LibGuides 2.0.

Notes

1. Eric S. Raymond, "The Cathedral and the Bazaar," *First Monday* 3, no. 3 (March 2, 1998), http://firstmonday.org/article/view/578/499; Eric S. Raymond, The Cathedral and the Bazaar website, February 18, 2010, www.catb.org/~esr/writings/cathedral-bazaar.
2. Bohyun Kim, "Playing with JavaScript and JQuery—The Ebook Link HTML String Generator and the EZproxy Bookmarklet Generator," *TechConnect Blog,* April 8, 2013, http://acrl.ala.org/techconnect/?p=3098.