2023

DRYing Our Library's LibGuides-Based Webpage by Introducing Vue.js

Mark E. Eaton
CUNY Kingsborough Community College

DRYing our library's LibGuides-based webpage by introducing Vue.js

By Mark E. Eaton
Reader Services Librarian / Associate Professor
Kingsborough Community College / City University of New York

Abstract

At the Kingsborough Community College library, we recently decided to bring the library's website more in line with DRY principles (Don't Repeat Yourself). We felt we could improve the site by creating more concise and maintainable code for our webpage by making it more DRY. DRYer code would be easier to read, understand and edit. We adopted the Vue.js framework in order to replace repetitive, hand-coded dropdown menus with programmatically generated markup. Using Vue allowed us to greatly simplify the HTML documents, while also improving maintainability.

Keeping it DRY

A common goal among programmers is to write code that is DRY, in other words, code where you don't repeat yourself. This is usually motivated by the insight that computers can often effectively automate repetitive tasks, making it unnecessary to repeat yourself in code. Taking advantage of the efficiencies of automation is widely regarded as a best practice among programmers.

However, HTML, when written by hand, is unfortunately not terribly suited to DRY practices. HTML is particularly declarative: all elements of the page are explicitly laid out by the programmer, so as to fully describe its structure. The problem with this is that it means that hand-written webpages are often not very DRY. Even those of a relatively modest amount of complexity can quickly grow into very long HTML documents.

This can be problematic, for a few reasons:

- It can become difficult to conceptualize the structure of a whole page when it stretches out over hundreds of lines.
- Even relatively trivial aspects of coding, such as indentation, can become difficult with the deeply nested HTML structures of a large page.

- It is easy to introduce syntax errors or formatting problems into long HTML documents, because typos can be easily overlooked. This is especially problematic in cases where there is no built-in linting or validation.[1]

At Our College

These challenges were familiar to us at Kingsborough Community College, a college of the City University of New York. Our homepage, built on LibGuides CMS, ran to over 500 lines, not including the <head> or <footer> sections. Much of this was owing to repetitive dropdown menus: our page relies heavily upon Bootstrap-based dropdown navigations to provide easy access to many of our services from the library homepage. These hand-coded menus, structured as lists of links, accounted for much of the length of the page's source code. Included below is the original code for our hamburger menu, which, despite its length, was in fact the shortest and simplest dropdown menu on our page:

```
<div class="dropdown" id="hamburger-container">
  <button class="btn btn-default dropdown-toggle" type="button" data-toggle="dropdown" aria-haspopup="true"
aria-expanded="true" id="hamburger">
    <i class="fas fa-bars" style="font-size: 2em;"></i>
  </button>
  <ul class="dropdown-menu fade" aria-labelledby="hamburger" id="hamburger-ul">
    <li>
      <a class="searchmenu" aria-label="OneSearch" href="https://library.kbcc.cuny.edu/onesearch">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-search fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>OneSearch</strong>
        </div>
      </a>
      <a class="searchmenu" aria-label="Databases A to Z" href="https://library.kbcc.cuny.edu/az.php">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-database fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>Databases A-Z</strong>
        </div>
      </a>
      <a class="searchmenu" aria-label="Research Guides" href="https://library.kbcc.cuny.edu/guides">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-telescope fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>Research Guides</strong>
        </div>
      </a>
      <a class="searchmenu" aria-label="FAQ" href="https://library.kbcc.cuny.edu/faq">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-question-circle fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>FAQ</strong>
        </div>
      </a>
      <a class="searchmenu" aria-label="Hours" href="https://library.kbcc.cuny.edu/calendar">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-clock fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>Library Hours</strong>
        </div>
      </a>
      <a class="searchmenu" aria-label="Site Map" href="https://library.kbcc.cuny.edu/sitemap">
        <div class="highlight-menu-item bigger-fancy-text">
          <i class="fas fa-location-arrow fa-fw bigger-icon" aria-hidden="true"></i>
          <strong>Site Map</strong>
        </div>
```

[1] The platform we use, LibGuides CMS, does not provide built-in linting or validation.

```
      </a>
    </li>
  </ul>
</div>
```

Abstracting away some of that repetition was, in some important ways, an obvious win for the maintainers of the library webpage. There were clear benefits to abstraction. Specifically, DRYing the page would:

- provide increased simplicity and maintainability;
- align us more with contemporary best practices in web development;
- allow us to write more aesthetically pleasing code;
- allow us to adopt and learn a modern JavaScript framework;
- raise the technical bar for what we are attempting to accomplish with our webpage.

In brief, it would make the site better, and make life easier for the maintainers.

These improvements were not undertaken without some hesitation. Our library has non-technical librarians who work with LibGuides daily, and who may also want to edit our webpage. We were worried that adding another layer of abstraction might be confusing to them, as they would no longer be able to "see" the full HTML Document Object Model (DOM) to their satisfaction, and therefore no longer be able to properly understand and manipulate it themselves. This was an important concern.

On the other hand, reducing the HTML devoted to dropdowns might in fact make other parts of the website more legible to our non-technical colleagues, because it would reduce the amount of noise that a non-expert user would need to filter through to accomplish their goals. In this sense, simplifying is also a way to improve access to the code.

We decided to proceed because we felt that, in sum, the benefits out-weighed the drawbacks. The tradeoff is that it will make the page more maintainable for some, while it is perhaps of mixed benefit to others. This project was the best way we could find to address these issues in a balanced way, while making sustained progress on the further development of the site.

Selecting and Using Vue.js

The tool we chose to do this work was Vue.js (referred to as Vue in the text that follows). Vue is what is referred to as a "progressive" JavaScript framework, in that it aims to scale up, as well as scale down. Scaling down was important to us, as our use case was not complicated, and we did not need the overhead of the complex build systems that are common to many JavaScript frameworks. We wanted something we could use within our CMS. Helpfully, it is possible to use Vue in this way. We were able to import Vue as a library with a simple call to a content delivery network (CDN), which allowed us to use it much in the same way that we would use other common libraries like Bootstrap or jQuery. We had access to many of Vue's abstractions by

simply including `<script src="https://cdn.jsdelivr.net/npm/vue@2.7.8">` in our page, without necessitating other complex overhead.

Vue provides a very useful templating system to build HTML programmatically. We were familiar with HTML templating from previous work that we had done with Python's Flask framework and its templating engine, Jinja. Jinja is somewhat conceptually similar to Vue's templating system, which helped us wrap our head around some parts of Vue. However, in our opinion, Vue provides added functionality beyond what is possible with Jinja, such as two-way binding and an even more broad-based control of the DOM.

Vue allowed us to write directives such as v-for, which is basically a for loop for constructing part of the DOM. Constructing a long list of links with a v-for loop is a huge improvement over typing out many lines of HTML. The content of the individual items, such as links, text, icons, and so on, can be stored as a JavaScript object in our Vue constructor. Vue's directives allow us to draw content from this object, while structuring the HTML with the templating syntax. This approach gives us the full power of JavaScript and Vue when building our HTML. It is truly much more powerful and accurate than patiently typing out HTML by hand.

Here is the HTML for the same hamburger menu as shown in the previous code snippet, but now DRYed with Vue:

```
<div class="dropdown" id="hamburger-container">
  <button class="btn btn-default dropdown-toggle" type="button" data-toggle="dropdown" aria-haspopup="true"
aria-expanded="true" aria-label="Hamburger Menu" id="hamburger">
    <i class="fas fa-bars" style="font-size: 2em;"></i>
  </button>
  <ul class="dropdown-menu fade" aria-labelledby="hamburger" id="hamburger-ul">
    <li>
      <a v-for="item in hamburger" v-bind:key="item.id" class="searchmenu" v-bind:aria-label="item.description"
v-bind:href="item.link" v-bind:target="item.target_blank">
        <menus-component v-bind:item="item"></menus-component>
      </a>
    </li>
  </ul>
</div>
```

The corresponding Vue component that makes this v-for loop happen looks like this:

```
const menusComponent = {
  Template:
  `<div class="bigger-fancy-text">
    <i v-bind:class="item.icon" aria-hidden="true"></i>
    <strong>{{ item.description }}</strong>
  </div>`,
    props: ['item']
}
```

And the Vue constructor:

```
new Vue({
  el: '#app',
  components: {
    'menus-component': menusComponent
    …
  },
  data() {
    return {
      hamburger: [
```

```
        {
          Link:
"https://cuny-kb.primo.exlibrisgroup.com/discovery/search?tab=Everything&vid=01CUNY_KB:CUNY_KB&lang=en",
          icon: "fas fa-search fa-fw bigger-icon",
          description: "OneSearch",
          target_blank: "",
          id: 1
        },
        …
```

We uploaded our Vue code as a single file to the "Upload Customization Files" section of the LibGuides CMS admin interface. While this was quite effective, nonetheless, there were a couple of notable downsides:

- The "Upload Customization Files" section of LibGuides CMS is a bit hidden away in the admin interface. This is not our preference, but it is a design decision by Springshare, the maker of LibGuides. The result of this is that someone new to the project, or new to LibGuides, might not immediately know where to look for the configuration files that are essential to rendering the page.
- It is very important that the uploaded JavaScript be valid, since formatting errors will mean that the data won't load when the page loads, which will cause major problems. Indeed, bad JavaScript often results in parts of the page not being rendered at all. The solution we adopted is to simply remember to validate our code before uploading it. We used babeljs (https://babeljs.io) for this purpose. Babeljs allows us to paste in our code – for example, our Vue constructor, including the data object – and it will flag any syntax errors. This is clearly not the most automated workflow, but it was simple enough to be an effective strategy for us.[2]

Assessing Our Approach

This project was not an entire re-write of our library webpage in the idiom of Vue.js. That was not our goal, and was far beyond the scope of this project. We simply took parts of the page that were easily DRY-able and used Vue to render them. For the most part, this process consisted of replacing the numerous hand-coded lists of links that produce our site's menus. We focused on these lists because they were the principal offenders that made our page source too long and unmanageable.

We tested this proposed approach ahead of time, by implementing Vue's v-for directive on the web librarian's personal projects page. This trial run worked surprisingly seamlessly, with basically no problems of any significance. This gave us the confidence to move ahead implementing Vue on the library homepage. And if there were major problems, we knew we

[2] Interestingly, this is a case where a more sophisticated build system would be an advantage, as linting and validation could be included as part of the build. This is something for us to consider in the future, if we decide to adopt Vue further.

could always use version control to roll back to a previous version, if needed (we use git and GitHub for version control).

Nonetheless, the transition was not entirely without problems. Our initial, most naive approach introduced new s to the page, which broke the existing CSS. We initially (and mistakenly) thought that these problems were on account of Vue, but they were in fact due to our CSS not behaving as expected on account of the new DOM structure. In hindsight, it should have been obvious to us that changing the page structure would break the CSS. The good news was that our Vue code was fine and was more or less working as expected. Tweaking the HTML created by Vue – so as not to break our CSS – was entirely doable. We solved the problem by configuring our v-for loops so as to faithfully recreate the original DOM structure, which allowed the CSS to work properly and as expected. In this way, the project was completed without needing to rewrite any of our CSS.

Moreover, we found that besides iterating through individual menus using Vue, we could DRY the next level of our webpage's hierarchy by having Vue iterate through the list of menus (in other words, through the entire nav bar), so as to create another layer of abstraction, automation, and benefit to the maintainers. While there appears to be more than one way to tackle this problem, we settled upon creating a second Vue component to handle the second-order DRYing logic. This higher-order abstraction is new to us, and we expect to make it better and more efficient over the coming weeks and months.

These limited goals and constrained scope for this project meant that the changes that we attempted were not too overwhelming to implement. We rolled out our improvements within a few weeks, without disrupting other, existing workflows. We did this work in our "sandbox" LibGuides group, before moving the code over to our production group. We did this to avoid breaking, even temporarily, the production homepage.

One especially pleasing part of this project was that, from the perspective of our users, there was no change at all to our website. From a user's point of view, the site remained identical. We were able to increase the maintainability of the site while causing no end-user effects, which was a big win.

To look at these changes quantitatively, our homepage was initially 501 lines of HTML, and after applying our DRYing techniques, was 279 lines. This is a difference of 44%. While our goal is not to play code golf, we felt that this alone made this project a worthwhile endeavor. Of course this appealing headline number is somewhat offset by the added cognitive load (and lines of code) of the Vue components and constructor, but nonetheless it is fair to say that the page is more concise. The new logic builds much of the DOM automatically, with less room for human error.

Conclusion

The result of our work is a more manageable and concise HTML document, which provides efficiencies in maintainability. Adopting Vue for our limited use case turned out to be a good decision. We hope to find compelling reasons to go further with Vue's more advanced features in the future. For example, as a next step, we intend to move more of the page's logic to Vue methods and computed properties.

One possible outcome of this project is that it may eventually lead to a full rewrite of the webpage which more fully embraces the Vue idiom. The chance to adopt Vue as the principal organizing framework for our page offers many exciting new possibilities beyond our current Bootstrap-oriented setup. Vue could expand the functionality available to us, and truly move the needle when it eventually comes time to fully redesign our webpage. DRYing our project is a small first step in that direction.