Improving Forms Performance

Author: Maya Nigrin

Claps: 260

Date: Nov 20, 2021

When Klaviyo customers decide to use our signup forms, they expect forms that load quickly and donâ€™t slow down their site, ideally only improving their usersâ€™ experience. Improving forms performance means minimizing the amount of JavaScript loaded on our customers sites (also known as bundle size), and decreasing main thread blocking time (also known as total blocking time) so that Klaviyo signup forms slow down our customerâ€™s sites as little as possible. Performance can also impact a siteâ€™s SEO score, which affects search engine optimization, ranking, and organic traffic. This past quarter, the rest of the Onsite Team and I worked on several different refactors to reduce both blocking time and bundle size on our customers sites and improve forms performance as much as possible. In this article, Iâ€™ll talk through the refactors we did to improve performance, as well as the results of those changes.

Performance Improvement Goals

There are many different strategies one can use to improve onsite content performance, but we mainly focused on three goals:

1. Load as little as possible before signup forms are rendered so that sites that are not currently displaying any forms spend less time downloading JavaScript and blocking the main thread
2. Rip out large or computationally expensive packages whenever possible, either by using a microservice to move the work to the server-side, or by replacing them with lighter-weight packages
3. Strategically chunk & asynchronously import modules when applicable so that our code only imports larger packages and code when necessary, while still trying to minimize the number of chunks so that we donâ€™t slow things down by needing to stop and download JavaScript too frequently

Splitting out triggering

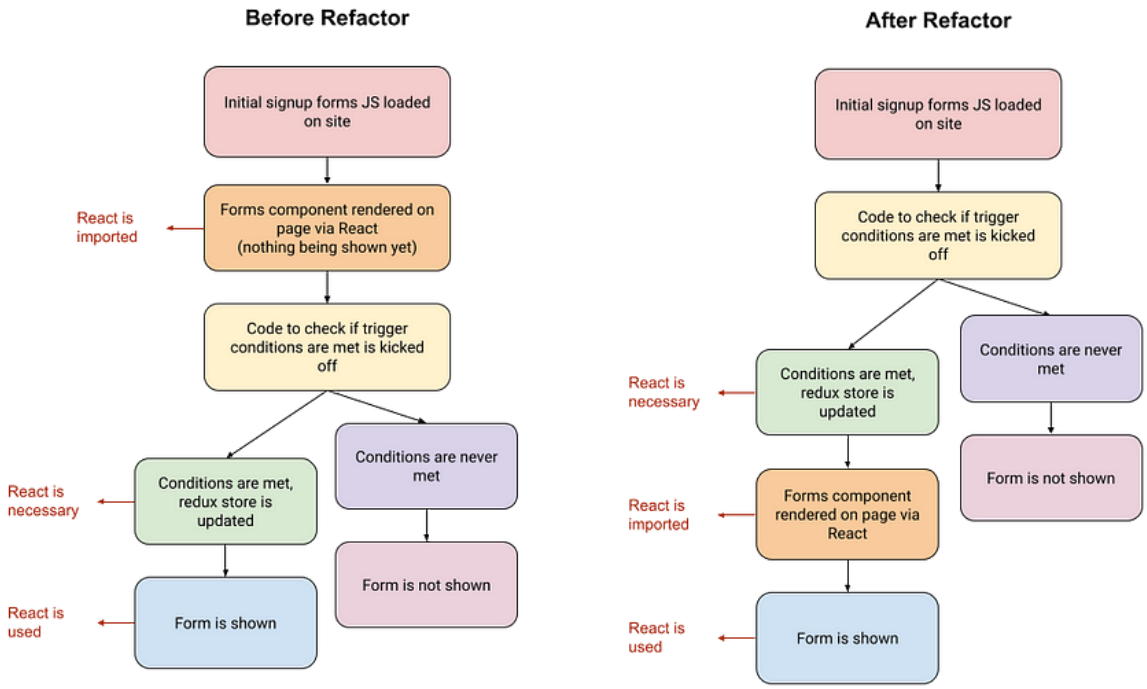
The first refactor that we opted to do to improve forms performance was to separate out the forms triggering logic from the rendering logic. Initially, all of the logic to detect when a form has been triggered and should show lived inside of a single React component. This meant that our initial JavaScript bundle included React, which is a relatively large package. Although that code did need to interact with the Redux store, it didnâ€™t actually render anything, so thereâ€™s no need for it to live inside of a React component.

To take advantage of this, I moved the logic that checks for trigger conditions to be satisfied into its own function in its own file, and then invoked that function directly in our initial signup forms JavaScript. Inside of our trigger-checking logic, once all necessary conditions have been satisfied, a Redux action is fired to update our store and display the form. Before, that action didnâ€™t have to worry about rendering at all because the component already existed on the page, but now that the triggering logic is directly triggered by the initial signup forms JavaScript instead of

being triggered on mount of the React component, thatâ€™s no longer the case. To remedy that, I wrote a function called *renderForms* that injects a div into the page body and renders our signup forms inside of that element using React. The *renderForms* function was then asynchronously imported inside of the aforementioned action, like so:

```
const { default: renderForms } = await import(    /* webpackChunkName: â€œ
```

This means that we hold off importing React and rendering any components until we know the form actually needs to be shown; consequently, React and the rest of the rendering code is not downloaded until we know itâ€™s actually necessary. You can see this more clearly in the following diagram:

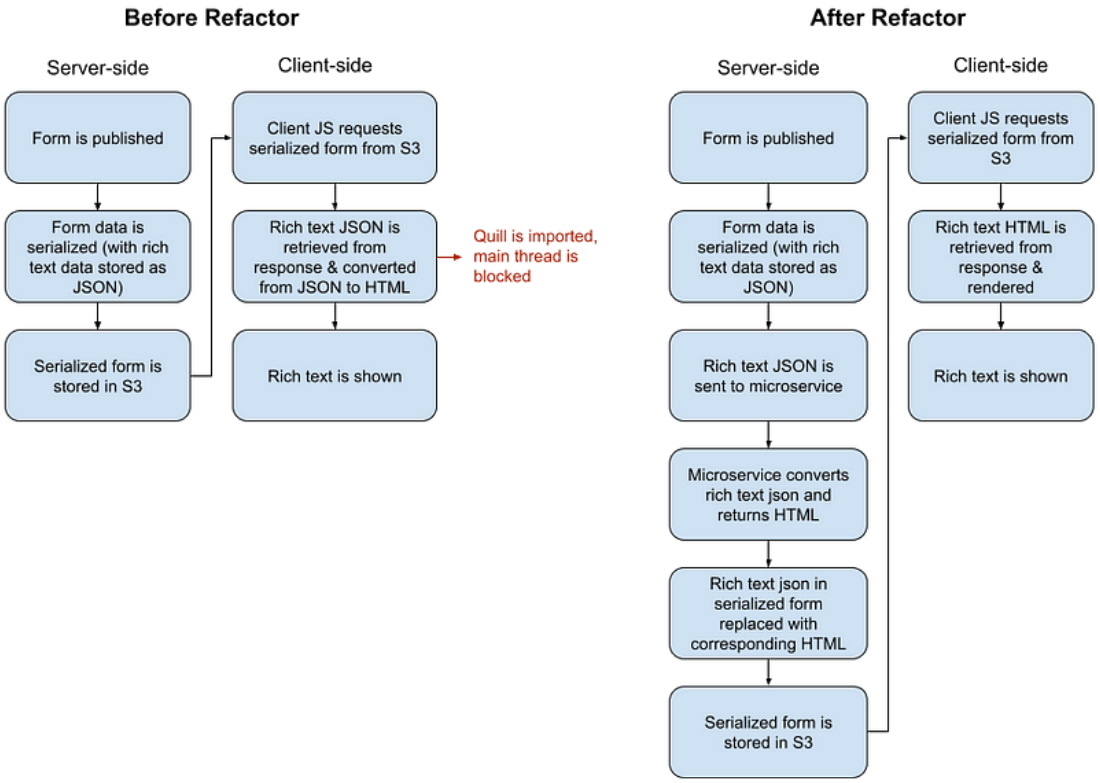


Ultimately, this change brought a ~10% decrease in initial bundle size!

Server-side rendering Quill

Another major refactor that we tackled was rendering Rich Text components in a microservice instead of on the client-side. We use Quill, a rich text editor & rendering package, for all of our rich text needs in Klaviyo forms. We use this library primarily because it allows us to store the rich text contents and styles as a serializable JSON. However, in order to convert this JSON into HTML, we had to import Quill in our client-side JavaScript. Since Quill is another relatively large package, this again contributed to our JavaScript bundle size. It also increased the main thread blocking time because it needed to convert the JSON to HTML before it could show the text. However, we realized that if we could convert the rich text JSON to HTML on the server-side instead of the client-side, we could avoid both importing Quill and blocking the main thread to render rich text.

We decided to accomplish this by creating a microservice that, given a rich text JSON, would render it using JSDOM and return the corresponding HTML. Then we could hit the service while serializing & storing the form data on publish, and the client-side would only have to insert the HTML instead of having to do computation to render the JSON.



You can see in the chart above that this does increase the work done on the server-side, which means that it takes slightly longer to go from hitting publish in the forms editor to seeing the updated form on your site; however, considering that even with these changes that process still takes at most a couple of minutes, we considered it a worthwhile tradeoff.

We decided to host the Quill-rendering microservice in Kubernetes and use an internal ELB to route requests to the service to pods within the cluster. Each pod in the Kubernetes cluster has a server and a workerpool. When the server receives a request, it instructs one of its workers to make a fresh JSDOM. It then gets the global context from that DOM, adds the request body to that context, and runs the quill rendering script in that context. It then sends back the results of that script as the server response. Using Nodeâ€™s VM API to scope the mutations to JSDOMâ€™s global apis helps prevent us from leaking document changes across network requests.

You can see the use of JSDOM and Nodeâ€™s VM API in action here, in the function that we call to kick off the rendering script:

```
const getRenderedHTMLForDelta = (deltas) => {    // Make a fresh dom    co
```

In the end, server-side rendering Quill helped reduce our main thread blocking time and decreased the amount of JavaScript we send to our customersâ€™ sites by about 15%. It also helped us learn how to build, tune, monitor, and deploy microservices in Kubernetes, which we

expect to be helpful both for future potential performance improvements, and also for other uses like performance and accessibility testing.

Remove styled-components from forms

After removing Quill and delaying the import of React, one of the largest remaining contributors to bundle size and main thread blocking time was styled-components. Styled-components let us write CSS in JS which makes it easier to use all the features of CSS while weâ€™re writing React code. However, we found that while those benefits might be useful when performance matters less, theyâ€™re less valuable when youâ€™re trying to keep your code as lightweight as possible.

Through some investigation, we found inline styles to be the fastest way of rendering dynamic styles. One limitation of this method is that pseudo elements cannot be inline, so we ended up using Goober, a styled-components replacement which is much smaller and faster. One other limitation of this was that we had to pass down the theme through component props instead of using the theme provider that comes with styled-components. We also ended up moving all non dynamic styles to a css file.

There are definitely some drawbacks to this change:

- We donâ€™t get any of the benefit Styled Components gave out of the box (like multiple nested theme providers and CSS prefixing)
- Styles per component are not re-used (e.g. inline styles are repeated)
- Pseudo elements have to be handled via CSS in JS

However, in the end, we think the benefits of this change outweigh those downsides:

- Bundle size was reduced by 8% (g-zipped)
- Main thread blocking time for rendering was reduced by 26% (according to our testing)
- We ultimately ended up with better typing for styles (thanks to React inline styles)

Building modern assets for onsite modules

Another refactor that helped reduce our bundle size was building modern assets for our onsite modules. Essentially, we build two different versions of our code: one for browsers that support ES Modules, and another for browsers that donâ€™t. We were able to do this thanks to webpack and the target.esmodules babel setting. Then in our initially loaded JS file, we determine whether or not the browser supports ES Modules using a JS script. This allows modern browsers to utilize built in JavaScript functions instead of having to ship extra code to support them. Ultimately, this reduced our bundle size by around 10% for customers that use browsers that support ES Modules, which is about 93% of users.

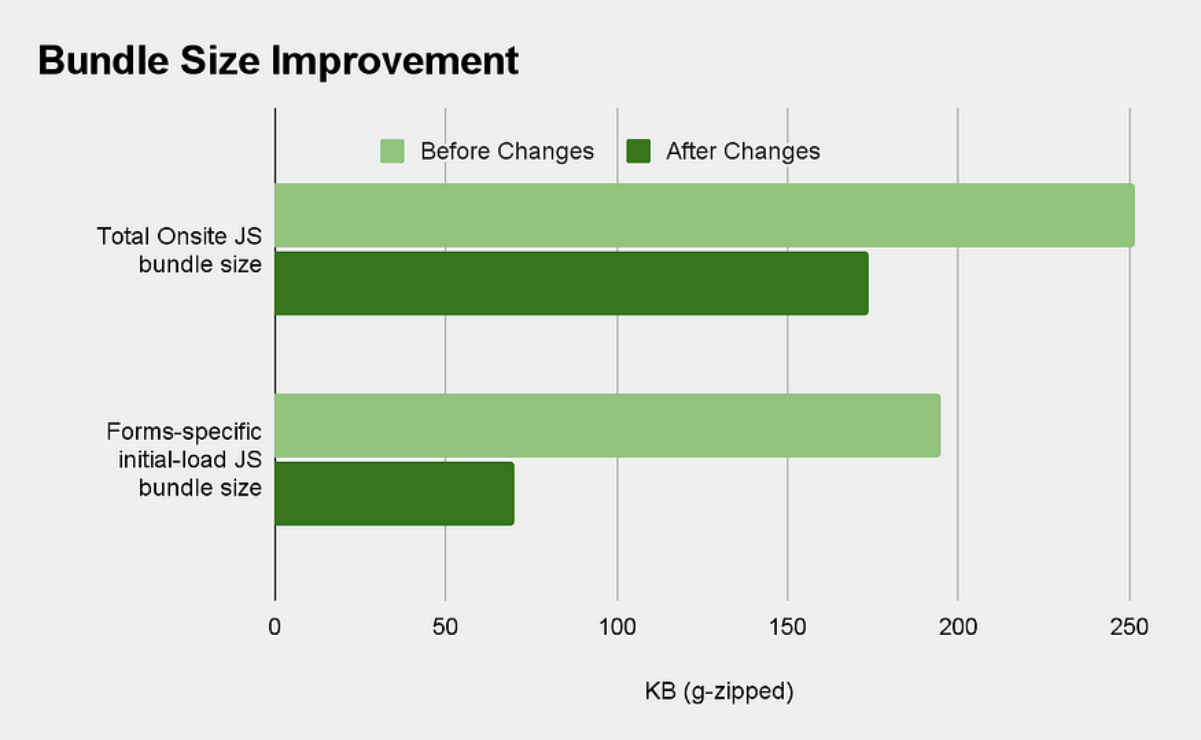
Other chunking optimizations

In addition to all of the larger refactors mentioned above, we also made some smaller changes to the code that individually didnâ€™t change much, but collectively helped decrease both the total and initial bundle size. Most of those changes acted on the same principle as splitting out the triggering and rendering code: instead of importing everything all at once, asynchronously import larger or more expensive code only once you know for sure that you need it. For example, we donâ€™t need to use phone number validation in every form, and the package we use to do it is

surprisingly large. To resolve this, we decided to asynchronously import the package inside of our phone number component, which means now weâ€™ll only download it when we know we need to use it. This pattern is relatively easy to implement (whether youâ€™re refactoring or writing code from scratch) and if you have large packages that you donâ€™t need all of the time, it can really help make your code more lightweight when possible. Keep in mind though that the more you chunk your JavaScript, the more times your code will have to stop & download content, so itâ€™s best not to overuse it but rather to find the largest contributors to your bundle size and investigate whether they are conditionally used; itâ€™s those cases that are the best candidates for asynchronous importing.

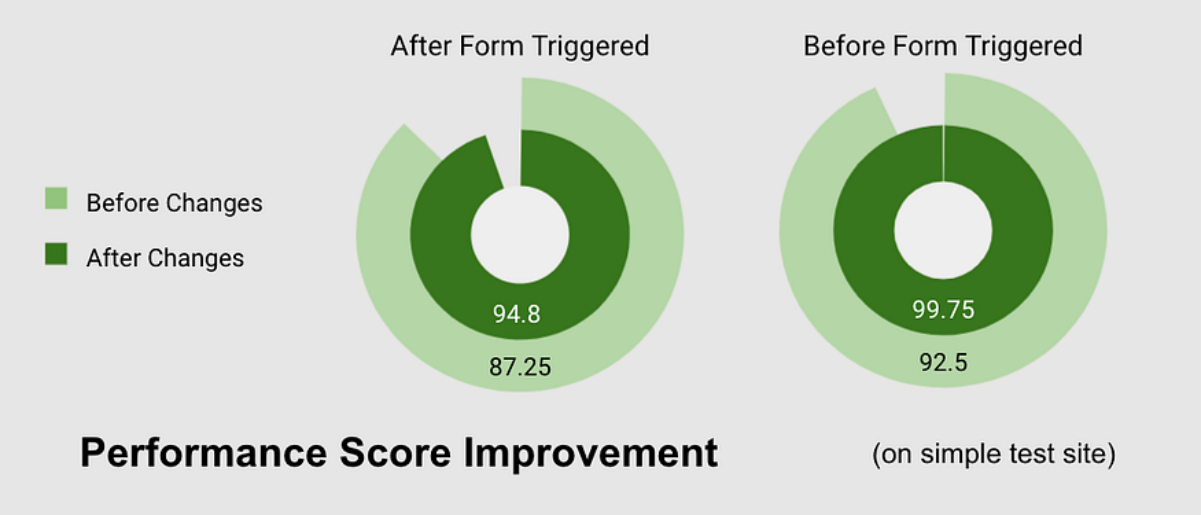
Results

The best part of working on performance improvements is that you get to exactly quantify the effects of your changes! Here are some of the combined results of the refactors listed above (not including the styled-system removal, since that was implemented after these calculations):



- **The total onsite JS bundle size was decreased by ~31%.** This means that the total amount of code that we ship to customers sites has decreased by 31%, which should improve page speed whether or not theyâ€™re using forms (assuming they use all potential forms features, e.g. phone numbers, date inputs, etc.)
- **The Forms-specific initial-load JS bundle size was decreased by ~64%.** This means that the amount of code that we initially download (i.e. on load of the page, before any forms are triggered) to customers sites for signup forms has decreased by 64%, which should notably improve page speed in situations where the site has forms but they havenâ€™t been triggered yet
- **The average main thread blocking time for a page where the form will show decreased by -240.35 ms (~51% decrease)**
- **The average main thread blocking time for a page where the form will not show decreased by -226.5 ms (~75% decrease)**

However, even though these metrics can help us understand the magnitude of the performance improvements, what most of our customers care about isnâ€™t necessarily bundle size and blocking time but rather the performance score of their page. While theyâ€™re definitely correlated, itâ€™s not always linearly so. Consequently, I used Lighthouse on a simple test site to get some idea of how these changes may impact performance scores:



- **The average performance score of a simple test site with a default form that loads immediately increased from 87.25 to 94.8.** Thatâ€™s an increase of +7.55 pts (out of 12.75 possible pts, aka we achieved 60% of all possible performance improvement according to lighthouse)
- **The average performance score of a simple test site with a default form that has not been triggered increased from 92.5 to 99.75.** Thatâ€™s an increase of +7.25 pts (out of 7.5 possible pts, aka we achieved 97% of all possible performance improvement in this case according to lighthouse)

This **does not mean** that our customers will see the exact same amount of improvement on their page speed score, but it does mean that they will see improvement in performance whether or not a form is triggered. In fact, we checked some Klaviyo customer sites and we have already moved down several ranks on the list of which 3rd party code owners contribute the most to slowness. We plan to continue making improvements to Klaviyo forms far into the future so stay tuned! Some potential future performance improvements include ideas like removing Redux from the triggering code and lazy-loading Sentry after an error has occurred.

I hope that this explanation of how we managed to improve our onsite code performance helps give you ideas for how to potentially improve your own frontend code performance. I learned a lot from talking through and implementing these changes and am glad Iâ€™ve gotten the chance to share some of that knowledge with you.