Shadow DOM

Author: Maya Nigrin

Claps: 60

Date: Sep 22

What it is, why we explored it, and why we chose not to use it.

The problem

One of the perpetual challenges of maintaining and developing Klaviyo sign-up forms is the fact that our code runs in an environment that we have little to no control over. The code is run on our customersâ€™ sites, which may contain CSS or Javascript that conflicts with ours.

Around 3% of all support tickets and 1â€“2 of the tickets escalated weekly to my team at Klaviyo are due to custom CSS overriding our form CSS. This type of issue is hard for our support team to identify, but not a real bug because itâ€™s not something we support; we generally close all of these tickets as expected behavior.

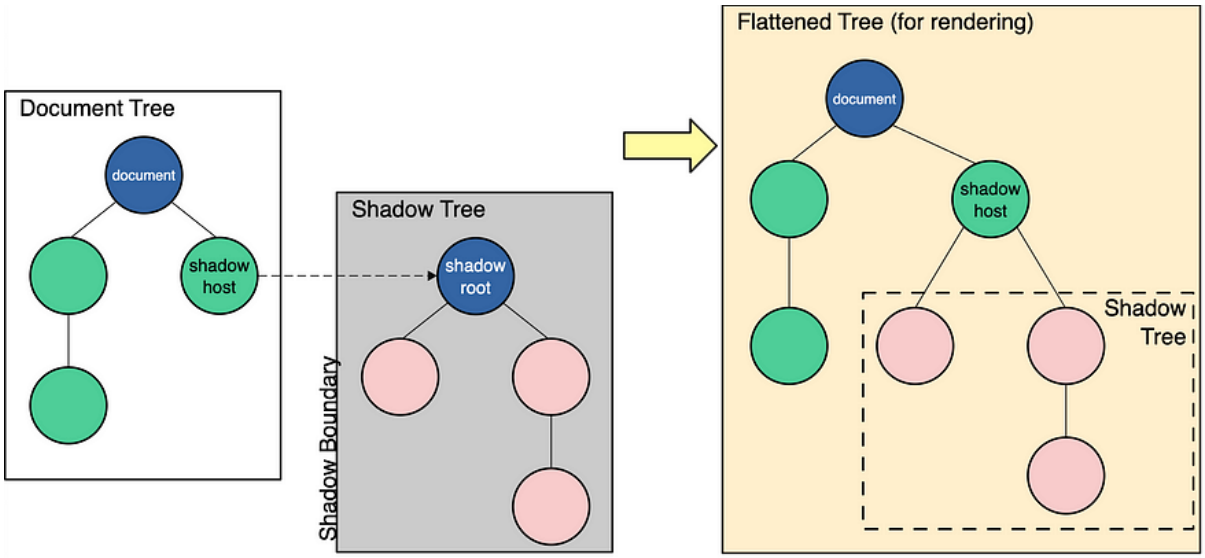
Because of the frequency of this issue, my team decided to investigate ways to encapsulate our code so that customersâ€™ code wouldnâ€™t affect ours and vice versa. It was that investigation that led us to look into the shadow DOM.

What is the shadow DOM?

If you work with HTML and CSS, you may have interacted with the shadow DOM without even realizing it. Itâ€™s a fundamental part of web components and itâ€™s a valuable tool when it comes to building encapsulation into web applications â€” keeping the markup structure, style, and behavior of different components hidden and separate from other code on the page so that different parts do not clash.

The second part of the name *shadow DOM* is something that might already be familiar to you: the DOM (or Document Object Model) is the tree-like structure of different HTML elements and text that defines the content of web page. Shadow DOM expands upon this structure, allowing hidden DOM trees to be attached to elements in the regular DOM tree.

There are four main terms to know when discussing the shadow DOM. The first is the **shadow host**, which is the regular DOM node that the shadow DOM is attached to. The second is the **shadow tree**, which is the hidden DOM tree inside of the shadow DOM. The **shadow root** is the root node of that hidden tree, and the **shadow boundary** is the place where the hidden tree ends and the regular DOM begins.



https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM

One key thing to understand is that when we say that the DOM tree is hidden, we mean that its contents are inaccessible via the standard methods of accessing the DOM, *not* invisible on the page. For example, if I have an element with id `"my-element"` outside of the shadow DOM, I can easily access it via `document.getElementById(â€œmy-elementâ€)`. However, if that same element lives inside of the shadow DOM, I would have to run `getElementById(â€œmy-elementâ€)` on the shadow tree instead of on `document` in order to find the element.

In terms of placement on the page, the contents of the shadow DOM are displayed within the shadow host. I can place the shadow DOM wherever Iâ€™d like on the page by manipulating the shadow host elementâ€™s styles. If youâ€™ve ever used an `iframe`, it works much the same way. For example, if I give the shadow host `position: absolute;` I can place it (and its contents) wherever I want on the page using the `top`, `left`, `right`, or `bottom` properties. I could also give it `display: block;` and `position: relative;` and have it positioned relative to the rest of my pageâ€™s elements.

You may ask â€” what is the point of having a hidden DOM tree when I could just use regular DOM elements? The key benefit of using the shadow DOM is that none of the Javascript or CSS inside it can access anything outside it. This can be incredibly useful when designing web components or other web code that will run on pages you have no control over, as it would be upsetting for users of your code to discover that your styling or manipulation of the DOM is unwittingly changing other parts of their page. Using a shadow DOM to encapsulate your code can also make it more difficult for code outside of it to unintentionally modify your styling or DOM structure.

Basics of using the shadow DOM

To attach a shadow root to an HTML element, you use the `Element.attachShadow()` method. It takes an options object with one parameter, *mode*, which you can either set to open or closed. An *open* shadow DOM is accessible using JavaScript in the main page context, whereas a *closed* one is not.

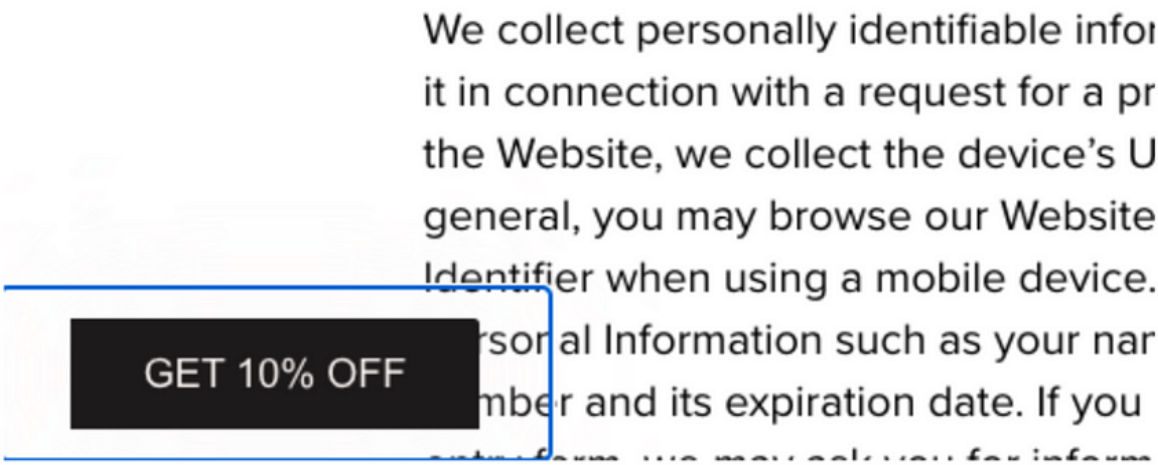
```
// Let's say openShadowHost and closedShadowHost are both regular
// HTML elements on your page
```

```
const openShadow = openShadowHost.attachShadow({ mode: â€œopenâ€ });
const closedShadow = closedShadowHost.attachShadow({ mode: "closed" });

// Since openShadowHost is the shadow host of an open shadow DOM,
// I can access the hidden tree by using the shadowRoot property,
// and then interact with it just like I would any other element of the DO
const newElement = document.createElement(â€œdivâ€)
openShadowHost.shadowRoot.appendChild(newElement)

// You cannot do the same with a closed shadow root,
// as trying to access the shadow root give returns null
console.log(closedShadowHost.shadowRoot) // null
```

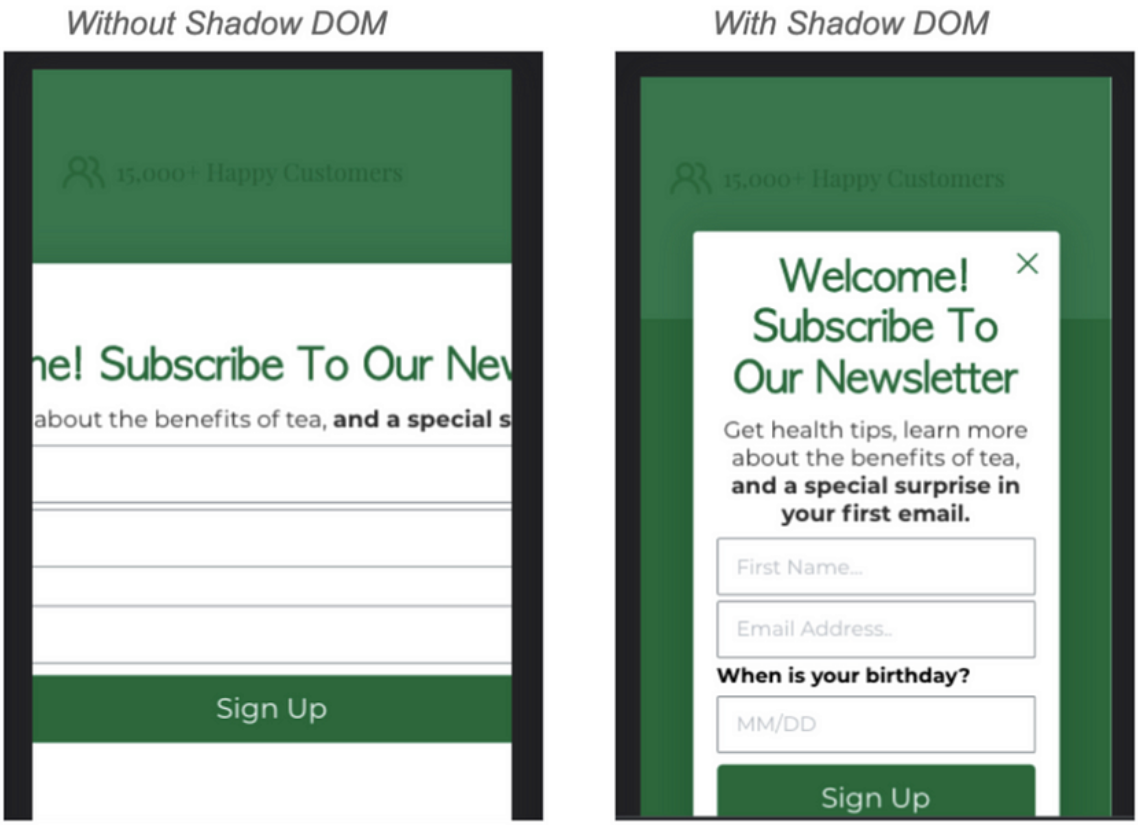
Because trying to access the shadow root from the shadow host yields `null` when the shadow DOM is closed, the only way to get a reference to a closed shadow DOM is to hold onto it when it was created (or set your own property on the element that contains a reference to the shadow DOM). For that reason, closed shadow DOMs tend to be used for set-it-and-forget applications â€” where you donâ€™t have any need to manipulate the elements inside of the hidden DOM tree once youâ€™ve created them.

Why shadow DOM for Klaviyo Forms?

As mentioned above, shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree, and external CSS wonâ€™t affect elements within the hidden DOM tree. Consequently, we thought that if we encapsulated the HTML for each form inside of a shadow DOM, then overriding CSS wouldnâ€™t affect our forms at all, thereby eliminating all problems with customer CSS overriding our form CSS.



Example of an overriding CSS issue: The customerâ€™s site had custom code adding a blue outline to their buttons and it unintentionally affected their Klaviyo teaser form.



> Another example of an overriding CSS issue: The customerâ€™s site had custom code modifying the width of dialogs and it make their form larger than the screen width.

We implemented a proof of concept to show that shadow DOM would protect form styles as expected, and to confirm that all other form-related javascript could run as usual.

Once we knew that shadow DOM would successfully protect our CSS from being overridden (and prevent our code from unintentionally changing anything else on the page), we planned work to fully implement the feature.

Integrating shadow DOM with Klaviyo Forms

There were several main points of consideration when integrating the shadow DOM with our on-site forms.

1. We only want to use the shadow DOM when the form is being shown on the customer site.

Klaviyo Forms code is used in two different places: on our customersâ€™ sites, and in our own app in the Forms Designer (the drag and drop editor where users actually create their form content). In the latter case, we actively *do not want* the forms code to be encapsulated because 1) we want easy access to element styles so that they can dynamically update as the customer interacts with different settings, and 2) the drag and drop libraries that we use within the designer need access to the DOM in order to restructure the order of elements when the user is dragging them around. These libraries are not built to accommodate the shadow DOM.

This meant that we had to modify all places where we accessed `document` or used `goober.css` to call a utility function instead that would return the right reference depending on whether we were in the designer or not. To implement the functions (one for `document` access and the other for CSS access), I used a module-scoped dictionary to store references to each formâ€™s shadow root. The function would then take in an identifier for the form and, if we were not in the designer, return the associated shadow root; otherwise, it would return `document` or `goober.css` respectively. Once those functions were written, all that remained was the relatively straightforward task of finding each place where we used `document` or `goober.css` and replacing the usage with a call to the appropriate utility function.

```
import { getShadowRootByFormVersionCId } from â€˜/constsâ€™;

// Use module scope to store root document for each form
const environments = {};

// Example function that I used to grab the correct document reference
// depdending on the form and designer status
const getRootDocument = ({ formVersionCId, isDesignWorkflow = false, }) =>
  if (isDesignWorkflow) {
    return document;
  }
  if (!environments[formVersionCId]) {
    const target = getShadowRootByFormVersionCId( formVersionCId, )
    if (!target) {
      return document;
    }
    environments[formVersionCId] = target;
  }
  return environments[formVersionCId];
};
```

2. Because of embeds, each form needs to be encapsulated in its own shadowDOM.

Because each shadow DOM is associated with only one shadow host, itâ€™s not possible to have one shadow DOM exist in multiple places in the original DOM. Since our customers may want to have more than one embedded form on their site, we needed to use a different shadow DOM for each form. This is why the aforementioned utility functions need to not only know whether weâ€™re in the designer or not, but also which form weâ€™re currently trying to interact with.

This also became important when adapting our Custom Font feature for use with shadow DOM. The font definitions (e.g. `@font-face`) needed to exist in the head of the original document, but all of the other css needed to be inserted into each shadow DOM separately.

3. Since we use webpack to insert our CSS, we had to make sure that that process could still inject styles at the right place in the right time.

In order to get webpackâ€™s CSS insertion to work with shadow DOM (e.g. insert a copy of the `<link>` inside of each shadow DOM instead of just injecting one in the `<head>`) I had to upgrade mini-css-extract-plugin from version 0.9.0 to 1.6.2.

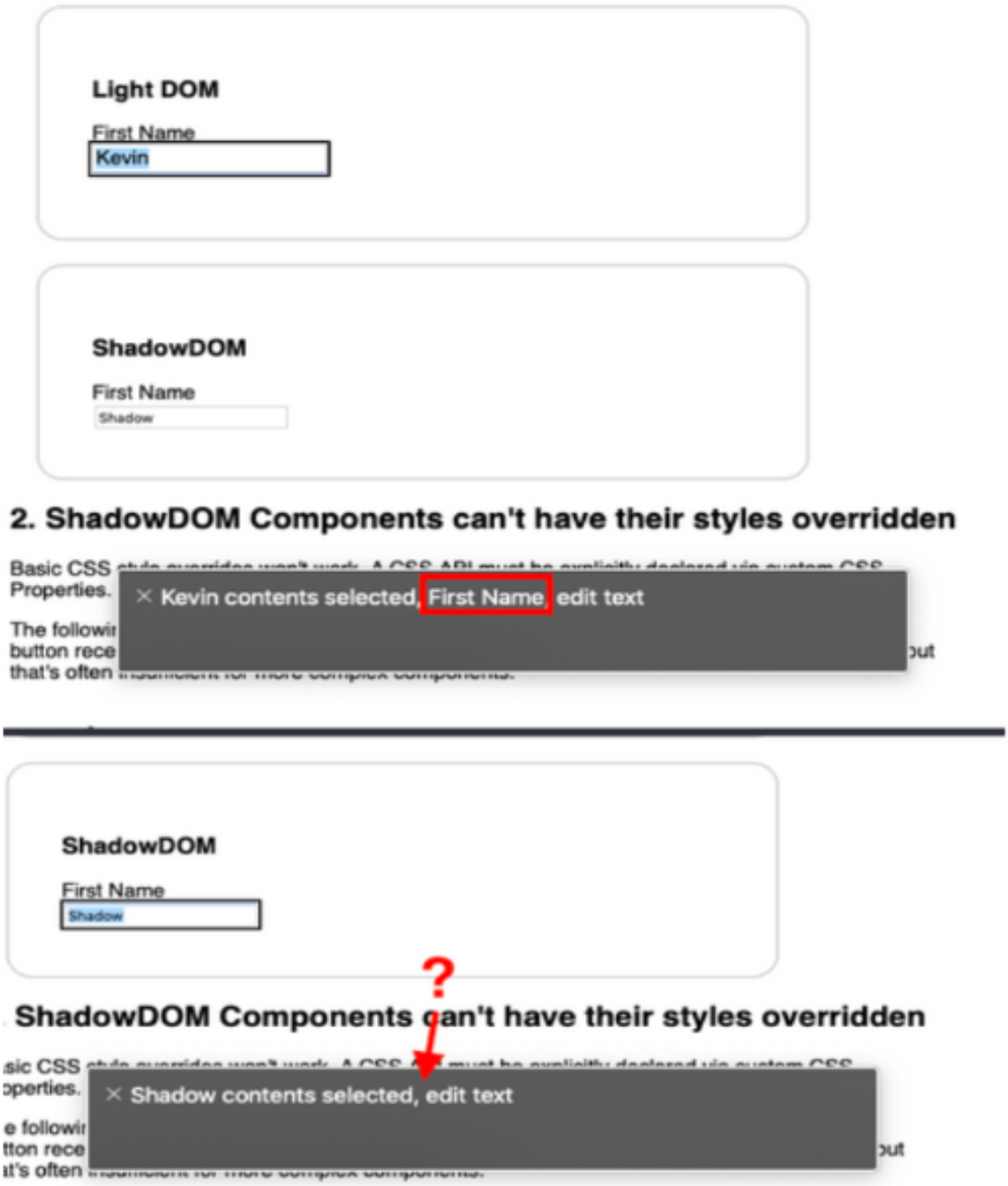
I also had to modify how and when we import the `.scss` files in order for the webpack CSS insertion to happen at the right time. Otherwise, the bundler would not be able to find the necessary files.

Those were the main hurdles and considerations for integrating shadow DOM into forms. Other than that, we didnâ€™t need to make many changes to our code, since as long as you can access the shadow root, you can manipulate the shadow DOM and its elements just like you would the regular DOM.

Deciding not to release shadow DOM in Forms

Once shadow DOM was implemented, we took time to manually test it. As part of that process, we discovered that Chrome extension screen readers and ADA compliance checkers werenâ€™t working properly. They both seemed to be unable to detect the contents of the shadow DOM.

After further investigation, we realized this was because Chromium does not (and will not, apparently) allow access into the shadow DOM to extensions. So all extension-based screen readers and accessibility assessors in Chromium-based web browsers (e.g. Chrome, the newest version of Edge, as well as a couple of smaller browsers we donâ€™t officially support) wouldnâ€™t be able to find form contents. (For some reason, the Google Translate extension seems to be unaffected). However, desktop-based screen readers like Appleâ€™s VoiceOver seem to be able traverse the form fine, and according to research done by a colleague on our Frontend Platform Team, the most popular screen reader (JAWS, which the government uses as their officially supported screen reader) can also traverse the shadow DOM without issue.



Example of an extension-based screen reader being unable to read the contents of the shadow DOM. From https://codepen.io/kevinmpowell/pen/JQwOyE

At the same time, from our product management team, concern was expressed about another (albeit previously known) effect of shadow DOM: Any customers that had custom CSS overriding form stylings would suddenly see their form change upon release. Some of our larger customers that have designers and developers rely upon the ability to modify their forms (even though we donâ€™t technically support that), and consequently there was some hesitation about whether or not this feature would actually end up helping customers; there was a good chance that we would just be replacing support tickets about â€œWhy doesnâ€™t my form look like the one in the designer?â€ with â€œWhy doesnâ€™t my custom CSS work?â€

Between these two concerns, we ultimately ended up deciding that releasing shadow DOM did not make sense.

Takeaways

While the shadow DOM is integral to web components and is appealing for developers who have to worry about conflicts between their code and the code of the page on which it will live, the accessibility concerns are a real drawback. The research we did into the accessibility of the shadow DOM said that it was accessible, but those articles did not take browser extensions into account. Many users rely on browser extensions for accessibility, both for interacting with and assessing web sites and applications. Until browsers end up opening access to the shadow DOM to extensions, it will be a tough sell to argue for using the shadow DOM for encapsulation more widely.

Regardless, I think the research into the feature was worth it, and as browsers evolve, I hope that one day we will be able to incorporate the shadow DOM into our code.