CSS Scoping Requirements

Author: yuzhehan@

What's All This About?

Slides: CSS style scoping for design systems & fw components - by nsull@

Styling is essential for web applications. Therefore, ensuring that styles are easily managed and applied is critical to the success of a web application. CSS scoping is a method of targeting a set of CSS rules to only a section of the DOM. Currently, CSS is globally scoped, which makes it challenging to determine which styles are applied to a given element precisely. This challenge is magnified as a web application grows. This document discusses CSS scoping, its history, current technologies, and required features. We hope to make strides to improve the ease with which designers and developers can scope CSS, further enhancing the performance, maintainability, and the interoperability of web applications.

CSS scoping aims to address the following areas:

- Style rules scoped to a connected fragment of DOM (could be a subtree, or a subtree up specified descendants)..
- CSS scoping works for light DOM and shadow DOM.
- Global styles can override scoped CSS based on specificity, allowing hosting applications to style components created by third parties.
- Eliminate the need for preprocessing tools to generate namespaced selector names in order to prevent selector naming collisions.
- Improve the interoperability of components from different design systems via scoping their styles to themselves.
- Enable developers to encapsulate component styles only to the component, preventing styles from breaking out (toward the document root) and bleeding in (toward nested components)

Target Audience for this Feature:

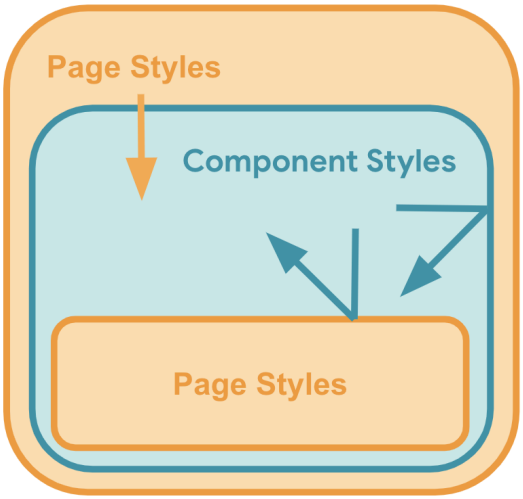
The target audience for this feature is HTML and CSS developers. Knowledge of javascript, frameworks, css preprocessors aren't required for utilizing this feature. With that said, we expect Scoped CSS still be leveraged by frameworks and css preprocessors.

Example use case:

```
Page.html
<style>          // Page styles apply to A, B, C and D

Component tree:  // Each node is a component, ex: custom-element or
                 // shadow host.
       A
     /   \
    B           // Styles scoped to B don't affect upwards (A) or
  /   \         // downwards (C, D)
 C     D
```

Blue box represents one component from the component tree above.



Current Issues:

The main issue with CSS is its global scope. All selectors reside in a global namespace. As such, it gets increasingly difficult to determine which styles get applied to an element as the project grows. It's common for the styling of one page to regress the appearance of another page unintentionally. Also, modifying and refactoring of styles becomes problematic as it requires manually checking all pages to make sure that the UI hasn't regressed.

A consequence of the CSS global scope is that it makes it difficult to provide interoperability of components from different design systems. Adding components from various component libraries requires importing their CSS. The web app runs the risk of style collision every time it imports a new CSS style sheet.

Lastly, the fear of unintended global style pollution can lead to the creation of non-performant CSS selectors. For example, nav.header .menu ul > li button, would be a non-performant (because of the complexity of the selector) choice to style a specific nav menu button while preventing its style from leaking to other buttons.

Scoping History:

Globally scoped CSS and its problems have plagued web developers for years. Back in 2012, an attempt to address these issues with the introduction of scoped attribute on the style element. The solution had some drawbacks. For one, it requires modification of HTML to inject the style tags. Later on, due to the non-uniform support from other browsers, this feature was removed [1] [2].

However, the removal caused many developers to raise the issue of the need for scoping CSS. The community created additional proposals and issues after scoped removal. These include CSS namespacing, CSS nesting, and bring back CSS Scoping.

Current Solutions:

The following are some of the popular solutions for addressing CSS scoping.

Manually: CSS rules are localized by adopting a consistent CSS selector naming scheme. Block, Element, and Modifier (BEM) is a common technique used. However, the management of the naming convention can be a challenge as the project grows. Also, selector names can get very long, and bloat both stylesheet and HTML.

```
Ex: <button class="x-button-primary">Save</button>
```

As a project grows, invariably, the demand for styling increases. The class selector for the same button placed inside a page content header menu becomes `x-header-nav-button_primary`.

```
<div class="x-detail-page">
  <div class="x-header">
    <button class="x-page-header-button_primary">Save</button>
  </div>
</div>
```

Additional CSS organizational methods exist, SMACSS and OOCSS. However, as with BEM, the management of the CSS class to ensure consistent naming gets increasingly burdensome as the web application grows.

CSS preprocessor: SASS, LESS, and other tools are commonly used to reduce the manual burden of managing CSS. They add a compilation step that can automate the scoping of CSS by defining scoped variables and attaching styles to those variables.

header.scss

```
$container: ".page";
$component: ".header";
#{$container} {
  #{$component} {
    &-button {
      &_primary {
         color: red;
      }
    }
  }
}
```

Compiles to:

header.css

```
.page .header-button_primary {
  color: red;
}
```

While these tools reduce manual CSS scoping work, they can create large stylesheets with multiple nested selectors. If CSS scoping were supported natively, these tools could leverage CSS scoping syntax to optimize the compiled CSS rules.

JS Frameworks: Frameworks like Vue and Angular have built-in support for CSS scoping. Primarily, they run a CSS preprocessor that generates a unique component hash and append it to the component style rules.

Ex: Vue

sample.vue

```
<style scoped>
  button.primary {
    color: red;
  }
</style>
<template>
  <button class=”primary”>Save</button>
</template>
```

Compiles to:

sample.css

```
button.primary[data-v-f3f3eg9] {
  color: red;
}
```

sample.html

```
<template>
  <button class=”primary” data-v-f3f3eg9>Save</button>
</template
```

Frameworks do a good job of scoping their CSS styles to the specific component that embeds them. Styles are encapsulated within the component boundary. Page styles propagate to components and can override their styles based on specificity. With all the benefits of using frameworks, it comes at a cost. Frameworks don't play well with each other within a web app. Components from one framework can't be used in another framework.

See demo built with Vue for more examples.

Web Components: Web components have native CSS scoping support. Styles defined within the shadow DOM are scoped only to the component. This frees developers from following a complex selector naming convention to prevent conflicts. However, styling

web components from the host application requires coordination between styles exposed by web components and style rules set by the host application, because global styles do not apply to shadow DOM content. Often, there is a coordination/API mismatch that prevents the web component from being styled as the host application desires.

Ex: The following shows an example of an Accordion web component that uses `var()` for page styling. See demo for more details:

```
Accordion.js
const styles = `
  :host(.section) .panel-content {
     color: var(--accordion-section-content-color);
     background-color: var(--accordion-section-content-bg-color);
 };`
const template = `
  <div id="sect" class="panel">
    <div class="panel-content"> Accordion Content</div>
      <br />
      <slot name="section-panel">section panel missing</slot>
      <slot name="section-description">section description is missing</slot>
  </div>`;
```

The accordion web component exposes two variables for styling from the parent container. Often, the container needs more styling options than is available. However, that's not possible without modification to the web component. This problem, to a lesser degree, exists for ::part() pseudo element.

Need for a new solution:

The existing solutions were created to solve the global scope of CSS rules. With all the existing solutions, why do we need to create another solution? To better illustrate this need, see the diagram below of scoping features to the existing solutions.

Scoped CSS Feature List vs. existing solution:

Features	CSS naming methodolgy: OSSCSS, SMACSS, BEM	CSS Preprocessor: SCSS & LESS	CSS in JS	Frameworks: Angular, Vue	Web Components
Component Style					

Scoped					
Prevent Scoped rules from applying outside of the scoped DOM subtree	NO	NO[3]	YES[1]	YES[1]	YES
Prevent scoped rules from applying inward towards nested components	NO	NO	YES[1]	YES[1]	YES
Global Styles					
Allow global styles to apply to component or DOM subtree	YES	YES	YES	YES	NO
Allow global styles to override scoped CSS styles.	YES	YES	YES	YES	YES[2]
Namespacing					
Scoped selectors don't conflict with selector of same name in other scoped context	NO	NO	YES	YES	YES
Combine multiple scoped CSS rules in a single stylesheet	NO	NO	YES	YES	NO
Additional features					
Scoped rules works with both light DOM and shadow DOM[2]	NO	NO	NO	NO	NO
Separation of Content and Style	YES	YES	NO	YES	NO

[1] - CSS compiler appends unique hash to the component selectors encapsulating their styles to the component.

[2] - Shadow DOM requires CSS var() or ::part() for style overriding.

[3]- CSS preprocessors uses descendant selectors for their scoping feature. However, there's no guarantee that the scoped styles wouldn't conflict with styles of other components.

Referencing the list above, all solutions have some missing features. The main problem is that CSS selectors are globally scoped. The ideal solution to CSS scoping should support all of the features above. Frameworks like Angular and Vue do a very good job ensuring their component CSS are scoped correctly. However, there's a common problem across all solutions that prevents styling across shadow boundaries. Having a scoped CSS will reduce the cognitive overhead required for managing a broad set of CSS styles in the global scope, increase interoperability between components, and boost performance via fewer nodes to traverse during style recalcs.

Summary:

This document is to capture the requirements for CSS scoping, its current state, and the missing features. This document doesn't go into solutions yet, as we are trying to make sure we've captured the problem space correctly and fully. Please comment on whether there are additional items to consider.

Open Questions:

This document is a work in progress! Please share your input!