Shadow DOM 101

- Introduction
- Hello, Shadow World
- Separating Content from Presentation
  - Step 1: Hide Presentation Details
  - Step 2: Separate Content from Presentation
  - Step 3: Profit
- Extra Credit: Advanced Projection
- You Pass Shadow DOM 101

{% Aside 'warning' %} This article describes an old version of Shadow DOM (v0). If you're interested in using Shadow DOM, check out our new article, "Shadow DOM v1: self-contained web components". It covers everything in the newer Shadow DOM v1 spec shipped in Chrome 53, Safari 10, and Firefox 63. {% endAside %}

Introduction

Web Components is a set of cutting edge standards that:

1. Make it possible to build widgets
2. …which can be reused reliably
3. …and which won't break pages if the next version of the component changes internal implementation details.

Does this mean you have to decide when to use HTML/JavaScript, and when to use Web Components? No! HTML and JavaScript can make interactive visual stuff. Widgets are interactive visual stuff. It makes sense to leverage your HTML and JavaScript skills when developing a widget. The Web Components standards are designed to help you do that.

{% Aside %} It doesn't make sense to have to switch to a different technology to build a widget. For example, I'm definitely not a fan of making your widget out of a `<canvas>`. It is reliable — pages won't break if you change what it paints — but it's hostile to accessibility, indexing, composition, and resolution independence. {% endAside %}

But there is a fundamental problem that makes widgets built out of HTML and JavaScript hard to use: The DOM tree inside a widget isn't encapsulated from the rest of the page. This lack of encapsulation means your document stylesheet might accidentally apply to parts inside the widget; your JavaScript might accidentally modify parts inside the widget; your IDs might overlap with IDs inside the widget; and so on.

{% Aside %} A particularly pernicious aspect of the lack of encapsulation is that if you upgrade the library and the internal details of the widget's DOM changes, your styles and scripts might break in unpredictable ways. {% endAside %}

Web Components is comprised of three parts:

1. Templates
2. Shadow DOM
3. Custom Elements

Shadow DOM addresses the DOM tree encapsulation problem. The four parts of Web Components are designed to work together, but you can also pick and choose which parts of Web Components to use. This tutorial shows you how to use Shadow DOM.

{% Aside %} Shadow DOM is now available in Chrome 35+. {% endAside %}

Hello, Shadow World

With Shadow DOM, elements can get a new kind of node associated with them. This new kind of node is called a **shadow root**. An element that has a shadow root associated with it is called a **shadow host.** The content of a shadow host isn't rendered; the content of the shadow root is rendered instead.

For example, if you had markup like this:

```
<button>Hello, world!</button>
<script>
var host = document.querySelector('button');
var root = host.createShadowRoot();
root.textContent = '□□□□□□□□□□□!';
</script>
```

then instead of

```
<button id="ex1a">Hello, world!</button>
<script>
```

```
function remove(selector) {
  Array.prototype.forEach.call(
      document.querySelectorAll(selector),
      function (node) { node.parentNode.removeChild(node); });
}

if (!HTMLElement.prototype.createShadowRoot) {
  remove('#ex1a');
  document.write('<img src="SS1.png" alt="Screenshot of a button with \'Hello, world!\' on
it.">');
}
</script>
```

your page looks like

```
<button id="ex1b">Hello, world!</button>
<script>
(function () {
  if (!HTMLElement.prototype.createShadowRoot) {
    remove('#ex1b');
    document.write('<img src="SS2.png" alt="Screenshot of a button with \'Hello, shadow
world!\' in Japanese on it.">');
    return;
  }
  var host = document.querySelector('#ex1b');
  var root = host.createShadowRoot();
  root.textContent = '□□□□□□□□□□□□!';
})();
</script>
```

Not only that, if JavaScript on the page asks what the button's `textContent` is, it isn't going to get "□□□□□□□□□□ !", but "Hello, world!" because the DOM subtree under the shadow root is encapsulated.

Separating Content from Presentation

Now we'll look at using Shadow DOM to separate content from presentation. Let's say we have this name tag:

```
<style>
.ex2a.outer {
  border: 2px solid brown;
  border-radius: 1em;
  background: red;
  font-size: 20pt;
  width: 12em;
  height: 7em;
  text-align: center;
}
.ex2a .boilerplate {
  color: white;
  font-family: sans-serif;
  padding: 0.5em;
}
.ex2a .name {
  color: black;
  background: white;
  font-family: "Marker Felt", cursive;
  font-size: 45pt;
  padding-top: 0.2em;
}
</style>
<div class="ex2a outer">
  <div class="boilerplate">
    Hi! My name is
  </div>
  <div class="name">
    Bob
  </div>
</div>
```

Here is the markup. This is what you'd write today. It doesn't use Shadow DOM:

```
<style>
.outer {
  border: 2px solid brown;
```

```
  border-radius: 1em;
  background: red;
  font-size: 20pt;
  width: 12em;
  height: 7em;
  text-align: center;
}
.boilerplate {
  color: white;
  font-family: sans-serif;
  padding: 0.5em;
}
.name {
  color: black;
  background: white;
  font-family: "Marker Felt", cursive;
  font-size: 45pt;
  padding-top: 0.2em;
}
</style>
<div class="outer">
  <div class="boilerplate">
    Hi! My name is
  </div>
  <div class="name">
    Bob
  </div>
</div>
```

Because the DOM tree lacks encapsulation, the entire structure of the name tag is exposed to the document. If other elements on the page accidentally use the same class names for styling or scripting, we're gonna have a bad time.

We can avoid having a bad time.

Step 1: Hide Presentation Details

Semantically we probably only care that:

- It is a name tag.
- The name is “Bob”.

First, we write markup that is closer to the true semantics we want:

```
<div id="nameTag">Bob</div>
```

Then we put all of the styles and divs used for presentation into a `<template>` element:

```
<div id="nameTag">Bob</div>
<template id="nameTagTemplate">
<span class="unchanged"><style>
.outer {
  border: 2px solid brown;

  … same as above …

</style>
<div class="outer">
  <div class="boilerplate">
    Hi! My name is
  </div>
  <div class="name">
    Bob
  </div>
</div></span>
</template>
```

At this point ‘Bob’ is the only thing that is rendered. Because we moved the presentational DOM elements inside a `<template>` element, they aren’t rendered, but they **can** be accessed from JavaScript. We do that now to populate the shadow root:

```
<script>
var shadow = document.querySelector('#nameTag').createShadowRoot();
var template = document.querySelector('#nameTagTemplate');
var clone = document.importNode(template.content, true);
shadow.appendChild(clone);
```

{% Aside %} Templates, like Shadow DOM, are an emerging standard. The `<template>` element is available in many modern browsers. You can also populate a shadow root using familiar properties and methods like `innerHTML`, `appendChild`, `getElementById`, and so on. This article is focused on Shadow DOM, so we won’t go further into how the template element works here. If you want to learn more about `<template>`, see HTML's New Template Tag. {% endAside %}

Now that we have set up a shadow root, the name tag is rendered again. If you were to right-click on the name tag and inspect the element you see that it is sweet, semantic markup:

```
<div id="nameTag">Bob</div>
```

This demonstrates that, by using Shadow DOM, we have hidden the presentation details of the name tag from the document. The presentation details are encapsulated in the Shadow DOM.

Step 2: Separate Content from Presentation

Our name tag now hides presentation details from the page, but it doesn't actually separate presentation from content, because although the content (the name "Bob") is in the page, the name that is rendered is the one we copied into the shadow root. If we want to change the name on the name tag, we'd need to do it in two places, and they might get out of sync.

HTML elements are compositional — you can put a button inside a table, for example. Composition is what we need here: The name tag must be a composition of the red background, the "Hi!" text, and the content that is on the name tag.

You, the component author, define how composition works with your widget using a new element called `<content>`. This creates an insertion point in the presentation of the widget, and the insertion point cherry-picks content from the shadow host to present at that point.

If we change the markup in the Shadow DOM to this:

```
<span class="unchanged"><template id="nameTagTemplate">
<style>
  …
</style></span>
<div class="outer">
  <div class="boilerplate">
    Hi! My name is
  </div>
  <div class="name">
    <content></content>
  </div>
</div>
<span class="unchanged"></template></span>
```

When the name tag is rendered, the content of the shadow host is projected into the spot that the `<content>` element appears.

Now the structure of the document is simpler because the name is only in one place — the document. If your page ever needs to update the user's name, you just write:

```
document.querySelector('#nameTag').textContent = 'Shellie';
```

and that is it. The rendering of the name tag is automatically updated by the browser, because we're **projecting** the content of the name tag into place with `<content>`.

```
<div id="ex2b">
```

Now we have achieved separation of content and presentation. **The content is in the document; the presentation is in the Shadow DOM.** They are automatically kept in sync by the browser when it comes time to render something.

Step 3: Profit

By separating content and presentation, we can simplify the code that manipulates the content — in the name tag example, that code only needs to deal with a simple structure containing one `<div>` instead of several.

Now if we change our presentation, we don't need to change any of the code!

For example, say we want to localize our name tag. It is still a name tag, so the semantic content in the document doesn't change:

```
<div id="nameTag">Bob</div>
```

The shadow root setup code stays the same. Just what gets put in the shadow root changes:

```
<template id="nameTagTemplate">
<style>
.outer {
  border: 2px solid pink;
  border-radius: 1em;
  background: url(sakura.jpg);
  font-size: 20pt;
  width: 12em;
  height: 7em;
  text-align: center;
```

```
  font-family: sans-serif;
  font-weight: bold;
}
.name {
  font-size: 45pt;
  font-weight: normal;
  margin-top: 0.8em;
  padding-top: 0.2em;
}
</style>
<div class="outer">
  <div class="name">
    <content></content>
  </div>
  □□□□□□
</div>
</template>
```

{% Aside %} Background image by Mike Dowman, reused under Creative Commons license. {% endAside %}

This is a big improvement over the situation on the web today, because your name update code can depend on the structure of the **component** which is simple and consistent. **Your name update code doesn't need to know the structure used for rendering.** If we consider what is rendered, the name appears second in English (after "Hi! My name is"), but first in Japanese (before "□□□□ "). That distinction is semantically meaningless from the point of view of updating the name that is being displayed, so the name update code doesn't have to know about that detail.

Extra Credit: Advanced Projection

In the above example, the `<content>` element cherry-picks all of the content from the shadow host. By using the `select` attribute, you can control what a content element projects. You can also use multiple content elements.

For example, if you have a document which contains this:

```
<div id="nameTag">
  <div class="first">Bob</div>
  <div>B. Love</div>
```

```
  <div class="email">bob@</div>
</div>
```

and a shadow root which uses CSS selectors to select specific content:

```
<div style="background: purple; padding: 1em;">
  <div style="color: red;">
    <content **select=".first"**></content>
  </div>
  <div style="color: yellow;">
    <content **select="div"**></content>
  </div>
  <div style="color: blue;">
    <content **select=".email">**</content>
  </div>
</div>
```

{% Aside %} `select` can only select elements which are immediate children of the host node. That is, you cannot select descendants (e.g. `select="table tr"`). {% endAside %}

The `<div class="email">` element is matched by both the `<content select="div">` and `<content select=".email">` elements. How many times does Bob's email address appear, and in what colors?

The answer is that Bob's email address appears once, and it is yellow.

The reason is that, as people who hack on Shadow DOM know, constructing the tree of what is actually rendered on screen is like a huge party. **The content element is the invitation that lets content from the document into the backstage Shadow DOM rendering party.** These invitations are delivered in order; who gets an invitation depends on to whom it is addressed (that is, the `select` attribute.) Content, once invited, always accepts the invitation (who wouldn't?!) and off it goes. If a subsequent invitation is sent to that address again, well, nobody is home, and it doesn't come to your party.

In the above example, `<div class="email">` matches both the `div` selector and the `.email` selector, but because the content element with the `div` selector comes earlier in the document, `<div class="email">` goes to the yellow party, and nobody is available to come to the blue party. (That might be **why** it is so blue, although misery loves company, so you never know.)

If something is invited to **no** parties, then it doesn't get rendered at all. That is what happened to the "Hello, world" text in the very first example. This is useful when you want to achieve a radically different rendering: Write the semantic model in the document, which is what is accessible to scripts in the page, but hide it for rendering purposes and connect it to a really different rendering model in Shadow DOM using JavaScript.

For example, HTML has a nice date picker. If you write `<input type="date">` you get a neat pop-up calendar. But what if you want to let the user pick a range of dates for their **dessert** island vacation (you know… with hammocks made out of Red Vines.) You set up your document this way:

```
<div class="dateRangePicker">
  <label for="start">Start:</label>
  <input type="date" name="startDate" id="start">
  <br>
  <label for="end">End:</label>
  <input type="date" name="endDate" id="end">
</div>
```

but create Shadow DOM that uses a table to create a slick calendar which highlights the range of dates and so on. When the user clicks on the days in the calendar, the component updates the state in the startDate and endDate inputs; when the user submits the form, the values from those input elements get submitted.

Why did I include labels in the document if they’re not going to be rendered? The reason is that if a user views the form with a browser that doesn’t support Shadow DOM, the form is still usable, just not as pretty. The user sees something like:

```
<div class="dateRangePicker">
  <label for="start">Start:</label>
  <input type="date" name="startDate" id="start">
  <br>
  <label for="end">End:</label>
  <input type="date" name="endDate" id="end">
</div>
```

You Pass Shadow DOM 101

Those are the basics of Shadow DOM — you pass Shadow DOM 101! You can do more with Shadow DOM, for example, you can use multiple shadow on one shadow host, or nested shadows for encapsulation, or architect your page using Model-Driven Views (MDV) and Shadow DOM. And Web Components are more than just Shadow DOM.

We explain those in later posts.

□□□ #2
□ □□□ □□□ 2026-10-31 22:16:03 UTC
□ □□□ □□□ 2026-10-31 22:16:05 UTC