Shadow DOM v1: Self-Contained Web Components

By Eric Bidelman
Engineer @ Google working on web tooling: Headless Chrome, Puppeteer, Lighthouse

TL;DR

Shadow DOM removes the brittleness of building web apps. The brittleness comes from the global nature of HTML, CSS, and JS. Over the years we've invented an exorbitant number of tools to circumvent the issues. For example, when you use a new HTML id/class, there's no telling if it will conflict with an existing name used by the page. Subtle bugs creep up, CSS specificity becomes a huge issue (`!important` all the things!), style selectors grow out of control, and performance can suffer. The list goes on.

Shadow DOM fixes CSS and DOM. It introduces **scoped styles** to the web platform. Without tools or naming conventions, you can **bundle CSS with markup**, hide implementation details, and **author self-contained components** in vanilla JavaScript.

Introduction

Note: Already familiar with Shadow DOM? This article describes the new Shadow DOM v1 spec. If you've been using Shadow DOM, chances are you're familiar with the v0 version that shipped in Chrome 35, and the webcomponents.js polyfills. The concepts are the same, but the v1 spec has important API differences. It's also the version that all major browsers have agreed to implement, with implementations already in Safari Tech Preview and Chrome Canary. Keep reading to see what's new or check out the section on History and browser support for more info.

Shadow DOM is one of the four Web Component standards: HTML Templates, Shadow DOM, Custom elements and HTML Imports.

You don't have to author web components that use shadow DOM. But when you do, you take advantage of its benefits (CSS scoping, DOM encapsulation, composition) and build reusable custom elements, which are resilient, highly configurable, and extremely reusable. If custom elements are the way to create a new HTML (with a JS API), shadow DOM is the way you provide its HTML and CSS. The two APIs combine to make a component with self-contained HTML, CSS, and JavaScript.

Shadow DOM is designed as a tool for building component-based apps. Therefore, it brings solutions for common problems in web development:

- **Isolated DOM**: A component's DOM is self-contained (e.g. `document.querySelector()` won't return nodes in the component's shadow DOM).
- **Scoped CSS**: CSS defined inside shadow DOM is scoped to it. Style rules don't leak out and page styles don't bleed in.
- **Composition**: Design a declarative, markup-based API for your component.
- **Simplifies CSS** - Scoped DOM means you can use simple CSS selectors, more generic id/class names, and not worry about naming conflicts.
- **Productivity** - Think of apps in chunks of DOM rather than one large (global) page.

Note: Although you can use the shadow DOM API and its benefits outside of web components, I'm only going to focus on examples that build on custom elements. I'll be using the custom elements v1 API in all examples.

`fancy-tabs` demo

Throughout this article, I'll be referring to a demo component (`<fancy-tabs>`) and referencing code snippets from it. If your browser supports the APIs, you should see a live demo of it just below. Otherwise, check out the full source on Github.

View source on Github

What is shadow DOM?

Background on DOM

HTML powers the web because it's easy to work with. By declaring a few tags, you can author a page in seconds that has both presentation and structure. However, by itself HTML isn't all that useful. It's easy for humans to understand a text- based language, but machines need something more. Enter the Document Object Model, or DOM.

When the browser loads a web page it does a bunch of interesting stuff. One of the things it does is transform the author's HTML into a live document. Basically, to understand the page's structure, the browser parses HTML (static strings of text) into a data model (objects/nodes). The browser preserves the HTML's hierarchy by creating a tree of these nodes: the DOM. The cool thing about DOM is that it's a live representation of your page. Unlike the static HTML we author, the browser-produced nodes contain properties, methods, and best of all...can be manipulated by programs! That's why we're able to create DOM elements directly using JavaScript:

```
const header = document.createElement('header');
const h1 = document.createElement('h1');
h1.textContent = 'Hello world!';
header.appendChild(h1);
document.body.appendChild(header);
```

produces the following HTML markup:

```
<body>
  <header>
    <h1>Hello DOM</h1>
  </header>
</body>
```

All that is well and good. Then what the heck is *shadow DOM*?

DOM...in the shadows

Shadow DOM is just normal DOM with two differences: 1) how it's created/used and 2) how it behaves in relation to the rest of the page. Normally, you create DOM nodes and append them as children of another element. With shadow DOM, you create a scoped DOM tree that's attached to the element, but separate from its actual children. This scoped subtree is called a **shadow tree**. The element it's attached to is its **shadow host**. Anything you add in the shadows becomes local to the hosting element, including `<style>`. This is how shadow DOM achieves CSS style scoping.

Creating shadow DOM

A **shadow root** is a document fragment that gets attached to a “host” element. The act of attaching a shadow root is how the element gains its shadow DOM. To create shadow DOM for an element, call `element.attachShadow()`:

```
const header = document.createElement('header');
const shadowRoot = header.attachShadow({mode: 'open'});
shadowRoot.innerHTML = '<h1>Hello Shadow DOM</h1>'; // Could also use appendChild().

// header.shadowRoot === shadowRoot
// shadowRoot.host === header
```

I'm using `.innerHTML` to fill the shadow root, but you could also use other DOM APIs. This is the web. We have choice.

The spec defines a list of elements that can't host a shadow tree. There are several reasons an element might be on the list:

- The browser already hosts its own internal shadow DOM for the element (`<textarea>`, `<input>`).
- It doesn't make sense for the element to host a shadow DOM (`<img>`).

For example, this doesn't work:

```
document.createElement('input').attachShadow({mode: 'open'});
// Error. `<input>` cannot host shadow dom.
```

Creating shadow DOM for a custom element

Shadow DOM is particularly useful when creating custom elements. Use shadow DOM to compartmentalize an element's HTML, CSS, and JS, thus producing a "web component".

Example - a custom element **attaches shadow DOM to itself**, encapsulating its DOM/CSS:

```
// Use custom elements API v1 to register a new HTML tag and define its JS beh
// using an ES6 class. Every instance of <fancy-tab> will have this same prototype.
customElements.define('fancy-tabs', class extends HTMLElement {
  constructor() {
    super(); // always call super() first in the constructor.
    // Attach a shadow root to <fancy-tabs>.
    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML = `
      <style>#tabs { ... }</style> <!-- styles are scoped to fancy-tabs! -->
      <div id="tabs">...</div>
      <div id="panels">...</div>
    `;
  }
  ...
});
```

There are a couple of interesting things going on here. The first is that the custom element **creates its own shadow DOM** when an instance of `<fancy-tabs>` is created. That's done in the `constructor()`. Secondly, because we're creating a shadow root, the CSS rules inside the `<style>` will be scoped to `<fancy-tabs>`.

Note: When you try to run this example, you'll probably notice that nothing renders. The user's markup seemingly disappears! That's because the **element's shadow DOM is rendered in place of its children**. If you want to display the children, you need to tell the browser where to render them by placing a `<slot>` element in your shadow DOM. More on that later.

Composition and slots

Composition is one of the least understood features of shadow DOM, but it's arguably the most important.

In our world of web development, composition is how we construct apps, declaratively out of HTML. Different building blocks (`<div>`s, `<header>`s, `<form>`s, `<input>`s) come together to form apps. Some of these tags even work with each other. Composition is why native elements like `<select>`, `<details>`, `<form>`, and `<video>` are so flexible. Each of those tags accepts certain HTML as children and does something special with them. For example, `<select>` knows how to render `<option>` and `<optgroup>` into dropdown and multi-select widgets. The `<details>` element renders `<summary>` as a expandable arrow. Even `<video>` knows how to deal with certain children: `<source>` elements don't get rendered, but they do affect the video's behavior. What magic!

Terminology: light DOM vs. shadow DOM

Shadow DOM composition introduces a bunch of new fundamentals in web development. Before getting into the weeds, let's standardize on some terminology so we're speaking the same lingo.

Light DOM

The markup a user of your component writes. This DOM lives outside the component's shadow DOM. It is the element's actual children.

```
<better-button>
  <!-- the image and span are better-button's light DOM -->
  <img src="gear.svg" slot="icon">
  <span>Settings</span>
</better-button>
```

Shadow DOM

The DOM a component author writes. Shadow DOM is local to the component and defines its internal structure, scoped CSS, and encapsulates your implementation details. It can also define how to render markup that's authored by the consumer of your component.

```
#shadow-root
  <style>...</style>
  <slot name="icon"></slot>
  <span id="wrapper">
    <slot>Button</slot>
  </span>
```

Flattened DOM tree

The result of the browser distributing the user's light DOM into your shadow DOM, rendering the final product. The flattened tree is what you ultimately see in the DevTools and what's rendered on the page.

```
<better-button>
  #shadow-root
    <style>...</style>
    <slot name="icon">
      <img src="gear.svg" slot="icon">
    </slot>
    <span id="wrapper">
      <slot>
        <span>Settings</span>
      </slot>
    </span>
</better-button>
```

The <slot> element

Shadow DOM composes different DOM trees together using the `<slot>` element. **Slots are placeholders inside your component that users *can* fill with their own markup**. By defining one or more slots, you invite outside markup to render in your component's shadow DOM. Essentially, you're saying *"Render the user's markup over here"*.

Note: Slots are a way of creating a "declarative API" for a web component. They mix-in the user's DOM to help render the overall component, thus, **composing different DOM trees together**.

Elements are allowed to "cross" the shadow DOM boundary when a `<slot>` invites them in. These elements are called **distributed nodes**. Conceptually, distributed nodes can seem a bit bizarre. Slots don't physically move DOM; they render it at another location inside the shadow DOM.

A component can define zero or more slots in its shadow DOM. Slots can be empty or provide fallback content. If the user doesn't provide light DOM content, the slot renders its fallback content.

```
<!-- Default slot. If there's more than one default slot, the first is used. -
<slot></slot>

<slot>fallback content</slot> <!-- default slot with fallback content -->

<slot> <!-- default slot entire DOM tree as fallback -->
  <h2>Title</h2>
  <summary>Description text</summary>
</slot>
```

You can also create **named slots**. Named slots are specific holes in your shadow DOM that users reference by name.

Example - the slots in `<fancy-tabs>`'s shadow DOM:

```
#shadow-root
  <div id="tabs">
    <slot id="tabsSlot" name="title"></slot> <!-- named slot -->
  </div>
  <div id="panels">
    <slot id="panelsSlot"></slot>
  </div>
```

Component users declare `<fancy-tabs>` like so:

```
<fancy-tabs>
  <button slot="title">Title</button>
  <button slot="title" selected>Title 2</button>
  <button slot="title">Title 3</button>
  <section>content panel 1</section>
  <section>content panel 2</section>
  <section>content panel 3</section>
</fancy-tabs>

<!-- Using <h2>'s and changing the ordering would also work! -->
<fancy-tabs>
  <h2 slot="title">Title</h2>
  <section>content panel 1</section>
  <h2 slot="title" selected>Title 2</h2>
  <section>content panel 2</section>
  <h2 slot="title">Title 3</h2>
  <section>content panel 3</section>
</fancy-tabs>
```

And if you're wondering, the flattened tree looks something like this:

```
<fancy-tabs>
  #shadow-root
    <div id="tabs">
      <slot id="tabsSlot" name="title">
        <button slot="title">Title</button>
        <button slot="title" selected>Title 2</button>
        <button slot="title">Title 3</button>
      </slot>
    </div>
    <div id="panels">
      <slot id="panelsSlot">
        <section>content panel 1</section>
        <section>content panel 2</section>
        <section>content panel 3</section>
      </slot>
    </div>
</fancy-tabs>
```

Notice our component is able to handle different configurations, but the flattened DOM tree remains the same. We can also switch from `<button>` to `<h2>`. This component was authored to handle different types of children...just like `<select>` does!

Styling

There are many options for styling web components. A component that uses shadow DOM can be styled by the main page, define its own styles, or provide hooks (in the form of CSS custom properties) for users to override defaults.

Component-defined styles

Hands down the most useful feature of shadow DOM is **scoped CSS**:

- CSS selectors from the outer page don't apply inside your component.
- Styles defined inside don't bleed out. They're scoped to the host element.

CSS selectors used inside shadow DOM apply locally to your component. In practice, this means we can use common id/class names again, without worrying about conflicts elsewhere on the page. Simpler CSS selectors are a best practice inside Shadow DOM. They're also good for performance.

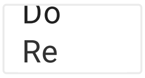
Example - styles defined in a shadow root are local

```
#shadow-root
  <style>
    #panels {
      box-shadow: 0 2px 2px rgba(0, 0, 0, .3);
      background: white;
      ...
    }
    #tabs {
      display: inline-flex;
      ...
    }
  </style>
  <div id="tabs">
    ...
  </div>
  <div id="panels">
    ...
  </div>
```

Stylesheets are also scoped to the shadow tree:

```
#shadow-root
  <!-- Available in Chrome 54+ -->
  <!-- WebKit bug: https://bugs.webkit.org/show_bug.cgi?id=160683 -->
  <link rel="stylesheet" href="styles.css">
  <div id="tabs">
    ...
  </div>
  <div id="panels">
    ...
  </div>
```

Ever wonder how the `<select>` element renders a multi-select widget (instead of a dropdown) when you add the `multiple` attribute:


Do
Re

`<select>` is able to style *itself* differently based on the attributes you declare on it. Web components can style themselves too, by using the `:host` selector.

Example - a component styling itself

```
<style>
:host {
  display: block; /* by default, custom elements are display: inline */
  contain: content; /* CSS containment FTW. */
}
</style>
```

One gotcha with `:host` is that rules in the parent page have higher specificity than `:host` rules defined in the element. That is, outside styles win. This allows users to override your top-level styling from the outside. Also, `:host` only works in the context of a shadow root, so you can't use it outside of shadow DOM.

The functional form of `:host(<selector>)` allows you to target the host if it matches a `<selector>`. This is a great way for your component to encapsulate behaviors that react to user interaction or state or style internal nodes based on the host.

```
<style>
:host {
  opacity: 0.4;
  will-change: opacity;
  transition: opacity 300ms ease-in-out;
}
:host(:hover) {
  opacity: 1;
}
:host([disabled]) { /* style when host has disabled attribute. */
  background: grey;
  pointer-events: none;
  opacity: 0.4;
}
:host(.blue) {
  color: blue; /* color host when it has class="blue" */
}
:host(.pink) > #tabs {
  color: pink; /* color internal #tabs node when host has class="pink". */
}
</style>
```

Styling based on context

`:host-context(<selector>)` matches the component if it or any of its ancestors matches **`<selector>`**. A common use for this is theming based on a component's surroundings. For example, many people do theming by applying a class to **`<html>`** or **`<body>`**:

```
<body class="darktheme">
  <fancy-tabs>
    ...
  </fancy-tabs>
</body>
```

`:host-context(.darktheme)` would style **`<fancy-tabs>`** when it's a descendant of **`.darktheme`**:

```
:host-context(.darktheme) {
  color: white;
  background: black;
}
```

`:host-context()` can be useful for theming, but an even better approach is to create style hooks using CSS custom properties.

Styling distributed nodes

`::slotted(<compound-selector>)` matches nodes that are distributed into a **`<slot>`**.

Let's say we've created a name badge component:

```
<name-badge>
  <h2>Eric Bidelman</h2>
  <span class="title">
    Digital Jedi, <span class="company">Google</span>
  </span>
</name-badge>
```

The component's shadow DOM can style the user's **`<h2>`** and **`.title`**:

```
<style>
::slotted(h2) {
  margin: 0;
  font-weight: 300;
  color: red;
}
::slotted(.title) {
   color: orange;
}
```

```
/* DOESN'T WORK (can only select top-level nodes).
::slotted(.company),
::slotted(.title .company) {
  text-transform: uppercase;
}
*/
</style>
<slot></slot>
```

If you remember from before, `<slot>`s do not move the user's light DOM. When nodes are distributed into a `<slot>`, the `<slot>` renders their DOM but the nodes physically stay put. **Styles that applied before distribution continue to apply after distribution**. However, when the light DOM is distributed, it *can* take on additional styles (ones defined by the shadow DOM).

Another, more in-depth example from `<fancy-tabs>`:

```
const shadowRoot = this.attachShadow({mode: 'open'});
shadowRoot.innerHTML = `
  <style>
    #panels {
      box-shadow: 0 2px 2px rgba(0, 0, 0, .3);
      background: white;
      border-radius: 3px;
      padding: 16px;
      height: 250px;
      overflow: auto;
    }
    #tabs {
      display: inline-flex;
      -webkit-user-select: none;
      user-select: none;
    }
    #tabsSlot::slotted(*) {
      font: 400 16px/22px 'Roboto';
      padding: 16px 8px;
      ...
    }
    #tabsSlot::slotted([aria-selected="true"]) {
      font-weight: 600;
      background: white;
      box-shadow: none;
    }
    #panelsSlot::slotted([aria-hidden="true"]) {
      display: none;
    }
  </style>
  <div id="tabs">
    <slot id="tabsSlot" name="title"></slot>
  </div>
  <div id="panels">
    <slot id="panelsSlot"></slot>
  </div>
`;
```

In this example, there are two slots: a named slot for the tab titles, and a slot for the tab panel content. When the user selects a tab, we bold their selection and reveal its panel. That's done by selecting distributed nodes that have the `selected` attribute. The custom element's JS (not shown here) adds that attribute at the correct time.

Styling a component from the outside

There are a couple of ways to style a component from the outside. The easiest way is to use the tag name as a selector:

```
fancy-tabs {
  width: 500px;
  color: red; /* Note: inheritable CSS properties pierce the shadow DOM boundary. */
}
fancy-tabs:hover {
  box-shadow: 0 3px 3px #ccc;
}
```

Outside styles always win over styles defined in shadow DOM. For example, if the user writes the selector **`fancy-tabs { width: 500px; }`**, it will trump the component's rule: **`:host { width: 650px;}`**.

Styling the component itself will only get you so far. But what happens if you want to style the internals of a component? For that, we need CSS custom properties.

Creating style hooks using CSS custom properties

Users can tweak internal styles if the component's author provides styling hooks using CSS custom properties. Conceptually, the idea is similar to **`<slot>`**. You create "style placeholders" for users to override.

Example - `<fancy-tabs>` allows users to override the background color:

```
<!-- main page -->
<style>
  fancy-tabs {
    margin-bottom: 32px;
    --fancy-tabs-bg: black;
  }
```

```
</style>
<fancy-tabs background>...</fancy-tabs>
```

Inside its shadow DOM:

```
:host([background]) {
  background: var(--fancy-tabs-bg, #9E9E9E);
  border-radius: 10px;
  padding: 10px;
}
```

In this case, the component will use `black` as the background value since the user provided it. Otherwise, it would default to `#9E9E9E`.

Note: As the component author, you're responsible for letting developers know about CSS custom properties they can use. Consider it part of your component's public interface. Make sure to document styling hooks!

Advanced topics

Creating closed shadow roots (should avoid)

There's another flavor of shadow DOM called "closed" mode. When you create a closed shadow tree, outside JavaScript won't be able to access the internal DOM of your component. This is similar to how native elements like `<video>` work. JavaScript cannot access the shadow DOM of `<video>` because the browser implements it using a closed-mode shadow root.

Example - creating a closed shadow tree:

```
const div = document.createElement('div');
const shadowRoot = div.attachShadow({mode: 'closed'}); // close shadow tree
// div.shadowRoot === null
// shadowRoot.host === div
```

Other APIs are also affected by closed-mode:

- `Element.assignedSlot` / `TextNode.assignedSlot` returns `null`
- `Event.composedPath()` for events associated with elements inside the shadow DOM, returns []

Note: Closed shadow roots are not very useful. Some developers will see closed mode as an artificial security feature. But let's be clear, it's **not** a security feature. Closed mode simply prevents outside JS from drilling into an element's internal DOM.

Here's my summary of why you should never create web components with `{mode: 'closed'}`:

1. Artificial sense of security. There's nothing stopping an attacker from hijacking `Element.prototype.attachShadow`.
2. Closed mode **prevents your custom element code from accessing its own shadow DOM**. That's complete fail. Instead, you'll have to stash a reference for later if you want to use things like `querySelector()`. This completely defeats the original purpose of closed mode!

   ```
   customElements.define('x-element', class extends HTMLElement {
     constructor() {
       super(); // always call super() first in the constructor.
       this._shadowRoot = this.attachShadow({mode: 'closed'});
       this._shadowRoot.innerHTML = '<div class="wrapper"></div>';
     }
     connectedCallback() {
       // When creating closed shadow trees, you'll need to stash the shadow root
       // for later if you want to use it again. Kinda pointless.
       const wrapper = this._shadowRoot.querySelector('.wrapper');
     }
     ...
   });
   ```

3. **Closed mode makes your component less flexible for end users**. As you build web components, there will come a time when you forget to add a feature. A configuration option. A use case the user wants. A common example is forgetting to include adequate styling hooks for internal nodes. With closed mode, there's no way for users to override defaults and tweak styles. Being able to access the component's internals is super helpful. Ultimately, users will fork your component, find another, or create their own if it doesn't do what they want :(

Working with slots in JS

The shadow DOM API provides utilities for working with slots and distributed nodes. These come in handy when authoring a custom element.

slotchange event

The `slotchange` event fires when a slot's distributed nodes changes. For example, if the user adds/removes children from the light DOM.

```
const slot = this.shadowRoot.querySelector('#slot');
slot.addEventListener('slotchange', e => {
```

```
  console.log('light dom children changed!');
});
```

Note: `slotchange` does not fire when an instance of the component is first initialized.

To monitor other types of changes to light DOM, you can setup a `MutationObserver` in your element's constructor.

What elements are being rendering in a slot?

Sometimes it's useful to know what elements are associated with a slot. Call `slot.assignedNodes()` to find which elements the slot is rendering. The `{flatten: true}` option will also return a slot's fallback content (if no nodes are being distributed).

As an example, let's say your shadow DOM looks like this:

```
<slot><b>fallback content</b></slot>
```

Usage	Call	Result
<my-component>component text</my-component>	`slot.assignedNodes();`	`[component text]`
<my-component></my-component>	`slot.assignedNodes();`	`[]`
<my-component></my-component>	`slot.assignedNodes({flatten: true});`	`[<b>fallback content</b>]`

What slot is an element assigned to?

Answering the reverse question is also possible. `element.assignedSlot` tells you which of the component slots your element is assigned to.

The Shadow DOM event model

When an event bubbles up from shadow DOM it's target is adjusted to maintain the encapsulation that shadow DOM provides. That is, events are re-targeted to look like they've come from the component rather than internal elements within your shadow DOM. Some events do not even propagate out of shadow DOM.

The events that **do** cross the shadow boundary are:

- Focus Events: `blur`, `focus`, `focusin`, `focusout`

- Mouse Events: **`click`**, **`dblclick`**, **`mousedown`**, **`mouseenter`**, **`mousemove`**, etc.
- Wheel Events: **`wheel`**
- Input Events: **`beforeinput`**, **`input`**
- Keyboard Events: **`keydown`**, **`keyup`**
- Composition Events: **`compositionstart`**, **`compositionupdate`**, **`compositionend`**
- DragEvent: **`dragstart`**, **`drag`**, **`dragend`**, **`drop`**, etc.

Tips

If the shadow tree is open, calling **`event.composedPath()`** will return an array of nodes that the event traveled through.

Using custom events

Custom DOM events which are fired on internal nodes in a shadow tree do not bubble out of the shadow boundary unless the event is created using the **`composed: true`** flag:

```
// Inside <fancy-tab> custom element class definition:
selectTab() {
  const tabs = this.shadowRoot.querySelector('#tabs');
  tabs.dispatchEvent(new Event('tab-select', {bubbles: true, composed: true}));
}
```

If **`composed: false`** (default), consumers won't be able to listen for the event outside of your shadow root.

```
<fancy-tabs></fancy-tabs>
<script>
  const tabs = document.querySelector('fancy-tabs');
  tabs.addEventListener('tab-select', e => {
    // won't fire if `tab-select` wasn't created with `composed: true`.
  });
</script>
```

Handling focus

If you recall from shadow DOM's event model, events that are fired inside shadow DOM are adjusted to look like they come from the hosting element. For example, let's say you click an **`<input>`** inside a shadow root:

```
<x-focus>
  #shadow-root
```

```
    <input type="text" placeholder="Input inside shadow dom">
```

The **focus** event will look like it came from **<x-focus>**, not the **<input>**. Similarly, **document.activeElement** will be **<x-focus>**. If the shadow root was created with **mode:'open'** (see closed mode), you'll also be able access the internal node that gained focus:

```
document.activeElement.shadowRoot.activeElement // only works with open mode.
```

If there are multiple levels of shadow DOM at play (say a custom element within another custom element), you need to recursively drill into the shadow roots to find the **activeElement**:

```
function deepActiveElement() {
  let a = document.activeElement;
  while (a && a.shadowRoot && a.shadowRoot.activeElement) {
    a = a.shadowRoot.activeElement;
  }
  return a;
}
```

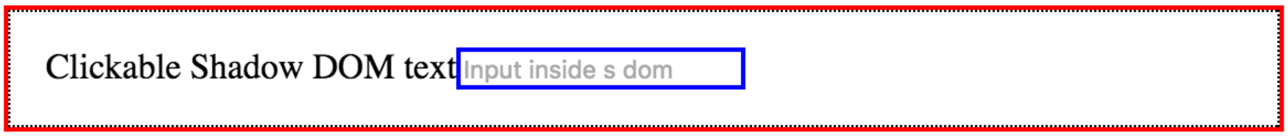
Another option for focus is the **delegatesFocus: true** option, which expands the focus behavior of element's within a shadow tree:

- If you click a node inside shadow DOM and the node is not a focusable area, the first focusable area becomes focused.
- When a node inside shadow DOM gains focus, **:focus** applies to the host in addition to the focused element.

Example - how **delegatesFocus: true** changes focus behavior

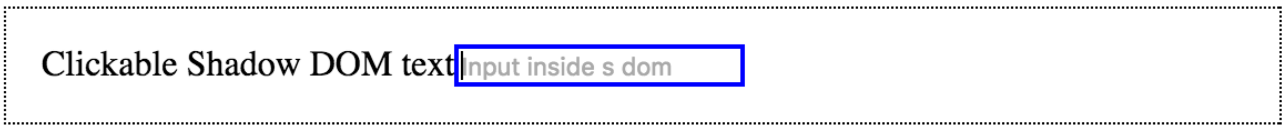
```
<style>
  :focus {
    outline: 2px solid red;
  }
</style>

<x-focus></x-focus>

<script>
customElements.define('x-focus', class extends HTMLElement {
  constructor() {
    super(); // always call super() first in the constructor.

    const root = this.attachShadow({mode: 'open', delegatesFocus: true});
    root.innerHTML = `
      <style>
        :host {
          display: flex;
          border: 1px dotted black;
          padding: 16px;
        }
        :focus {
          outline: 2px solid blue;
        }
      </style>
      <div>Clickable Shadow DOM text</div>
      <input type="text" placeholder="Input inside shadow dom">`;

    // Know the focused element inside shadow DOM:
    this.addEventListener('focus', function(e) {
      console.log('Active element (inside shadow dom):',
                  this.shadowRoot.activeElement);
    });
  }
});
</script>
```

Result

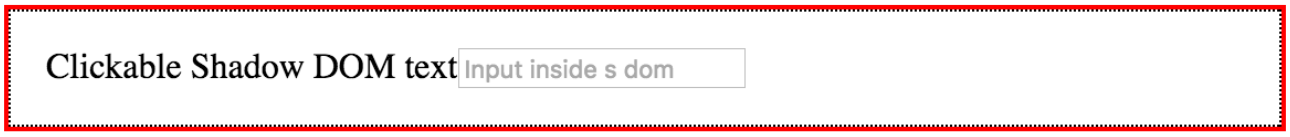


Above is the result when `<x-focus>` is focused (user click, tabbed into, `focus()`, etc.), "Clickable Shadow DOM text" is clicked, or the internal `<input>` is focused (including `autofocus`).

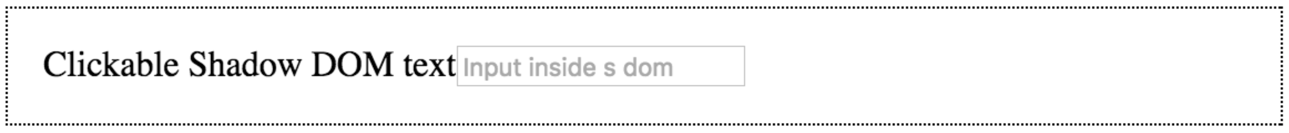
If you were to set `delegatesFocus: false`, here's what you would see instead:



`delegatesFocus: false` and the internal `<input>` is focused.



`delegatesFocus: false` and `<x-focus>` gains focus (e.g. it has `tabindex="0"`).



`delegatesFocus: false` and "Clickable Shadow DOM text" is clicked (or other empty area within the element's shadow DOM is clicked).

Tips & Tricks

Over the years I've learned a thing or two about authoring web components. I think you'll find some of these tips useful for authoring components and debugging shadow DOM.

Use CSS containment

Typically, a web component's layout/style/paint is fairly self-contained. Use CSS containment in `:host` for a perf win:

```
<style>
:host {
  display: block;
  contain: content; /* Boom. CSS containment FTW. */
}
</style>
```

Resetting inheritable styles

Inheritable styles (`background`, `color`, `font`, `line-height`, etc.) continue to inherit in shadow DOM. That is, they pierce the shadow DOM boundary by default. If you want to start with a fresh slate, use `all: initial;` to reset inheritable styles to their initial value when they cross the shadow boundary.

```
<style>
  div {
    padding: 10px;
    background: red;
    font-size: 25px;
    text-transform: uppercase;
    color: white;
  }
</style>

<div>
  <p>I'm outside the element (big/white)</p>
  <my-element>Light DOM content is also affected.</my-element>
  <p>I'm outside the element (big/white)</p>
</div>
```

```
<script>
const el = document.querySelector('my-element');
el.attachShadow({mode: 'open'}).innerHTML = `
  <style>
    :host {
      all: initial; /* 1st rule so subsequent properties are reset. */
      display: block;
      background: white;
    }
  </style>
  <p>my-element: all CSS properties are reset to their
     initial value using <code>all: initial</code>.</p>
  <slot></slot>
`;
</script>
```

Finding all the custom elements used by a page

Sometimes it's useful to find custom elements used on the page. To do so, you need to recursively traverse the shadow DOM of all elements used on the page.

```
const allCustomElements = [];

function isCustomElement(el) {
  const isAttr = el.getAttribute('is');
  // Check for <super-button> and <button is="super-button">.
  return el.localName.includes('-') || isAttr && isAttr.includes('-');
}

function findAllCustomElements(nodes) {
  for (let i = 0, el; el = nodes[i]; ++i) {
    if (isCustomElement(el)) {
      allCustomElements.push(el);
    }
    // If the element has shadow DOM, dig deeper.
    if (el.shadowRoot) {
      findAllCustomElements(el.shadowRoot.querySelectorAll('*'));
    }
  }
}

findAllCustomElements(document.querySelectorAll('*'));
```

Creating elements from a <template>

Instead of populating a shadow root using `.innerHTML`, we can use a declarative `<template>`. Templates are an ideal placeholder for declaring the structure of a web component.

See the example in "Custom elements: building reusable web components".

History & browser support

If you've been following web components for the last couple of years, you'll know that Chrome 35+/Opera have been shipping an older version of shadow DOM for some time. Blink will continue to support both versions in parallel for some time. The v0 spec provided a different method to create a shadow root (`element.createShadowRoot` instead of v1's `element.attachShadow`). Calling the older method continues to create a shadow root with v0 semantics, so existing v0 code won't break.

If you happen to be interested in the old v0 spec, check out the html5rocks articles: 1, 2, 3. There's also a great comparison of the differences between shadow DOM v0 and v1.

Browser support

Chrome 53 (status), Opera 40, and Safari 10 are shipping shadow DOM v1. Edge is under consideration with high priority. Mozilla has an open bug to implement.

To feature detect shadow DOM, check for the existence of `attachShadow`:

```
const supportsShadowDOMV1 = !!HTMLElement.prototype.attachShadow;
```

Polyfill

Until browser support is widely available, the shadydom and shadycss polyfills give you v1 feature. Shady DOM mimics the DOM scoping of Shadow DOM and shadycss polyfills CSS custom properties and the style scoping the native API provides.

Install the polyfills:

```
bower install --save webcomponents/shadydom
bower install --save webcomponents/shadycss
```

Use the polyfills:

```
function loadScript(src) {
 return new Promise(function(resolve, reject) {
   const script = document.createElement('script');
   script.async = true;
   script.src = src;
   script.onload = resolve;
   script.onerror = reject;
   document.head.appendChild(script);
 });
}

// Lazy load the polyfill if necessary.
if (!supportsShadowDOMV1) {
  loadScript('/bower_components/shadydom/shadydom.min.js')
    .then(e => loadScript('/bower_components/shadycss/shadycss.min.js'))
    .then(e => {
      // Polyfills loaded.
    });
} else {
  // Native shadow dom v1 support. Go to go!
}
```

See the https://github.com/webcomponents/shadycss#usage for instructions on how to shim/scope your styles.

Conclusion

For the first time ever, we have an API primitive that does proper CSS scoping, DOM scoping, and has true composition. Combined with other web component APIs like custom elements, shadow DOM provides a way to author truly encapsulated components without hacks or using older baggage like `<iframe>`s.

Don't get me wrong. Shadow DOM is certainly a complex beast! But it's a beast worth learning. Spend some time with it. Learn it and ask questions!

Further reading

- Differences between Shadow DOM v1 and v0
- "Introducing Slot-Based Shadow DOM API" from the WebKit Blog.

- Web Components and the future of Modular CSS by Philip Walton
- "Custom elements: building reusable web components" from Google's WebFundamentals.
- Shadow DOM v1 spec
- Custom elements v1 spec

FAQ

Can I use Shadow DOM v1 today?

With a polyfill, yes. See Browser support.

What security features does shadow DOM provide?

Shadow DOM is not a security feature. It's a lightweight tool for scoping CSS and hiding away DOM trees in component. If you want a true security boundary, use an `<iframe>`.

Does a web component have to use shadow DOM?

Nope! You don't have to create web components that use shadow DOM. However, authoring custom elements that use Shadow DOM means you can take advantage of features like CSS scoping, DOM encapsulation, and composition.

What's the difference between open and closed shadow roots?

See Closed shadow roots.

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License. For details, see our Site Policies. Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 23, 2018.