Shadow DOM 301

- Using multiple shadow roots
  - Shadow Insertion Points
- Obtaining a host's shadow root
- Building Shadow DOM in JS
- Working with insertion points
  - Element.getDistributedNodes()
  - Element.getDestinationInsertionPoints()
- Tool: Shadow DOM Visualizer
- Event Model
  - Events that are always stopped
- Conclusion

{% 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 %}

This article discusses more of the amazing things you can do with Shadow DOM! It builds on the concepts discussed in Shadow DOM 101 and Shadow DOM 201.

{% Aside %} In Chrome, turn on the "Enable experimental Web Platform features" in about:flags to experiment with everything covered in this article. {% endAside %}

Using multiple shadow roots

If you're hosting a party, it gets stuffy if everyone is crammed into the same room. You want the option of distributing groups of people across multiple rooms. Elements hosting Shadow DOM can do this too, that is to say, they can host more than one shadow root at a time.

Let's see what happens if we try to attach multiple shadow roots to a host:

```
<div id="example1">Light DOM</div>
<script>
  var container = document.querySelector('#example1');
  var root1 = container.createShadowRoot();
  var root2 = container.createShadowRoot();
  root1.innerHTML = '<div>Root 1 FTW</div>';
  root2.innerHTML = '<div>Root 2 FTW</div>';
</script>
```

{% Aside %} In the DevTools, turn on "Show Shadow DOM" to be able to inspect ShadowRoots. {% endAside %}

What renders is "Root 2 FTW", despite the fact that we had already attached a shadow tree. This is because the last shadow tree added to a host, wins. It's a LIFO stack as far as rendering is concerned. Examining the DevTools verifies this behavior.

{% Aside %} Shadow trees added to a host are stacked in the order they're added, starting with the most recent first. The last one added is the one that renders. {% endAside %}

{% Aside %} The most recently added tree is called the **younger tree**. Previous trees are called **older trees**. In this example, `root2` is the younger tree and `root1` is the older tree. {% endAside %}

So what's the point of using multiple shadows if only the last is invited to the rendering party? Enter shadow insertion points.

Shadow Insertion Points

"Shadow insertion points" (`<shadow>`) are similar to normal insertion points (`<content>`) in that they're placeholders. However, instead of being placeholders for a host's *content*, they're hosts for other *shadow trees*. It's Shadow DOM Inception!

As you can probably imagine, things get more complicated the further you drill down the rabbit hole. For this reason, the spec is very clear about what happens when multiple `<shadow>` elements are in play:

{% Aside %} If multiple `<shadow>` insertion points exist in a shadow tree, only the first is recognized. The rest are ignored. {% endAside %}

Looking back to our original example, the first shadow `root1` got left off the invite list. Adding a `<shadow>` insertion point brings it back:

```
<div id="example2">Light DOM</div>
<script>
var container = document.querySelector('#example2');
var root1 = container.createShadowRoot();
var root2 = container.createShadowRoot();
root1.innerHTML = '<div>Root 1 FTW</div><content></content>';
**root2.innerHTML = '<div>Root 2 FTW</div><shadow></shadow>';**
</script>
```

There are a couple of interesting things about this example:

1. "Root 2 FTW" still renders above "Root 1 FTW". This is because of where we've placed the `<shadow>` insertion point. If you want the reverse, move the insertion point: `root2.innerHTML = '<shadow></shadow><div>Root 2 FTW</div>';`.

- Notice there's now a `<content>` insertion point in root1. This makes the text node "Light DOM" come along for the rendering ride.

What's rendered at `<shadow>`?

Sometimes it's useful to know the older shadow tree being rendered at a `<shadow>`. You can get a reference to that tree through `.olderShadowRoot`:

```
**root2.olderShadowRoot** === root1 //true
```

Obtaining a host's shadow root

If an element is hosting Shadow DOM you can access its youngest shadow root using `.shadowRoot`:

```
var root = host.createShadowRoot();
console.log(host.shadowRoot === root); // true
console.log(document.body.shadowRoot); // null
```

If you're worried about people crossing into your shadows, redefine `.shadowRoot` to be null:

```
Object.defineProperty(host, 'shadowRoot', {
  get: function() { return null; },
  set: function(value) { }
});
```

A bit of a hack, but it works. In the end, it's important to remember that while amazingly fantastic, **Shadow DOM has not been designed to be a security feature**. Don't rely on it for complete content isolation.

Building Shadow DOM in JS

If you prefer building DOM in JS, `HTMLContentElement` and `HTMLShadowElement` have interfaces for that.

```
<div id="example3">
  <span>Light DOM</span>
</div>
<script>
var container = document.querySelector('#example3');
var root1 = container.createShadowRoot();
var root2 = container.createShadowRoot();

var div = document.createElement('div');
div.textContent = 'Root 1 FTW';
root1.appendChild(div);

 // HTMLContentElement
var content = document.createElement('content');
content.select = 'span'; // selects any spans the host node contains
root1.appendChild(content);

var div = document.createElement('div');
div.textContent = 'Root 2 FTW';
root2.appendChild(div);

// HTMLShadowElement
var shadow = document.createElement('shadow');
root2.appendChild(shadow);
</script>
```

This example is nearly identical to the one in the previous section. The only difference is that now I'm using `select` to pull out the newly added `<span>`.

Working with insertion points

Nodes that are selected out of the host element and "distribute" into the shadow tree are called…drumroll…distributed nodes! They're allowed to cross the shadow boundary when insertion points invite them in.

What's conceptually bizarre about insertion points is that they don't physically move DOM. The host's nodes stay intact. Insertion points merely re-project nodes from the host into the shadow tree. It's a presentation/rendering thing: ~~"Move these nodes over here"~~ "Render these nodes at this location."

{% Aside %} You cannot traverse the DOM into a `<content>`. {% endAside %}

For example:

```
<div><h2>Light DOM</h2></div>
<script>
var root = document.querySelector('div').createShadowRoot();
root.innerHTML = '<content select="h2"></content>';

var h2 = document.querySelector('h2');
console.log(root.querySelector('content[select="h2"] h2')); // null;
console.log(root.querySelector('content').contains(h2)); // false
</script>
```

Voilà! The `h2` isn't a child of the shadow DOM. This leads to another tid bit:

{% Aside %} Insertion points are incredibly powerful. Think of them as a way to create a "declarative API" for your Shadow DOM. A host element can include all the markup in the world, but unless I invite it into my Shadow DOM with an insertion point, it's meaningless. {% endAside %}

Element.getDistributedNodes()

We can't traverse into a `<content>`, but the `.getDistributedNodes()` API allows us to query the distributed nodes at an insertion point:

```
<div id="example4">
  <h2>Eric</h2>
  <h2>Bidelman</h2>
  <div>Digital Jedi</div>
  <h4>footer text</h4>
```

```
</div>

<template id="sdom">
  <header>
    <content select="h2"></content>
  </header>
  <section>
    <content select="div"></content>
  </section>
  <footer>
    <content select="h4:first-of-type"></content>
  </footer>
</template>

<script>
var container = document.querySelector('#example4');

var root = container.createShadowRoot();

var t = document.querySelector('#sdom');
var clone = document.importNode(t.content, true);
root.appendChild(clone);

var html = [];
[].forEach.call(root.querySelectorAll('content'), function(el) {
  html.push(el.outerHTML + ': ');
  var nodes = el.getDistributedNodes();
  [].forEach.call(nodes, function(node) {
    html.push(node.outerHTML);
  });
  html.push('\n');
});
</script>
```

Element.getDestinationInsertionPoints()

Similar to `.getDistributedNodes()`, you can check what insertion points a node is distributed into by calling its `.getDestinationInsertionPoints()`:

```
<div id="host">
  <h2>Light DOM
</div>

<script>
  var container = document.querySelector('div');

  var root1 = container.createShadowRoot();
  var root2 = container.createShadowRoot();
  root1.innerHTML = '<content select="h2"></content>';
  root2.innerHTML = '<shadow></shadow>';

  var h2 = document.querySelector('#host h2');
  var insertionPoints = h2.getDestinationInsertionPoints();
  [].forEach.call(insertionPoints, function(contentEl) {
    console.log(contentEl);
  });
</script>
```

Tool: Shadow DOM Visualizer

Understanding the black magic that is Shadow DOM is difficult. I remember trying to wrap my head around it for the first time.

To help visualize how Shadow DOM rendering works, I've built a tool using d3.js. Both markup boxes on the left-hand side are editable. Feel free to paste in your own markup and play around to see how things work and insertion points swizzle host nodes into the shadow tree.

> {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/vby0ZDehF9RhL3Ouuupk.png", alt="Shadow DOM Visualizer", width="500", height="207" %}
>
> Launch Shadow DOM Visualizer

Give it a try and let me know what you think!

Event Model

Some events cross the shadow boundary and some do not. In the cases where events cross the boundary, the event target is adjusted in order to maintain the encapsulation that the shadow

root's upper boundary provides. That is, **events are retargeted to look like they've come from the host element rather than internal elements to the Shadow DOM**.

{% Aside %} Access `event.path` to see the adjusted event path. {% endAside %}

Play Action 1

- This one is interesting. You should see a `mouseout` from the host element (`<div data-host>`) to the blue node. Even though it's a distributed node, it's still in the host, not the ShadowDOM. Mousing further down into yellow again causes a `mouseout` on the blue node.

Play Action 2

- There is one `mouseout` that appears on host (at the very end). Normally you'd see `mouseout` events trigger for all of the yellow blocks. However, in this case these elements are internal to the Shadow DOM and the event doesn't bubble through its upper boundary.

Play Action 3

- Notice that when you click the input, the `focusin` doesn't appear on the input but on the host node itself. It's been retargeted!

Events that are always stopped

The following events never cross the shadow boundary:

- abort
- error
- select
- change
- load
- reset
- resize
- scroll
- selectstart

Conclusion

I hope you'll agree that **Shadow DOM is incredibly powerful**. For the first time ever, we have proper encapsulation without the extra baggage of `<iframe>`s or other older techniques.

Shadow DOM is certainly complex beast, but it's a beast worth adding to the web platform. Spend some time with it. Learn it. Ask questions.

If you want to learn more, see Dominic's intro article Shadow DOM 101 and my Shadow DOM 201: CSS & Styling article.

□□□ #2
□ □□□ □□□ 2026-10-31 22:15:54 UTC
□ □□□ □□□ 2026-10-31 22:15:56 UTC