Working with the Shadow DOM

The shadow DOM is not the dark side of the DOM, but if it were I would definitely give in to my hatred of the lack of encapsulation the DOM normally affords and cross over.

One of the aspects of the DOM that makes development of widgets/components difficult is this lack of encapsulation. For instance, one major problem has always been CSS rules bleeding into or out of a component’s branch of the DOM tree: it forces one to write overly specific selectors or abuse !important so that styles do not conflict, and even then conflicts still happen in large applications. Another issue caused by lack of encapsulation is that code external to a component can still traverse into the component’s branch of the DOM tree. These problems and others can be prevented by using the shadow DOM.

What Is the Shadow DOM?

So what exactly is this mysterious-sounding shadow DOM? According to [the W3C](http://www.w3.org/TR/components-intro/):

*Shadow DOM is an adjunct tree of DOM nodes. These shadow DOM subtrees can be associated with an element, but do not appear as child nodes of the element. Instead the subtrees form their own scope. For example, a shadow DOM subtree can contain IDs and styles that overlap with IDs and styles in the document, but because the shadow DOM subtree (unlike the child node list) is separate from the document, the IDs and styles in the shadow DOM subtree do not clash with those in the document.*

So what benefits does the shadow DOM provide to developers? It essentially provides encapsulation for a subtree from the parent page. This subtree can contain markup, CSS, JavaScript, or any asset that can be included in a web page. This allows you to create widgets without being concerned that of any of the assets will impact the parent page, or vice versa. Previously this level of encapsulation was only achievable by using an <iframe>.

Shadow DOM Basics

The shadow DOM is a simple concept, but it has some intricacies that make it appear more complex than it really is. This section will focus on the basics. The intricacies that afford the developer even more control and power will be covered later in this chapter.

Shadow Host

A shadow host is a DOM node that contains a shadow root. It is a regular element node within the parent page that hosts the scoped shadow subtree. Any child nodes that reside under the shadow host are still selectable, with the exception of the shadow root.

Shadow Root

A shadow root is an element that gets added to a shadow host. The shadow root is the root node for the shadow DOM branch. Shadow root child nodes are not returned by DOM queries even if a child node matches the given query selector. Creating a shadow root on a node in the parent page makes the node upon which it was created a shadow host.

Creating a shadow root

Creating a shadow root is a straightforward process. First a shadow host node is selected, and then a shadow root is created in the shadow host.

**TIP**

To inspect shadow DOM branches using the Chrome debugger, check the “Show Shadow DOM” box under the “Elements” section in the “General” settings panel of the debugger.

The code to create a shadow root looks like this:

**<div** id="host"**></div>**

**var** host = document.querySelector('#host');

**var** root = host.createShadowRoot();

**WARNING**

If you do not prefix createShadowRoot with “webkit” in Chrome 34 and below you are going to have a [bad time](http://caniuse.com/shadowdom). All calls to createShadowRoot should look like host.webkitCreateShadowRoot().

It is possible to attach multiple shadow roots to a single shadow host. However, only the last shadow root attached is rendered. A shadow host follows the LIFO pattern (last in, first out) when attaching shadow roots. At this point you might be asking yourself, “So what is the point of hosting multiple shadow roots if only the last one attached is rendered?” Excellent question, but you are getting ahead of the game! This will be covered later in this chapter (see [“Shadow Insertion Points”](https://www.oreilly.com/library/view/modern-javascript/9781491971420/ch05.html#shadow-insertion)).

Using a Template with the Shadow DOM

Using a template to populate a shadow root involves almost the same process as using a template to add content to a DOM node in the parent page. The only difference is that the template.content is added to the shadow root.

The first step is to create a template node. This example leverages the template from the previous chapter, with the addition of an element that will be the shadow host:

**<head>**

**<template** id="atcq"**>**

**<p** class="response"**></p>**

**<script** type="text/javascript"**>**

(**function** () {

**var** p = confirm('You on point Tip?');

**var** responeEl = document.querySelector('#atcq-root')

.shadowRoot

.querySelector('.response');

**if** (p) {

responeEl.innerHTML = 'All the time Phife';

} **else** {

responeEl.innerHTML = 'Check the rhyme y\'all';

}

})();

**</script>**

**</template>**

**</head>**

**<body>**

**<div** id="atcq-root"**></div>**

**</body>**

Next, we create a shadow root using the shadow host element, get a reference to the template node, and finally append the template content to the shadow root:

*// create a shadow root*

**var** root = document.querySelector('#atcq-root').createShadowRoot();

*// get a reference to the template node*

**var** template = document.querySelector('#atcq');

*// append the cloned content to the shadow root*

root.appendChild(template.content);

Shadow DOM Styling

I cannot count the number of times I have encountered CSS scoping issues throughout my career. Some of them were due to broad selectors such as div, the overusage of !important, or improperly namespaced CSS. Other times it has been difficult to override widget CSS or widget CSS has bled out, impacting application-level CSS. As an application grows in size, especially if multiple developers are working on the code base, it becomes even more difficult to prevent these problems. Good standards can help to mitigate these issues, but most applications leverage open source libraries such as jQuery UI, Kendo UI, Bootstrap, and others, which makes good standards alone inadequate. Addressing these problems and providing a standard way of applying styles to scoped elements are two of the benefits of using a shadow DOM.

Style Encapsulation

Any styles defined in the shadow DOM are scoped to the shadow root. They are not applied to any elements outside of this scope, even if their selector matches an element in the parent page. Styles defined outside of a shadow DOM are not applied to elements in the shadow root either.

In the example that follows, the text within the <p> that resides outside of the shadow root will be blue because that style is external to the shadow DOM that is created. The text within the <p> that is a child node of the shadow root will initially be the default color. This is because the styles defined outside of the shadow root are not applied to elements within the shadow root. After two seconds, the text within the <p> inside the shadow root will turn green, because the callback for the setTimeout function injects a <style> tag into the shadow root. The text within the <p> that resides outside of the shadow root will remain blue because the style injected into the shadow root is scoped to elements that are children of the shadow root. Here’s the code that achieves this styling:

**<head>**

**<style>**

**p** {

**color**: blue;

}

**</style>**

**<template><p>**I am the default color, then green.**</p></template>**

**</head>**

**<body>**

**<div** id="host"**></div>**

**<p>**I am blue.**</p>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('template');

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

setTimeout(**function** () {

root.innerHTML += '<style>p { color: green; }</style>';

}, 2000)

**</script>**

**</body>**

Styling the Host Element

In some cases you will want to style the host element itself. This is easily accomplished by creating a style anywhere within the parent page, because the host element is not part of the shadow root. This works fine, but what if you have a shadow host that needs different styling depending on the contents of the shadow root? And what if you have multiple shadow hosts that need to be styled based on their contents? As you can imagine, this would get very difficult to maintain. Fortunately, there is a new selector, :host, that provides access to the shadow host from within the shadow root. This allows you to encapsulate your host styling to the shadow root:

**<head>**

**<template** id="template"**>**

**<style>**

:host {

**border**: 1px solid red;

**padding**: 10px;

}

**</style>**

My host element will have a red border!

**</template>**

**</head>**

**<body>**

**<div** id="host"**></div>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('#template')

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

**</script>**

**</body>**

The parent page selector has a higher specificity than the :host selector, so it will trump any shadow host styles defined within the shadow root:

**<head>**

**<style>**

#host {

**border**: 1px solid green;

}

**</style>**

**<template** id="template"**>**

**<style>**

:host {

**border**: 1px solid red;

**padding**: 10px;

}

**</style>**

My host element will have a green border!

**</template>**

**</head>**

**<body>**

**<div** id="host"**></div>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('#template')

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

**</script>**

**</body>**

If you want to override styles set in the parent page, this must be done inline on the host element:

**<head>**

**<template** id="template"**>**

**<style>**

:host {

**border**: 1px solid red;

**padding**: 10px;

}

**</style>**

My host element will have a blue border!

**</template>**

**</head>**

**<body>**

**<div** id="host" style="border: 1px solid blue;"**></div>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('#template')

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

**</script>**

**</body>**

The :host selector also has a functional form that accepts a selector, :host*selector*, allowing you to set styles for specific hosts. This functionality is useful for theming and managing states on the host element.

Styling Shadow Root Elements from the Parent Page

Encapsulation is all well and good, but what if you want to target specific shadow root elements with a styling update? What if you want to reuse templates and shadow host elements in a completely different application? What if you do not have control over the shadow root’s content? For instance, you could be pulling the code from a repository that is maintained by another department internal to your organization, or a shared repository that is maintained by an external entity. In either case you might not have control over the shadow root’s contents, or the update process might take a significant amount of time, which would block your development. Additionally, you might not *want* control over the content. Sometimes it is best to let domain experts maintain certain modules and to simply override the default module styling to suit your needs. Fortunately, the drafters of the W3C specification thought of these cases (and probably many more), so they created a selector that allows you to apply styling to shadow root elements from the parent page.

The ::shadow pseudoelement selects the shadow root, allowing you to target child elements within the selected shadow root:

**<head>**

**<style>**

#host::shadow **p** {

**color**: blue;

}

**</style>**

**<template><p>**I am blue.**</p></template>**

**</head>**

**<body>**

**<div** id="host"**></div>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('template');

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

**</script>**

**</body>**

The ::shadow pseudoelement selector can be used to style nested shadow roots:

**<head>**

**<style>**

#parent-host::shadow #child-host::shadow **p** {

**color**: blue;

}

**</style>**

**<template** id="child-template"**><p>**I am blue.**</p></template>**

**<template** id="parent-template"**>**

**<p>**I am the default color.**</p>**

**<div** id="child-host"**></div>**

**</template>**

**</head>**

**<body>**

**<div** id="parent-host"**></div>**

**<script** type="text/javascript"**>**

**var** parentTemplate = document.querySelector('#parent-template');

**var** childTemplate = document.querySelector('#child-template');

**var** parentRoot = document.querySelector('#parent-host')

.createShadowRoot();

**var** childRoot;

parentRoot.appendChild(parentTemplate.content);

childRoot = parentRoot.querySelector('#child-host').createShadowRoot();

childRoot.appendChild(childTemplate.content);

**</script>**

**</body>**

Sometimes targeting individual shadow roots using the ::shadow pseudoelement is very inefficient, especially if you are applying a theme to an entire application of shadow roots. Again, the drafters of the W3C specification had the foresight to anticipate this use case and specified the /deep/ combinator. The /deep/ combinator allows you cross through all shadow roots with a single selector:

**<style>**

*/\* colors all <p> text within all shadow roots blue \*/*

**body** /**deep**/ **p** {

**color**: blue;

}

*/\* colors all <p> text within the child shadow root blue \*/*

#parent-host /**deep**/ #child-host # **p** {

**color**: blue;

}

*/\* targets a library theme/skin \*/*

**body** /**deep**/ **p.skin** {

**color**: blue;

}

**</style>**

**NOTE**

At this point you might be asking yourself, “Doesn’t this defeat the purpose of encapsulation?” But encapsulation does not mean putting up an impenetrable force field that makes crossing boundaries for appropriate use cases, such as theming UI components, impossible. The problem with the Web is that it has never had a formalized method of encapsulation or a defined API for breaking through an encapsulated component, like in other development platforms. The formalization of encapsulation and associated methods makes it clear in the code what the developer’s intent is when encapsulation is breached. It also helps to prevent the bugs that plague a web platform that lacks formalized encapsulation.

Content Projection

One of the main tenets of web development best practices is the separation of content from presentation, the rationale being that it makes application maintenance easier and more accessible.

In the past separation of content from presentation has simply meant not placing styling details in markup. The shadow DOM takes this principle one step further.

In the examples we have seen thus far the content has been contained within a template and injected into the shadow root. In these examples no significant changes were made to the presentation, other than the text color. Most cases are not this simple.

In some cases it is necessary to place the content inside of the shadow host element for maintenance and accessibility purposes. However, that content needs to be projected into the shadow root in order to be presented. Luckily, the building blocks for projecting the content from the shadow host into the shadow root exist.

Projection via a Content Tag

One way to project content from the shadow host into the shadow root is by using a <content> element inside of a <template>. Any content inside the shadow host will be automatically projected into the <content> of the <template> used to compose the shadow root:

**<head>**

**<meta** charset="utf-8"**>**

**<title>**Test Code**</title>**

**<template>**

**<p>**I am NOT projected content.**</p>**

**<content></content>**

**</template>**

**</head>**

**<body>**

**<div** id="host"**>**

**<p>**I am projected content.**</p>**

**</div>**

**<script** type="text/javascript"**>**

**var** template = document.querySelector('template');

**var** root = document.querySelector('#host').createShadowRoot();

root.appendChild(template.content);

**</script>**

**</body>**

Projection via Content Selectors

In some cases you may not want to project all of the content from the shadow host. You might want to select specific content for injection in different <content> elements in the shadow root. A common case is that some markup in the shadow host has semantic meaning and helps with accessibility, but doesn’t add anything to the presentation of the shadow root. The mechanism for extracting content from the shadow host is the select attribute. This attribute can be added to a <content> element with a query selector value that will match an element in the shadow host. The matched element’s content is then injected into the <content> tag.

**TIP**

Only the first element matched by a <content> element’s select attribute is injected into the element—keep this in mind when using selectors that are likely to match a number of elements, such as tag selectors (e.g., div).

The following example is a product listing with a review widget. The shadow host contains semantic markup that is accessible, and the template contains the presentation details. The template presentation does not lend itself well to accessibility and contains extraneous markup that is used for presentation purposes only—e.g., the column containers are there for positioning purposes only and the review <ul> items do not contain text (the interface would be purely graphical). In this example, select attributes are used in the <template> <content> elements to extract content from the shadow root:

*<!--*

*Only the relevant markup is shown. All other details,*

*such as template CSS and JavaScript, have been omitted,*

*so that the focus is on the selector projection use case.*

*-->*

**<template>**

**<div** class="product"**>**

**<div** class="column main"**>**

**<content** select="h2"**></content>**

**<content** select=".description"**></content>**

**</div>**

**<div** class="column sidebar"**>**

**<content** select="h3"**></content>**

**<ul** class="ratings"**>**

**<li** class="1-star"**></li>**

**<li** class="2-star"**></li>**

**<li** class="3-star"**></li>**

**<li** class="4-star"**></li>**

**</ul>**

**</div>**

**</div>**

**</template>**

**<div** id="host" class="product"**>**

**<h2>**ShamWow**</h2>**

**<p** class="description"**>**

ShamWow washes, dries, and polishes any surface. It's like a towel,

chamois, and sponge all in one!

**</p>**

**<h3>**Ratings**</h3>**

**<ul** class="ratings"**>**

**<li>**1 star**</li>**

**<li>**2 stars**</li>**

**<li>**3 stars**</li>**

**<li>**4 stars**</li>**

**</ul>**

**</div>**

**WARNING**

Only nodes that are children of the shadow host can be projected, so you cannot select content from any lower descendant in the shadow host.

Getting Distributed Nodes and Insertion Points

Nodes that are projected from a host are referred to as *distributed nodes*. These nodes do not actually move locations in the DOM, which makes sense because the same host child node can be projected to different insertion points across shadow roots. As you can imagine, things can get complicated rather quickly, and at times you may need to do some inspecting and take action on distributed nodes in your application. There are two different methods that support this, Element.getDistributedNodes and Element.getDestinationInsertionPoints.

**NOTE**

You cannot traverse into a <content> tree, because a <content> node does not have any descendant nodes. It is helpful to think of a <content> node as a television that is displaying a program. The television’s only role in producing the program is to display it. The program itself was filmed and edited elsewhere for consumption by an unlimited number of televisions.

We use these methods as follows:

*// Element.getDistributedNodes*

**var** root = document.querySelector('#some-host').createShadowRoot();

*// iterate over all the content nodes in the root*

[].forEach.call(root.querySelectorAll('content'), **function** (contentNode) {

*// get the distributed nodes for each content node*

*// and iterate over the distributed nodes*

[].forEach.call(contentNode.getDistributedNodes(),

**function** (distributedNode) {

*// do something cool with the contentNode*

});

});

*// Element.getDestinationInsertionPoints*

**var** hostChildNode = document.querySelector('#some-host .some-child-node');

*// get child node insertion points and iterate over them*

[].forEach.call(hostChildNode.getDestinationInsertionPoints(),

**function** (contentNode) {

*// do something cool with the contentNode*

});

Shadow Insertion Points

In the previous section we examined how shadow host content can be projected into insertion points, <content>. Just as content can be projected into insertion points, so can shadow roots. Shadow root insertion points are defined using <shadow> tags. Like any other tags, these can be created directly in markup or added to the DOM programmatically.

**NOTE**

Shadow roots are stacked in the order they are added, with the youngest shadow root tree appearing last and rendering. Trees appearing earlier in the stack are referred to as *older trees*, while trees appearing after a given shadow root are referred to as *younger trees*.

At the beginning of this chapter it was stated that a shadow host could contain multiple shadow roots, but that only the last shadow root defined would be rendered. This is true in the absence of <shadow> elements. The <shadow> tag provides a point for projecting an older shadow root using a younger shadow root tree. If a shadow root tree contains more than one <shadow> element, the first one is used and the rest are ignored. Essentially, <shadow> allows you to render an older shadow root in a stack by providing an insertion point for it to be projected into.

**NOTE**

Projecting nodes to an insertion point does not affect the tree structure. The projected nodes remain in their original locations within the tree. They are simply displayed in the assigned insertion point.

Here’s an example:

**<template** id="t-1"**>**I am t1. **</template>**

**<template** id="t-2"**><shadow></shadow>**I am t2. **</template>**

**<template** id="t-3"**><shadow></shadow>**I am t3. **</template>**

**<div** id="root"**></div>**

**<script** type="text/javascript"**>**

(**function** () {

**var** t1 = document.querySelector('#t-1');

**var** t2 = document.querySelector('#t-2');

**var** t3 = document.querySelector('#t-3');

**var** host = document.querySelector('#root')

**var** r1 = host.createShadowRoot();

**var** r2 = host.createShadowRoot();

**var** r3 = host.createShadowRoot();

r1.appendChild(t1.content);

r2.appendChild(t2.content);

r3.appendChild(t3.content);

})();

**</script>**

*<!-- renders: "I am t1. I am t2. I am t3." -->*

The previous code block renders from the bottom (youngest tree) up, projecting the next-oldest shadow root tree into the <shadow> insertion point. This was a very simple example. In a real application, the code will be more complicated and dynamic. Because of this it is helpful to have a way to inspect a <shadow> programmatically or to determine a shadow host’s root:

*// using the previous code block as an example*

*// determine older shadow root*

r1.olderShadowRoot === **null**; *// true; first in the stack*

r2.olderShadowRoot === r1; *// true*

r3.olderShadowRoot === r2; *// true*

*// determine a host's shadow root*

host.shadowRoot === r1; *// false; there can only be one (LIFO)*

host.shadowRoot === r2; *// false; ditto*

host.shadowRoot === r3; *// true*

*// determine a shadow root's host*

r1.host === host; *// true*

r2.host === host; *// true*

r3.host === host; *// true*

Events and the Shadow DOM

At this point you might be thinking that projecting nodes instead of cloning them is a great optimization that will help to keep changes synchronized—but what about events bound to these projected nodes? How exactly does this work if they are not copied? In order to normalize these events, they are sometimes [retargeted](http://bit.ly/dwc-shadow-dom) to appear as if they were triggered by the host element rather than the projected element in the shadow root. In these cases you can still determine the shadow root of the projected node by examining the path property of the event object. Some events are never retargeted, though, which makes sense if you think about it. For instance, how would a scroll event be retargeted? If a user scrolls one projected node, should the others scroll? The events that are not retargeted are:

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

Updating the Dialog Template to Use the Shadow DOM

You might have noticed generic id values such as title and content were used in the dialog component, and you probably thought, “This idiot is going to have duplicate id values, which are supposed to be unique in the DOM, colliding left and right…” This was intentional, though, and has been leading up to this moment!

One of the benefits of the shadow DOM is the encapsulation of markup, which means the encapsulation of id values and a decrease in the likelihood of id value collisions in your application.

This code will leverage the previous chapter’s code that demonstrated using a template to make the dialog component markup and JavaScript inert until it was appended to the DOM.

Dialog Markup

The dialog component will utilize a template, like the previous example, but this template will be appended to a shadow root that is hosted by <div id="dialog-host">. The interesting part about this example is that it is practically the reverse of our review widget example in terms of accessibility and readability. The [aria](http://bit.ly/dwc-aria) attributes are contained within the shadow DOM, and the markup a developer writes is not exactly semantic. However, if you think about it, the aria attributes are primarily used for accessibility implementation details, so it makes sense that these details are obfuscated from the developer. The part that does not make sense is that the host markup is not very semantic, but please reserve judgment on that until the next chapter!

Here’s the code for our updated dialog template:

**<head>**

**<script** type="text/javascript" src="/vendor/jquery.js"**></script>**

**<template** id="dialog"**>**

**<style>**

// styling src

**</style>**

**<script** type="text/javascript"**>**

*// dialog component source*

**</script>**

**<div** role="dialog" aria-labelledby="title" aria-describedby="content"**>**

**<h2** id="title"**></h2>**

**<p** id="content"**></p>**

**</div>**

**</template>**

**</head>**

*<!-- example host node -->*

**<div** id="dialog-host"**>**

**<h2>**I am a title**</h2>**

**<p>**Look at me! I am content.**</p>**

**</div>**

Dialog API

If you want to encapsulate the creation of the shadow root, the cloning and appending of the template, and the dialog component instantiation, then it is best to create a wrapper constructor function that encapsulates all of these implementation details:

**function** DialogShadow(options) {

**this**.options = options;

*// get the host node using the hostQrySelector option*

**this**.host = document.querySelector(options.hostQrySelector);

*// grab the template*

**this**.template = document.querySelector('#dialog');

**this**.root = **this**.host.createShadowRoot();

*// append the template content to the root*

**this**.root.appendChild(**this**.template.content);

*// get a reference to the dialog container element in the root*

**this**.el = **this**.root.querySelector('[role="dialog"]');

**this**.options.$el = **this**.el;

*// align element to body since it is a fragment*

**this**.options.alignToEl = document.body;

**this**.options.align = 'M';

*// do not clone node*

**this**.options.clone = **false**;

*// get the content from the host node; projecting would retain host*

*// node styles and not allow for encapsulation of template styles*

**this**.el.querySelector('#title').innerHTML = **this**.host.querySelector('h2')

.innerHTML;

**this**.el.querySelector('#content').innerHTML = **this**.host.querySelector('p')

.innerHTML;

*// create a dialog component instance*

**this**.api = **new** Dialog(**this**.options);

**return** **this**;

}

Updating the Dialog show Method

Since the shadow root and its children are a subtree that is not part of the parent document, we have to ensure that the host element’s z-index value is modified so that it appears on the top of its stacking context, the <body>:

*// see GitHub repo for full example*

(**function** (window, $, Voltron, Duvet, Shamen, ApacheChief, jenga) {

'use strict';

*// makes dialog visible in the UI*

Dialog.prototype.show = **function** () {

*// this will adjust the z-index, set the display property,*

*// and position the dialog*

**this**.overlay.position();

*// bring the host element to the top of the stack*

jenga.bringToFront(**this**.$el[0].parentNode.host);

};

})(window, jQuery, Voltron, Duvet, Shamen, ApacheChief, jenga);

Instantiating a Dialog Component Instance

The dialog component can now be instantiated just as before, but it will now be scoped to the shadow root:

**var** dialog = **new** DialogShadow({

draggable: **true**,

resizable: **true**,

hostQrySelector: '#dialog-host'

});

dialog.api.show();

Shadow DOM Structure Xpath

Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree. Shadow DOM enables encapsulation. With Shadow DOM, a component can have its own â€œshadowâ€� DOM tree that cannot be accidentally accessed from the main document. This subtree may have local style rules and more.

Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which you can attach any element, in the same way as the normal DOM.

In this JavaScript executor is used, normal xpaths would not work.

JavaScriptExecutor jse = (JavascriptExecutor)driver;

WebElement username = (WebElement) jse.executeScript("return document.querySelector("body > downloads-manager").shadowRoot.querySelector("#toolbar").shadowRoot.querySelector("#toolbar").shadowRoot.querySelector("#search").shadowRoot.querySelector("#searchInput")");

WebElement password = (WebElement) jse.executeScript("return document.querySelector("body > downloads-manager").shadowRoot.querySelector("#toolbar").shadowRoot.querySelector("#moreActions").shadowRoot.querySelector("#icon > iron-icon").shadowRoot.querySelector("svg > g > path")");

WebElement resetPwdLink= (WebElement) jse.executeScript("return document.querySelector("body > downloads-manager").shadowRoot.querySelector("#toolbar").shadowRoot.querySelector("#moreActions").shadowRoot.querySelector("#icon > iron-icon").shadowRoot.querySelector("svg > g > path")");

Sring js = "arguments[0].setAttribute('value', 'vaicha.oracle@gmail.com')";

((JavaScriptExecutor)driver).executeScript(js, username);

((JavaScriptExecutor)driver).executeScript(js, password);

((JavaScriptExecutor)driver).executeScript("arguments[0].click();", resetPwdLink);

}