Web Components: what they are and where are they?

Web Components are a set of specifications that address one of the biggest shortcomings of HTML. We're stuck with what the spec says we have and the browser implementors decide to give us...

At least that's the theory.

Web Components allow developers to create their own elements along with the styles and scripts associated with the elements. These custom elements would be treated the same as the built-in elements available on browsers, as long as the browser supports the APIs.

Early version of the web components family of specifications included the following specifications:

- Custom Elements
- HTML Imports
- Shadow DOM
- HTML Templates

The initial proposals from Google had a lot of pushback from Apple and Mozilla, particularly the concept of HTML Imports. They wanted to wait what would ECMAScript modules would look like and how they would work with non-HTML resources. We're still waiting...

This has changed considerably since the first introduction of the concept.

- Shadow DOM
    - Most of the parts are now maintained in DOM Standard, and are called shadow trees
- Custom Elements
    - Custom elements were upstreamed into the HTML Standard (and bits in the DOM Standard) and are maintained there
- HTML Templates
    - HTML Templates were upstreamed into the HTML Standard

- JSON, CSS, HTML Modules
    - Successor to the abandoned HTML Imports, allows JSON, CSS, and HTML markup to be requested by a component
    - HTML Modules Spec work is being incubated in the whatwg/html repository (see PR)
    - See the following docs:
        - HTML Modules explainer
        - initial proposal
        - earlier design ideas
- CSS changes
    - The CSS WG works on CSS Scoping and CSS Shadow Parts, which help dealing with shadow trees with various selectors
    - Various other parts of CSS and its object model are also impacted by shadow trees and directly worked on in the various CSS specificaions

There have also been two versions of the web components specifications.

V0 was the original version of web components that shipped in older Chromium browsers; it has been deprecated in Chromium browsers and is no longer supported.

V1 was developed as a replacement for V0 and incorporated feedback from the community and lessons learned from the original designs.

A component created with Polymer 3.0 looks like this:

```
import {PolymerElement, html} from '@polymer/polymer/polymer-element.js';

class XCustom extends PolymerElement {

  // Optional Shadow DOM template
  static get template() {
    return html`
      <style>
        /* CSS rules for your element */
      </style>

        <!-- shadow DOM for your element -->

      <div>[[greeting]]</div>/* CSS rules for your element */`
```

```
      <style>
        /* CSS rules for your element */
      </style>

        <!-- shadow DOM for your element -->

      <div>[[greeting]]</div> <!-- data bindings in shadow DOM -->
    `eeting = 'Hello!';
  }
  greetMe() {
    console.log(this.greeting);
  }

}

customElements.define('x-custom', XCustom);
'Hello!'
```

Then you would add the custom element to your page like any other HTML element

```
<app-drawer></app-drawer>
```

An example of a web component created using plain Javascript looks like this. Note that the main difference is the element it extends, `HTMLElement` rather than `PolymerElement`

```
class VCard extends HTMLElement {
  constructor() {
    super();

    this.heading = "";
    this.subheading = "";
  }

  connectedCallback() {
```

```
    this.heading = this.getAttribute("heading");
    t""s.subheading = this.getAttribute("subheading");

    this.render();
  }

  render() {
    this.innerHTML = "subheading"`
      <div style="text-align: center; font-family: sans-serif">
        <h1>${this.heading}</h1>
        <p>${this.subheading}</p>
      </div>
      `;
  }
}

customElements.define("v-card", VCard);
```

And include the element in HTML passing the values for the fields we want to use as attribute/value pairs.

```
<v-card
  heading="Carlos Araya"
  subheading="Technical Writer"
></v-card>
```

Shadow DOM

The idea has always been that the styles for a web components are encapsulated to the element but that makes it very hard to style elements based on the page's themes or styles.

The Shadow DOM V0 specification had several pseudo elements and classes that would allow designers to style the shadow DOM from the host page, but they were deprecated as they defeat the purpose of encapsulating styles to an element.

Shadown Parts

Since the removal of the piercing combinators, the current best practice is to use the ::part pseudo elements to style custom elements from outside. Using parts requires we add the `part` attribute in the custom element HTML template. Like the `class` attribute, `part` can take a list of values separated from spaces; we'll be able to match on them individually.

```
<template id="tabbed-custom-element">
<style type="text/css"><style type="text/css">
  /* Internal styles go here */
</style>
<div part="tab active">Tab 1</div>
<div part="tab">Tab 2</div>
<div part="tab">Tab 3</div>
</template>

<tabbed-custom-element></tabbed-custom-element>
```

Then, once we have instantiated the custom element, we can target the parts we defined in it using the `::part` pseudo element from an external style sheet.

`::part` takes the value we declared in the markup as the attribute and can be followed by other selectors.

Using parts would allow authors to add styles like the ones below to the stylesheet on the host page. To style the component above, we could use styles like the ones below:

```
tabbed-custom-element::part(tab) {
  color: #0c0dcc;
  border-bottom: transparent solid 2px;
}

tabbed-custom-element::part(tab):hover {
  background-color: #0c0d19;
  border-color: #0c0d33;
}
```

```
tabbed-custom-element::part(tab):hover:active {
  background-color: #0c0d33;
}

tabbed-custom-element::part(active) {
  color: #0060df;
  border-color: #0a84ff !important;
}
```

So now we have a component that we can style from the outside with minimal styling coming with the component in the shadow DOM.

Slots and composition

So far our custom elements are static. We've hardcoded the content of our elements and styled them based on what the template offers. Using slots we can mix elements from inside and outside the template to create more flexible elements.

The template below uses two slots by using the `slot` element with a name attribute. This will allow you to compose your content with both shadow DOM and "Light" or regular DOM.

In the code below, the shadow DOM provides the slots' destination using the name attribute and a basic structure if the instance of the custom element doesn't provide a value for the slot content.

```
<template id="element-details-template">
  <style><style>
    /*
      Private styles to the component go here
    */
  </style>
  <details>
    <summary>
      <span>
        <code class="name">&lt;<slot name="element-name">NEED NAME</slot>&
```

```
        <i class="desc">
          <slot name="description">NEED DESCRIPTION</slot>
        </i>
      </span>
    </summary>
    <div class="attributes">
      <h4><span>Attributes</span></h4>
      <slot name="attributes">
        <p>None</p>
      </slot>
    </div>
  </details>
  <hr>
</template>
```

The instance of the custom element fills the slot in the template using the value of the element with a matching slot value.

```
<element-details>
  <span slot="element-name">slot</span>
  <span slot="description">A placeholder
  inside a web component that users
  can fill with their own markup,
  with the effect of composing
  different DOM trees together.</span>
  <dl slot="attributes">
    <dt>name</dt>
    <dd>The name of the slot.</dd>
  </dl>
</element-details>
```

If you leave the instance blank, then the default values from the shadow DOM will be used.

Declarative Shadow DOM

Declarative Shadow DOM, first available in Chrome 90, presents an easier way to create shadow trees for custom elements.

By placing the template inside the custom element and declaring the shadow root using the `shadowroot` attribute with one of the two possible values for the attribute (open or closed).

```
<nineties-button>
  <template shadowr<template shadowroot="open">
      button {
        background-color: navy;
        color: white;
      }
    </style>
    <button>
      <slot></slot>
    </button>
  </template>
  I'm Blue
</nineties-button>
```

Using declarative shadow DOM also allows for server side rendering web components. Since you are no longer tied to Javascript to build the element's shadow root, you can render custom elements on the server and present them to the clients as any other server side rendered element.

As far as I know only Chromium browsers support declarative shadow DOM (tested in Chrome stable and Canary and Edge Canary) so we should use feature detection for the features.

```
function supportsDeclarativeShadowDOM() {
  return HTMLTemplateElement.prototype.hasOwnProperty('shadowRoot');
}

if (! supportsDeclarativeShadowDOM) {
  console.log('No support for DSD, boo!')
} else {
  console.log('we support DSD, do something')
}
```

See Declarative Shadow DOM for more information.

Importing web components

Up to this point we've built components that are embedded in the hosting document. They won't affect performance since the templates themselves are innert and are not processed until they are instantiated but they will still affect the weight of the page.

The question then becomes: How do we import web components defined outside our document?

HTML Imports provided a solution that allowed you to import self-contained web components as HTML files. The idea was rejected by Apple and Mozilla and eventually deprecated from Chrome (since version 73). So we're back at square one.

There is an interesting thread about HTML Modules in the WICG Github Repository but it's still far from having implementations available.

Here are the HTML Modules explainer and the Chrome Status Entry.

The criticisms

The criticisms for web components come from several angles. I've chosen to summarize some of them here.

Jeremy Keith

In Evaluating Technology, Jeremy Keith ask a very interesting question: ***How do web components fail?*** His answer to the question is illustrative of one of the problems with web components.

Until declarative shadow DOM came into the picture, web components were tied to Javascript to do anything.

If we created a component and instantiated it like this:

```
<image-gallery></image-gallery>
```

The assumption is that Javascript will hydrate the document witht content from the

shadow DOM and the styles encapsulated in the Template.

But what happens if we don't have Javascript enabled? Or if we loose connectivity on our mobile device or WI-FI?

Take the same element but populate it light DOM elements, in this case the images that will go inside the gallery.

If Javascript is enabled we'll get the enhanced version but if we don't then we get the images inside the custom element (HTML will ignore elements that it doesn't understand).

```
<image-gallery>
  <img src="..." alt="...">
  <img src="..." alt="...">
  <img src="..." alt="...">
</image-gallery>
```

The second element will also work for older browsers and provide a basic experience for browsers that don't support custom elements. **It is up to use to make web components fail well**.

Lea Verou

Lea Verou expresses her missgivings about web components in The failed promise of Web Components. Her concerns lie more in the realms of usability and best practices.

Her first point is reliance on Javascript makes it much harder it makes it to write good web components because people are not as comfortable or proficient in writing JS as they are writing HTML or CSS:

> ... HTML is a lower barrier to entry language. Far more people can write HTML than JS. Even for those who do eventually write JS, it often comes after spending years writing HTML & CSS.
>
> Lea Verou — The failed promise of web components

Traditional web components rely on Javascript but that doesn't mean we need

massive amount of Javascript to make them work, or as Lea puts it:

> Even when JS is unavoidable, it's not black and white. A well designed HTML element can reduce the amount and complexity of JS needed to a minimum. Think of the `<dialog>` element: it usually does require ***some*** JS, but it's usually rather simple JS. Similarly, the `<video>` element is perfectly usable just by writing HTML, and has a comprehensive JS API for anyone who wants to do fancy custom things.
>
> Lea Verou — The failed promise of web components

So, if we consider, these criticisms, where are we in the Web Components journey?

Still getting there

Things have evolved since I last looked at web components but some things remain the same, there are no real models to follow when creating web components or, to my knowledge, discussions about best practices and models we can follow when creating our own.

It would be nice to have a resource where we can do a best of breed set of components as Lea outlines on her post, particularly when she discusses possible solutions: Can we fix this?. I will paraphrase some of her solutions and add some of my own

- **Plug and play**. No dependencies, no setup beyond including one `<script>` tag
    - Include any essential dependency automatically (like a map library for a map custom element)
- Syntax and APIs follow the same design patterns that we see in HTML elements
    - See HTML APIs: What They Are And How To Design A Good One
    - You should be honest when deciding if your element needs an API at all
- **Anything that can be done without the component user writing JS, is doable without JS**, per the W3C principle of least power
    - We should explore if declarative shadow DOM is a solution
- **Accessible by default** via sensible ARIA defaults, just like normal HTML elements

        - Since we have to add our own ARIA attributes we should teach people how to use ARIA properly
        - No ARIA is better than bad ARIA
        - See To ARIA! The Cause of, and Solution to, All Our Accessibility Problems
    - **Themable** via ::part(), and regular CSS properties as much as possible
    - Individual elements should be **Composable** using slots
        - Component users should be able to add content to the custom element without having to edit the component itself
        - This user-side content becomes the full content of the element when the browser doesn't support custom elements or Javascript fails for some reason
        - Custom elements should provide sensible default for when no content is added to the light DOM.

While these are all sensible solutions, they only solve part of the problem dealing with web components.

We also need an education component to make web components successful.

We're past the stage where everyone wanted to experiment and create their own versions and see what they could do. We need a central collection of best practices and examples of how to apply them to our own web components.