Custom Elements v1: Reusable Web Components

By Eric Bidelman

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

TL;DR

With Custom Elements, web developers can **create new HTML tags**, beef-up existing HTML tags, or extend the components other developers have authored. The API is the foundation of web components. It brings a web standards-based way to create reusable components using nothing more than vanilla JS/HTML/CSS. The result is less code, modular code, and more reuse in our apps.

Introduction

Note: This article describes the new Custom Elements spec. If you've been using custom elements, chances are you're familiar with the version 0 that shipped in Chrome 33. The concepts are the same, but the version 1 spec has important API differences. Keep reading to see what's new or check out the section on History and browser support for more info.

The browser gives us an excellent tool for structuring web applications. It's called HTML. You may have heard of it! It's declarative, portable, well supported, and easy to work with. Great as HTML may be, its vocabulary and extensibility are limited. The HTML living standard has always lacked a way to automatically associate JS behavior with your markup... until now.

Custom elements are the answer to modernizing HTML, filling in the missing pieces, and bundling structure with behavior. If HTML doesn't provide the solution to a problem, we can create a custom element that does. **Custom elements teach the browser new tricks while preserving the benefits of HTML**.

Defining a new element

To define a new HTML element we need the power of JavaScript!

The `customElements` global is used for defining a custom element and teaching the browser about a new tag. Call `customElements.define()` with the tag name you want to create and a JavaScript `class` that extends the base `HTMLElement`.

Example - defining a mobile drawer panel, `<app-drawer>`:

```
class AppDrawer extends HTMLElement {...}
window.customElements.define('app-drawer', AppDrawer);

// Or use an anonymous class if you don't want a named constructor in current scope.
window.customElements.define('app-drawer', class extends HTMLElement {...});
```

Example usage:

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

It's important to remember that using a custom element is no different than using a `<div>` or any other element. Instances can be declared on the page, created dynamically in JavaScript, event listeners can be attached, etc. Keep reading for more examples.

Defining an element's JavaScript API

The functionality of a custom element is defined using an ES2015 `class` which extends `HTMLElement`. Extending `HTMLElement` ensures the custom element inherits the entire DOM API and means any properties/methods that you add to the class become part of the element's DOM interface. Essentially, use the class to create a **public JavaScript API** for your tag.

Example - defining the DOM interface of `<app-drawer>`:

```
class AppDrawer extends HTMLElement {

  // A getter/setter for an open property.
  get open() {
    return this.hasAttribute('open');
  }

  set open(val) {
    // Reflect the value of the open property as an HTML attribute.
    if (val) {
      this.setAttribute('open', '');
    } else {
      this.removeAttribute('open');
    }
    this.toggleDrawer();
  }
```

```
  // A getter/setter for a disabled property.
  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    // Reflect the value of the disabled property as an HTML attribute.
    if (val) {
      this.setAttribute('disabled', '');
    } else {
      this.removeAttribute('disabled');
    }
  }

  // Can define constructor arguments if you wish.
  constructor() {
    // If you define a constructor, always call super() first!
    // This is specific to CE and required by the spec.
    super();

    // Setup a click listener on <app-drawer> itself.
    this.addEventListener('click', e => {
      // Don't toggle the drawer if it's disabled.
      if (this.disabled) {
        return;
      }
      this.toggleDrawer();
    });
  }

  toggleDrawer() {
    ...
  }
}

customElements.define('app-drawer', AppDrawer);
```

In this example, we're creating a drawer that has an `open` property, `disabled` property, and a `toggleDrawer()` method. It also reflects properties as HTML attributes.

A neat feature of custom elements is that **`this` inside a class definition refers to the DOM element itself** i.e. the instance of the class. In our example, `this` refers to `<app-drawer>`. This (☺) is how the element can attach a `click` listener to itself! And you're not limited to event listeners. The entire DOM API is available inside element code. Use `this` to access the element's properties, inspect its children (`this.children`), query nodes (`this.querySelectorAll('.items')`), etc.

Rules on creating custom elements

1. The name of a custom element **must contain a dash (-)**. So `<x-tags>`, `<my-element>`, and `<my-awesome-app>` are all valid names, while `<tabs>` and `<foo_bar>` are not. This requirement is so the HTML parser can distinguish custom elements from regular elements. It also ensures forward compatibility when new tags are added to HTML.
2. You can't register the same tag more than once. Attempting to do so will throw a `DOMException`. Once you've told the browser about a new tag, that's it. No take backs.
3. Custom elements cannot be self-closing because HTML only allows a few elements to be self-closing. Always write a closing tag (`<app-drawer></app-drawer>`).

Custom element reactions

A custom element can define special lifecycle hooks for running code during interesting times of its existence. These are called **custom element reactions**.

Name	Called when
`constructor`	An instance of the element is created or upgraded. Useful for initializing state, settings up event listeners, or creating shadow dom. See the spec for restrictions on what you can do in the `constructor`.
`connectedCallback`	Called every time the element is inserted into the DOM. Useful for running setup code, such as fetching resources or rendering. Generally, you should try to delay work until this time.
`disconnectedCallback`	Called every time the element is removed from the DOM. Useful for running clean up code.
`attributeChangedCallback(attrName, oldVal, newVal)`	Called when an observed attribute has been added, removed, updated, or replaced. Also called for initial values when an element is created by the parser, or upgraded. **Note:** only attributes listed in the `observedAttributes` property will receive this callback.
`adoptedCallback()`	The custom element has been moved into a new `document` (e.g. someone called `document.adoptNode(el)`).

Note: The browser calls the `attributeChangedCallback()` for any attributes whitelisted in the `observedAttributes` array (see Observing changes to attributes). Essentially, this is a performance optimization. When users change a common attribute like `style` or `class`, you don't want to be spammed with tons of callbacks.

Reaction callbacks are synchronous. If someone calls `el.setAttribute()` on your element, the browser will immediately call `attributeChangedCallback()`. Similarly, you'll receive a

`disconnectedCallback()` right after your element is removed from the DOM (e.g. the user calls `el.remove()`).

Example: adding custom element reactions to `<app-drawer>`:

```
class AppDrawer extends HTMLElement {
  constructor() {
    super(); // always call super() first in the constructor.
    ...
  }
  connectedCallback() {
    ...
  }
  disconnectedCallback() {
    ...
  }
  attributeChangedCallback(attrName, oldVal, newVal) {
    ...
  }
}
```

Define reactions if/when it make senses. If your element is sufficiently complex and opens a connection to IndexedDB in `connectedCallback()`, do the necessary cleanup work in `disconnectedCallback()`. But be careful! You can't rely on your element being removed from the DOM in all circumstances. For example, `disconnectedCallback()` will never be called if the user closes the tab.

Properties and attributes

Reflecting properties to attributes

It's common for HTML properties to reflect their value back to the DOM as an HTML attribute. For example, when the values of `hidden` or `id` are changed in JS:

```
div.id = 'my-id';
div.hidden = true;
```

the values are applied to the live DOM as attributes:

```
<div id="my-id" hidden>
```

This is called "reflecting properties to attributes". Almost every property in HTML does this. Why? Attributes are also useful for configuring an element declaratively and certain APIs like accessibility and CSS selectors rely on attributes to work.

Reflecting a property is useful anywhere you want to **keep the element's DOM representation in sync with its JavaScript state**. One reason you might want to reflect a property is so user-defined styling applies when JS state changes.

Recall our `<app-drawer>`. A consumer of this component may want to fade it out and/or prevent user interaction when it's disabled:

```
app-drawer[disabled] {
  opacity: 0.5;
  pointer-events: none;
}
```

When the `disabled` property is changed in JS, we want that attribute to be added to the DOM so the user's selector matches. The element can provide that behavior by reflecting the value to an attribute of the same name:

```
...

get disabled() {
  return this.hasAttribute('disabled');
}

set disabled(val) {
  // Reflect the value of `disabled` as an attribute.
  if (val) {
    this.setAttribute('disabled', '');
  } else {
    this.removeAttribute('disabled');
  }
  this.toggleDrawer();
}
```

Observing changes to attributes

HTML attributes are a convenient way for users to declare initial state:

```
<app-drawer open disabled></app-drawer>
```

Elements can react to attribute changes by defining a `attributeChangedCallback`. The browser will call this method for every change to attributes listed in the `observedAttributes` array.

```
class AppDrawer extends HTMLElement {
  ...

  static get observedAttributes() {
```

```
    return ['disabled', 'open'];
  }

  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    if (val) {
      this.setAttribute('disabled', '');
    } else {
      this.removeAttribute('disabled');
    }
  }

  // Only called for the disabled and open attributes due to observedAttributes
  attributeChangedCallback(name, oldValue, newValue) {
    // When the drawer is disabled, update keyboard/screen reader behavior.
    if (this.disabled) {
      this.setAttribute('tabindex', '-1');
      this.setAttribute('aria-disabled', 'true');
    } else {
      this.setAttribute('tabindex', '0');
      this.setAttribute('aria-disabled', 'false');
    }
    // TODO: also react to the open attribute changing.
  }
}
```

In the example, we're setting additional attributes on the `<app-drawer>` when a `disabled` attribute is changed. Although we're not doing it here, you could also **use the `attributeChangedCallback` to keep a JS property in sync with its attribute**.

Element upgrades

Progressively enhanced HTML

We've already learned that custom elements are defined by calling `customElements.define()`. But this doesn't mean you have to define + register a custom element all in one go.

Custom elements can be used *before* their definition is registered.

Progressive enhancement is a feature of custom elements. In other words, you can declare a bunch of `<app-drawer>` elements on the page and never invoke `customElements.define('app-drawer', ...)` until much later. This is because the browser treats potential custom elements

differently thanks to unknown tags. The process of calling **`define()`** and endowing an existing element with a class definition is called "element upgrades".

To know when a tag name becomes defined, you can use **`window.customElements.whenDefined()`**. It vends a Promise that resolves when the element becomes defined.

```
customElements.whenDefined('app-drawer').then(() => {
  console.log('app-drawer defined');
});
```

Example - delay work until a set of child elements are upgraded

```
<share-buttons>
  <social-button type="twitter"><a href="...">Twitter</a></social-button>
  <social-button type="fb"><a href="...">Facebook</a></social-button>
  <social-button type="plus"><a href="...">G+</a></social-button>
</share-buttons>

// Fetch all the children of <share-buttons> that are not defined yet.
let undefinedButtons = buttons.querySelectorAll(':not(:defined)');

let promises = [...undefinedButtons].map(socialButton => {
  return customElements.whenDefined(socialButton.localName);
));

// Wait for all the social-buttons to be upgraded.
Promise.all(promises).then(() => {
  // All social-button children are ready.
});
```

Note: I think of custom elements as being in a state of limbo before they're defined. The spec defines an element's state as "undefined", "uncustomized", or "custom". Built-in elements like **`<div>`** are always "defined".

Element-defined content

Custom elements can manage their own content by using the DOM APIs inside element code. Reactions come in handy for this.

Example - create an element with some default HTML:

```
customElements.define('x-foo-with-markup', class extends HTMLElement {
  connectedCallback() {
    this.innerHTML = "<b>I'm an x-foo-with-markup!</b>";
```

```
  }
  ...
});
```

Declaring this tag will produce:

```
<x-foo-with-markup>
 <b>I'm an x-foo-with-markup!</b>
</x-foo-with-markup>
```

Note: Overwriting an element's children with new content is generally not a good idea because it's unexpected. Users would be surprised to have their markup thrown out. A better way to add element-defined content is to use shadow DOM, which we'll talk about next.

Creating an element that uses Shadow DOM

Note: I'm not going to cover the features of Shadow DOM in this article, but it's a powerful API to combine with custom elements. By itself, Shadow DOM is composition tool. When it's used in conjunction with custom elements, magical things happen.

Shadow DOM provides a way for an element to own, render, and style a chunk of DOM that's separate from the rest of the page. Heck, you could even hide away an entire app within a single tag:

```
<!-- chat-app's implementation details are hidden away in Shadow DOM. -->
<chat-app></chat-app>
```

To use Shadow DOM in a custom element, call `this.attachShadow` inside your `constructor`:

```
let tmpl = document.createElement('template');
tmpl.innerHTML = `
  <style>:host { ... }</style> <!-- look ma, scoped styles -->
  <b>I'm in shadow dom!</b>
  <slot></slot>
`;

customElements.define('x-foo-shadowdom', class extends HTMLElement {
```

```
  constructor() {
    super(); // always call super() first in the constructor.

    // Attach a shadow root to the element.
    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.appendChild(tmpl.content.cloneNode(true));
  }
  ...
});
```

Note: In the above snippet we use a `template` element to clone DOM, instead of setting the `innerHTML` of the `shadowRoot`. This technique cuts down on HTML parse costs because the content of the template is only parsed once, whereas calling `innerHTML` on the `shadowRoot` will parse the HTML for each instance. We'll talk more about templates in the next section.

Example usage:

```
<x-foo-shadowdom>
  <p><b>User's</b> custom text</p>
</x-foo-shadowdom>

<!-- renders as -->
<x-foo-shadowdom>
  #shadow-root
    <b>I'm in shadow dom!</b>
    <slot></slot> <!-- slotted content appears here -->
</x-foo-shadowdom>
```

Creating elements from a `<template>`

For those unfamiliar, the `<template>` element allows you to declare fragments of DOM which are parsed, inert at page load, and can be activated later at runtime. It's another API primitive in the web components family. **Templates are an ideal placeholder for declaring the structure of a custom element**.

Example: registering an element with Shadow DOM content created from a `<template>`:

```
<template id="x-foo-from-template">
  <style>
    p { color: green; }
  </style>
  <p>I'm in Shadow DOM. My markup was stamped from a &lt;template&gt;.</p>
</template>

<script>
  let tmpl = document.querySelector('#x-foo-from-template');
  // If your code is inside of an HTML Import you'll need to change the above line t
  // let tmpl = document.currentScript.ownerDocument.querySelector('#x-foo-from-temp

  customElements.define('x-foo-from-template', class extends HTMLElement {
    constructor() {
      super(); // always call super() first in the constructor.
      let shadowRoot = this.attachShadow({mode: 'open'});
      shadowRoot.appendChild(tmpl.content.cloneNode(true));
    }
    ...
  });
</script>
```

These few lines of code pack a punch. Let's understand the key things going on:

1. We're defining a new element in HTML: `<x-foo-from-template>`
2. The element's Shadow DOM is created from a `<template>`
3. The element's DOM is local to the element thanks to Shadow DOM
4. The element's internal CSS is scoped to the element thanks to Shadow DOM

Styling a custom element

Even if your element defines its own styling using Shadow DOM, users can style your custom element from their page. These are called "user-defined styles".

```
<!-- user-defined styling -->
<style>
  app-drawer {
    display: flex;
```

```
  }
  panel-item {
    transition: opacity 400ms ease-in-out;
    opacity: 0.3;
    flex: 1;
    text-align: center;
    border-radius: 50%;
  }
  panel-item:hover {
    opacity: 1.0;
    background: rgb(255, 0, 255);
    color: white;
  }
  app-panel > panel-item {
    padding: 5px;
    list-style: none;
    margin: 0 7px;
  }
</style>

<app-drawer>
  <panel-item>Do</panel-item>
  <panel-item>Re</panel-item>
  <panel-item>Mi</panel-item>
</app-drawer>
```

You might be asking yourself how CSS specificity works if the element has styles defined within Shadow DOM. In terms of specificity, user styles win. They'll always override element-defined styling. See the section on Creating an element that uses Shadow DOM.

Pre-styling unregistered elements

Before an element is upgraded you can target it in CSS using the **`:defined`** pseudo-class. This is useful for pre-styling a component. For example, you may wish to prevent layout or other visual FOUC by hiding undefined components and fading them in when they become defined.

Example - hide **`<app-drawer>`** before it's defined:

```
app-drawer:not(:defined) {
  /* Pre-style, give layout, replicate app-drawer's eventual styles, etc. */
  display: inline-block;
  height: 100vh;
  opacity: 0;
  transition: opacity 0.3s ease-in-out;
}
```

After `<app-drawer>` becomes defined, the selector (`app-drawer:not(:defined)`) no longer matches.

Extending elements

The Custom Elements API is useful for creating new HTML elements, but it's also useful for extending other custom elements or even the browser's built-in HTML.

Extending a custom element

Extending another custom element is done by extending its class definition.

Example - create `<fancy-app-drawer>` that extends `<app-drawer>`:

```
class FancyDrawer extends AppDrawer {
  constructor() {
    super(); // always call super() first in the constructor. This also calls the ex
    ...
  }

  toggleDrawer() {
    // Possibly different toggle implementation?
    // Use ES2015 if you need to call the parent method.
    // super.toggleDrawer()
  }

  anotherMethod() {
    ...
  }
}

customElements.define('fancy-app-drawer', FancyDrawer);
```

Extending native HTML elements

Let's say you wanted to create a fancier `<button>`. Instead of replicating the behavior and functionality of `<button>`, a better option is to progressively enhance the existing element using custom elements.

A **customized built-in element** is a custom element that extends one of the browser's built-in HTML tags. The primary benefit of extending an existing element is to gain all of its features (DOM properties, methods, accessibility). There's no better way to write a progressive web app than to **progressively enhance existing HTML elements**.

Note: Only Chrome 67 supports customized built-in elements (status) right now. Edge and Firefox will implement it, but Safari has chosen not to implement it. This is unfortunate for accessibility and progressive enhancement. If you think extending native HTML elements is useful, voice your thoughts on 509 and 662 on Github.

To extend an element, you'll need to create a class definition that inherits from the correct DOM interface. For example, a custom element that extends `<button>` needs to inherit from `HTMLButtonElement` instead of `HTMLElement`. Similarly, an element that extends `<img>` needs to extend `HTMLImageElement`.

Example - extending `<button>`:

```
// See https://html.spec.whatwg.org/multipage/indices.html#element-interfaces
// for the list of other DOM interfaces.
class FancyButton extends HTMLButtonElement {
  constructor() {
    super(); // always call super() first in the constructor.
    this.addEventListener('click', e => this.drawRipple(e.offsetX, e.offsetY));
  }

  // Material design ripple animation.
  drawRipple(x, y) {
    let div = document.createElement('div');
    div.classList.add('ripple');
    this.appendChild(div);
    div.style.top = `${y - div.clientHeight/2}px`;
    div.style.left = `${x - div.clientWidth/2}px`;
    div.style.backgroundColor = 'currentColor';
    div.classList.add('run');
    div.addEventListener('transitionend', e => div.remove());
  }
}

customElements.define('fancy-button', FancyButton, {extends: 'button'});
```

Notice that the call to `define()` changes slightly when extending a native element. The required third parameter tells the browser which tag you're extending. This is necessary because many HTML tags share the same DOM interface. `<section>`, `<address>`, and `<em>` (among others) all share `HTMLElement`; both `<q>` and `<blockquote>` share `HTMLQuoteElement`; etc.. Specifying `{extends: 'blockquote'}` lets the browser know you're creating a souped-up `<blockquote>` instead of a `<q>`. See the HTML spec for the full list of HTML's DOM interfaces.

Note: Extending `HTMLButtonElement` endows our fancy button with all the DOM properties/methods of `<button>`. That checks off a bunch of stuff we don't have to implement ourselves: `disabled` property,

`click()` method, `keydown` listeners, `tabindex` management. Instead, our focus can be progressively enhancing `<button>` with custom functionality, namely, the `drawRipple()` method. Less code, more reuse!

Consumers of a customized built-in element can use it in several ways. They can declare it by adding the `is=""` attribute on the native tag:

```
<!-- This <button> is a fancy button. -->
<button is="fancy-button" disabled>Fancy button!</button>
```

create an instance in JavaScript:

```
// Custom elements overload createElement() to support the is="" attribute.
let button = document.createElement('button', {is: 'fancy-button'});
button.textContent = 'Fancy button!';
button.disabled = true;
document.body.appendChild(button);
```

or use the `new` operator:

```
let button = new FancyButton();
button.textContent = 'Fancy button!';
button.disabled = true;
```

Here's another example that extends `<img>`.

Example - extending `<img>`:

```
customElements.define('bigger-img', class extends Image {
  // Give img default size if users don't specify.
  constructor(width=50, height=50) {
    super(width * 10, height * 10);
  }
}, {extends: 'img'});
```

Users declare this component as:

```
<!-- This <img> is a bigger img. -->
<img is="bigger-img" width="15" height="20">
```

or create an instance in JavaScript:

```
const BiggerImage = customElements.get('bigger-img');
const image = new BiggerImage(15, 20); // pass constructor values like so.
console.assert(image.width === 150);
console.assert(image.height === 200);
```

Misc details

Unknown elements vs. undefined custom elements

HTML is lenient and flexible to work with. For example, declare `<randomtagthatdoesntexist>` on a page and the browser is perfectly happy accepting it. Why do non-standard tags work? The answer is the HTML specification allows it. Elements that are not defined by the specification get parsed as `HTMLUnknownElement`.

The same is not true for custom elements. Potential custom elements are parsed as an `HTMLElement` if they're created with a valid name (includes a "-"). You can check this in a browser that supports custom elements. Fire up the Console: Ctrl+Shift+J (or Cmd+Opt+J on Mac) and paste in the following lines of code:

```
// "tabs" is not a valid custom element name
document.createElement('tabs') instanceof HTMLUnknownElement === true

// "x-tabs" is a valid custom element name
document.createElement('x-tabs') instanceof HTMLElement === true
```

API reference

The `customElements` global defines useful methods for working with custom elements.

`define(tagName, constructor, options)`

Defines a new custom element in the browser.

Example

```
customElements.define('my-app', class extends HTMLElement { ... });
customElements.define(
  'fancy-button', class extends HTMLButtonElement { ... }, {extends: 'button'});
```

`get(tagName)`

Given a valid custom element tag name, returns the element's constructor. Returns `undefined` if no element definition has been registered.

Example

```
let Drawer = customElements.get('app-drawer');
let drawer = new Drawer();
```

`whenDefined(tagName)`

Returns a Promise that resolves when the custom element is defined. If the element is already defined, resolve immediately. Rejects if the tag name is not a valid custom element name

Example

```
customElements.whenDefined('app-drawer').then(() => {
  console.log('ready!');
});
```

History and browser support

If you've been following web components for the last couple of years, you'll know that Chrome 36+ implemented a version of the Custom Elements API that uses **`document.registerElement()`** instead of **`customElements.define()`**. That's now considered a deprecated version of the standard, called v0. **`customElements.define()`** is the new hotness and what browser vendors are starting to implement. It's called Custom Elements v1.

If you happen to be interested in the old v0 spec, check out the html5rocks article.

Browser support

Chrome 54 (status) and Safari 10.1 (status) have Custom Elements v1. Edge has begun prototyping. Mozilla has an open bug to implement.

To feature detect custom elements, check for the existence of **`window.customElements`**:

```
const supportsCustomElementsV1 = 'customElements' in window;
```

Polyfill

Until browser support is widely available, there's a standalone polyfill available for Custom Elements v1. However, we recommend using the webcomponents.js loader to optimally load the web components polyfills. The loader uses feature detection to asynchronously load only the necessary pollyfills required by the browser.

Note: If your project transpiles to or uses ES5, be sure to see the notes on including custom-elements-es5-adapter.js in addition to the polyfills.

Install it:

```
npm install --save @webcomponents/webcomponentsjs
```

Usage:

```
<!-- Use the custom element on the page. -->
<my-element></my-element>

<!-- Load polyfills; note that "loader" will load these async -->
<script src="node_modules/@webcomponents/webcomponentsjs/webcomponents-loader.js" de

<!-- Load a custom element definitions in `waitFor` and return a promise -->
<script type="module">
  function loadScript(src) {
    return new Promise(function(resolve, reject) {
      const script = document.createElement('script');
      script.src = src;
      script.onload = resolve;
      script.onerror = reject;
      document.head.appendChild(script);
    });
  }

  WebComponents.waitFor(() => {
    // At this point we are guaranteed that all required polyfills have
    // loaded, and can use web components APIs.
    // Next, load element definitions that call `customElements.define`.
    // Note: returning a promise causes the custom elements
    // polyfill to wait until all definitions are loaded and then upgrade
    // the document in one batch, for better performance.
    return loadScript('my-element.js');
  });
</script>
```

Note: the `:defined` CSS pseudo-class cannot be polyfilled.

Conclusion

Custom elements give us a new tool for defining new HTML tags in the browser and creating reusable components. Combine them with the other new platform primitives like Shadow DOM and `<template>`, and we start to realize the grand picture of Web Components:

- Cross-browser (web standard) for creating and extending reusable components.
- Requires no library or framework to get started. Vanilla JS/HTML FTW!

- Provides a familiar programming model. It's just DOM/CSS/HTML.
- Works well with other new web platform features (Shadow DOM, `<template>`, CSS custom properties, etc.)
- Tightly integrated with the browser's DevTools.
- Leverage existing accessibility features.

Last updated July 23, 2018.