Introduction to Web Components

Arthur Barrett
abarrett@fas.harvard.edu
April 1, 2022

Today's Talk

- Introduction to Web Components
- What, Why, How?
- React/Vue vs Web Components
- Limitations
- Wrap-up

What are Web Components?

Web components are a set of web APIs that allow you to create custom, reusable, and encapsulated HTML elements.

- Based on web standards (three core APIs).
- All you need is JS, HTML, and CSS.
- No libraries or frameworks required.

A Brief History

- **2011** Web Components introduced by Alex Russell.
- **2013** Polymer (polyfill) released by Google.
- **2018** Chrome 67 and Firefox 63.
- **2020** Edge 79.
- **2022** Opera 83 and Android 99.

Web Components are now well supported in modern browsers.

Building Blocks

Web Components depend on 3 core APIs:

1. Custom Elements (Spec)
    - Provides a way to create custom, fully-featured DOM elements.
2. Shadow DOM (Spec)
    - Provides a way to encapsulate and scope style and markup.
3. Templates (Spec)
    - Provides a way to declare fragments of markup for reuse later.

Building Block #1: Custom Elements

Web Components don't exist without the features unlocked by custom elements.

```
class HelloWorld extends HTMLElement {
  connectedCallback() { // called when component inserted into DOM
    this.textContent = "Hello World"
  }
}
customElements.define('hello-world', HelloWorld); // must have a dash
```

```
<hello-world></hello-world>
```

Types of Custom Elements

There are two types of custom elements:

- **Autonomous** custom elements
  - Component class inherits from generic `HTMLElement`
  - Standalone (e.g. `<hello-world>`)
- **Customized** built-in elements
  - Inherit from basic HTML elements like `HTMLButtonElement`
  - Must specify `extends` option in `customElements.define()`
  - Must use `is` attribute: `<button is="my-btn">`

Autonomous vs Customized

Autonomous elements represent their children with no special meaning. Customized elements may inherit semantics for:

- Accessibility
  - Your `<my-button>` may not be identified as a `<button>`.
  - But `<button is="my-button">` ensures it is.
  - Can be mitigated using ARIA attributes.
- Search engine optimization

Lifecycle hooks

Web components have their own lifecycle:

- `constructor()`: When Web Component is created
- `connectedCallback()`: Element is added to the DOM
- `disconnectedCallback()`: Element is removed
- `adoptedCallback()`: Element is moved from one doc to another
- `attributeChangedCallback()`: Attribute is added/removed/changed

Building Block #2: Shadow DOM

> 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.

https://web.dev/shadowdom-v1/

Light tree and Shadow tree

A DOM element can have a *Light tree* or *Shadow tree*.

- **Light tree**: regular DOM tree we're used to working with.
- **Shadow tree**: hidden DOM tree, not reflected in HTML.

```
<custom-element>
    ^^^^^^^^^^^^^^^^^^^^^^^^^ ISOLATION
    #shadow-root
        ...
    ------------------------- DOCUMENT FRAGMENT
    <!--LIGHT DOM-->
</custom-element>
```

Shadow Boundary

One of the core features is the **shadow boundary**:

- Selectors don't cross the boundary.
- CSS is scoped to the shadow root.
- Events only cross shadow boundary if `compose` flag is `true`.
  - Built-in events mostly have `composed:true`.
  - Custom events must set this explicitly.
- Events that cross are retargeted to the host.

Shadow DOM example

```
el.attachShadow({mode: "open"}); // open = allow access to .shadowRoot
el.shadowRoot // the shadow root
el.shadowRoot.host // the element itself

// put something in shadow DOM
el.shadowRoot.innerHTML = "Hello from the shadows!";

// Like any other normal DOM operation.
let hello = document.createElement("span");
hello.textContent = "Hello from a shadow span";
el.shadowRoot.appendChild(hello);
```

Shadow DOM in a Component

```
class MyWebComponent extends HTMLElement {
    constructor() {
        super();
        this.attachShadow({ mode: "open" });
    }
    connectedCallback() {
        this.shadowRoot.innerHTML = `
            <p>I'm in the Shadow Root!</p>
        `;
    }
}

window.customElements.define("my-web-component", MyWebComponent);
```

Templates

A built-in `<template>` element provides storage for complex (or simple) HTML markup templates. Parsed, but not rendered.

```
<template id="my-paragraph">
  <p>My paragraph</p>
</template>
```

```
let template = document.getElementById('my-paragraph');
let templateContent = template.content;
document.body.appendChild(templateContent);
```

Examples

React/Vue and Web Components

React and Vue consider Web Components a complementary technology.

- React is about keeping the DOM in sync with data.
- React and Vue can both use Web Components.
- Vue tries to resolve non-native HTML elements itself.

Limitations

- May not be appropriate for building a whole application:
  - Relatively low-level and bare bones.
  - No state management or other niceties (SSR, etc).
- Eager slot evaluation may be limiting
  - Can't control when/whether to rener slot content.
- Shadow DOM doesn't play well with native forms.
  - Can't extend native form elements.
  - Form elements inside shadow DOM aren't considered by parent.

Wrap-up

- Web Components are based on 3 fundamental web APIs to create custom, reusable, encapsulated HTML elements.
- Web Components are well supported in modern browsers.
- They aren't going to replace React, Vue, etc.
- Slow adoption, but here to stay.

Links

- References: Webcomponents.org, MDN, Google, javascript.info
- Collections: Component Gallery and Github Web Component Collection
- Perspectives: React and Web Components, Vue and Web Components