Web Components

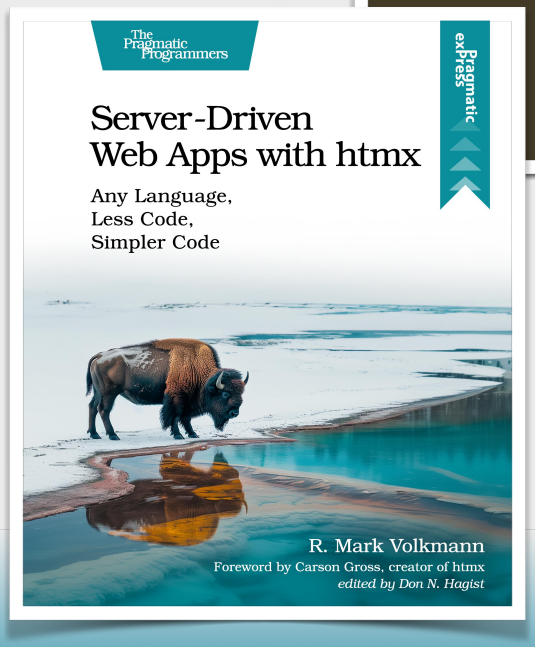
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at **https://github.com/mvolkmann/talks/**

About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- **Blog** at https://mvolkmann.github.io/blog/
- Author of Manning book "**Svelte ... in Action**"
- Author of Pragmatic Bookshelf book "**Server-Driven Web Apps with htmx**"



Web Components

- Define **custom HTML elements** that can be used like standard HTML elements
  - names must be all lower-case and contain at least one hyphen
  - tags cannot be self-closing ex. `<radio-group></radio-group>`, not `<radio-group />`
- Requires **a bit more effort** than implementing components using a framework like Angular, React, Svelte, or Cue
  - worthwhile for components that may someday be used in multiple apps written using multiple frameworks

Pros

- Standard way to implement web-based UI components
- Just use the “platform”
  - no libraries, tooling, or build process required
- Can use in any web page, with any web framework, and in Markdown files
  - can be shared between projects using different frameworks
- Shadow DOM provides CSS and DOM encapsulation
  - styles defined in a component do not affect HTML outside it
  - can control ability to style a component from outside it
  - can prevent code outside the component from querying and modifying its DOM

Some issues remain when using in React-based frameworks.

Cons

- Learning curve, but relatively small
- Somewhat more verbose unless using a library like Lit
- Reactivity must be implemented manually unless using a library like Lit
- Implementing web components with inputs that should be included in form submissions requires extra work see “Form Associated” slide
- Server-side rendering is currently challenging Do you need it?

Standards Based ...

- **Custom Elements**
  - enables defining custom HTML elements by creating a class that extends **`HTMLElement`** class or one of its subclasses
  - supports several lifecycle methods
  - each class defines a reusable component that has attributes, properties, and methods
- **Shadow DOM**
  - encapsulates styling and DOM of custom HTML elements
  - adds **`attachShadow`** method, **`ShadowRoot`** class, and **`shadowRoot`** property
  - supports “open” and “closed” modes

... Standards Based

- **HTML Templates**
  - adds HTML element **`template`** defined by **`HTMLTemplateElement`** class
  - **`content`** property is an instance of **`DocumentFragment`** that can be created without rendering
  - can be efficiently cloned and rendered later
- **ES Modules** (a.k.a. JavaScript Modules)
  - adds import and export syntax, module scope, asynchronous loading, and more

Web Component Libraries

- **Shoelace** - https://shoelace.style/
  - “forward-thinking library of web components”
- **Web Awesome** - https://webawesome.com/
  - “make something awesome with open-source web components”
  - eventual successor to Shoelace; still in beta
- **FAST** from Microsoft - https://fast.design
  - “dedicated to providing support for native Web Components and modern Web Standards”
- **Lion** - https://lion.js.org/
  - “fundamental white label web components for building your design system”

Shoelace Example


```
<html>
  <head>
    <link
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/@shoelace-style
        /shoelace@2.20.1/cdn/themes/light.css"
    />
    <style>
      sl-radio-group {
        margin-top: 1rem;
      }
      sl-radio-group::part(form-control),
      sl-radio-group::part(form-control-input) {
        display: flex;
        gap: 1rem;
      }
      sl-radio-group::part(form-control-label) {
        font-family: var(--sl-input-font-family);
        font-weight: bold;
      }
    </style>
    <script
      type="module"
      src="https://cdn.jsdelivr.net/npm/@shoelace-style
        /shoelace@2.20.1/cdn/shoelace-autoloader.js"
    ></script>
```

```
    <script>
      window.onload = () => {
        const slSwitch = document.querySelector("sl-switch");
        slSwitch.addEventListener("sl-change", (event) => {
          console.log("switch is",
            event.target.checked ? "on" : "off");
        });

        const slRadioGroup =
          document.querySelector("sl-radio-group");
        slRadioGroup.addEventListener("sl-change", (event) => {
          console.log("color =", event.target.value);
        });
      };
    </script>
  </head>
  <body>
    <div>
      <sl-switch>Happy?</sl-switch>
    </div>
    <sl-radio-group label="Color:" value="red">
      <sl-radio value="red">Red</sl-radio>
      <sl-radio value="green">Green</sl-radio>
      <sl-radio value="blue">Blue</sl-radio>
    </sl-radio-group>
  </body>
</html>
```

Implementation Options

- **Vanilla** from W3C
- **WCE** from me (more on this later)

No libraries, tooling, or build process is required.

- **Lit** from Google
  - most popular web components library
- **Stencil** from Ionic
- **FAST** from Microsoft

All of these require installing a library and using tooling in a build process.

Show Me Code!

- Simple web component

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

<hello-world name="Mark"></hello-world>
```

Hello, World!

Hello, Mark!

- Module benefits
- We'll see five ways to implement
  - vanilla with no shadow DOM, setting innerHTML
  - vanilla with no shadow DOM, using DOM API
  - vanilla with shadow DOM, setting innerHTML
  - vanilla with shadow DOM, using DOM API
  - using Lit (a bit later)

Each requires
a small amount of code
that is easy to understand.

Starting Simple

None of these implement **attributeChangedCallback**, so modifying the **name** attribute in DevTools does not update the UI.

index.html

```
<html>
  <head>
    <script src="hello-world1.js" type="module"></script>
    <script src="hello-world2.js" type="module"></script>
    <script src="hello-world3.js" type="module"></script>
    <script src="hello-world4.js" type="module"></script>
  </head>
  <body>
    <hello-world1 name="innerHTML"></hello-world1>
    <hello-world2 name="appendChild"></hello-world2>
    <hello-world3 name="shadow-innerHTML"></hello-world3>
    <hello-world4 name="shadow-appendChild"></hello-world4>
  </body>
</html>
```

hello-world1.js

```
class HelloWorld1 extends HTMLElement {
  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    this.innerHTML = `<p>Hello, ${name}!</p>`;
  }
}
customElements.define("hello-world1", HelloWorld1);
```

hello-world2.js

```
class HelloWorld2 extends HTMLElement {
  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    const p = document.createElement("p");
    p.textContent = `Hello, ${name}!`;
    this.appendChild(p);
  }
}
customElements.define("hello-world2", HelloWorld2);
```

hello-world3.js

```
class HelloWorld3 extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    this.shadowRoot.innerHTML = `<p>Hello, ${name}!</p>`;
  }
}
customElements.define("hello-world3", HelloWorld3);
```

hello-world4.js

```
class HelloWorld4 extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    const p = document.createElement("p");
    p.textContent = `Hello, ${name}!`;
    this.shadowRoot.appendChild(p);
  }
}
customElements.define("hello-world4", HelloWorld4);
```

innerHTML vs. DOM API

- Two approaches for building web component DOM
- Set **`innerHTML`**
  - typically using JavaScript template strings
  - less verbose
  - faster for bulk updates
  - destroys existing nodes and loses event listeners
  - builds DOM synchronously so it can be queried immediately
  - danger of XSS when using untrusted content
- Call **DOM API** methods like **`createElement`** and **`appendChild`**
  - provides more fine-grained control
  - can preserve existing nodes and event listeners
  - avoids XSS from untrusted content

Middle Ground

- set **`innerHTML`** to render initial DOM
- use DOM API to make targeted updates in event handlers

Template Strings

- JavaScript feature that can be used to build a string of HTML that is used as an innerHTML value
- Capabilities include:
  - conditional logic with ternary operator
    - one use is to conditionally add an attribute
  - iteration with **`map`** method
  - break up content rendering into multiple functions

examples come from "Radio Group" component we will review later

```
<input
  type="radio"
  id="${option}"
  name="${this.#name}"
  value="${option}"
  ${option === this.value ? "checked" : ""}
/>
```

```
<div class="radio-group">
  ${options.map((option) => this.#makeRadio(option)).join("")}
</div>
```

```
#makeRadio(option) {
  return /*html*/ `
    ...
  `;
}
```

Can AI Tools Generate?

- **Request:** Write a hello-world vanilla web component that has a name attribute with a default value of “World”.
- **Contenders:**
  - **ChatGPT**, **Claude**, **Google Gemini**, **Microsoft Copilot**, and **Perplexity**
- While the generated code uses slightly different approaches, each successfully generated code that meets the requirements
  - Google Gemini produce worst code

Using Lit

package.json

```
{
  "name": "hello-world-web-components",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "lit": "^3.0.0"
  },
  "devDependencies": {
    "typescript": "^5.0.0",
    "vite": "^5.0.0"
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "experimentalDecorators": true,
    "useDefineForClassFields": false
  }
}
```

When using Lit, modifying the **name** attribute in DevTools automatically updates the UI!

hello-world5.js

```
import { html, LitElement } from "lit";

class HelloWorld5 extends LitElement {
  static properties = { name: { type: String } };

  render() {
    return html`<p>Hello, ${this.name || "World"}!</p>`;
  }
}
customElements.define("hello-world5", HelloWorld5);
```

JavaScript version

index.html

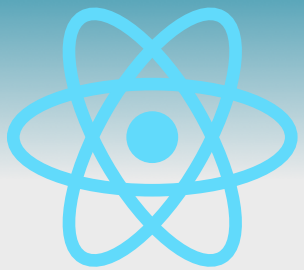
```
<html>
  <head>
    <script src="hello-world5.js" type="module"></script>
    <script src="hello-world6.ts" type="module"></script>
  </head>
  <body>
    <hello-world5 name="Lit JS"></hello-world5>
    <hello-world6 name="Lit TS"></hello-world6>
  </body>
</html>
```

hello-world6.ts

```
import { html, LitElement } from "lit";
import { customElement, property } from "lit/decorators.js";

@customElement("hello-world6")
export class HelloWorld6 extends LitElement {
  @property({ type: String }) name = "";

  render() {
    return html`<p>Hello, ${this.name || "World"}!</p>`;
  }
}
```

TypeScript version

React Comparison



- Steps to create
  - **`npm create vite@latest react-hello-world`**
    - select React and TypeScript
  - **`cd react-hello-world`**
  - **`npm install`**
  - create **`src/HelloWorld.tsx`**
  - modify **`App.tsx`**
  - **`npm run dev`**
  - browse localhost:5173
- Component can only be used in React apps
- Lots of libraries and tools required

HelloWorld.tsx

```tsx
import React from "react";

interface HelloWorldProps {
  name?: string;
}

const HelloWorld: React.FC<HelloWorldProps> =
  ({ name = "World" }) => {
    return <p>Hello, {name}!</p>;
  };

export default HelloWorld;
```

App.tsx

```tsx
import HelloWorld from "./HelloWorld";

function App() {
  return (
    <>
      <HelloWorld />
      <HelloWorld name="Mark" />
    </>
  );
}

export default App;
```

Hello, World!

Hello, Mark!

Including Web Components

- Include web components with **`script`** tags

```
<script src="hello-world.js" type="module"></script>
```

- Module benefits
  - can use **`import`** and **`export`** keywords
  - top-level variables are scoped to the module — useful when using **`template`** element
  - loading is deferred, same is as when **`defer`** attribute is included
    - fetches in parallel, but delays execution until DOM is built
  - uses strict mode for better error checking
  - can use top-level **`await`**
  - enforces Cross Origin Resource Sharing (CORS) restrictions
    - prevents testing web components with **`file://`** URLs
    - have to run a local server (like Vite which is described next)

Vite Server

- Many options for starting an HTTP server that serves local files
- One option is to use Vite
  - has many other benefits include live reload
- Steps
  - make new directory and **`cd`** to it in a terminal window
  - create **`package.json`** file by entering **`npm init`**
  - install Vite by entering **`npm i -D vite`**
  - create **`index.html`**
  - start server by entering **`npm run dev`**
  - browse localhost:5173

If port is in use,
Vite will try 5174, 5175, ...
until it finds an open port.

minimal **`package.json`**
after installing Vite

```
{
  "name": "web-server",
  "scripts": {
    "dev": "vite"
  },
  "devDependencies": {
    "vite": "^7.0.1"
  }
}
```

Cloning `template` Elements

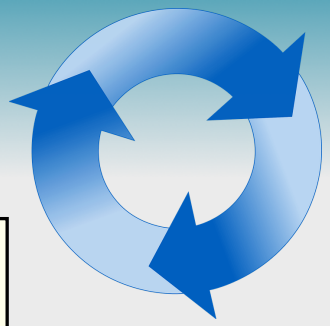
- Faster than building component DOM from scratch for each instance

```
const template = document.createElement("template");
template.innerHTML = /*html*/ `
  <p>Hello, <span id="name"></span>!</p>
`;

class HelloWorld7 extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const { shadowRoot } = this;
    shadowRoot.appendChild(template.content.cloneNode(true));
    const span = shadowRoot.querySelector("#name");
    span.textContent = this.getAttribute("name") || "World";
  }
}

customElements.define("hello-world7", HelloWorld7);
```

When `script` tag includes `type="module"`, the `template` variable is scoped to this module.

Preceding a template string with the comment `/*html*/` triggers **VS Code extension** "es6-string-html" to add syntax highlighting.

Lifecycle Methods ...



- **`constructor`**
  - called when an instance is created
  - recommended place to attach a **`ShadowRoot`**
- **`connectedCallback`**
  - called after an instance is added to DOM
  - recommended place to populate web component DOM and register event handlers
- **`attributeChangedCallback`**
  - called when the value of any “observed” attribute changes
  - specify with

```
constructor() {
  super();
  this.attachShadow({ mode: "open" });
}
```

`shadowRoot` property is set when mode is **`"open"`**, but not when it is **`"closed"`**

```
connectedCallback() {
  this.shadowRoot.innerHTML = `...some HTML...`;
  const button = this.shadowRoot.querySelector('button');
  button.addEventListener('click', event => {
    alert('got click');
  });
}
```

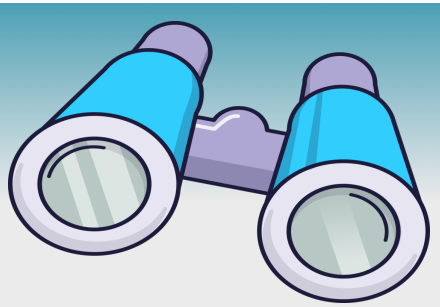
```
attributeChangedCallback(name, oldValue, newValue) {
  ...
}
```

```
static get observedAttributes() {
  return ['name1', 'name2', ...];
}
```

... Lifecycle Methods

- **`disconnectedCallback`** (rarely used)
  - called after an instance is removed from DOM
  - cleanup resources created in **`connectedCallback`**
    - resources could include event listeners, timeouts, intervals, and pending network requests
- **`adoptedCallback`** (rarely used)
  - called when an instance is moved to a different document

Observing Attributes



- Let's modify the **`HelloWorld1`** class to update what it renders when the **`name`** attribute is modified
  - same approach works for other **`HelloWorld*`** classes

```
class HelloWorld1 extends HTMLElement {
  static get observedAttributes() {
    return ["name"];
  }

  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    this.innerHTML = `<p>Hello, ${name}!</p>`;
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === "name") {
      const p = this.querySelector("p");
      if (p) p.textContent = `Hello, ${newValue}!`;
    }
  }
}
customElements.define("hello-world1", HelloWorld1);
```

Shadow DOM

- Encapsulates styling and content of a custom element
- Many standard HTML elements utilize a shadow DOM
  - for example, **`input`**, **`audio`**, **`video`**, and **`detail`**
- To use a shadow DOM in a web component

Inspect these elements in Chrome DevTools to see **`#shadow-root`** and their contents. This requires checking Settings ... Preferences ... Elements ... Show user agent shadow DOM. The **`#shadow-root`** of custom elements is always visible.

```
constructor() {
  this.attachShadow({mode: 'open'});
}
```

```
constructor() {
  this.#root = this.attachShadow({mode: 'closed'});
}
```

When mode is "**`closed`**", capture shadow root in a private property so it can be accessed here.

- To add content

```
connectedCallback() {
  this.shadowRoot.innerHTML = 'some HTML string';
  // OR
  this.shadowRoot.appendChild(someElement);
}
```

```
connectedCallback() {
  this.#root.innerHTML = 'some HTML string';
  // OR
  this.#root.appendChild(someElement);
}
```

Form Associated

- Instances of web components nested in a **`form`** element by default do not contribute to the set of name/value pairs that are submitted by the **`form`** focus still moves properly
  - prevented by shadow DOM
- Solution demonstrated in **`radio-group`** web component on next few slides
  - set static property **`formAssociated`** to **`true`**
  - declare private property **`#internals`**
  - initialize in constructor with
    `this.#internals = this.attachInternals();`
  - when value to contribute changes,
    call **`this.#internals.setFormValue(newValue);`**

Radio Group Component

- Renders a set of associated radio buttons
- Uses a vanilla web component with an “open” shadow DOM
- Will use to demonstrate approaches for styling from outside component

RadioGroup Component ...

Example instance

```
<radio-group
  name="favoriteColor"
  options="red,green,blue"
  default="blue"
  value="green"
><radio-group>
```

`default` attribute is optional and defaults to first option;

`value` attribute is optional and defaults to `default`

```
class RadioGroup extends HTMLElement {
  static formAssociated = true;
  #default;
  #internals;
  #name;
  #value;

  constructor() {
    super();
    this.attachShadow({ mode: "open" });
    this.#internals = this.attachInternals();
  }
```

radio-group.js

```
  connectedCallback() {
    this.#name = this.getAttribute("name");
    const options = this.getAttribute("options")
      .split(",")
      .map((option) => option.trim());
    this.#default = this.getAttribute("default") || options[0];
    this.#value = this.getAttribute("value") || this.#default;

    this.shadowRoot.innerHTML = /*html*/ `
      <style>
        :not(:defined) {
          visibility: hidden;
        }

        .radio-group {
          display: flex;
          gap: 0.25rem;

          > div {
            display: flex;
            align-items: center;
          }
        }
      </style>
      <div class="radio-group">
        ${options.map((option) => this.#makeRadio(option)).join("")}
      </div>
    `;

    // Add event listeners to the radio buttons.
    const inputs = this.shadowRoot.querySelectorAll("input");
    for (const input of inputs) {
      input.addEventListener("change", (event) => {
        this.value = event.target.value;
      });
    }
  }
```

VS Code extension **es6-string-html** adds syntax highlight to HTML in template literals that are preceded by the comment `/*html*/`

demonstrates factoring out some HTML generation to separate functions

invokes “set value” method on next slide

... RadioGroup Component

called when containing **form** is reset by clicking a button with **type="reset"** or calling **reset** method on DOM **Form** object

```
formResetCallback() {
  const value = (this.value = this.#default);
  for (const input of this.shadowRoot.querySelectorAll("input")) {
    input.checked = input.value === value;
  }
}

#makeRadio(option) {
  return /*html*/ `
    <div>
      <input
        type="radio"
        id="${option}"
        name="${this.#name}"
        value="${option}"
        ${option === this.value ? "checked" : ""}
      />
      <label for="${option}">${option}</label>
    </div>
  `;
}
```

called in **connectedCallback** on previous slide to specify the DOM to be built

called when a value is assigned to **value** property

```
  get value() {
    return this.#value;
  }

  set value(newValue) {
    if (newValue === this.#value) return;
    this.#value = newValue;
    this.#internals.setFormValue(newValue);

    // This demonstrates how a web component
    // can contribute multiple values to a form.
    /*
    const data = new FormData();
    data.append(this.#name, newValue);
    data.append("favoriteNumber", 19);
    this.#internals.setFormValue(data);
    */
  }
}

customElements.define("radio-group", RadioGroup);
```

Shadow DOM Styling

- CSS rules defined in a web component
  do not leak out to affect external elements
- By default, web component styling is
  not affected by styles defined outside it
- Four ways to “pierce the Shadow DOM” to style from outside
  1. inherit CSS properties
  2. expose CSS variables
  3. use `part` attributes
  4. share CSS files

Inherit CSS Properties

- Inheritable CSS properties are automatically used by web components unless they specify other values in their own CSS
  - include **`color`**, **`cursor`**, **`font`**, **`font-family`**, **`font-size`**, **`font-style`**, **`font-variant`**, **`font-weight`**, **`letter-spacing`**, **`line-height`**, **`text-align`**, **`text-indent`**, **`text-transform`**, **`visibility`**, **`white-space`**, and **`word-spacing`**
- Example
  - in HTML that uses the web component, include this CSS rule which causes every element the web component renders that does not specify **`color`** to use **`blue`**
    - in this case it will be the **`label`** elements

```
radio-group {
  color: blue;
}
```

Despite the fact that **`inherit`** is not the default value of inheritable CSS properties, they inherit the value from their parent element, unless explicitly overridden.

Expose CSS Variables

- Web components can specify CSS properties in a way that allows them to be overridden by setting CSS variables (a.k.a. CSS custom properties)
- Example
  - in **`radio-group.js`**, include this CSS rule

```
label {
  color: var(--radio-group-label-color, black);
}
```

  - in HTML that uses the web component, include this CSS rule

```
radio-group {
  --radio-group-label-color: blue;
}
```

Use part Attributes

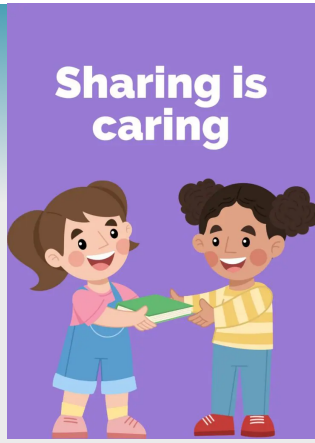
- Add part attributes to each web component element that wishes to allow external styling
- Example
  - in **`radio-group.js`**, include this CSS rule

```
<label for="${option}" part="radio-label">${option}</label>
```

  - in HTML that uses the web component, include this CSS rule

```
radio-group::part(radio-label) {
  color: blue;
}
```

Share CSS Files



- Define styles to be shared by multiple pages and web components in a **`.css`** file
- Add **`link`** element in each page and web component that refers to the **`.css`** file
  - only downloaded once and cached
- Example
  - create **`style.css`** containing this

```
label {
  color: blue;
}
```

  - in **`radio-group.js`**, include this **`link`** element

```
<link rel="stylesheet" href="style.css" />
```

CSS Concerns

- To specify inheritable CSS properties on a custom element, use the **`:host`** selector

```
:host {
  font-family: sans-serif;
}
```

- Style custom elements with the following CSS rule to avoid “Flash of Undefined Custom Elements” (FOUCE) and layout shift

```
:not(:defined) {
  visibility: hidden;
}
```

Shadow DOM Access ...

- DOM provides many methods that find elements including **`querySelector`**, **`querySelectorAll`**, and **`getElementById`**
- Whether these methods can be used to find and manipulate elements in a web component depends on whether/how it uses a shadow DOM
  - no shadow DOM - YES
  - shadow DOM with “closed” mode - NO
  - shadow DOM with “open” mode - YES, if methods are called on **`shadowRoot`** property

`shadowRoot` property is not set, but **DevTools** can still view and manipulate the elements

The **`shadowRoot`** property holds an instance of the **`ShadowRoot`** class which inherits from **`DocumentFragment`** class.

... Shadow DOM Access

- Suppose a document contains an instance of **`radio-group`** web component that uses shadow DOM in “open” mode
  - this **does not** find **`label`** elements inside web component

```
const labels = document.querySelectorAll('label');
```

  - this **does** find **`label`** elements inside web component

```
const labels = document.querySelector('radio-group').shadowRoot.querySelectorAll('label');
```

  - this changes the **`label`** of first radio button inside web component

```
document
  .querySelector('radio-group')
  .shadowRoot
  .querySelector('label')
  .textContent = 'Rouge'
```

Events

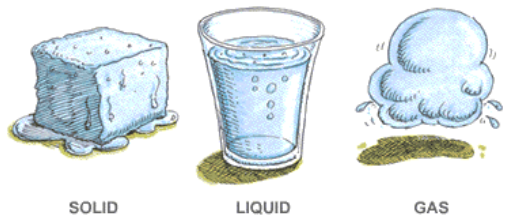
- Web components can dispatch any number of events, perhaps to advertise their state changes
- Code outside web components can listen for the events
- Example
  - in **`radio-group.js`**, add following at end of **`set value`** method

```
this.dispatchEvent(new Event("change"));
```

  - listen for the event with code like this

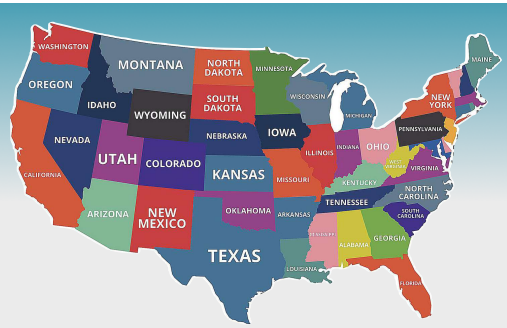
```
const rg = document.querySelector("radio-group");
rg.addEventListener("change", (event) => {
  const { value } = event.target;
  console.log("value =", value);
});
```

Sharing State ...



- Many approaches can be used to share state between web component instances
- A simple approach is to hold all shared state in a singleton object that
  - has a private property, getter method, and setter method for each piece of state
  - has a method that registers callback functions to be called when a specific piece of state changes
  - implements setter methods to call each registered callback function when the value is changed

... Sharing State ...




```
class State {
  static instance = new State();
  #favoriteColor = "transparent";
  #propertyToCallbacksMap = new Map();

  constructor() {
    if (State.instance) {
      throw new Error(
        "get singleton instance with State.instance"
      );
    }
    State.instance = this;
  }

  addCallback(property, callback) {
    let callbacks = this.#propertyToCallbacksMap.get(property);
    if (!callbacks) {
      callbacks = [];
      this.#propertyToCallbacksMap.set(property, callbacks);
    }
    callbacks.push(callback);
  }

  changed(property) {
    const callbacks =
      this.#propertyToCallbacksMap.get(property) || [];
    for (const callback of callbacks) {
      callback(this[property]);
    }
  }
```

Add a private instance variable for each piece of state to be shared.

Add getter and setter methods like these for each piece of state to be shared.

```
  get favoriteColor() {
    return this.#favoriteColor;
  }

  set favoriteColor(color) {
    if (color === this.#favoriteColor) return;
    this.#favoriteColor = color;
    this.changed("favoriteColor");
  }
}

window.State = State;
```

... Sharing State

Add setter methods similar to this in each web component that wishes to tie its state to the singleton **`State`** object.

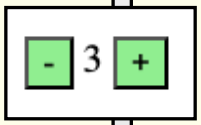
```
set value(newValue) {
  if (newValue === this.#value) return;

  this.#value = newValue;
  this.#internals.setFormValue(newValue);
  const input = this.shadowRoot.getElementById(newValue);
  if (input) input.checked = true;
  this.dispatchEvent(new Event("change"));
}
```

Add code similar to this to wire up web components to the singleton State object.

```
window.onload = () => {
  const state = State.instance;

  const rgs = document.querySelectorAll("radio-group");

  // Add an event listener to each radio-group web component.
  for (const rg of rgs) {
    rg.addEventListener("change", (event) => {
      state.favoriteColor = event.target.value;
    });
  }

  // When the favoriteColor state changes,
  // update all the radio-group web components.
  state.addListener("favoriteColor", (color) => {
    for (const rg of rgs) {
      rg.value = color;
    }
  });
};
```

Vanilla Counter Component

```
const template =
document.createElement("template");
template.innerHTML = /*html*/ `
  <style>
    :not(:defined) {
      visibility: hidden;
    }

    .counter {
      display: flex;
      align-items: center;
      gap: 0.5rem;
    }

    button {
      background-color: lightgreen;
    }

    button:disabled {
      background-color: gray;
    }
  </style>
  <div>
    <button id="decrement-btn"
      type="button">-</button>
    <span></span>
    <button id="increment-btn"
      type="button">+</button>
  </div>
`;
```




```
class CounterVanilla extends HTMLElement {
  static get observedAttributes() {
    return ["count"];
  }

  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  attributeChangedCallback() {
    if (this.isConnected) this.#update();
  }

  connectedCallback() {
    const root = this.shadowRoot;
    root.appendChild(template.content.cloneNode(true));

    this.decrementBtn = root.querySelector("#decrement-btn");
    this.decrementBtn.addEventListener("click", () => {
      this.decrement();
    });
    root.querySelector("#increment-btn")
      .addEventListener("click", () => {
        this.increment();
      });

    this.span = root.querySelector("span");
    this.update();
  }
```

Vanilla Counter Component

```
  get count() {
    return this.getAttribute("count") || 0;
  }

  set count(newCount) {
    this.setAttribute("count", newCount);
  }

  decrement() {
    if (this.count == 0) return;

    this.count--;
    if (this.count == 0) {
      this.decrementBtn.setAttribute("disabled", "disabled");
    }
    this.#update();
  }

  increment() {
    this.count++;
    this.decrementBtn.removeAttribute("disabled");
    this.#update();
  }

  #update() {
    if (this.span) this.span.textContent = this.count;
  }
}

customElements.define("counter-vanilla", CounterVanilla);
```

treating `count` attribute as the source of truth

`this.count` gets converted to a string, so we need to use == instead === here

Simplifying Counter

- Previous example requires a lot of code
- Can simplify using a custom superclass of **`HTMLElement`** that I call **`Wreck`** (**W**eb **RE**active **C**omponent **K**it)
- Fewer features than Lit, but much smaller and requires no tooling or build process
  - automatically wires event listeners
  - automatically implements reactivity
  - supports 2-way data binding for HTML form elements (**`input`**, **`select`**, and **`textarea`**)
- Let's see how Wreck simplifies the counter component

Wreck Counter

```
import Wreck from "./wreck.js";

class CounterWreck extends Wreck {
  static properties = {
    count: { type: Number, reflect: true },
  };

  css() {
    return /*css*/ `
      :not(:defined) {
        visibility: hidden;
      }

      .counter {
        display: flex;
        align-items: center;
        gap: 0.5rem;
      }

      button {
        background-color: lightgreen;
      }

      button:disabled {
        background-color: gray;
      }
    `;
  }
```

352 lines of code including comments and blank lines

Add following to class to contribute to form submissions:
`static formAssociated = true;`

Setting a property `reflect` option to `true` causes the corresponding attribute to be updated when the property value changes.

```
  html() {
    return /*html*/ `
      <div>
        <button
          disabled="this.count === 0"
          onclick="decrement"
          type="button">-</button>
        <span>this.count</span>
        <button onclick="this.count++"
          type="button">+</button>
      </div>
    `;
  }

  decrement() {
    if (this.count > 0) this.count--;
  }
}
CounterWreck.register();
```

Wreck looks for attributes whose name begins with "`on`" and whose value is the name of a method.

Wreck looks for attribute values and text content containing "`this.propertyName`" and adds reactivity.

DevTools Tips

These work in Chrome, Firefox, and Safari.

- **To change a web component attribute value**
  - select Elements tab
  - select web component
  - double-click an attribute value
    and enter a new value
  - commit change by pressing return or tab key,
    or clicking elsewhere
  - verify that the component UI updates

- **To change a web component property**
  - select Elements tab
  - select web component
  - select Console tab
  - enter `$0.propertyName = newValue`
  - verify that the component UI updates
  - if property has `reflect` option set to `true`,
    verify that the corresponding attribute updates

Resources

- **WebComponents.org** - https://www.webcomponents.org/
  - “building blocks for the web”
- **MDN docs** - https://developer.mozilla.org/en-US/docs/Web/API/Web_components
- **Open Web Components** - https://open-wc.org/
  - “guides, tools and libraries for developing web components”
- **Custom Elements Everywhere** - https://custom-elements-everywhere.com/
  - “making sure frameworks and custom elements can be BFFs”
- **Kinsta** - https://kinsta.com/blog/web-components/
  - “complete introduction to web components”
- **My blog** - https://mvolkmann.github.io/blog/
  - select “Web Components”

hard cover **soft cover** **mine**

Wrap Up

- Web components provide a **nice alternative** to implementing UI components using popular frameworks
- They are **standards-based** and "use the platform"
- They **can be shared** across apps that use frameworks
- They don't necessarily require libraries, tooling, or build processes
- They have a **small learning curve**, but that mostly involves learning more about the DOM ... which is good to know