Vanilla JS Web Components

A vanilla approach to single-page applications

K. Devin McIntyre

ConFoo 2022

Why vanilla web components?

- Self-contained and reusable
- Usable in any framework or none
- Synchronous events and lifecycles
- Supported in all modern browsers*

Custom Elements

- Created as a JavaScript class
- Synchronous lifecycle listeners: DOM attach, DOM detach, and attribute changes
- "Autonomous": extends HTMLElement
  - <custom-tag-name></custom-tag-name>
- "Customized built-in": extends other elements
  - <div is="custom-tag-name"></div>
  - Safari refuses to implement support for them and requires a polyfill

Shadow DOM

- Can attach to any element
- Creates "shadowRoot" document fragment inside the element to contain child elements
- <style> tags in the shadow root affect only the elements inside
  - Use CSS properties to "reach in" if needed

<template> and <slot>

- <template>: reusable HTML without render
- <slot> positions Light DOM elements within your shadow DOM
  - Variable contents
  - Browser autofill inputs must be in Light DOM
- Element children not inside its shadowRoot will show up inside <slot> element(s)
- Can listen for 'slotchange' event

Things in a single-page application (SPA)

- Code architecture
- Encapsulated components
- Two-way data binding
- State management
- Navigation
- Automatic translations

Code architecture ideas

- View and model: Custom Elements
- Controller: (singleton) modules
- Non-coupled communication: DOM events, CustomEvent
- File load: ES6 import/export
- HTML content: fetch + DOMParser
- Global/variable CSS: CSS properties
- HTML structure: semantic HTML with aria

Encapsulated components

- Custom Element with optional shadow DOM
- Put all listeners, properties, and methods into the component class
- DOM attachment: "connectedCallback" and "disconnectedCallback"
- HTML content
  - fetch + DOMParser if content is outside the file
  - Template strings for content kept in the same file
  - Use <template> element if there are multiples: lists, rows, grids

Two-way data binding

- Custom Element "attributeChangedHandler"
- Keep your data as attributes on the element
- Use addEventListener for inputs, drag events, etc.
  - Note: use this.shadowRoot.addEventListener

State management

- Communicate up
  - Use "dispatchEvent" to send up
    - Use { composed: true, bubbles: true} if inside shadowRoot
  - Listen for bubbled events from child elements
- Communicate down one level at a time
  - Change observed attributes on child custom elements
- Communicate global, sideways, deep down, etc.
  - Send and listen to custom events on document or window

Navigation

- Listen for events "hashchange" and "popstate"
- Use hash-based navigation
  - Otherwise, you need an interval to check the URL for changes
- Old-school anchor links: <a href="#cats">Cats page</a>
- history.pushState(null, null, path + query + hash)
- On "hashchange"/"popstate", load new view

Automatic translations

- Pick a set of attributes to hold translation keys
  - E.g. i18n="text.content", i18n-alt="alt.content"
- Use MutationObserver
  - Listen for attribute changes of i18n-*
  - Listen for new nodes for when new things load in the DOM
  - Observe all document fragments (shadowRoot) separately
- Translate each affected element when the observer is triggered
- For label inserts, you can do more attributes
  - E.g. i18n-insert-name="Michael"

What about Virtual DOM?

- Not available in vanilla JS
- Why is it needed? Rendering only the things that changed
  - The attributeChangedCallback rendering for one attribute at a time naturally limits what gets changed
  - An attributeChangedCallback-sourced change occurs in the same event cycle as the attribute that changed, so extra rendering does not occur
  - Need to be careful with MutationObserver-sourced changes because they cross over multiple event cycles

https://github.com/miyasudokoro/web-component-demo

see also: https://developer.mozilla.org/en-US/docs/Web/Web_Components