BUILDING A SHARED COMPONENT LIBRARY IN LIT

WHAT IS THIS TALK?

- What drove me to Lit
- How Lit makes the solution so much better
- Lets Build A Thing™ In Lit
- Who Actually Uses Lit in Production Right Now

NYPD
BURTON SMITH'S CODEMASH 2023 WEB COMPONENT PRE-COMPILER
THIS TALK
imgflip.com

Speaker notes

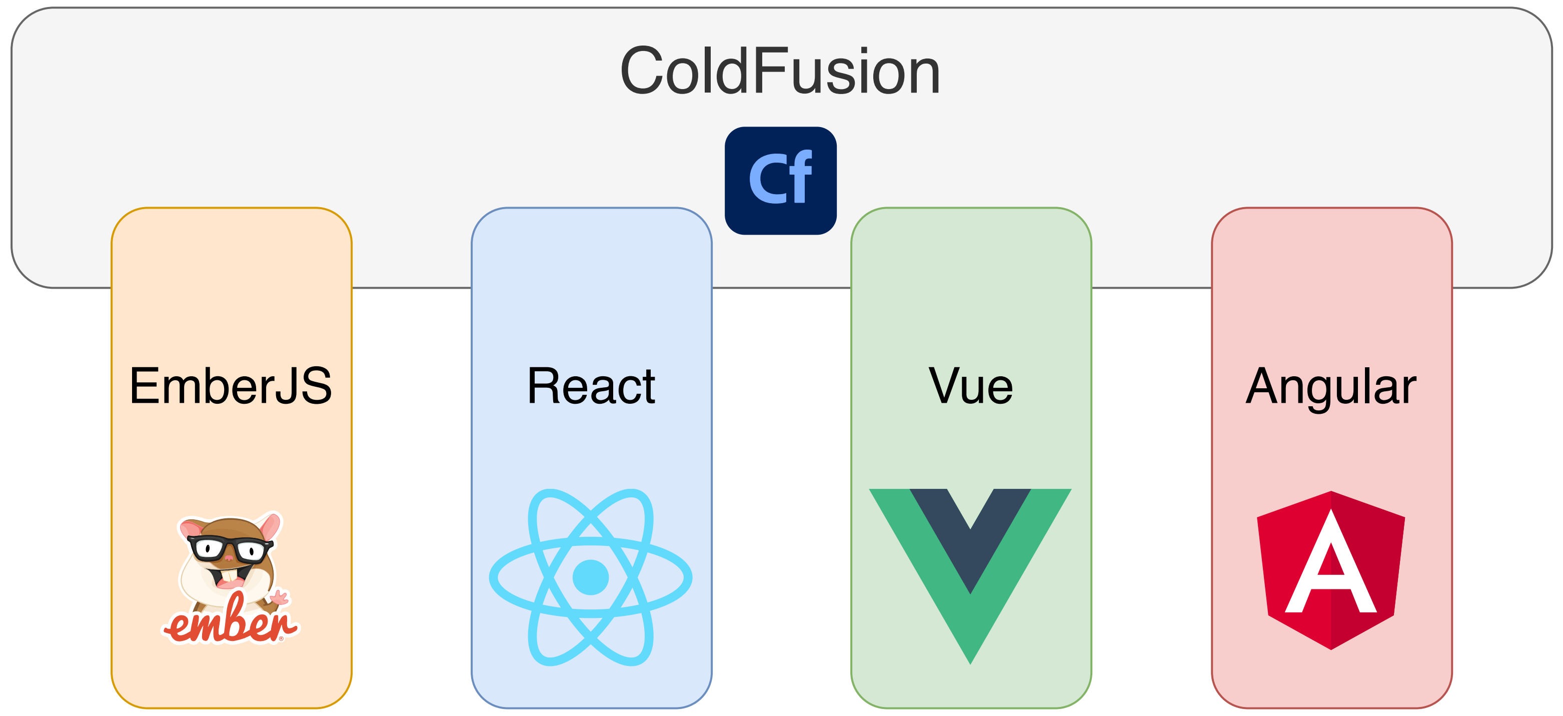
So, if any of you went to the Web Components pre compiler on Teusday morning, you might get the slightest hint of deja-vu from this talk. In the wildest stroke of unintentional luck, it turns out Burton stole my entire talk and padded it out to 4 hours (and expertly done at that). Jokes aside, I'm not kidding that this talk is basically a highly compressed version of that pre-compiler. There are a couple differences/new things here, but if you want to leave now, you'll only hurt my feelings a little bit.

SO, WHO AM I?

Fellow Engineer at Fuse

Really into Javascript and Javascript Accessories

THE WHY



Speaker notes

So, the product I work on at Fuse was originally written in ColdFusion. Just one big monolith, 90% of the business logic in the database, the whole 9 yards. The people who built it were solid developers, but they were decidedly not front end devs, and so the css in particular suffered for it.

Then, still before my time, new devs joined, old devs left, it was decided they should start modernizing the front end, they chose Ember for the task, and they got to it. Replaced most of the external facing pages wrote their own shiny new CSS, and everything was great.

But same old same old, teams shifted, etc, etc, the new devs decided that React was the new hotness. They modernized a few of the remaining external pages that'd been left in CF, and even started in on the older ember pages. Complete with their own, even shinier and newer CSS, and everything was awesome.

At the same time, another team in a more sectioned off part of the app decided they wanted to use Angular to start modernizing their peice. And so they did. With their own shiny new CSS. All built in the aim of adhering to the style guide granted to us from on high.

New teams, new devs, etc, etc, now Vue is the new hotness. Complete with those devs opinions on how best to implement the style guide, complete with the newest and shiniest CSS. You can see where this is going.

Now, the real fun part is that our styleguide is Material-ish. Its close enough that you can almost get away with using whatever material library for your framework, but you will inevitably have to tweak something. This leads to multiple teams using multiple material libraries, each having to tweak them in unique ways, and not tweaking them consistently with each other, such that you'd end up in an uncanny valley situation as you went between pages that used different frameworks, where certain things would like *almost* the same, and function *almost* the same, but they'd be off in vague and unsettling ways.

Now, there was a component library developed by an in house team, and written in Angular. Obviously, that doesn't help the non-Angular product teams, so they also compiled those components using Angular Elements. Works great, except. Now you're shipping Angular + that pages main framework. And there's angular-isms in Angular that don't actually translate well, or at all, in to Angular Elements. There were actually 3 of the original Angular components that just never made the jump to Angular Elements, and there were a handful more that worked, but only in the most simple cases.

Eventually, that component library fell to me to maintain, and I eventually just got tired of dealing w/ Angular and went hunting for something else that would consistently work across all of our frameworks, and I landed on web components.

SO WHY WEB COMPONENTS?

- Web Standard since HTML5
- Use them in any framework
- Shadow DOM
- Progressive Enhancement

Speaker notes

Web components have been a thing for almost a decade now. All current browsers support them. You could even run them in IE11 thanks to polyfills, but now that Microsoft is ended support for IE11 we don't even have to worry about that.

They work in any framework. Merely load the JS and use the tags and you're good to go. Now.. there are some caveats to the actual useability in certain frameworks. In Vue, you need to tell the compiler that any tag matching a pattern is a web-component and it should treat them as such. It'll still do it, but it'll warn you in dev mode that its assuming this is what you meant. Angular needs you to setup the custom elements schema in the module where you're rendering them. React is another story. Basic attributes work, but event binding is a mess, and so are any complex attributes/properties. There are wrappers around to wrap any web component in a way that React can use them "like native", tho, and support for just being able to use them *should* be landing in React 19, so things are looking up there.

The Shadow DOM makes it so that your styles and html are encapsulated from the host document. No more fighting styles (mostly), no more important wars. Your component will look exactly as you intended on every page, no matter what css the host has loaded. There are of course ways to enable styling of internals of your web components, but those methods now become part of the API of your component, and must be fairly explicitly coded to in order to affect the styling.

You can just rewrite a single control into a web-component, load its javascript on the page, and replace the existing legacy control. You don't have to whole-sale rip your app apart to be able to make use of web components.

SO WHY LIT?

- The replacement for Polymer
- Builds standards complaint web-components
- Tiny Bundle

Speaker notes

At Google I/O 2018, Google announced that LitElement would be the replacement for the PolymerElement base class from polymer.

LitElement extends from HtmlElement and just adds a few niceties. This means you are just writing straight web-components, with a few extra features thrown in to handle reactive state and declarative rendering.

The entirety of Lit is only 5kb minified and gzipped, so it adds basically nothing to your bundle. Lit components in general are already pretty small already, so you end up with impressively small bundles.

The shared library I'm building at Fuse has somewhere around 15 different components, and the entire bundle is only 50kb minified and gzipped, and the majority of that is styles.

A COMPARISON

```
<template id="template">
  <style>
    .message {
      font-size: var(--hello-world-size, 5rem);
      color: var(--hello-world-color, blue);
    }
  </style>
  <span class="message"></span>
  <button class="count-button"></button>
</template>
<script type="text/javascript">
  class HelloWorldComponent extends HTMLElement {
    // in order to respond to changes to the name attribute
    static observedAttributes = ['name']

    // internal state
    count = 0

    constructor () {
      super()

      // clone the template
      const content =
        document.getElementById('template').content.cloneNode(true)

      // set the message content
      content.querySelector('.message').textContent =
        `Hello, ${this.getAttribute('name') ?? 'World'}`

      // set the button content
      const button = content.querySelector('.count-button')
      button.textContent =
```

```
// register the component
@customElement('hello-world')
class HelloWorldComponent extends LitElement {
  // styles
  static styles = css`
    .message {
      font-size: var(--hello-world-size, 5rem);
      color: var(--hello-world-color, blue);
    }
  `

  // wire up an attribute called `name` and default it to `World`
  @property() name = 'World'

  // internal state
  @state() private count = 0

  // the content of the element
  // will automatically rerender when name or count change
  render () {
    return html`
      <span class="message">Hello, ${this.name}</span>
      <button class="count-button" @click="${this.click}">
        This button has been clicked ${this.count} times!
      </button>
    `
  }

  // handle the button click
  click () { this.count += 1 }
}
```

Speaker notes

- element registration
- styling & shared stylesheets
- rendering
- attribute/state handling & re-rendering

ACCESSIBILITY IN WEB COMPONENTS

LABELS AND THE SHADOW DOM

Id's are only unique within a specific shadow layer.

This means that you can repeat id's, so long as they are in different shadows.

```
<shadow-tag>
  <!-- shadow dom -->
    <span id="something">
  <!-- end shadow -->
</shadow-tag>
<shadow-tag>
  <!-- shadow dom -->
    <span id="something">
  <!-- end shadow -->
</shadow-tag>
```

This also means that labels must be in the same DOM level as the element they are tied to.

Speaker notes

- id's don't cross the shadow boundary. This means that if you need an id for an internal element in your component, feel free to call it whatever. It will not and cannot collide with ids somewhere else in the dom.
- however, this also means that things which use ids to link themselves together (like labels and their target element) must be in the same DOM.
- this also applies to any aria-* attributes that use ids as their values

LABELS AND THE SHADOW DOM

So there are two ways to do this:

- Require label and the element its for to be passed in as slotted elements. This way they both come from the application and are at the same level.

```
<shadow-tag>
  <label for="my-input">label text</label>
  <input id="my-input" />
</shadow-tag>
```

- Take the label text in as a prop (or slot it if you feel like it) and render both controls internally

```
<shadow-tag label-text="whatever">
  <!-- shadow dom -->
    <label for="my-internal-input">${this.label-text}</label>
    <input id="my-internal-input" />
  <!-- end shadow -->
</shadow-tag>
```

Speaker notes

- Because of this, there are 2 ways to go about handling this.
- Option 1: make the application using your component pass in both the label and the input.
    - This gives the application the most control over its inputs and such, and makes the lit component be responsible mostly for rendering/styles.
    - You can still get ahold of the input in lit if you need to attach events / otherwise control it.
- Option 2: you render the label and the input internally.
    - You can always take in the label text as a prop or a slot if necessary to allow that level of customization.
    - Change & Input events are composed, so the application can still listen for those events on the root lit tag and they'll come through from the internal input just fine.
- The option is up to you (and I've used both styles of implementation, it really depends on what the purpose/complexity of your component is)

KEYBOARD FOCUS

Just works.

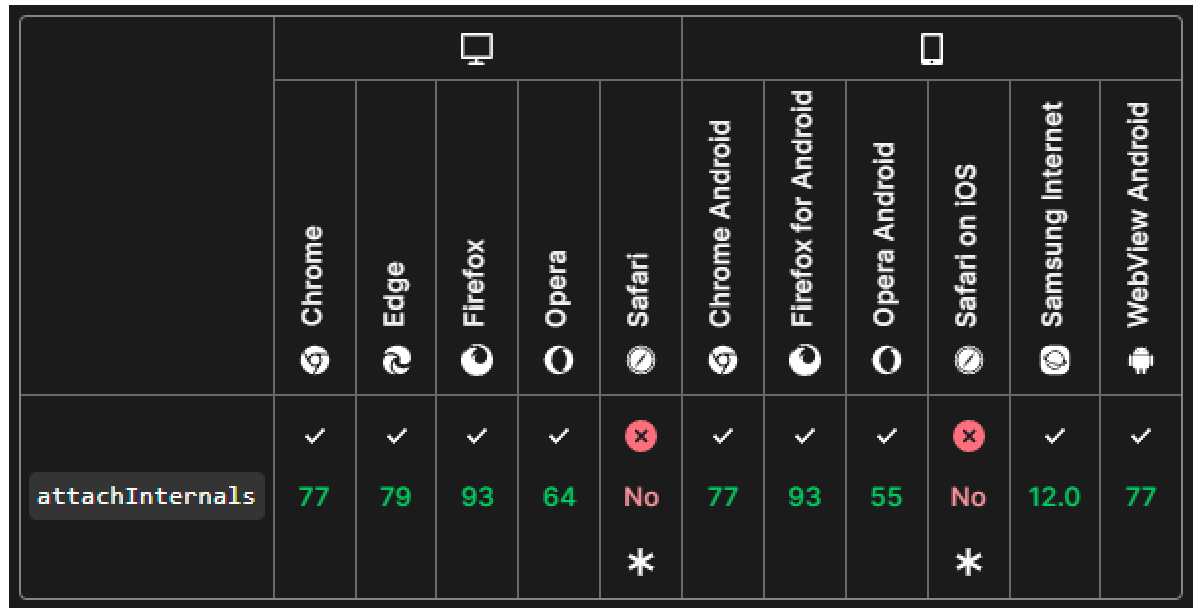
Anything that is normally keyboard focusable outside of a web-component will still be focusable inside it.

This includes but is not limited to:

- inputs
- things with an explicit tab-order
- everything you'd expect to be tabable

FORMS

You can even make a component interact with a form like a native element using `attachInternals`.



	🖥					📱					
	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
attachInternals	✓ 77	✓ 79	✓ 93	✓ 64	✗ No *	✓ 77	✓ 93	✓ 55	✗ No *	✓ 12.0	✓ 77

- `this.internals = this.attachInternals()` in the constructor of the element
- `this.internals.setFormValue` whenever the internal state/value of your component changes
- stick it in an html form, give it a name, ???, profit

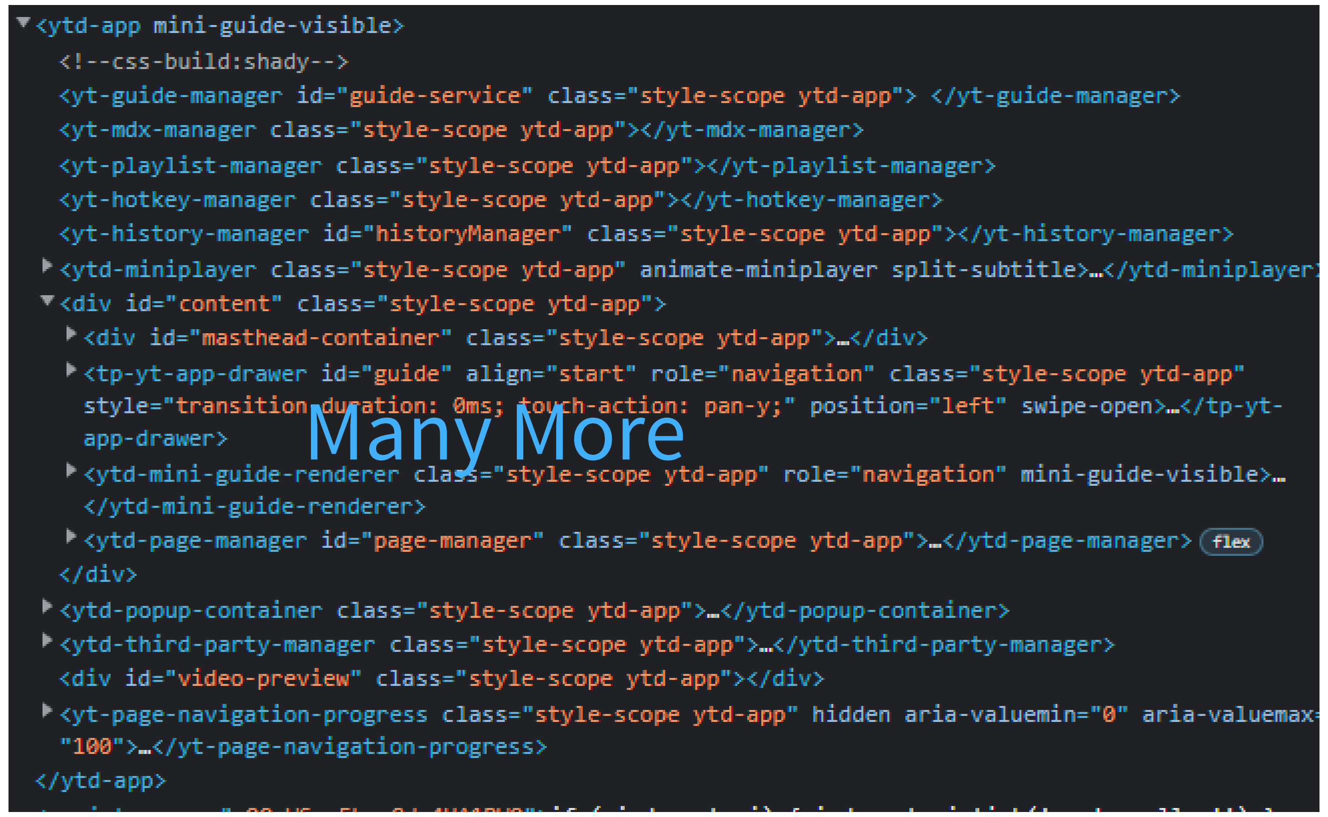
Speaker notes

- in every browser except safari its pretty easy
- you attachInternals in the constructor and store a reference to your new internals
- then when the value of your component changes, just call internals.setFormValue with that value and you're good to go
- internals also has methods to update form validity
- in safari, you'll need to locate the closest form when your element is connected and then add an eventListener for `formData` and set the value of your element in the form data on that event
- it is entirely possible to write a base class that handles all of this for you, and then all you need to do is implement it and set `this.value` correctly for any future component that needs to interact with a form

Demo Time!

IS THIS ACTUALLY IN PRODUCTION ANYWHERE?

YouTube:



```
<ytd-app mini-guide-visible>
  <!--css-build:shady-->
  <yt-guide-manager id="guide-service" class="style-scope ytd-app"> </yt-guide-manager>
  <yt-mdx-manager class="style-scope ytd-app"></yt-mdx-manager>
  <yt-playlist-manager class="style-scope ytd-app"></yt-playlist-manager>
  <yt-hotkey-manager class="style-scope ytd-app"></yt-hotkey-manager>
  <yt-history-manager id="historyManager" class="style-scope ytd-app"></yt-history-manager>
  <ytd-miniplayer class="style-scope ytd-app" animate-miniplayer split-subtitle>…</ytd-miniplayer
  <div id="content" class="style-scope ytd-app">
    <div id="masthead-container" class="style-scope ytd-app">…</div>
    <tp-yt-app-drawer id="guide" align="start" role="navigation" class="style-scope ytd-app"
    style="transition duration: 0ms; touch-action: pan-y;" position="left" swipe-open>…</tp-yt-
    app-drawer>
    <ytd-mini-guide-renderer class="style-scope ytd-app" role="navigation" mini-guide-visible>…
    </ytd-mini-guide-renderer>
    <ytd-page-manager id="page-manager" class="style-scope ytd-app">…</ytd-page-manager> flex
  </div>
  <ytd-popup-container class="style-scope ytd-app">…</ytd-popup-container>
  <ytd-third-party-manager class="style-scope ytd-app">…</ytd-third-party-manager>
  <div id="video-preview" class="style-scope ytd-app"></div>
  <yt-page-navigation-progress class="style-scope ytd-app" hidden aria-valuemin="0" aria-valuemax
  "100">…</yt-page-navigation-progress>
</ytd-app>
```

LINKS AND THINGS

- Github: https://github.com/kaiyote
- This Talk: https://github.com/kaiyote/lit-component-library-talk
- LinkedIn: https://www.linkedin.com/in/timothy-huddle/
- I don't do socials, tbh ¯_(ツ)_/¯