BUILDING A SHARED COMPONENT LIBRARY IN LIT

WHAT IS THIS TALK?

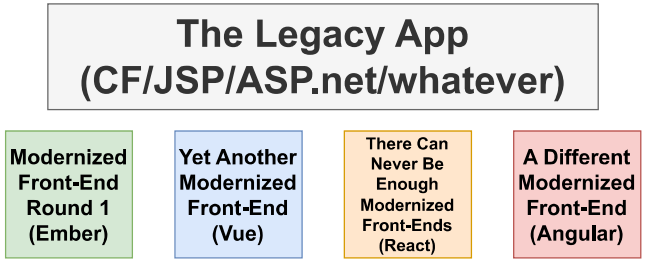
- The Scenario that led me down this path
- How Lit solves a lot of the problems inherent in The Scenario
- Lets Build A Thing™ In Lit
- Who Actually Uses Lit in Production Right Now

SO, WHO AM I?

Fellow Engineer at Fuse

Really into Javascript and Javascript Accessories

THE NIGHTMARE SCENARIO



Speaker notes

In the distant past, the original dev team made the, I'll call it questionable, decision to write the entire application as a monolithic ColdFusion application, complete with 90% of the business logic embedded into the database.

In the less distant past, a new dev team was brought in, and started using the new-and-shiny-at-the-time front-end framework of Ember 3.0 to rebuild the user experience. They also built their own microservices to talk to, but that is unimportant to this talk.

Then, corporate bought another product, that dev team likes React, so they start building their chunk of the app in React. It doesn't interact with the ember portion, so its "fine".

History repeats itself, and new teams are acquired, each with their own existing front-end preferences, and so everything just gets stitched together from various single-page applications.

You can get all of the pages to mostly look the same with some global CSS and some gentle guidance, but when it comes to user interactions with more complex controls, it starts to break down. I can tell you from experience that even things as simple as the speed of the ripple in a material button differs across the various implementations.

SO WHY LIT?

- The replacement for Polymer
- Builds standards compliant web-components
- Use them in any framework*
- Tiny bundle
- Scoped styles

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.
- You can use web components in any framework. There are a couple of things you need to do however
    - React: LitLabs actually has a package to generate React components from the class of a Lit component. So they `just work` for the most part
    - Vue: You'll need to tell the vue compiler that tags that match a specific pattern are web-components and it should treat them as such, but even if you don't, the worst that happens is just a lot of console warnings in dev mode.
    - Angular: You'll need to add CUSTOM_ELEMENTS_SCHEMA to the NGModule that is trying to render it.
- 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 work has somewhere around 15 different components, and the entire bundle is only 50kb minified and gzipped, and the majority of that is styles.
- So long as you don't turn off shadowDom, all your styles are contained within the shadow tree. This means they can't leak out, and styles from the host application can't leak in (except for fonts and css custom properties). What this really means is your component will look exactly as its supposed to look in all circumstances, no matter what the host application is doing style-wise.

REACTIVE STATE

`@property` - Exactly what it sounds like. Identical to `props` in React, Angular, Vue

- sets up watchers for both attribute and object property versions of the `@property`
  - `<my-tag prop=""></my-tag>` or `myTag.prop = whatever`
- has conversion support for the basic JS types (String, Boolean, Number, Object, Array) for things passed in as attributes
- you can write your own custom converter if you're feeling fancy

Speaker notes

- You just decorate each class property that you want exposed as a property with this decorator.
- It generates custom getter/setters that handle all the of the change detection to trigger rerenders, etc
- If necessary, you can write custom converters for converting from attributes on the element tag to the type/shape you need them to be
    - TBH, these shouldn't be necessary very often
- State works just like property does, except it doesn't expose anything to the outside

REACTIVE STATE

`@state` - Internal component state. Triggers a re-render on change just like `@property` does

- generates custom getters and setters which make `this.prop = newValue` trigger a rerender automatically
- doesn't need "dirty checking"

EVENTS

Properties down, Events up.

Aka, transmit data down to child components via properties, child components communicate back up to their parents via events.

```
<tag-with-event @event-name=${handlerFunc}></tag-with-event>
```

A couple of things to watch for with events in the land of web-components however:

- `composed: true` in the event init options to make an event bubble out of the shadow dom it was dispatched in
- This can be useful for keeping "internal" events truly internal, just dont set `composed: true`
- `this.dispatchEvent(new CustomEvent('child-did-a-thing', { detail: something, composed: true }))`

Speaker notes

- Under most circumstances, anything that needs to be transmitted to a child should be done via its defined reactive properties
- Anything a child needs to communicate to a parent element should be done via events (most likely CustomEvents)
- Under some circumstances, it may make more sense for elements to have a public functional api, and this is also doable, but it shouldn't be the rule.
- According to MDN: The read-only composed property of the Event interface returns a boolean value which indicates whether or not the event will propagate across the shadow DOM boundary into the standard DOM.
- User-Agent dispatched events are composed (Click/Mouseover/Touch/Copy/Paste/etc), but most other events are not by default.
- This means any "click" event that triggers from anything inside of your web-component will be visible to the outside, but that's probably a good thing (think of having a web-component that is a button, you'd want the buttons click to propagate out of your shadow so the host app can see it and respond).

COMPOSITION IN LIT

This works like every other framework.

Define a child component, render it in the parents template. Do all the normal things with it.

Lit also defines some "directives" that can help w/ repetition/render performance/etc.

- `repeat` - render a template for each element of an iterable, with optional keying
- `choose` - its a switch statement, but as a function
- `when` - its an if statement, but as a function
- `ifDefined` - only render this attribute at all if the value of the argument is defined

There are also `ref`'s and memoization available.

Speaker notes

- `repeat(listOfItems, () => keyFun, () => html``)`
- `choose(something, [['array', of], ['tuples', and], ['render', functions]], () => defaultCase)`
- `when(booleanExpression, () => trueFun, () => optionalFalseFun)`
- `attr=${ifDefined(prop)}`

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.

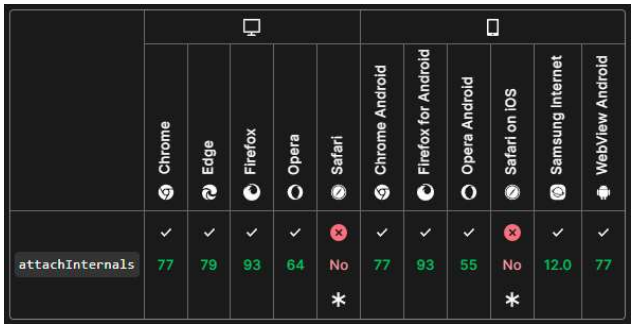
Anything that is normally keyboard focusable outside of a web-component will still be focusable inside it.

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:

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)_/¯