Bryan Braun

How searching for a bundle-free React led me to web components

Aug 31, 2020

I really like the ergonomics of React. I like the explicit state management, the intuitive templating, the flow of data, and the lack of boilerplate. It's onto some really good ideas!
So it was kind of a bummer that I couldn't really use it in the bundle-free dev workflows I've been experimenting with.
The problem is JSX—you gotta run it through Babel. There's no way around it!
As far as I could tell, I had a few options:

Skip JSX and use React.createElement instead. 😞

Replace JSX with a bundle-free alternative like hyperscript or htm. 🤔

Use a React-alternative written in Vanilla JS. 🏗

I was in the process of playing with the option three, when I discovered a fourth option that completely took me by surprise.

Web components

I know what you're thinking, but hear me out! I experimented with web components back in 2015, and things were bad. The API was awkward, Shadow DOM had weird constraints, browser support was terrible, and Polymer felt like a bloated attempt to make it all work. Worst of all, it wasn't clear what problem web components would help me solve. I moved on and didn't look back.
But recently, I learned about two things that caught my attention.
First, web components bring lifecycle hooks to DOM elements. If you add or remove a web component from the DOM, the lifecycle hooks will fire. This is a DOM feature... no frameworks involved!
Second, you can instantiate a web component the same way you instantiate a React component (at least, a class-based one). It looks like this:

```
class MyComponent extends HTMLElement {
  constructor() {
    super();
    ...
```

```
    }

    // Lifecycle hooks
    connectedCallback() {
      ...
    }
    disconnectedCallback() {
      ...
    }
  }
```

This is not the awkward API I remembered. All it needed was a `render` function to run whenever it changes state, and we'd be pretty dang close to the ergonomics of React. Maybe I could write a thin wrapper around `HTMLElement`, and we'd be there.

I was about to try it out when I discovered that this had already been done in a project called LitElement.

LitElement

LitElement calls itself "A simple base class for creating fast, lightweight web components." Here's an example component built with LitElement implementing a "Dark Mode" checkbox:

```
import { LitElement, html } from '../web_modules/lit-element.js';

export class DarkMode extends LitElement {
  constructor() {
    super();

    this.isDark = false;
  }

  static get properties() {
    return {
      isDark: { type: Boolean }
    };
  }

  handleChange() {
    this.isDark = !this.isDark;
  }

  render() {
    const theme = this.isDark ? 'dark' : 'light';
    document.body.setAttribute('data-theme', theme);

    return html`
      <label>
        <span>Dark Mode</span>
        <input type="checkbox" ?checked="${this.isDark}" @change="${this.
      </label>
    `;
  }
}
```

That looks pretty familiar! It's got state management, event handling, and intuitive templating. Also: no bundler needed.

I was still a little skeptical so I decided to build a little demo app to put it to the test. These were my first impressions:

The templating (which uses lit-html) is really great. The `render` function works just like React's, event handlers are nice, and I love that it uses template strings. This was my favorite part!

Styling is done with template strings, and the styles are scoped to the component. It has constraints, but I was able to work around them.

Props took some getting used to. `LitElement` props are like a combination of React props and local state. At first I was bummed to lose the word "state" but it wasn't a big deal once I got used to it.

I was pleasantly surprised at how well it worked with form inputs (which are often a sticking point because they are stateful).

It was nice seeing my custom element tags in the real DOM. No special browser plugins needed for debugging.

It was easy to integrate with a global state management solution.

The docs were pretty good, though the interactive tutorial was broken for me.

There's a VSCode plugin which was great for adding syntax highlighting to my template strings.

Overall, I was pretty satisfied with the experience. That being said, my app was pretty small, so I still have a few questions, like:

How well does it work with nested components? It looks promising, but the React apps I've worked on were *very* heavily nested.

Would the CSS scoping run into issues for much larger projects?

Also, browser support seems pretty good, but I haven't done much testing so I don't know for sure.

Could React ever be refactored to use web components?

Seeing the similarities between LitElement and React made me wonder: Could React ever be refactored to use web components?

Maybe! Web components support lifecycle hooks, props, custom element tags, scoped CSS, children (or "slots"), and more. The basic features are there. Also, it seems like these APIs are fairly low-level, so React could hide at least some of them under an abstraction.

Still, I think there are some major obstacles. React is big on functional components right now (see hooks), and it's unclear how well these would map to web components (update: this has already been done too—see Haunted). React's lifecycle hooks look pretty different from the DOM ones. And things like Shadow DOM would also behave pretty differently.

I suspect that any refactoring to use web components would be a major (and backwards-incompatible) change.

I think it's more likely that we see new frameworks pop up, borrowing ideas from React, and building on web components to reduce overhead. Indeed, it looks like LitElement has done exactly that.

Edit this post

Related posts: A minimalist development workflow using ES6 modules and Snowpack , Alt-React and Nonsense

---

If you liked this, subscribe and get future posts in your inbox:

you@email.com

Subscribe

Comments

This site is open source. See it on Github.
Content is Licensed CC-BY and
available via RSS & JSON.
© Copyright 2024