Generic Blog (Web) Components

A while back I wrote a set of Vue 2 components for a WordPress blog. It wasn't a complete project, was missing functionality that I couldn't figure out how to implement and I was afraid it would lock me in to a framework (although I would rather get locked into Vue than any of the other frameworks I've seen out there).

Rather than work with a specific framework, I decided to revisit this as custom elements/web components.

according to Custom Elements Everywhere most frameworks, at least those that were tested by the site, have pretty good support for custom elements. Even React works with custom elements as long as you make some modifications. There is also an experimental React branch that fully supports custom elements but there's no guarantee that the branch will be merged in to the main React codebase.

So what are web components?

Web components are a set of technologies that enable developers to create custom HTML elements that can be used on web pages.

The component technologies for web components are:

- **Custom elements**: A set of JavaScript APIs that allow you to define custom elements and their behavior, which can then be used as desired in your user interface
- **Shadow DOM**: A set of JavaScript APIs for attaching an encapsulated "shadow" DOM tree to an element — which is rendered separately from the main document DOM — and controlling associated functionality. In this way, you can keep an element's features private, so they can be scripted and styled without the fear of collision with other parts of the document.
- **HTML templates**: The <template> and <slot> elements enable you to write markup templates that are not displayed in the rendered page. These can then be reused multiple times as the basis of a custom element's structure.

The basic approach for implementing a web component generally looks something

like this:

- Using the ECMAScript 2015 class syntax create a class in which you specify your web component functionality
- Register your new custom element using the `CustomElementRegistry.define()` method, passing it the element name to be defined, the class or function in which its functionality is specified
- Attach a shadow DOM to the custom element using Element.attachShadow() method
    - Add child elements, event listeners, etc., to the shadow DOM using regular DOM methods.
- Define an HTML template using <template> and <slot>
    - Again use regular DOM methods to clone the template and attach it to your shadow DOM.
- Use your custom element wherever you like on your page, just like you would any regular HTML element.

```
// Create a class for the element
class PopUpInfo extends HTMLElement {
  constructor() {
    super();

    // Create a shadow root
    const shadow = this.attachShadow({mode: 'open'});

    'open'// Create structure elementsonst wrapper = document.createEleme
    wrapper.setAttribute('class', 'wrapper');

    const icon = document.createElement('span');
    icon.setAttribute('class', 'icon');
    icon.setAttribute('tabindex', 0);

    const info = document.createElement(''span';
    info.setAttribute('class', 'info');

    const text = this.getAttribute('data-text');
    info.textContent = text;
```

```
let imgUrl;
if(this.hasAttribute('img')) {
  imgUrl = this.getAttribute('img');
} else {
  imgUrl = 'img/default.png';
}

const img = document.createElement('img');
img.src = imgUrl;
icon.appendChild(img);

const style = document.createElement('style');

style.textContent = `
  .wrapper {
    position: relative;
  }
  .info {
    'class'ze: 0.8rem;
    width: 200px;
    display: inline-block;
    border: 1px solid black;
    padding: 10px;
    background: `
  .wrapper {
    position: relative;
  }
  .info {
    font-size: 0.8rem;
    width: 200px;
    display: inline-block;
    border: 1px solid black;
    padding: 10px;
    background: white;
    border-radius: 10px;
    opacity: 0;
    transition: 0.6s all;
    position: absolute;
```

```
        bottom: 20px;
        left: 10px;
        z-index: 3;
      }
      img {
        width: 1.2rem;
      }
      .icon:hover + .info, .icon:focus + .info {
        opacity: 1;
      }
    `
```

Using the Lit library

Lit is the spiritual successor to the Polymer library and it makes it easier to work with Web Components.

Instead of using plain custom elements, we'll take advantage of Lit features that will make the work easier.

An example custom element built with Lit:

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

export class CEElement extends LitElement {
  constructor() {
    super();
    //implementation
  }

  render() {
    return html`<span part="textspan">This text will be red</span>`;
  }
}

window.customElements.define('c-e', CEElement);
"textspan"`<span part="textspan">This text will be red</span>``<span part=
```

Additional specifications that enhance web components

There are additional specifications that enhance the existing web component applies

CSS modules and Constructable Stylesheets eliminate the need for <style> elements in your custom elements.

Using CSS modules you can import stylesheets using Javascript and then attach them to your custom elements' shadow roots.

The main advantage of this method is that you can do so with any number of custom elements on a page, as well as the root stylesheet for your page.

```
import sheet from './styles.css'
  assert {
    type: 'css'
  };

'css'// adds the sheet to the root document
document.adoptedStyleSheets = [sheet];

// Adds the imported styelsheet
// to a shadowroot
shadowRoot.adoptedStyleSheets = [sheet];
```

Earlier versions of the Web Components specifications had shadow piercing combinators to offer a way for the host document to style content inside custom elements. This was powerful but broke encapsulation and was later removed from the specifications.

Instead of the piercing combinator we now have CSS Shadow Parts as way to signal that we want to apply styles from an external stylesheet to the custom element.

the `c-e` element does this by using the `part` attribute with a name value in the template.

```
class CEElement extends HTMLElement {
  constructor() {
    super();
    //implementation
  }
  <template id="c-e-template">
    <span part="textspan">This text will be red</span>
  </template>
}

w"c-e-template"ements.define('c-e', CEElement);
```

In the CSS, we use the name of the custom element and the `::part` pseudo-element to select the part we want to style.

```
c-e::part(textspan) {
  color: red;
}
```

For more information, see

- Scoped Styles
- Keeping Your CSS Small

What components to build?

Here's a list of the minimal set of components I want to build for this project:

- Show all posts
- Show single posts
- Show all pages
- Show all categories
- Show post matching a category
- Show all tags
- Show posts matching a tags
- Post pagination
- Page pagination

As an example, I will build a component to show all posts (or at least the latest 10).

Building an example component

The easiest component to get started is the `blog-posts` component. It displays a list of the latest 10 posts.

We first import the components of the Lit library that we want to use:

- `LitElement` and `html` from the `lit` package
- `unsafeHTML` from its own package in the `lit/directives` there

We then create a class that extends the `LitElement` class. This means that we can use the features of the base class and expand it with functionality specific to the custom element we're creating.

We first define the properties for this component. In this case we have only one: the data that we'll retrieve from the API as an object.

We then run our constructor function. We call the super() to call the parent class's constructor (in this case LitElement's constructor) and then we run fetchData() using `this.fetchData()`.

This has initialized our class with both the properties from the parent class and our properties and methods.

We could also run `this.fetchData()` in one of the component lifecycle methods but I want to be absolutely sure that the data is loaded before we render the component.

`fetchData()` runs a fetch request for the posts endpoint at `https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?embedded=true` and sets the `data` property to the response data.

Promises are one way to handle asynchronous code. We could also use async/await to achieve the same goal.

The renderer is where the magic happens.

IF the data variable `this.data` is not set, we display a loading message since the fetch promises haven't fulfilled yet and these is nothing to show.

Once the data is loaded we can populate the template. We use the array.map method to ensure that we have an array to work with and the `html` tagged template to render the HTML for each component

In this instance WordPress has already sanitized the data for use so it's OK to use the unsafeHTML directive to render the content as HTML rather than text. If you can't ensure that the data is sanitized you should not use the `unsafeHTML` directive or you'll open your application to XSS attcks.

The template also provides part attributes so we can use the ::part pseudo-element to style the content inside the custom element from one master stylesheet outside it.

The final step is to define the element as a custom element. We use the customElements.define to associate the name that we want to use `blog-posts` witht he function that defines it `BlogPosts`.

```
import { LitElement, html } from 'lit';
import { unsafeHTML } from 'lit/directives/unsafe-html.js';

clas'lit/directives/unsafe-html.js'{
  static get properties() {
    return {
      data: Object,
    }
  }

  constructor() {
    super();
    this.fetchData();
  }

  fetchData() {}

  render() {
    if (!this.data) {
      return html`
          <h2>Loading...</h2>
        `;
```

```
    }
    return html`
      ${this.data
        .map((post) =>
          html`<article part="article-post">
            <h2><a href="${post.link}">${post.title.rendered}</a></h2>

            <div class="post-content" part="article-content">
              ${unsafeHTML(post.excerpt.rendered)}
            </div>
          </article>
          `
        )
      }
    `
  }

}

customElements.define('blog-posts', BlogPosts);
```

We can then use the custom element anywhere we want to by running the following steps:

Import the script as a module.

```
<script type="module" src="/path/to/blog-post.js"></script>
```

and use the tag you defined in the script wherever you want to place the element at

```
<blog-posts></blog-posts>
```

Future Evolution

The basic element works and it displays the content as we intended.

There are a few things that I will save for a future interation of the component.

Change the links

Right now the links point to the original server. The first iteration will change the links to individual posts.

Because the links point to individual pages, we need to figure out how to display the individual posts. Do we link to the JSON content of the individual post or do we create a custom URL?

Customize the element

The element currently uses the default values for the number of pages and the page number where we want to start:

- `per_page`: how many posts per page. The default is 10
- `page`: what page (or group) of posts we want to see. The default is 1

Use Storybook

Once we have our component ready, we can look at Storybook as a way to show all the available components to potential users.

Storybook provides a way to use it with web components.Introduction to Storybook for Web Components

Code Repository

The code for this project, both the custom elements and the Storybook data are available on Github at https://github.com/caraya/blog-components