Custom Elements revisited (once more)

This post will look at technologies allow us to create fully encapsulated and accessible web content.

- **Custom Elements** allows developers to create specialized elements for the web
- **Shadow DOM** gives us encapsualated styles that won't bleed into the parent document
- **Templates** provide inert chunks of HTML content that we can instantiate after creaation
- **Custom Properties** provide a way to create reusable styles that will work through the shadow DOM

Reviewing Custom elements

The idea behind custom elements is to allow developers to create their own valid HTML elements that encapsulate HTML content, CSS styles and Javascript behavior that is unique to that instance of the component.

At the most basic level the component is a class that inherits from HTMLElement, the base class for all HTML elements on the page, whether visible or not.

We can create constructor methods and use them to set up default behavior for the element. This behavior comes in handy when we extend a custom element rather than the default.

```
class DemoElement extends HTMLElement {

  // Can define constructor arguments if you wish.
  constructor() {
    // If you define a constructor, always call super()
    // first! This is specific to CE and required by
    // the spec.
    super();
```

```
  }

}

customElements.define('app-drawer', DemoElement);
'app-drawer'
```

To load the element we created we need to do two things:

First we load the web comoponents polyfill loader. This will take care of loading all the necessary polyfills asynchronously.

```
<script src="node_modules/@webcomponents/webcomponentsjs/webcomponents-loa
```

Next, we use methods of the web components polyfill to dynamically load the component's Javascript file.

```
<script type="module">
  function loadScript(src) {
    return new Promise(function(resolve, reject) {
      const script = document.createElement('script');
      script.src = src;
      script.onload = resolve;
      script.onerror = reject;
      document.head.appendChild(script);
    });
  }

  WebComponents.waitFor(() => {
    return loadScript('my-element.js');
  });
</script>
```

Adding Content to the component: Templates and ShadowDOM

So we've loaded the component but it has no content in it. To fix this we'll look at shadow DOM, templates, and Shadow Parts.

To fully understand how these things work together we will look at a working example using shdow DOM, templates and slots together. We will then discuss each part in depth.

The template has two named slots, one for the title and one for text. These will be the defaults if no content is entered in the custom element.

```
<template id="my-content">
  <h1><slot name="my-title">My Default Title</slot></h1>
  <p><slot name="my-text">My default text</slot></p>
</template>
```

The class we use to define the custom element changes the constructor to acommodate adding the shadow DOM to the element.

We capture references to the template and its content.

Then we create and attach a shadow root to the custom element and, finally, we add the content of the template to the shadow root.

```
class DemoElement extends HTMLElement {

  constructor() {
    // If you define a constructor, always call super()
    // first! This is specific to CE and required by
    // the spec.
    super();
    // Capture a reference to template
    let template = document.getElementById('my-content');
    'my-content'// Capture a reference to template content
    let templateContent = template.content;
```

```
    // Attach the shadow root to the custom elementhadowRoot = this.attach
    // Attach a 'open'// Attach a clone of the template to the shadow root
    .appendChild(templateContent.cloneNode(true));
  }
}
// Define a custom element using the class we createdfine('demo-element',
'demo-element'
```

Now that we know how they work, let's dive into the component pieces.

Templates: Inert pieces

Templates are the native solution to the problem of creating inert chunks of content that will not render until we instantiate them, can be used multiple times and will not impact layout until they are instantiated.

Because it is part of the HTML standard, we can use it for everything where we could have used custom solutions.

```
<template>
  <h1>This is my card title.</h1>
  <div class="content">
    <p>This is my content</p>
  </div>
</template>
```

For custom elements, templates become important when we add slots to customize the template content as we'll discuss in the next section.

Adding content from light DOM into the shadow

We now have a template that we can instantiate from, but the question is how to compose our template with information from the custom element.

Slots and named slots are the answer to this question. Using slots we're basically telling the browser: ***"Whatever content matches the name of this slot will replace the content of the slot in the template"***.

To make our lives easier we'll only talk about named slots.

Taking our template from the last section we will turn it into a reusable template for custom elements.

```
<template>
  <h1><slot name="card-title">Default title</slot></h1>
  <div class="content">
    <slot name="card-content">
          <p>This is my content</p>
    </slot>
  </div>
</template>
```

Shadow DOM: Encapsulate all things.

The final step we'll take is to wrap the slotted template in a shadow DOM.

The shadow DOM gives developers the ability to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree. Adding the template to a shadow DOM isolates the component's content from the page.

Elements like select, video or slider all use the shadow DOM to hide its children from the end user or the developer using your element. They don't need too know what you did under the hood.

In the example we used earlier in the post, we captured the template and the template content into variables and then attached a clone of the content into the element's shadow root (the root of the element's DOM subtree).

```
class DemoElement extends HTMLElement {
```

```
  constructor() {
    super();
    let template = document.getElementById('my-content');
    let templateContent = template.content;

    const shadowRoot = this.attachShadow({mode: 'open'})
    .appendChild(templateContent.cloneNode(true));
  }
}
customElements.define('demo-element', DemoElement);
```

With templates, slots and shadow DOM we have a fully customizable custom element. We have two other things to look at: how do add attributes to the custom elements and how to style them, both internally and from the host page.

Attributes and properties in custom elements

So far, all the code we've written assumes no attributes and no way to change the styles inside the document. We'll first tackle attributes, how to add them and how to reflectt the changes back and forth between the custom element and the page.

Let's add two attributes to the `demo-element`.

We start by listing the attributes that we want to observe using a static getter method, `getObservedAttributes`. This will enable to change the attribute from Javascript. Static methods are private to the class and unavailable on class instances.

Next, we define the attributes that we want to use by adding setters and getters for the attributes that we want.

To track changes we use a lifecycle callback method, `attributeChangedCallback`. In this case we just log the change to the console.

The `DemoElement` now looks like this:

```
class DemoElement extends HTMLElement {
  static get observedAttributes() {
    return ['height', 'width'];
  }

  get height() {
    return this.getAttribute('height');
  }

  set height(newVal'width'   this.setAttribute('height', newValue);
  }

  get width() {
    return this.getAttribute('width');
  }

  set width(newValue) {
    this.setAttribute('width', newValue);
  }

  constructor() {
    super();
    let templ'height'cument.getElementById('my-content');
    let templateContent = template.content;

    const shadowRoot = this.attachShadow({mode: 'open'})
    .appendChild(templateContent.cloneNode(true));
  }

  attributeChangedCallback(name, oldValue, newValue) {
    console.log('Custom demo-element attributes changed.');
    console.log('my-content'`${name} changed from ${oldValue} to ${newValu
  }
}
customElements.define('demo-element', DemoElement);
```

Styling custom elements from the

inside

The easiest way to style a custom element is to include the styles in a style element and put it inside the template. This example will leverage the single file component that we'll discuss later to show how we can style components from the inside.

In this stripped down example we've included the style for the component on the template. Since the template is inert until it's instantiated, we don't instantiate until we built the element instance and the styles are encapsulated to the instance the styles will not bleed

```
 const template = document.createElement('template');
template.innerHTML = `
<style>
  :host {
    display: flex;
  }
  .card {
    height: 300px;
    width: 400px;
    border: 2px solid red;
    margin-top: 2em;
    margin-left: 2em;
    margin-bottom: 2em;
  }
</style>
<div class="card">
  <h1><slot name="my-title">Default Title</slot></h1>
  <p><slot name="my-content">Default Content</slot></p>
</div>
`;

class DemoElement extends HTMLElement {

  constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.appendChild(template.content.cloneNode(true));
```

```
  }

}
window.customElements.define('demo-element', DemoElement);
```

Styling custom elements from oustide the custom element

Styling the component from the inside means that we can't easily match the styles with the host pages. Chris Coyier's Styling a Web Component show different ways to style your web components.

I've picked CSS variables as a way to style components from a master style sheet. I do it this way to reuse the same variables across the app and its components, ennsuring some level of uniformity on the user interface.

We define the custom variables to set up the attributes of our components.

```
:root {
  --border-color: blue;
  --background-color: lightgray;
  --title-size: 1.5rem;
}
```

The style inside the custom element uses the variables defined in the CSS stylesheet, otherwise it's the same as the internal version of our component.

```
 const template = document.createElement('template');
template.innerHTML = `
<style>
  :host {
    display: flex;
  }
  .card {
    background-color: var(--background-color);
```

```
    height: 300px;
    width: 400px;
    border: 2px solid var(--border-color);
    margin-top: 2em;
    margin-left: 2em;
    margin-bottom: 2em;
  }

  h1 {
    text-align: center;
    font-size: var(--title-size);
  }

  p {
    padding: 1em;
  }
</style>
<div class="card">
  <h1><slot name="my-title">Default Title</slot></h1>
  <p><slot name="my-content">Default Content</slot></p>
</div>
`;

class DemoElement extends HTMLElement {
  static get observedAttributes() {
    return ['height', 'width'];
  }

  get height() {
    return this.getAttribute('height');
  }

  set height(newValue) {
    this.setAttribute('height', newValue);
  }

  get width() {
    return this.getAttribute('width');
```

```
  }

  set width(newValue) {
    this.setAttribute('width', newValue);
  }

  constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.appendChild(template.content.cloneNode(true));
  }

  attributeChangedCallback(name, oldValue, newValue) {
    console.log('Custom demo-element attributes changed.');
    console.log('height'`${name} changed from ${oldValue} to ${newValue}`
  }
}
window.customElements.define('demo-element', DemoElement);
```

This means that we, as the creators of the custom element (or elements), have to be careful in what aspects of the component we surface for users to style.

When working on these examples I realized that the default color for the component (black), would not work with a dark colored background, so I had to add a text color variable and ensure that the header and text used this new variable to set the color of the text in a way that works well with whatever background color we want to use

Bonus points: putting the template in the same file as the component definition

Something that took me forever to figure out was how to incorporate the template into the same file as the custom element defintion.

This makes the element portable as we don't have to paste the template in every page where we'll use it. It also helps in keeping the styles for the component

in one place, particularly since the styles will not be needed anywhere else.

The code looks like this:

```
 const template = document.createElement('template');
template.innerHTML = `
<style>
  .card {
    height: 300px;
    width: 400px;
    border: 2px solid red;
    margin-top: 2em;
    margin-left: 2em;
    margin-bottom: 2em;
  }

  h1 {
    text-align:center;
  }

  p {
    padding: 1em;
  }
</style>
<div class="card">
  <h1><slot name="my-title">Default Title</slot></h1>
  <p><slot name="my-content">Default Content</slot></p>
</div>
`;

class DemoElement extends HTMLElement {

  constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.appendChild(template.content.cloneNode(true));
  }
```

```
}
window.customElements.define('demo-element', DemoElement);
```

Bonus Points: Using custom element attributes to style the element

When I created the `demo-element` element I added width and height attributes with the idea that I could use them to control the width and height of the custom element without having to modify the CSS of the element.

The way that I found out to do this is a little cumbersoome but it works as intended. Rather than copying all the code from the final version of `demo-element`, I'll just highlight the parts that changed.

In the constructor we add an empty style element. We add it here but will use it elsewhere in the code.

```
 constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.appendChild(template.content.cloneNode(true));
    const newStyle = document.createElement('style');
    this.shadowRoot.appendChild(newStyle);
  }
```

`attributeChangedCallback` checks if the width and height attributes exists at all and, if they exist, if they are empty. If either of those conditions are true then we set a default value. If not it assumes that the values are correct and leaves them as is.

It then calls `updateStyle` passing this as the parameter. This is what will change the appearance of the page

```
attributeChangedCallback(name, oldValue, newValue) {
  if (!this.hasAttribute('width')
```

```
      || (this.getAttribute('width') == '')) {
    this.setAttribute('width', 400);
  }

  if (!this.hasAttribute('height')
      || (this.getAttribute('height') == '')) {
    this.setAttribute('height', 300);
  }

  updateStyle(this);
};
```

The `updateStyle` function is where the magic happens. It creates a new style element and attaches it to the element's shadow root.

Next it puts a copy of the element's CSS using a string template literal with the values for width and height being interpolated from the corresponding attribute values.

```
function updateStyle(elem) {
  // Create new style element
  const myStyle = elem.shadowRoot.querySelector('style');
  myStyle.textContent = `
  :host {
    display:'style'  }
  .card {
    backgroun`
  :host {
    display: flex;
  }
  .card {
    background-color: var(--background-color);
    width: ${elem.getAttribute('w'width'}px;
    height: ${elem.getAttribute('h'height'}px;
    border: 2px solid var(--border-color);
    margin-top: 2em;
    margin-left: 2em;
```

```
    margin-bottom: 2em;
  }

  h1 {
    text-align: center;
    font-size: var(--title-size);
    color: var(--font-color);
  }

  p {
    padding: 1em;
    color: var(--font-color);
  }
`
```

SO now we have a custom element that can be styled from a master stylesheet for the application and that will respond to changes in the element attributes, while protecting developers from mistakes that are easy to overlook.

Future Looking: ::part and ::theme

There is an interesting specification in the pipeline regarding the styling of custom elements. CSS Shadow Parts defines ways to pierce the shadow boundary (the separation between the custom element's shadow DOM and the hosting page)

I will leave the explanation to the experts. Monica Dinculescu has written an explainer about ::part and ::theme that does a much better job than I can.

Links and Resources

- Attributes and Properties in Custom Elements
- Styling Your Custom Elements
- Composing Custom Elements With Slots And Named Slots
- CSS Shadow Parts
- ::part and ::theme, an ::explainer — Monica Dinculescu