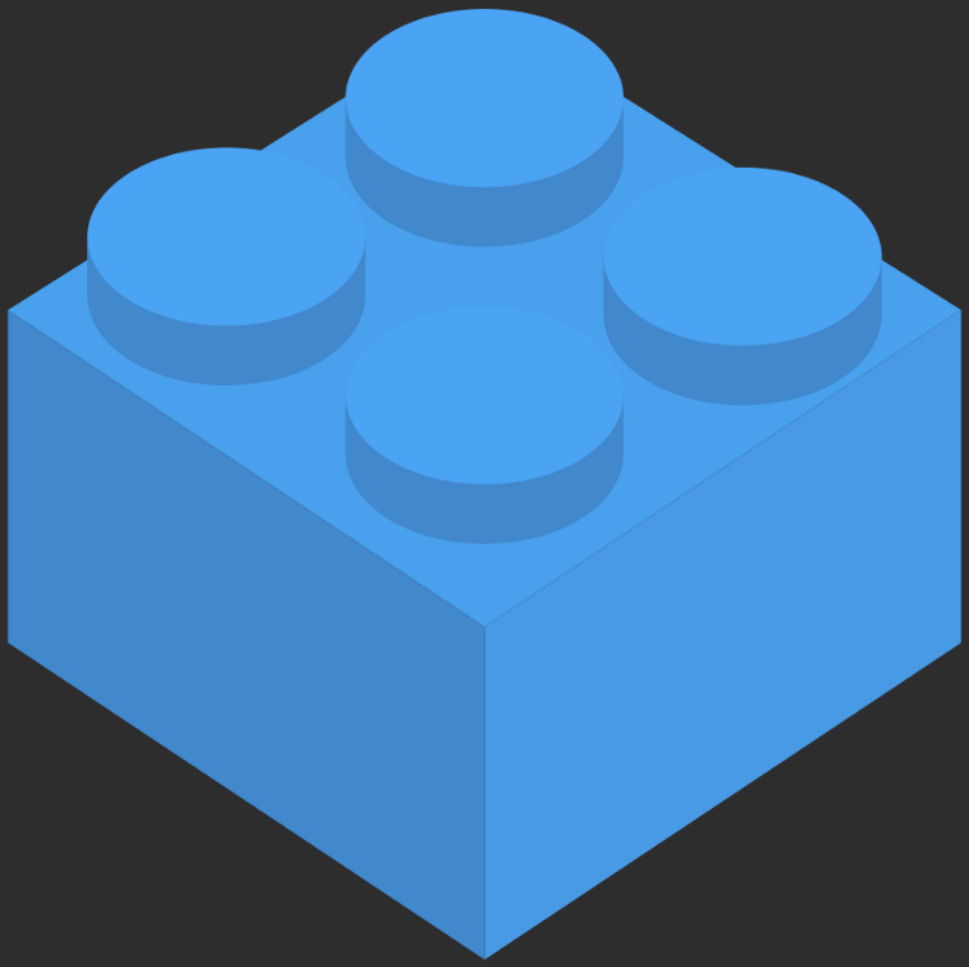
Sold to
peter.stanik@gmail.com

# Web Component Essentials



An introduction to creating reusable user interfaces with Web Components.

Cory Rylan

# Web Component Essentials

## An introduction to creating reusable user interfaces with Web Components. Full working code examples included!

Cory Rylan

This book is for sale at http://leanpub.com/web-component-essentials

This version was published on 2021-10-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2021 Cory Rylan

# Contents

**Introduction** . . . . . **1**
About the Author . . . . . 1
What is this Book About? . . . . . 1
Technical Prerequisites . . . . . 1
Source Code Examples . . . . . 2

**Chapter 1 - The Component** . . . . . **3**
History of the Component . . . . . 3
What is a Web Component? . . . . . 4
Custom Elements . . . . . 4

**Chapter 2 - Templates** . . . . . **7**
Attaching a Template to the Shadow DOM . . . . . 8
Content Slot API . . . . . 10
Named Slots . . . . . 12
Use Case - Dropdown Component . . . . . 14

**Chapter 3 - Component Communication** . . . . . **17**
Component Properties . . . . . 17
Component Events . . . . . 18
Use Case - Dropdown Component . . . . . 21
Component Custom Attributes . . . . . 25

**Chapter 4 - Component Lifecycle** . . . . . **29**
JavaScript Class Constructor and Connected Callback . . . . . 29
Disconnected Callback . . . . . 29
Attribute Changed Callback . . . . . 31
Adopted Callback . . . . . 31

**Chapter 5 - Styling with CSS and the Shadow DOM** . . . . . **32**
Global Styles . . . . . 32
CSS Encapsulation . . . . . 33
CSS Custom Properties . . . . . 36
Dynamic CSS Custom Properties . . . . . 38
Using :host and CSS Custom Properties . . . . . 41

CSS Parts API . . . . . 44
Styling Slotted Elements . . . . . 45

**Chapter 6 - Theming Web Components . . . . . 48**
Root Scope and Host Scope . . . . . 48
Implementing Dark Themes . . . . . 51

**Chapter 7 - Component Hierarchy and Architecture . . . . . 53**
Component Data Flow . . . . . 54
Character List . . . . . 56
Character Detail . . . . . 59

**Chapter 8 - Production Ready(ish) Web Components . . . . . 62**
Publishing with NPM . . . . . 62
Browser Support . . . . . 65
Polyfills . . . . . 65
Installing Webpack and Babel . . . . . 68
Webpack . . . . . 69

**Chapter 9 - Using Web Components in Angular and VueJS . . . . . 71**
Angular . . . . . 71
VueJS . . . . . 74

**Chapter 10 - Using Web Components in React . . . . . 78**
React Compatibility . . . . . 78
Create React App . . . . . 78
Properties and Events . . . . . 79
Prop Updates . . . . . 80

**Chapter 11 - Lit . . . . . 84**
Lit and Template Literal Strings . . . . . 84
Templates and Event Listeners . . . . . 85
Properties and Decorators . . . . . 86
Custom Events . . . . . 88
Binding to other Web Components with Lit . . . . . 89

**Chapter 12 - Stencil JS . . . . . 92**
The Stencil CLI . . . . . 93
Decorators . . . . . 94
JSX Templates . . . . . 95
JSX Component Bindings . . . . . 96
Building your Stencil Components . . . . . 97

**Chapter 13 - Building a Todo App with Lit . . . . . 98**
Implementation . . . . . 99

Todos Data Service . . . . . 100
Todos List . . . . . 102
Todo Item . . . . . 106

**Chapter 14 - Unit Testing Basics . . . . . 110**
Setting up the Test Runner . . . . . 110
Unit Test Structure . . . . . 111
First Unit Test . . . . . 112

**Chapter 15 - Conclusion . . . . . 114**

**Chapter 16 - Code Examples . . . . . 115**

# Introduction

## About the Author

Cory Rylan is a Front End Web Developer and Google Developer Expert for Angular and Web technologies. He specializes in building high-performance Web applications and enterprise Design Systems with Web Components. He currently works on the VMware open-source design system, Clarity[1]. He speaks at conferences, meet-ups, and workshops. When not speaking and teaching, he writes all about Angular and Web Components on his blog at coryrylan.com[2]. You can also follow him on Twitter @coryrylan[3];

## What is this Book About?

This book aims to provide an introduction and practical guide on how to create and use Web Components. By the end of this book, readers can feel confident in making a reusable UI component and share it across Web technologies and projects.

## Technical Prerequisites

The code examples accompanied with this book use the latest in Web technologies. Currently to get started with as little tooling as possible, these examples will work only work in Chrome and Firefox nightly. In later chapters, we will cover how to build Web Components and get cross-browser support.

This book covers beginner to intermediate web-specific technologies in HTML, CSS, and JavaScript. Some of the topics used in this book are listed below.

- HTML
  - HTML5 tags
  - HTML inputs and forms
  - Familiar with Web accessibility
- CSS
  - Familiar with responsive design concepts

[1] https://clarity.design/storybook/core
[2] https://coryrylan.com
[3] https://twitter.com/coryrylan

  - Familiar with some layout techniques such as Flexbox or CSS Grid
  - Media Queries
  - Familiar with CSS pre-processors such as Less or Sass
- JavaScript
  - Familiar with modern JavaScript syntax (ES modules, classes, template strings)
  - Familiar with any front end build system (rollup, webpack, etc)
  - Basic use of NPM (node package manager)
  - Beneficial but not required to be familiar with a JavaScript front end framework such as Angular, React, Ember, or Vue

# Source Code Examples

You can find a link to access and download the source code of all examples in the last chapter of this book.

# Chapter 1 - The Component

Building User Interfaces on the Web seems to be a problem with a million possible solutions. Every day there is a new JavaScript framework or library trying to help build your UIs. How many programming development years have been used rewriting a date-picker component to work in the latest framework? In web development, it is not uncommon for our UI widgets and components to be rewritten frequently in the latest popular JavaScript framework.

With the fast pace of the ever-evolving JavaScript and Web ecosystem, it can feel daunting to share code between websites and projects. We have to learn different frameworks, browser support, ever-changing web APIs, versioning, performance, accessibility, and more. All are all things we have to think about when building our UI components. It's becoming critical that we have stable and portable components that work everywhere. We have come a long way with the Web, but how did we get to Web Components, and what are they?

## History of the Component

The idea of the component for Web development has been around for a long time. Other names such as widget or plugin also align with the idea of components if we think back to JQuery and JQuery UI days. We could quickly add interactive widgets to our websites. I remember how excited my fellow developers and I were when we added that first little bit of JQuery to our Web pages, and suddenly, a new world of possibilities was open to us. Then we had libraries and frameworks like AngularJS, Backbone, and Knockout. Each one of these technologies tried to solve specific parts of UI development. The Web since then has moved on and evolved. We have come so far but have a ways to go.

Fast forward to today, we have frameworks and libraries such as React, Angular, Ember, Vue, and Polymer, which all embrace the idea of the component. Angular was one of the first significant frameworks to have something component-like. Angular's Directive API used custom element tags. React shook up Web development through its simple component model. Current JavaScript libraries and frameworks have their own definition of what a component is, but there is a lot of common overlap.

We still have this issue of components being built in framework silos. That slick React datepicker you found? No, sorry, you can't use that in Angular. That Vue datagrid, sorry that won't work in your Ember app. This repetition happens more often than we want to admit, but it is a problem we can solve.

# What is a Web Component?

What is a Web Component from a high-level design perspective? A component is an isolated piece of user interface with a single responsibility. When we think of components, things like date-pickers, dropdowns, and modals are the first that come to mind. Most modern frameworks such as Angular, React, and Vue have all embraced the idea of components, but all still go about it slightly differently. The Web Component API is here to help solve this fragmentation and give you true reusability across multiple web applications regardless of your chosen framework.

The Web Component API is a collection of different Web standard APIs that make a robust tool-set to create highly reusable UI components when combined. This book will cover each of these core APIs and then combine them to build reusable components for our applications. After an API introduction, we will learn about the available Web Component authoring tools that make it easy to author and publish our Web Components. Let's get started!

# Custom Elements

First, we need to start with the basics, HTML. No matter how complex or straightforward, from static blog to single page application, HTML is our starting point. We use many built-in components already when writing HTML. Some of these built-in components have been around for quite a while. HTML elements like text inputs, dropdowns, and even native date-pickers all are encapsulated single-purpose components. While the Web has provided some of these primitive components, most real-world applications need some customization or more robust features that these primitives lack.

The starting foundation of Web Components is the Custom Elements API. The Custom Elements API allows us to define our own custom HTML tags and attach the behavior to them. Let's define our first custom element. In a single JavaScript file, we are going to add the following:

```
1  class XComponent extends HTMLElement {
2    constructor() {
3      super();
4    }
5
6    connectedCallback() {
7      this.innerHTML = 'Hello World from a Web Component!';
8    }
9  }
10
11 customElements.define('x-component', XComponent);
```

As you can see in our JavaScript file, we use some of the newer ES2015 JavaScript features, such as Classes. To create our custom element, we extend the `HTMLElement` base class. When extending

a Class in JavaScript, we must call `super()` in the constructor. The constructor is called whenever a new instance of our component is created. Every time we use the `x-component` tag in our HTML, we will have a new instance. The constructor is where we will initialize any component state or initialization logic.

Our component class `XComponent` has a single method called `connectedCallback()`. The `connectedCallback()` is a lifecycle hook invoked when our component is created and added to the DOM. The `connectedCallback()` method is where most heavy lifting initialization logic should occur, like data fetching or initial render logic.

In the `connectedCallback()` method we can set the `innerHTML` of our component. Because we are extending the `HTMLElement`, we get all the common properties we get with any standard HTML DOM element.

Lastly, we need to register our component to the DOM (Document Object Model).

```
customElements.define('x-component', XComponent);
```

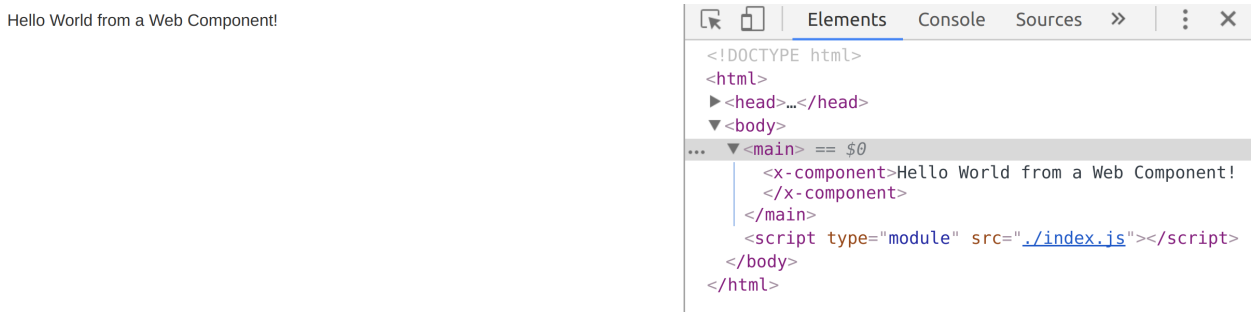
We pass two parameters to the custom elements API. The first is the tag name we will use in the DOM, and the second is a reference to our component class definition. In our example, we are going to call our component `x-component`. Notice we have a dash in our tag name. With the custom elements API, it is necessary to have at least one dash in the name. The dash convention protects us from defining a component that may come to a later version of HTML as native HTML elements have no dashes.

By registering the element, the browser can now instantiate our custom element whenever the browser sees an instance of the tag name in our HTML. Now that we have our component defined, let's use it in our HTML file.

```
<!DOCTYPE HTML>

<html>
<head>
  <title>Web Components Rock!</title>
</head>

<body>
  <x-component></x-component>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

Notice we have the component in our `index.html` template as well as a script tag. Take note that the script tag to our `index.js` also has `type="module"`. In this book, we will be using the latest browser

features, including ES2015 Modules. Later in the book, we will cover techniques to use these features with cross-browser support, including Web Components all the way back to IE11. If we now check the browser, we will see our component being rendered.




**Our First Web Component**

Congratulations, you have created your first Custom HTML Element! Our component doesn't do anything useful just yet. We need to learn a few more APIs before we can start building something a bit more interesting. Our next chapter will develop a simple dropdown component and learn some of the built-in template APIs.

# Chapter 2 - Templates

This chapter will cover the templating features provided by Web Component APIs and build a simple dropdown component using these APIs. The template API provides many low-level features needed to make UI components. The template API does not support templating language features we are accustomed to in many JavaScript frameworks. For example, use cases such as loops and conditionals are not built into the Template API.

Many of the Web Component APIs are specifically designed to be low level to allow more powerful opinionated tools to be built on top of them. We will see in later chapters how these APIs are helpful for high-level authoring tools for Web Components.

The Template API allows us to create neutral HTML templates in our components. In the past templating languages would use script tags with alternate types such as `<script type="templating-lib"></script>`. Script tags with invalid types ensured that the encapsulated HTML in the template would not render by the browser until the JavaScript had time to execute.

Now with the `<template>` tag, the browser ignores any code within those tags. Let's try it out. We are going to add the following to our HTML file.

```
<!DOCTYPE HTML>

<html>
<head>
  <title>Web Component Essentials</title>
</head>

<body>
  <template>
    <h1>Not going to render!</h1>
    <script>
      alert('not going to happen!');
    </script>
  </template>
  <script type="module" src="./index.js"></script>
</body>

</html>
```

If we open the HTML file in the browser, we can see the h1 heading doesn't render, and we don't get an alert from the inner script tag. We can clone the template, manipulate it with JavaScript, and then update the DOM.

If we are using ES2015/JavaScript modules for our components, why cover the template tag at all? Template tags are still helpful for performance in creating templates in our components. Let's take a look at the code below.

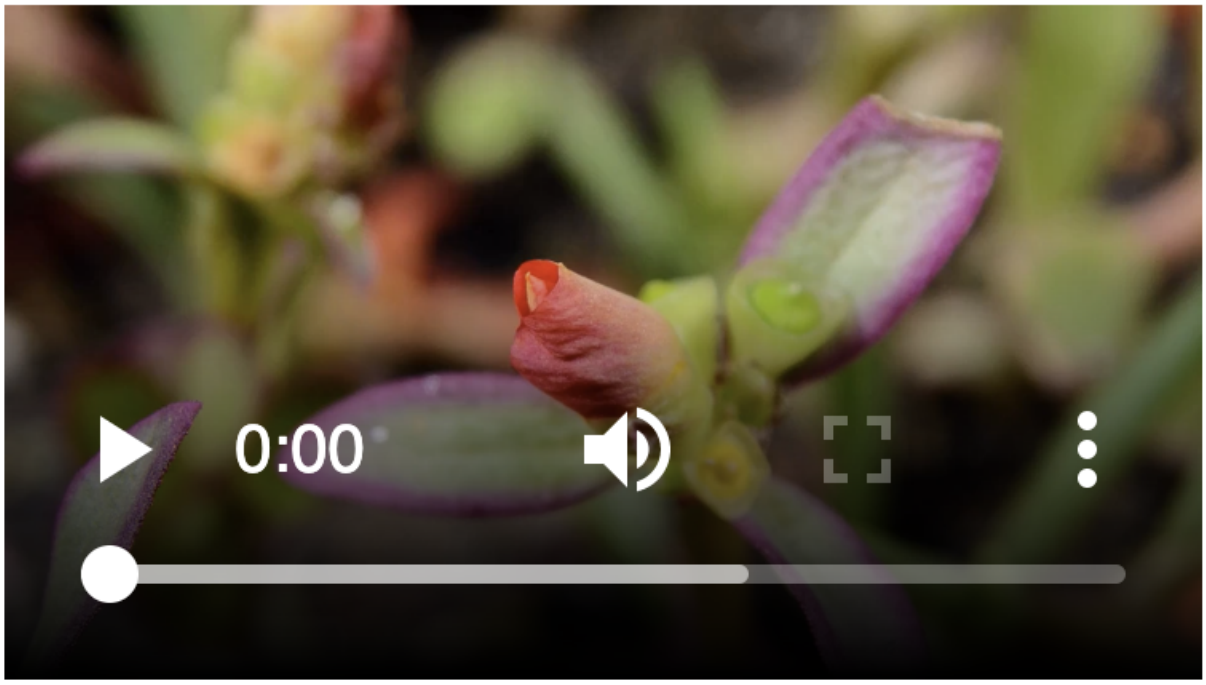
```
const template = document.createElement('template');
template.innerHTML = `<p>Hello World from Template</p>`;

class XComponent extends HTMLElement {
  constructor() {
    super();
    this.appendChild(template.content.cloneNode(true));
  }
}

customElements.define('x-component', XComponent);
```

With this technique, we can create a single reference for our template. Now every time we use `x-component` we reduce the parsing costs of creating the template. This technique gives us a performance gain when instantiating our components and cloning a new copy of the template to each component instance. In later chapters, we will see how web component authoring tools automatically handle this optimization for you.

## Attaching a Template to the Shadow DOM

Before we can take advantage of some of the other templating features built into the web, we must first learn the Shadow DOM API. The Shadow DOM API allows us to declare isolated and encapsulated pieces of DOM in our applications. These isolated templates using the Shadow DOM API can encapsulate their CSS and use some advanced declarative template APIs.

The idea of the Shadow DOM has been around for a long time in HTML and browsers. Take the `<video>` element, for example. The `<video>` element can take in a video source and play a video. Within that `<video>` element, there are many other elements such as the play/pause button, the time track, full-screen button, and more.




**Video Element (Mozilla Developer Docs)**

All of these elements are encapsulated into a single `<video>` element. This encapsulation makes it easy to reuse the video element as well as hide its complexity. This encapsulation is possible because of the Shadow DOM APIs.

Let's take a look at how we instantiate a component template with the Shadow DOM.

```
const template = document.createElement('template');
template.innerHTML = `<p>Hello World from Template</p>`;

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

customElements.define('x-component', XComponent);
```

Similar to the previous example, we create our template reference for our component. Instead of appending the template directly to our component, we instead create a Shadow DOM root for our component to use.

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

The first line tells the browser that this particular custom element should have a Shadow Root attached to it. By creating a shadow root, we will have an isolated template we can update in our component. The `mode` option allows us to configure if we want the shadow root accessible outside of the component. Unfortunately, even by setting it to "closed" there are ways with JavaScript to bypass this as there is no `private` construct in the language yet. Because of this, it is common to leave the mode to "open". Once we create a shadow root, we can start appending and updating the template.

```
this.shadowRoot.appendChild(template.content.cloneNode(true));
```

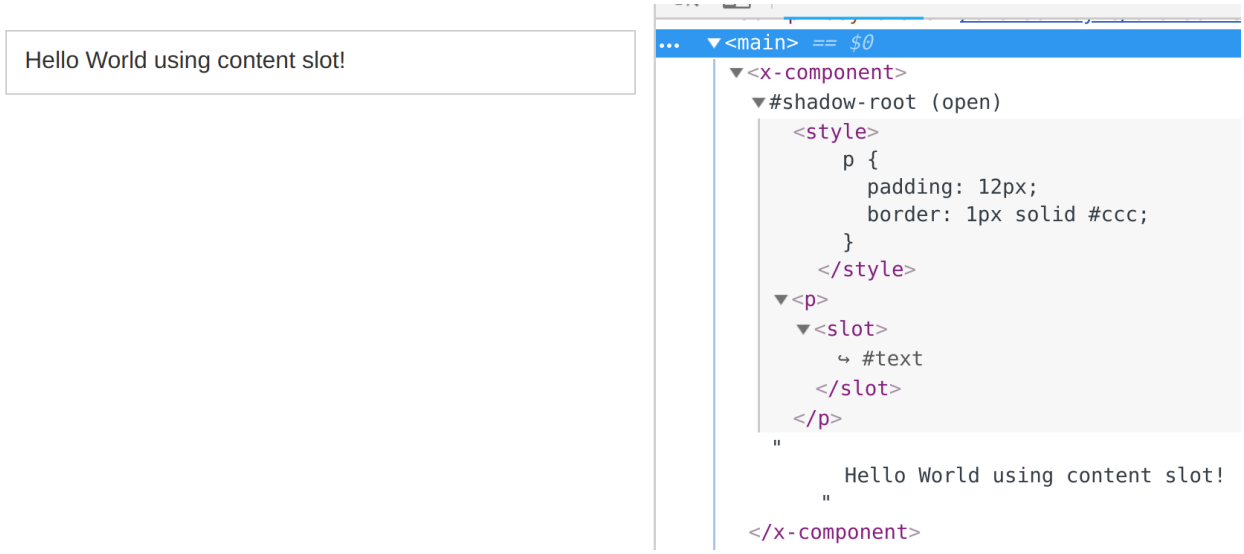
Now that we have a template created that is isolated via the Shadow DOM, we can take advantage of some of the other declarative APIs it provides.

## Content Slot API

The Content Slot API is a useful API that allows us to pass content into a component template declaratively. To use this API, you must be using the Shadow DOM API with your component templates. Let's look at this use of our `<x-component>`.

```
<x-component>
  Hello World!
</x-component>
```

If we look at the rendered output, we will see the message "Hello World!" inside the `<x-component>` tags.




**Rendered Slot Component**

Let's take a look at the `<x-component>` code.

```
const template = document.createElement('template');
template.innerHTML = `
  <p>
    <slot></slot>
  </p>
`;

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

customElements.define('x-component', XComponent);
```

In the template, we have a paragraph element, and within the paragraph, a `<slot>` element. The slot element allows us to declare where the content passed into our component should render. If we look at the output of `<x-component>` we will see that "Hello World!" is rendered between the component tags and our paragraph element. The Slot API is only available if your template is initialized with a shadow DOM instance, as we see in our `constructor()`.

# Named Slots

The template slot API also supports multiple Slot insertion points in components. Multiple slots are helpful when we want a structured component with lots of dynamic content. For example, we have a detail card component that displays an article or some blog post content.

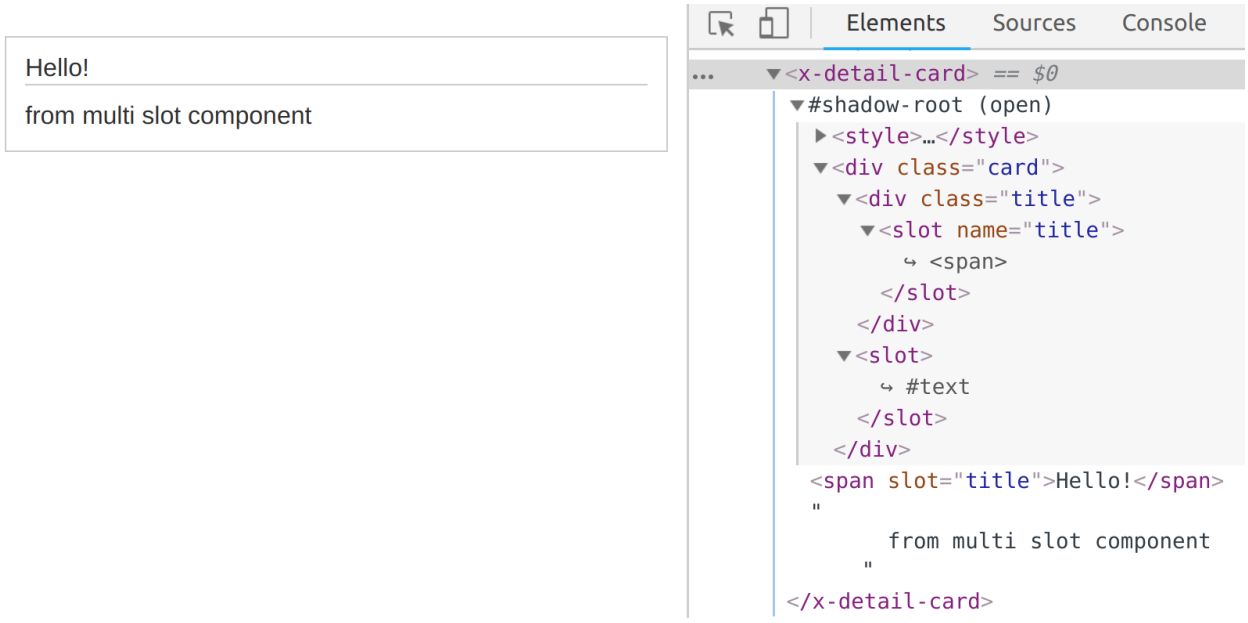
In this detail card, we want to display a title and snippet of text. With named slots, we can define multiple points where the dynamic content can be inserted. One can be inserted for the heading of the card and another for the content. Let's go ahead and take a look at an updated template to get a better understanding.

```
<style>
  .card {
    padding: 12px;
    border: 1px solid #ccc;
  }

  .title {
    border-bottom: 1px solid #ccc;
    margin-bottom: 12px;
  }
</style>
<div class="card">
  <div class="title">
    <slot name="title"></slot>
  </div>
  <slot></slot>
</div>
```

In the template of our new component, `x-detail-card`, we have some styles associated with our template. Notice we have two `<slot>` elements. The first has a name attribute with the value of `title`. Slots allow you to give specific names, so you can define where each piece of content should be rendered when you use the component. If you don't define a name, then the content will be rendered in the default unnamed `<slot>`. Let's take a look at what it looks like to use our `x-detail-card`.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
</head>

<body>
  <x-detail-card>
    <span slot="title">Hello!</span>
    from multi slot component
  </x-detail-card>

  <script type="module" src="./index.js"></script>
</body>

</html>
```

When we use the `x-detail-card`, our span has a `slot` attribute that defines which slot it should render in. For ours, it's the `title` slot. This inserts the span into the `title` slot, which means we get any styles associated with it. Our content `from multi-slot component` is rendered in the default slot as we did not define a named slot. Here is the rendered output of using the `x-detail-card`:

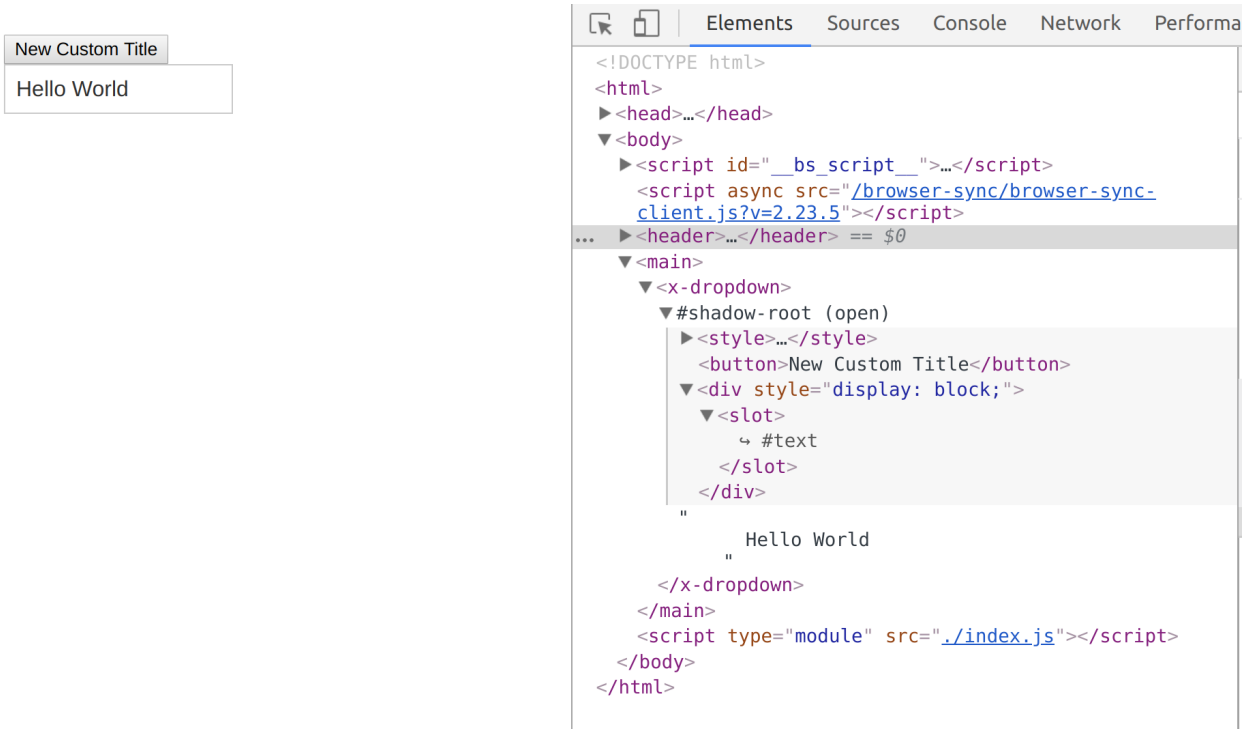



**Rendered Multi Named Slot Component**

We will see in our next section how the slot API has many practical uses.

# Use Case - Dropdown Component

We will take what we learned about the custom elements API and template tag API to build a simple dropdown style component. First, let's take a look at the rendered output of the dropdown component.




**Dropdown component**

To create this component, we will need to define our custom element class.

```
const template = document.createElement('template');
template.innerHTML = `
  <button>Dropdown</button>
  <div>
    <slot></slot>
  </div>
`;

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

```
}

customElements.define('x-dropdown', XDropdown);
```

Our dropdown component template has a single button to toggle our content. We also use the Content Slot API we saw earlier to take in content easily for our component template to render.

Next, we need to add some DOM references to our component to access the button and content.

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

  connectedCallback() {
    // Root also gives us access to the inner template of our component
    this.button = this.shadowRoot.querySelector('button');
    this.button.addEventListener('click', () => this.toggle());

    // Set the initial dropdown content to be hidden
    this.content = this.shadowRoot.querySelector('div');
    this.content.style.display = 'none';
  }

  toggle() {
    const show = this.content.style.display === 'block'
    this.content.style.display = show ? 'none' : 'block';
  }
}
```

In our constructor, we can query the template with our root element property. We create references to the button and content. With the content DOM references set, we can set the style property to display none to hide the slot content by default. We next create a click event listener on our button. This event listener will call the `toggle()` method whenever the button is clicked.

On an important note, we are having to set up events and set properties of our template manually. In most modern JavaScript templating libraries/frameworks, it abstracts this boilerplate code away. In a later chapter, we will see how we can add custom templating systems to make it easier to handle events and set custom properties. Now in our `index.html` we can use our new dropdown component.

```
<!DOCTYPE HTML>

<html>
<head>
  <title>Web Component Essentials</title>
</head>
<body>
  <x-dropdown>
    Hello World!
  </x-dropdown>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

Congratulations, you now have a functioning dropdown component! However, what if we want to change the text value of the button? What if we need to know when the dropdown has been opened or closed? How do we accomplish this? Our next chapter will dive into component communication with properties and events to extend our new dropdown component's functionality!

# Chapter 3 - Component Communication

In our previous chapter, we successfully built our dropdown component. Now we need the ability to customize some aspects of it. Ideally, we want to change the button text to whatever makes sense for our UI. How do we accomplish this? We could use content slots to project the content into the button, but we will show another way to pass data around between components for this example.

With Web Components, slots are ideal for content, but we often need to pass data or configuration into a component. With Web Components, we pass data into the component by setting properties or attributes. In this chapter, we will cover both techniques and the best practices when passing data to components.

## Component Properties

First, we will cover how to set the inner text of our dropdown button using custom properties. With properties, we can pass any JavaScript data type to our components; this includes things like Objects and Arrays. Let's go ahead and dive into our component and see how we define these properties.

```
const template = document.createElement('template');
template.innerHTML = `
  <p></p>
`;

class XComponent extends HTMLElement {
  // ES2015 classes support Getters and Setters
  set name(value) {
    this._name = value;
    this.nameElement.innerText = this._name;
  }

  get name() {
    return this._name;
  }

  constructor() {
    super();
```

```
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.appendChild(template.content.cloneNode(true));
    this.nameElement = this.shadowRoot.querySelector('p');
  }
}

customElements.define('x-component', XComponent);
```

ES2015 Classes support Getters and Setters. With Getters and Setters, we can define properties on our component that, when set, we can execute some logic on our component. In this example, we have a pubic Getter and Setter for the `name` property. Our getter returns the value of `_name`, our private property, to hold the value that will be rendered. JavaScript, unfortunately, does not support private properties yet. Because of the lack of a formal private keyword, we follow the standard JavaScript convention of prefixing private properties with an underscore.

Our setter for `name` sets the private `_name` and sets the inner text of the `nameElement` reference we created in our `constructor()`. Anyone using our component can set our component's name property, and the template will reflect those changes.

```
import './component.js';

const component = document.querySelector('x-component');
component.name = 'Hello World from Web Component property!';
```

Setting custom properties on components is the primary and recommended way to pass data into components. By setting properties, we can pass any data type to the component. Many frameworks follow this pattern and provide an easy way to bind data to components in declaratively in HTML templates. Some Web Component libraries also simplify declaring properties on your component by automatically creating your Getters and Setters or some advanced form of data change detection. In later chapters, we will take a look at these useful tools in more detail.

Now we know how to set or pass data to a component; what about when we want a component to communicate or notify us of a change or user interaction?

## Component Events

Component Events provide a mechanism to notify us of a change of some kind. For example, when we use the HTML `<button>` element, we can listen to the `click` event to notify when the user clicks our button. We can set up a similar custom event on our own custom component. Let's look at an example below.

```
const template = document.createElement('template');
template.innerHTML = `
  <button>click me!</button>
`;

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

    this.shadowRoot
      .querySelector('button')
      .addEventListener('click', () => console.log('button clicked'));
  }
}

customElements.define('x-component', XComponent);
```

In this example, we have a simple component with a single button in the template. Our `constructor()` we select the button from our template and add an event listener for the click event.

```
this.root
  .querySelector('button')
  .addEventListener('click', () => console.log('button clicked'));
```

Now we can log when the button is clicked, but this is only part of the solution. We need to create a custom event for our `XComponent`, so anyone using our `XComponent` can notify when the user clicks the button.

```
const template = document.createElement('template');
template.innerHTML = `
  <button>click me!</button>
`;

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

    this.shadowRoot
```

```
      .querySelector('button')
      .addEventListener('click', () => this.handleClick());
  }

  handleClick() {
    this.dispatchEvent(
      new CustomEvent('customClick', { detail: Math.random() })
    );
  }
}

customElements.define('x-component', XComponent);
```

In our example above, we can see we added a new method, `handleClick`. This method will get called whenever the click event of the button occurs. In `handleClick` we trigger a new custom event.

```
handleClick() {
  this.dispatchEvent(new CustomEvent('customClick', { detail: Math.random() }));
}
```

When we dispatch a new custom event, we provide two parameters. First is the name we want for our custom event. In this example, we have `customClick`. Now when anyone uses our component, they can listen to the `customClick` event.

The second parameter is a configuration object for our event. In our configuration, we have a single property called `detail`. The `detail` property is where we can attach any value to our event to pass to any listener of the event. In our example, we are just going to pass back a random number. Let's take a look at some code that listens to our new custom event.

```
import './component.js';

const component = document.querySelector('x-component');
component.addEventListener('customClick', e => console.log(e));
```

You can see we can select the DOM instance of our component and create an event listener for our `customClick` event just like any other DOM event. If we look at our event's value, we get a lot more back than only the event value.

```
▼ CustomEvent {isTrusted: false, detail: 0.4379201446977461, type: "customClick",
  x-component, …} i
    bubbles: false
    cancelBubble: false
    cancelable: false
    composed: false
    currentTarget: null
    defaultPrevented: false
    detail: 0.4379201446977461
    eventPhase: 0
    isTrusted: false
  ▶ path: (6) [x-component, main, body, html, document, Window]
    returnValue: true
  ▶ srcElement: x-component
  ▶ target: x-component
    timeStamp: 75839.69999999681
    type: "customClick"
  ▶ __proto__: CustomEvent
```

**Custom Event Output**

In the custom event, we get all the useful DOM details about our component and our `detail` property with the custom value we passed back.

In a later chapter, we will see some examples of how Web Component libraries can help easily declare and listen to component events. Now that we have learned the API for custom properties and events let's look at how we can improve our dropdown component using these APIs.

# Use Case - Dropdown Component

Going back to our dropdown component, we want to add some additional functionality. First, we want to be able to change the text of the dropdown button. Second, we want to create a custom event listener to know when the user has opened or closed our dropdown. Let's start with the custom button text.

## Dropdown Component - Custom Properties

We want to set the button text to a custom string value. To accomplish this, we need to pass data down to the dropdown. By using custom properties, we learned earlier, we can achieve this.

```
const template = document.createElement('template');
template.innerHTML = `
  <button></button>
  <div>
    <slot></slot>
  </div>
`;

class XDropdown extends HTMLElement {
  get title() {
    return this._title;
  }

  set title(value) {
    this._title = value;
    this.buttonElement.innerText = this._title;
  }

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

    this.buttonElement = this.shadowRoot.querySelector('button');
    this.buttonElement.innerText = this.title;
    this.buttonElement.addEventListener('click', () => this.toggle());

    this.contentElement = this.shadowRoot.querySelector('div');
    this.contentElement.style.display = 'none';
  }

  toggle() {
    this.show = !this.show;
    this.contentElement.style.display = this.show ? 'block' : 'none';
  }
}

customElements.define('x-dropdown', XDropdown);
```

On our dropdown, we create the `title` property getter and setter and a private `_title` property. Using the getter and setters, we can change the button `innerText` when the `title` property is set.

```
import './dropdown.js';

const dropdown = document.querySelector('x-dropdown');

dropdown.title = 'Custom Title';
// wait three seconds then update the property
setTimeout(() => (dropdown.title = 'New Custom Title'), 3000);
```

Like any DOM element, once we have defined a custom property, we can pass data. In this example, we set the dropdown button to `Custom Title`. The dropdown is also updated if this property is changed later. In this snippet, we create a `setTimeout` that will change the button text to `New Custom Title` three seconds later.

We have successfully passed data to our dropdown component, and the component updates its template. Next, we want to get notified when a user has clicked on our dropdown component.

## Dropdown Component - Custom Events

To get notified of when the user has clicked our dropdown component, we need to create a custom event similar to our example we learned earlier. We will need to have a click event to trigger the custom event to be dispatched. Let's take a look at our dropdown code.

```
const template = document.createElement('template');
template.innerHTML = `
  <button></button>
  <div>
    <slot></slot>
  </div>
`;

class XDropdown extends HTMLElement {
  get title() {
    return this._title;
  }

  set title(value) {
    this._title = value;
    this.buttonElement.innerText = this._title;
  }

  constructor() {
    super();
    this._title = 'dropdown';
```

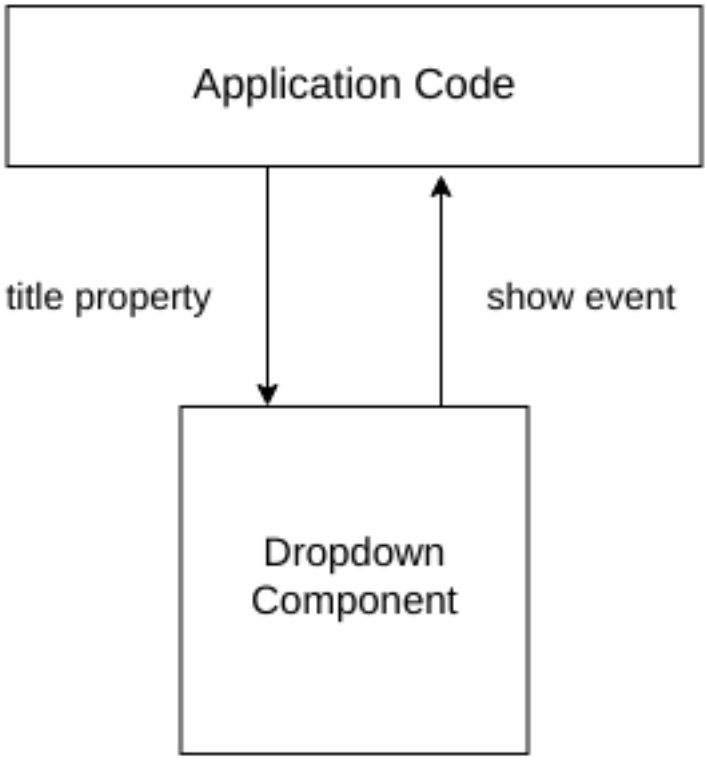
```
    this.show = false;

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

    this.buttonElement = this.shadowRoot.querySelector('button');
    this.buttonElement.innerText = this.title;
    this.buttonElement.addEventListener('click', () => this.toggle());

    this.contentElement = this.shadowRoot.querySelector('div');
    this.contentElement.style.display = 'none';
  }

  toggle() {
    this.show = !this.show;
    this.contentElement.style.display = this.show ? 'block' : 'none';

    // trigger our custom event 'show' for others to listen to
    this.dispatchEvent(new CustomEvent('show', { detail: this.show }));
  }
}

customElements.define('x-dropdown', XDropdown);
```

In our dropdown component, we added our custom event in the `toggle()` method. When the user clicks the dropdown button, our dropdown will now dispatch a custom event `show`, which will return a boolean detail value to tell us if the dropdown is visible or not.

```
import './dropdown.js';

const dropdown = document.querySelector('x-dropdown');
dropdown.addEventListener('show', e => console.log(e));
```

As you can see, we can now listen to our custom `show` event and get notified when the user has clicked the dropdown and know if the dropdown has opened or closed.

The use of custom properties and events is our primary communication mechanism for components. We can pass data down to components via properties and listen to data being passed back via events.




**Component Communication**

Before we move onto the next chapter, there is one more topic we need to cover when interacting with our components. So far, we have been communicating or passing data down to our components via JavaScript properties. What about using DOM attributes instead? In our next section, we will discuss the pros and cons of using DOM attributes for component communication.

# Component Custom Attributes

It is common when using Web Components to set HTML attributes to pass information to the component. Let's look at the following code example.

```
1 <x-component message="hello world"></x-component>
```

As you can see, we are setting a custom HTML attribute on our `x-component`. We can read the values of the attributes on our component. Attributes can be a convenient way to pass information to a component without the need for additional JavaScript. The downside to HTML Attributes is that the attribute is always treated as a string, so you cannot use other data types such as numbers

or objects without additional parsing of the string value. Let's take a look at what it takes to read attribute values on a custom element.

```
const template = document.createElement('template');
template.innerHTML = `
  <p></p>
`;

class XComponent extends HTMLElement {
  static get observedAttributes() {
    return ['name'];
  }

  set name(value) {
    this._name = value;
    this.nameElement.innerText = this._name;
  }

  get name() {
    return this._name;
  }

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

  attributeChangedCallback(attrName, oldValue, newValue) {
    if (attrName === 'name') {
      this.name = newValue;
    }
  }
}

customElements.define('x-component', XComponent);
```

The first part of the component is the `observedAttributes()` method.

```
static get observedAttributes() {
  return ['name'];
}
```

The static method `observedAttributes` expects a list of named attributes that the DOM should watch for changes. This is a required performance optimization, so you receive updates about which attributes have changed. The second part of the custom Attributes API is the `attributeChangedCallback()` method.

```
attributeChangedCallback(attrName, oldValue, newValue) {
  if (attrName === 'name') {
    this.name = newValue;
  }
}
```

This callback method is called whenever one of our listed attribute values has been updated. The method takes three parameters. The first is the name of the attribute that changed to determine if and how to update our component. The second and third attribute gives you the previous and new value of the attribute value to efficiently update your component.

Let's take what we learned with custom attributes and add support to our dropdown component to change the name via an Attribute.

```
const template = document.createElement('template');
template.innerHTML = `
  <button></button>
  <div>
    <slot></slot>
  </div>
`;

class XDropdown extends HTMLElement {
  static get observedAttributes() {
    return ['title'];
  }

  get title() {
    return this._title;
  }

  set title(value) {
    this._title = value;
    this.buttonElement.innerText = this._title;
```

```
21    }
22
23    constructor() {
24      super();
25      this._title = 'dropdown';
26      this.show = false;
27
28      this.attachShadow({ mode: 'open' });
29      this.shadowRoot.appendChild(template.content.cloneNode(true));
30
31      this.buttonElement = this.shadowRoot.querySelector('button');
32      this.buttonElement.innerText = this.title;
33      this.buttonElement.addEventListener('click', () => this.toggle());
34
35      this.contentElement = this.shadowRoot.querySelector('div');
36      this.contentElement.style.display = 'none';
37    }
38
39    attributeChangedCallback(attrName, oldValue, newValue) {
40      if (attrName === 'title' && oldValue !== newValue) {
41        this.title = newValue;
42      }
43    }
44
45    toggle() {
46      this.show = !this.show;
47      this.contentElement.style.display = this.show ? 'block' : 'none';
48      this.dispatchEvent(new CustomEvent('show', { detail: this.show }));
49    }
50  }
51
52  customElements.define('x-dropdown', XDropdown);
```

Component input Properties and output Events are our primary communication mechanism to communicate with our components. In later chapters, we will also cover how we can use these same mechanisms to have components communicate with each other. In our next chapter, we will cover component styles and themes.

# Chapter 4 - Component Lifecycle

This chapter will cover the Custom Element lifecycle. We have already covered a few lifecycle methods in previous chapters. Let's walk through them in orderâ€"first, the constructor.

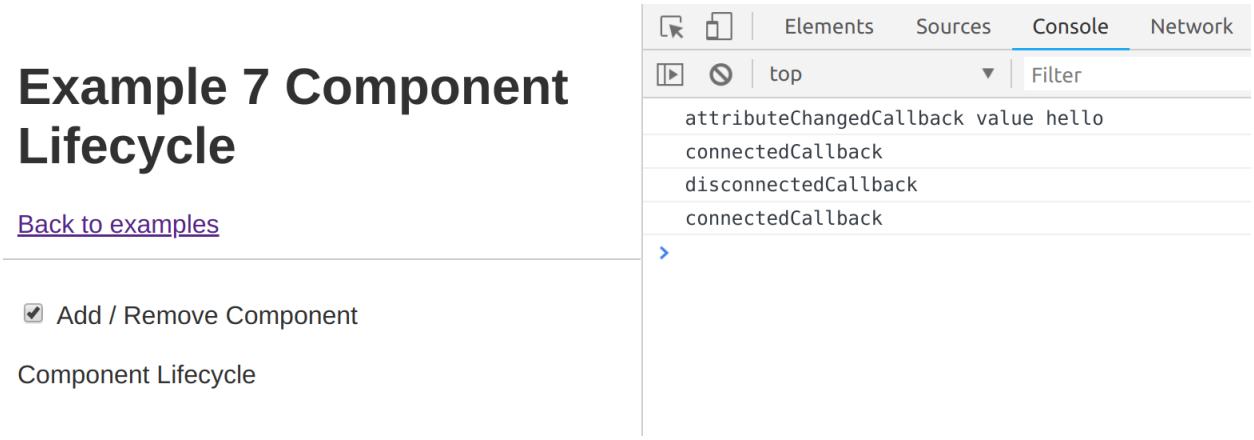
## JavaScript Class Constructor and Connected Callback

```
class XComponent extends HTMLElement {
  constructor() {
    super();
  }

  connectedCallback() {
    this.innerHTML = 'hello world';
  }
}

customElements.define('x-component', XComponent);
```

The constructor is executed whenever an instance of our component is created. However, if we need to instantiate DOM or initialization logic, we will want to run this code in the `connectedCallback()` lifecycle hook. The `connectedCallback()` is executed whenever our component is added to the DOM. Once added, we can start rendering and interacting with the DOM.

## Disconnected Callback

Our next lifecycle hook is the `disconnectedCallback()` method. The `disconnectedCallback()` method is called whenever our component is removed from the DOM. Let's take a look at the following example:




**Removing and adding a component**

In our template, we render a component then add or remove the component-based on a checkbox value.

```
<x-component></x-component>
```

```
class XComponent extends HTMLElement {
  disconnectedCallback() {
    console.log('disconnectedCallback');
  }
}

customElements.define('x-component', XComponent);
```

```
import './component.js';

const toggle = document.querySelector('#toggle');
const main = document.querySelector('main');

toggle.addEventListener('change', e => {
  if (e.target.checked) {
    main.appendChild(document.createElement('x-component'));
  } else {
    main.removeChild(document.querySelector('x-component'))
  }
});
```

Using `disconnectedCallback()` helps when we need to do any kind of cleanup work when our component is removed, for example, disconnecting from Web Sockets or event listeners.

## Attribute Changed Callback

Our next lifecycle hook we have covered with our dropdown component. The `attributeChangedCallback()` helps notify us whenever an attribute on our element has changed.

```
class XComponent extends HTMLElement {
  static get observedAttributes() {
    return ['value'];
  }

  attributeChangedCallback(attrName, oldVal, newVal) {
    console.log('attributeChangedCallback', attrName, newVal);
  }
}

customElements.define('x-component', XComponent);
```

To get notified, we must list the attributes for the element to watch. We then add the `attributeChangedCallback()`, which passes what attribute changed and the previous and next values.

```
<x-component value="hello"></x-component>
<!-- updated to the following -->
<x-component value="hello world"></x-component>
```

```
// logged values
{
  attrName: 'value',
  oldVal: 'hello',
  newVal: 'hello world'
}
```

## Adopted Callback

The last lifecycle hook, `adoptedCallback()` is likely the least commonly used. The `adoptedCallback()` is called whenever the element has been moved to a new document. Using `adoptedCallback()` most commonly happens when cloning from an element in an iframe and moving it to the parent document with `Document.adoptNode()`. You can read more on the MDN Documentation[4].

[4] https://developer.mozilla.org/en-US/docs/Web/API/Document/adoptNode

# Chapter 5 - Styling with CSS and the Shadow DOM

In this chapter, we are going to cover CSS Custom Properties. We will address some of the pain points of CSS and how the Shadow DOM API can help. By using CSS Custom Properties and Shadow DOM, we can build powerful theming capabilities into our components.

## Global Styles

One of the big pain points of CSS is that CSS, by default, is global. For example, let's look at this simple CSS rule below.

```
h1 {
  color: red;
}
```

This CSS rule makes all `h1` headings red. By default with CSS, this rule will apply to all `h1` elements in the document or view. Many times we only want to apply specific rules for specific components or views. To get around this in the past, developers have come up with clever naming conventions for CSS classes to avoid global styles from accidentally overriding component-specific styles. For example, if we wanted to apply some CSS to only `h1` elements in our article view, we might write something like this:

```
h1.article-heading {
  color: blue;
}
```

Now when the `h1` has the `.article-heading` class, it will be blue and not apply to all `h1` elements. This works but does not scale well as it's easy to have name collisions. What can we do to solve this?

With Web Components, we can use the Shadow DOM API to simplify how we write CSS for our Web Applications. We have already covered some of the Shadow DOM features like the Slot API for our dropdown component. The next feature we will cover is one of the significant benefits of using the Shadow DOM, CSS Encapsulation.

## CSS Encapsulation

CSS Encapsulation has been a long-awaited feature of the Web. CSS Encapsulation allows us to write component-specific CSS that will only apply to a particular DOM or template subset. To illustrate this, let's go ahead and jump right into some code.

```
<!DOCTYPE HTML>

<html>
<head>
  <title>Web Component Essentials</title>

  <style>
    p {
      color: blue;
    }
  </style>
</head>
<body>
  <p>
    I'm a blue paragraph from the global styles.
  </p>

  <x-component></x-component>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

So in our example, we have an `index.html` file. In this file, we have declared a single global style. This style rule sets all `p`, paragraph tags to be blue. We then also have a single component on the view, our `x-component`. Let's take a look at what it looks like rendered.

# Example 8 Shadow DOM

Back to examples

I'm a blue praragraph from the global styles.

I'm a red paragraph in a web component

**CSS Encapsulation**

We can see a blue paragraph in our rendered output, but we also see a paragraph rendered by our custom `x-component`. Notice this component rendered its paragraph red. Let's go ahead and jump into the component code and see what is going on here.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    p {
      color: red;
    }
  </style>
  <p>
    I'm a red paragraph in a web component
  </p>
`;

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

customElements.define('x-component', XComponent);
```

Our component is pretty simple, with no behavior, just a single template with a static paragraph tag and a style tag. Because we register the template as a shadow element, we automatically get CSS Encapsulation for free! Notice in the style tag we set the following rule:

```
p {
  color: red;
}
```

Typically that would cause all paragraph elements to be read in our entire application. In our component, this style is encapsulated to only apply to this component's template. Our global styles are left unchanged. Let's take a look at the rendered DOM output to see how the browser treats our component.

```
▼<main> == $0
    <p>
          I'm a blue praragraph from the global styles.
        </p>
  ▼<x-component>
    ▼#shadow-root (open)
        <style>
            p {
              color: red;
            }
          </style>
        <p>
            I'm a red paragraph in a web component
          </p>
    </x-component>
```

**CSS Encapsulation Shadow DOM Output**

In our DOM output, we can see that the browser created a Shadow Root for our component. You can kind of think of this as a subset of an isolated subtree of DOM. This mechanism is what provides our CSS Encapsulation behavior. In later chapters, when we dive into Component authoring tools, we will cover tooling that will allow you to quickly write CSS in stand-alone CSS files or even authoring with Sass or Less.

The CSS encapsulation mechanism works both ways; component styles don't leak out to the global scope, and global styles don't override component styles. CSS encapsulation solves so many of the headaches we can have with CSS. But there are a few scenarios still left for questioning. What if I do want to override the component styles? Maybe I want a custom theme? These are all questions we will answer in our following sections, CSS Custom Properties and Component Themes.

# CSS Custom Properties

In our previous section, we learned about CSS Encapsulation and the problems it solves for us. This section will cover some new CSS features that, when combined with Shadow DOM, give us powerful theming capabilities with our components.

CSS Custom Properties solve a few different issues for us. CSS Custom properties will allow us to define dynamic variables we can use to theme and style our applications and individual components. Let's take a look at a simple example showing how to use variables in our CSS.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
  <style>
    :root {
      --primary-color: blue;
      --heading-size: 18px;
    }

    h2 {
      color: var(--primary-color);
      font-size: var(--heading-size);
    }

    p {
      color: var(--primary-color);
    }
  </style>
</head>
<body>
  <h2>Heading styled with a CSS Custom Property</h2>
  <p>
    Paragraph styled with a CSS Custom Property
  </p>
</body>
</html>
```

Our HTML file has a single style tag containing an `h2` and a `p` selector. In our styles, we have our first glimpse of the CSS Custom Properties feature.

```
:root {
  --primary-color: blue;
  --heading-size: 18px;
}

h2 {
  color: var(--primary-color);
  font-size: var(--heading-size);
}

p {
  color: var(--primary-color);
}
```

The `:root` selector selects the highest root DOM element. In our use case, this is the global document. With the `:root` selector, we can define our first CSS Custom Properties. In this example, we have two custom properties, `--primary-color` and `--heading-size`. Custom properties can be any name but must be prefixed with two dashes. To use these variables in our CSS, we can look at the `h2` and `p` selectors.

```
h2 {
  color: var(--primary-color);
  font-size: var(--heading-size);
}
```

We use the `var()` keyword passing in the variable we want to assign to assign a value to a property. These variables are like any other language variable in that we can use them in multiple places. Note in our `p` tag selector, we can reuse the `--primary-color` variable.

```
p {
  color: var(--primary-color);
}
```

Custom Properties are a fantastic addition to CSS. We have built-in variables now. We once had to rely on Sass and Less to get this kind of behavior. CSS Custom Properties take it one step further than Sass or Less. Sass and Less take your variables and compiles them into static CSS. Example:

```
$primary-color: blue;
$size: 18px;

h2 {
  color: $primary-color;
  font-size: $size;
}

p {
  color: $primary-color;
}
```

This Sass code above compiles to:

```
h2 {
  color: blue;
  font-size: 18px;
}

p {
  color: blue;
}
```

Notice how the CSS generated by Sass is not truly dynamic. There is no way to change the variable as the Sass variable to converted static string values at runtime in the browser. CSS Custom Properties are genuinely dynamic and can be switched on the fly with no build step required.

## Dynamic CSS Custom Properties

In our previous example, we had a heading and paragraph with styles defined with CSS Custom Properties. One of CSS Custom Properties' significant benefits is we can dynamically change the variables at runtime with JavaScript.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
  <style>
    :root {
      --primary-color: blue;
      --heading-size: 18px;
    }

    h2 {
      color: var(--primary-color);
      font-size: var(--heading-size);
    }

    p {
      color: var(--primary-color);
    }
  </style>
</head>

<body>
  <h2>Heading styled with a CSS Custom Property</h2>
  <p>
    Paragraph styled with a CSS Custom Property
  </p>

  <button>Toggle Theme</button>

  <script type="module" src="./index.js"></script>
</body>

</html>
```

Our HTML is the same as before with a couple of additions. We now have a button and JavaScript file to toggle the styles whenever we click the button. Let's take a look at the two different toggled states.

CSS Custom Properties initial state

CSS Custom Properties second state

When the view first renders, the color of the text is blue. When we click the button, the text then turns green. Let's take a look at the JavaScript that makes this possible.

```
document
  .querySelector('button')
  .addEventListener('click', () => toggleStyles());

function toggleStyles() {
  const styles = getComputedStyle(document.documentElement);
  // You can get CSS Custom Properties
  const colorValue = styles.getPropertyValue('--primary-color');

  if (colorValue === 'green') {
    // You can also set CSS Custom Properties
    document.documentElement.style.setProperty('--primary-color', 'blue');
```

```
  } else {
    document.documentElement.style.setProperty('--primary-color', 'green');
  }
}
```

As you can see in our example, we can read and write our custom properties dynamically, allowing us to make style updates across multiple CSS selectors. Custom Properties makes theming much more straightforward than previous techniques.

## Using :host and CSS Custom Properties

Our previous example showed the basics of CSS Custom Properties and how they function. In this example, we will see that we can scope our CSS Custom Properties to specific components using the CSS `:host` selector.

In the following example, we have a paragraph that is styled globally. The example also contains a paragraph in a web component with some default styles. The global styles make the paragraphs blue while the Web Component styles its paragraphs red. Here is the rendered output.

I am a paragraph.

I'm a web component that can have custom styles thanks to CSS Properties!

**Theming Custom Web Components**

We want to customize our web component and change the color to be green instead of red. Let's start with the component code first.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    :host {
      --color: red;
      --font-size: 16px;
    }

    p {
      color: var(--color);
      font-size: var(--font-size);
```

```
    }
  </style>
  <p>
    I'm a web component that can have custom styles thanks to CSS Properties!
  </p>
`;

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

customElements.define('x-component', XComponent);
```

Our component has a small static template with a single paragraph tag. In our CSS, we have a new selector, the `:host` selector. The host selector refers to the host element. For our component, that would be the `x-component` tag. Using the `:host` selector, we can set Custom CSS Properties on our component to use.

```
:host {
  --color: red;
  --font-size: 16px;
}

p {
  color: var(--color);
  font-size: var(--font-size);
}
```

On our `:host` selector, we create two properties, `--color` and `--size`. We will use these to style our component. In our styles, we see that the `p` tag is styled using our custom properties. When creating Custom CSS Properties on a Web Component, we can easily set these properties with additional CSS, making theming and customizing components easier. Let's take a look a the `index.html` file.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Web Component Essentials</title>
  <style>
    p {
      color: blue;
    }

    x-component {
      --color: green;
      --size: 24px;
    }
  </style>
</head>
<body>
  <p>I am a paragraph.</p>
  <x-component></x-component>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

In our global styles, we can set our `x-component` custom properties with little effort.

```
x-component {
  --color: green;
  --size: 24px;
}
```

I am a paragraph.

I'm a web component that can have custom styles thanks to CSS Properties!

#### CSS Custom Properties for Web Components

Custom CSS Properties allow consumers of our components to easily style and theme our components while still keeping the benefits of CSS encapsulation. While CSS Custom Properties can expose specific properties for customization, sometimes you may need to provide more flexibility by enabling styling of entire DOM elements within your component.

# CSS Parts API

The CSS Parts API provides a way to expose specific DOM elements in your Web Component template for public styling. By default, any element in your component is shielded from being styled globally. We can customize with CSS Custom Properties. However, CSS Custom Properties can be difficult to maintain as they only represent a single CSS property value.

CSS Parts expose the entire element allowing the consumer to style the element any way needed. Let's take a look at an example.

---

**CSS Parts API**

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    button {
      background: green;
      color: #fff;
      border: 0;
      padding: 12px;
    }
  </style>
  <button part="button-one">one</button>
  <button part="button-two">two</button>
  <button>three</button>
```

```
`;

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

customElements.define('x-component', XComponent);
```

Here we have a basic component with three buttons and some internal styles. On two of the buttons, we use the `part` attribute with a unique name. The `part` attribute enables the button to be accessed by consumers of your component directly for CSS styling. We use the `:part` selector and pass in the name of the part element we want to access to access the element.

```
x-component::part(button-one) {
  background: red;
}

x-component::part(button-two) {
  background: blue;
}
```

In our CSS, we can get access to button one and two directly and apply any styles that we want, just like we would outside the Shadow DOM. However, the third button is still protected within the Shadow DOM since we did not expose it via the `part` attribute.

CSS Parts can make your code more maintainable and more flexible styles. These advantages have tradeoffs as well. If you expose with CSS Parts, your styles may break in unexpected ways for the component. If you ship your components as part of a design system, this may be too much flexibility in your components' visual appearance.

## Styling Slotted Elements

We have covered how to style our Web Components with Shadow DOM. How do we style elements that are used in Shadow DOM slots? The `::slotted()` CSS selector styling slotted elements is relatively easy to accomplish.

```
<x-component>
  <button>styled with ::slotted</button>
</x-component>
```

With our example component, we slot a button element into the default slot. The `x-component` can apply styles to the `button` by using the `::slotted` CSS selector.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    ::slotted(button) {
      background: #2d2d2d;
      color: #fff;
      padding: 12px;
      border: 0;
      cursor: pointer;
    }
  </style>
  <div>
    <slot></slot>
  </div>
`;

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

customElements.define('x-component', XComponent);
```

The slotted selector takes a selector as a parameter to select which slotted element you would like to style.

```
::slotted(button) {
  background: #2d2d2d;
  color: #fff;
  padding: 12px;
  border: 0;
  cursor: pointer;
}
```

The `::slotted` selector can help theme as now parent elements can override the styles for slotted child elements. However, there are some limitations for the slotted selector. The selector can only select direct children of your component. If our button had an inner span element, the `x-component` would not be able to use `::slotted` to style the span tag. We will dig into slot styles in further detail in our advanced chapters.

In the next chapter, we will examine how we can take what we have learned with styling Web Components and build out our own custom themes.

# Chapter 6 - Theming Web Components

This chapter will cover how we can leverage Custom CSS Properties to style and theme our Web Components. Using CSS Custom Properties, we can quickly build out flexible and easy to maintain themes. To do this, we need to learn how to leverage global and component scoping when managing our CSS styles.

## Root Scope and Host Scope

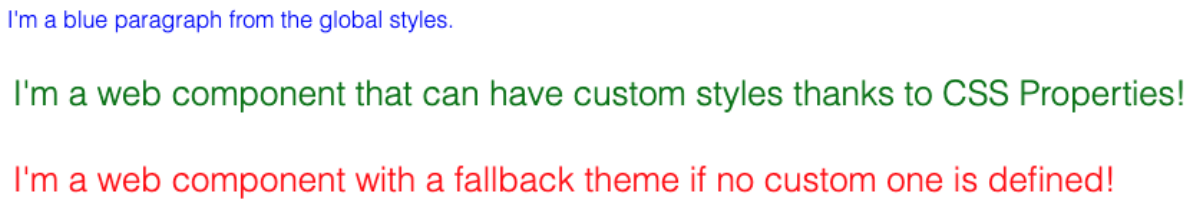
Our web component defined our CSS custom properties at the component `:host` level. We can also set CSS Custom Properties at the `:root` level and have a default value in our component. Let's take a look.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    p {
      color: var(--z-component-color, orange);
      font-size: var(--z-component-font-size, 16px);
      padding: 4px;
    }
  </style>
  <p>
    I'm a web component with a fallback theme if no custom one is defined!
  </p>
`;

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

customElements.define('z-component', ZComponent);
```

In our `z-component` we have two CSS custom properties defined similarly to the `x-component`. However, in the `z-component` we do not have the CSS custom properties defined on the `:host`. Instead of using `:host`, we set them at the point they are used.

```
p {
  color: var(--z-component-color, orange);
  font-size: var(--z-component-font-size, 16px);
  padding: 4px;
}
```

With CSS Custom Properties, we can define a second parameter which will be the default value if the property is not set elsewhere in the parent scope or `:root` selector.




**CSS Custom Properties with Fallback Values**

In our `z-component` we can see the values are not the defaults we provided but were instead overridden by defining them in the `:root` selector.

```
:root {
  --z-component-color: red;
  --z-component-font-size: 24px;
}
```

When defining properties at the root, it is important to prefix the property name to avoid any naming collisions. If we look at the `index.html` we can compare and see the API's differences when using `:root` and `:host` when creating CSS custom properties for our components.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Web Component Essentials</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="/index.css" rel="stylesheet" />
  <style>
    p {
      color: blue;
    }

    /* variables scoped to component :host */
    x-component {
      --color: green;
      --font-size: 24px;
    }

    /* variables scoped to global :root */
    :root {
      --z-component-color: red;
      --z-component-font-size: 24px;
    }
  </style>
</head>
<body>
  <main>
    <p>
      I'm a blue paragraph from the global styles.
    </p>

    <x-component>I'm a web component that can have custom styles thanks to CSS Prope\
rties!</x-component>
    <z-component>I'm a web component with a fallback theme if no custom one is defin\
ed!</z-component>
  </main>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

The tradeoff benefit of defining properties at a root level instead of host level is we get a flat selector of all the properties. This single flat selector of all the properties available can make custom theming easier.

# Implementing Dark Themes

When implementing themes in web components, we have a few strategies available for use. We can provide a separate style sheet that contains an updated list of the updated CSS Custom Properties for that particular theme.

```
:root {
  --modal-background-color: black;
  --modal-color: white;
  --dropdown-background-color: black;
  --dropdown-color: white;
  ...
}
```

We could apply this style sheet when we want to change the theme. We could also apply the styles by scoping them to a particular CSS class. When the CSS class is applied to say the body element, the theme is applied.

```
:root .dark-theme {
  --modal-background-color: black;
  --modal-color: white;
  --dropdown-background-color: black;
  --dropdown-color: white;
  ...
}
```

Either strategy will work to apply a custom theme across an entire application. We can take our style customization options one step further by providing built-in themes for our components. To do this, we can use a host selector to apply many different style properties based on any attribute that may exist on our host element.

```
/* if the host element has a dark attribute style the paragraph */
:host([dark]) p {
  color: #fff;
  background: #2d2d2d;
}
```

Instead of using a Custom CSS Property we can use the `:host` selector to apply a style when a class `:host(.dark)` or attribute `:host([dark])` is applied to our component.

```
<x-component dark></x-component>
```

When someone adds the `dark` attribute to our component, we can style our component to have a dark theme.

I am a paragraph.

I'm a web component that can have custom styles thanks to CSS Properties!

I'm a web component that can have custom styles thanks to CSS Properties!

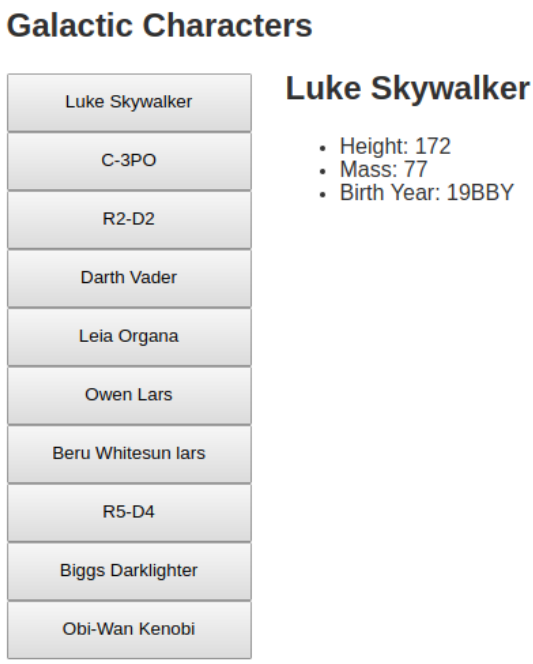
**Dark Theme with Custom Web Components**

There is a tradeoff when using a built-in theme into a component. There is a minor performance penalty when the theme is built-in as all users of the component will download the custom theme styles even if the custom theme is unused. Whereas the loading of a custom theme style sheet containing all the Custom CSS Properties will prevent this excess bloat. Each strategy is valid; it depends on how you would want developers to interact with your components theming API.

# Chapter 7 - Component Hierarchy and Architecture

In this chapter, we are going to discuss component hierarchy and communication. We have briefly touched on these topics in our previous chapter, where we built a dropdown component. We will look at how components can communicate with each other and where the Web Platform falls a bit short in aiding application architecture.

In our use case, we are going to walk through a list/detail view. A typical UI pattern is to have a list view, and when a list item is selected, a detail view of that particular item is displayed. We will walk through how to build this out with a recommended component structure. Here is what our list/detail view looks like:



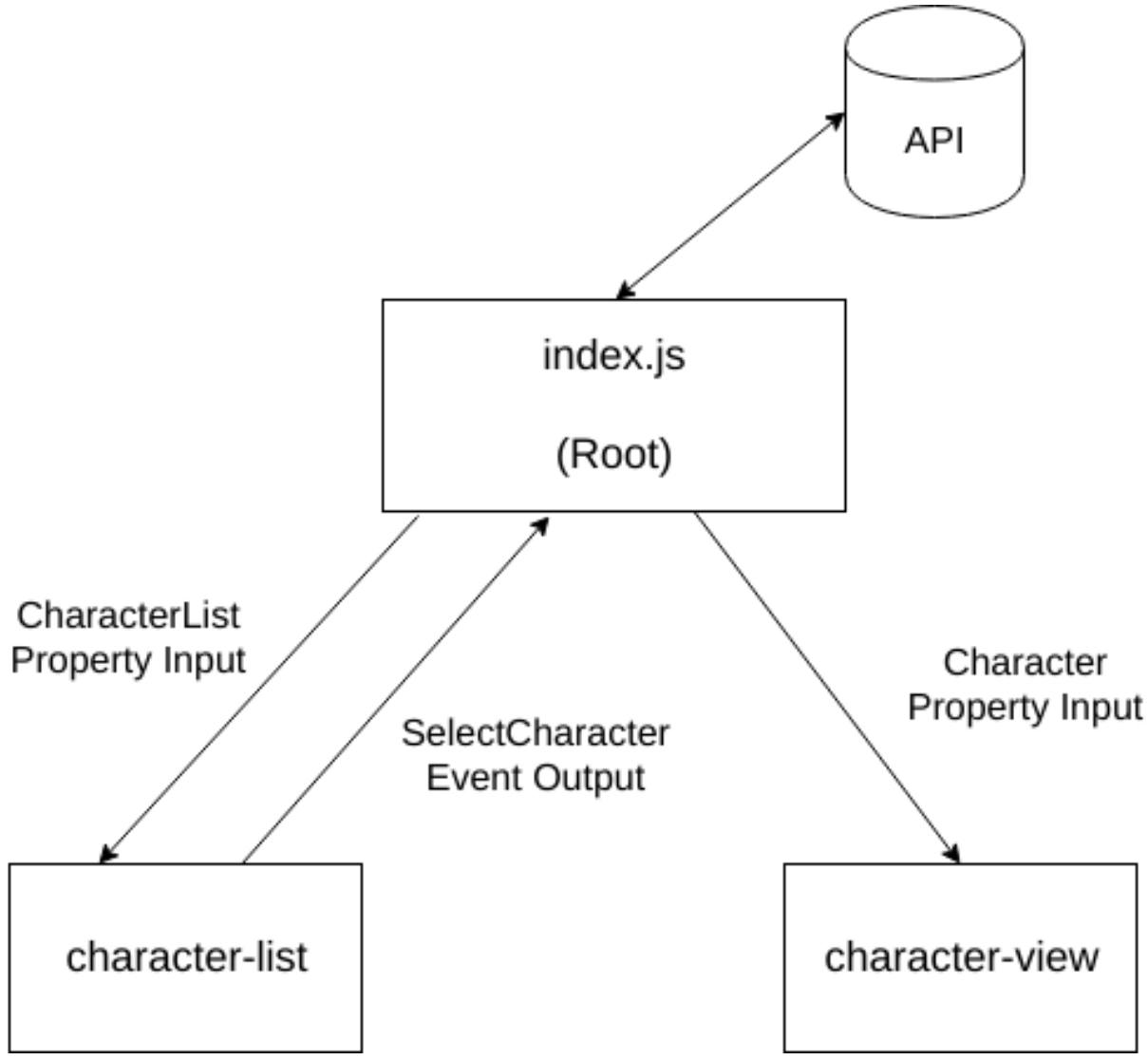

**List/Detail View with Web Components**

In our view, we have a list of Star Wars characters loaded from the open-source Star Wars API[5]. When we click one of the characters in our list, we will see a detail view of that character to the right of the screen.

[5] https://swapi.dev/

# Component Data Flow

We are going to follow some general best practices for component architecture/design. First, we want to keep the character list and character detail components generic and reusable. This means the character list and detail components should not fetch or request data but have that data passed in via custom properties. If we were to draw out the data flow of our application, it would look like the following:




**Data flow with Web Components**

We fetch data from our root or main application code. Our code then passes that data to our characters list component via the `characters` custom property. The characters property keeps the characters list component generic and reusable as it does not care where the data comes from.

Our character list component also needs to communicate back to the root when the user has selected. The character list will have a custom event `selectedCharacter` for our root component to listen to. In the event, we will pass back which character was selected by the user. This event allows the root component to take the chosen character and pass it to the character detail component. Like the list component, the detail component is generic as it takes its data as an input property. The detail component does not care where the data comes from, give it the data, and it does its one responsibility to render that data.

This pattern we see above is a typical component pattern used across almost all component libraries and frameworks; examples include Angular, Vue, and React. The primary idea is to have the majority of our components to be rendered or pure rendering components. Similar to a pure function, they take in data and render a template as its output. This makes components reusable and typically easier to unit test. Let's go ahead and jump into the code for our example.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
</head>
<body>
  <main>
    <h2>Galactic Characters</h2>
    <x-character-list></x-character-list>
    <x-character-detail></x-character-detail>
  </main>

  <script type="module" src="./index.js"></script>
</body>
</html>
```

Our root template or `index.html`, we have our starting index.js file and our top-level components, `x-character-list` and `x-character-detail`. Our `index.js` file will fetch our API data and pass that data along to our child list and detail components.

```
import './character-list.js';
import './character-detail.js';

const characterListComponent =  document.querySelector('x-character-list');
const characterDetailComponent =  document.querySelector('x-character-detail');

characterListComponent.addEventListener('selectCharacter', e => characterDetailCompo\
nent.character = e.detail);

fetch('https://swapi.dev/api/people/')
  .then(res => res.json())
  .then(data => {
    characterListComponent.characters = data.results;
    characterDetailComponent.character = data.results[0];
  });
```

In our `index.js` file, we get DOM references to our components. We listen to the custom event `selectCharacter` from our `characterListComponent`. This event will notify us when a user has selected a character. When the event fires, it passes a reference of the selected character so we can pass it along to our `characterDetailComponent` to render.

## Character List

We then fetch our user data from the API we pass the list of characters to the custom input property `characters` on the `characterListComponent`. We also give an initial character to the `character` input property on the `characterDetailComponent`. Let's take a look at the `characterListComponent` next and then break it down, line by line.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    button {
      display: block;
      padding: 12px;
      width: 100%;
    }
  </style>
  <section></section>
`;

class XCharacterList extends HTMLElement {
  get characters() {
```

```
    return this._characters;
  }

  set characters(value) {
    this._characters = value;
    this._render();
  }

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

  _render() {
    this._characters.forEach(character => {
      const button = document.createElement('button');
      button.appendChild(document.createTextNode(character.name));
      button.addEventListener('click', () => this.dispatchEvent(new CustomEvent('sel\
ectCharacter', { detail: character })));
      this.characterListElement.appendChild(button);
    });
  }
}

customElements.define('x-character-list', XCharacterList);
```

Ok, so lets walk through this component piece by piece. First we have our declared template:

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    button {
      display: block;
      padding: 12px;
      width: 100%;
    }
  </style>
  <section></section>
`;
```

Our template has some minor styles for the buttons we will be creating and a `section` tag for us to

select and add buttons to. Next is the class definition:

```
get characters() {
  return this._characters;
}

set characters(value) {
  this._characters = value;
  this._render();
}

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

We have a single custom property, `characters`. We have our getter and setter for this property. The constructor does the typical boilerplate setup we have seen in our previous examples. We create a reference to the `section` element for us to generate the list of buttons. Notice in the setter for `characters` we all `this._render()`. Let's go ahead and take a look at the `_render()` method.

```
_render() {
  this._characters.forEach(character => {
    const button = document.createElement('button');
    button.appendChild(document.createTextNode(character.name));
    button.addEventListener('click', () => this.dispatchEvent(new CustomEvent('selec\
tCharacter', { detail: character })));
    this.characterListElement.appendChild(button);
  });
}
```

Our render method is called any time the `characters` property is updated. When the render method runs, we loop through our given characters, creating our buttons. Because we have no built-in templating support, we have to create the DOM nodes, buttons manually and then add each click event to each button.

In the click event, we trigger an output custom event `selectCharacter`. As you can see, templates can get very tedious to maintain without a proper templating system. In a later chapter, we will cover a few options that make it significantly easier to create dynamic templates and events.

## Character Detail

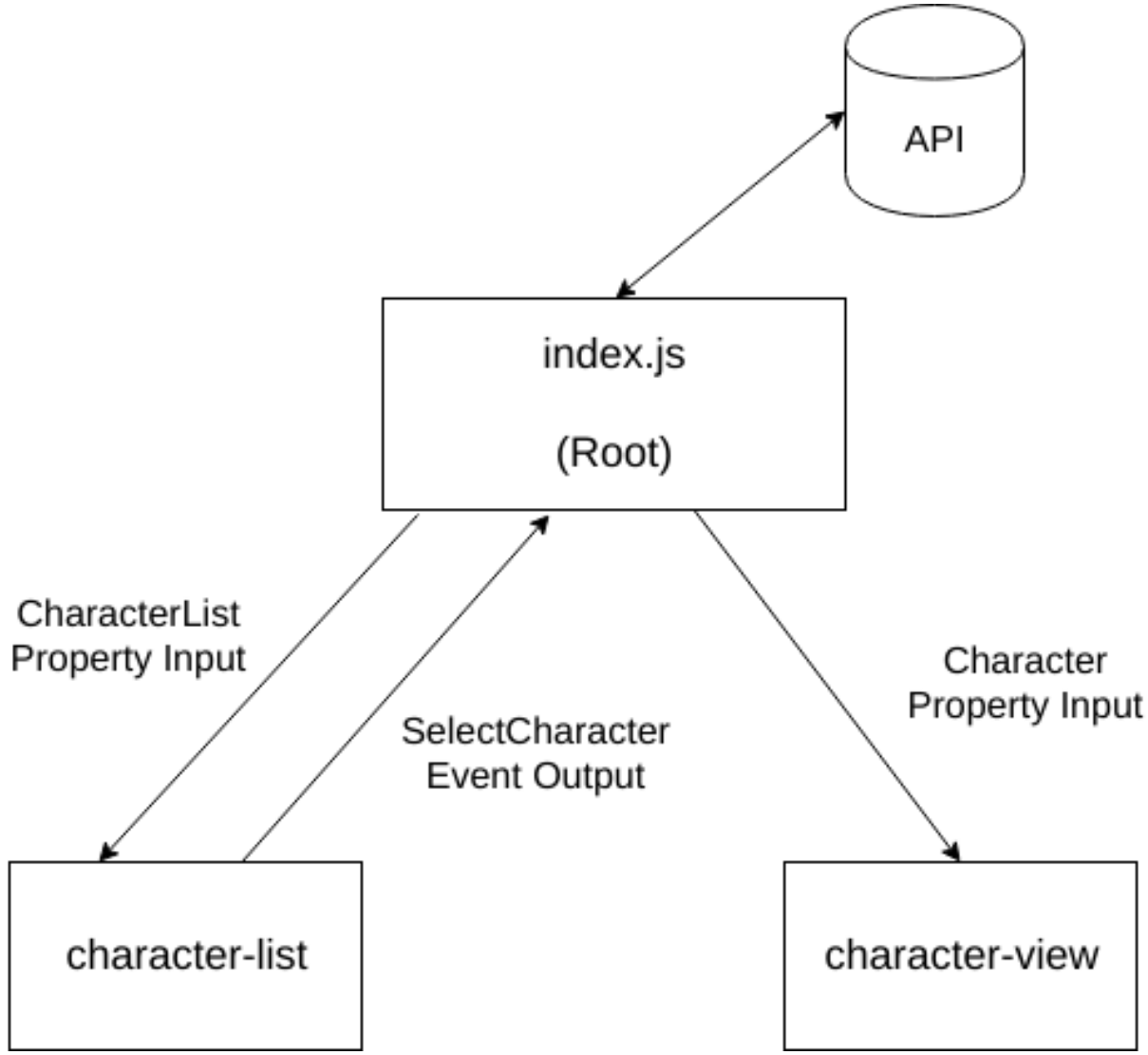
The `x-character-detail` component will be responsible for rendering the details about a particular character in our application. Let's take a look at the code.

```
class XCharacterList extends HTMLElement {
  get character() {
    return this._character;
  }

  set character(value) {
    this._character = value;
    this._render();
  }

  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  _render() {
    this.shadowRoot.innerHTML = `
      <h2>${this.character.name}</h2>
      <ul>
        <li>Height: ${this.character.height}</li>
        <li>Mass: ${this.character.mass}</li>
        <li>Birth Year: ${this.character.birth_year}</li>
      </ul>
    `;
  }
}

customElements.define('x-character-detail', XCharacterList);
```

Starting with the class definition, we can see we have the one input property, `character`. Whenever the property changes, we call the `_render()` method. Because this template has no dynamic event handlers like our list component, it is quite more straightforward.

```
_render() {
  this.shadowRoot.innerHTML = `
    <h2>${this.character.name}</h2>
    <ul>
      <li>Height: ${this.character.height}</li>
      <li>Mass: ${this.character.mass}</li>
      <li>Birth Year: ${this.character.birth_year}</li>
    </ul>
  `;
}
```

Looking back at our original diagram, we can see how we can leverage input properties and output events to accomplish component-to-component communication in a decoupled way.




**Data flow with Web Components**

Now our detail component has the sole responsibility of rendering the details of one character. This decouples the component from its data source. By decoupling the component, we make it more reusable and testable.

The idea of pure rendering components is a common pattern with many component-based JavaScript frameworks. This is especially important for Web Components since it is likely that our Web Components will be mixed with other opinionated JavaScript frameworks. Keeping the components generic or render only will make them easier to integrate into existing applications.

In our next chapter, we will learn the basics of how to bundle and publish our Web Components for other developers to use.

# Chapter 8 - Production Ready(ish) Web Components

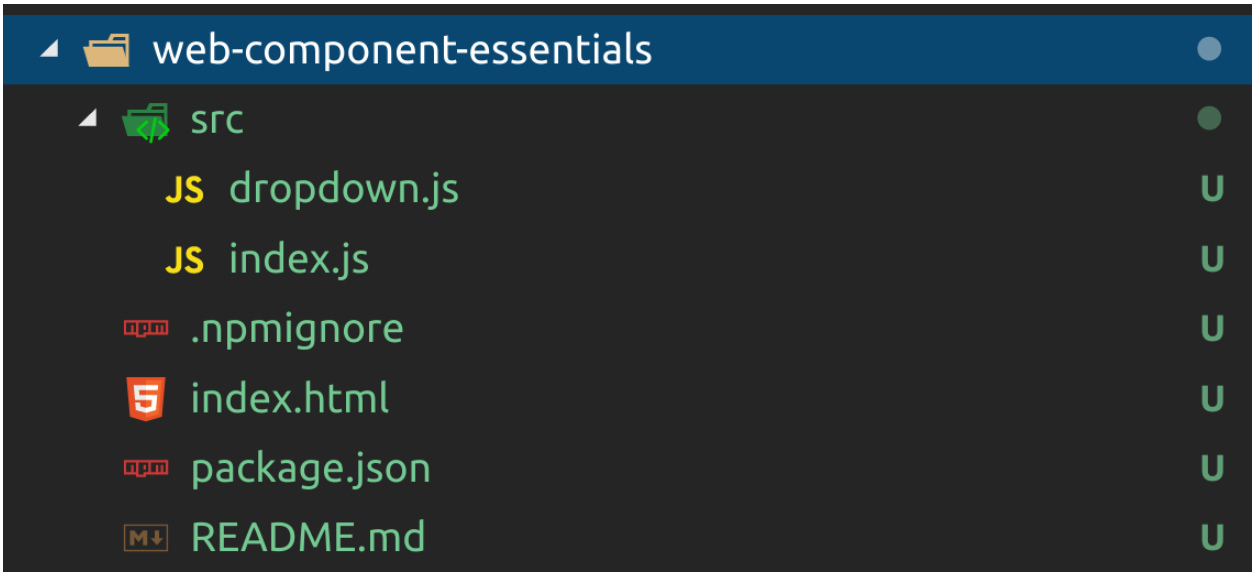
This chapter will go over a brief overview of what it takes to publish Web Components. The chapter is named "Production Ready(ish)" because while this chapter introduces what it takes to publish components, we will later cover additional tooling that automates much of the work we will cover in this chapter. This chapter emphasizes why we still need a build process to distribute Web Components with cross-browser support.

We have covered the basics of making a Web Component, but to make it useful, we need to be able to use it in multiple applications. We will include the simplest way to publish our component for others to use and then slowly add more complexity to cover more use cases.

## Publishing with NPM

With our first example, we will make a few assumptions about how our Web Component will be used. First, we are supporting only browsers that support the core Web Component APIs. We will cover later how we can get better broad browser support. We will also assume that the consumers of our Web Component will use the Node Package Manager (NPM) to install and use our component.

In our example, we will have a basic folder structure like the following:




**Basic Web Component Library**

In our `dropdown.js` we have the dropdown component we built in our previous chapter. In the `index.js`, we re-export the component and any other components in our library.

```
// index.js
export * from './dropdown';
```

To publish to NPM, we need first to create our `package.json` file. The `package.json` file defines some metadata about our package as well as any other dependencies it may have on other NPM packages.

To create our `package.json`, we will need to have NodeJS[6] installed. Once installed, the NPM command line/terminal CLI will be available for us to use. To generate the `package.json`, we will run the following command in the directory of our component library:

```
npm init
```

Running this command will prompt us with a few basic settings such as the package name, version, and license. Once that is done, we will need to add some additional information.

```
{
  "name": "web-component-essentials",
  "version": "0.0.1",
  "description": "",
  "main": "src/index.js",
  "module": "src/index.js",
  "directories": {
    "src": "src"
  },
  "author": "Cory Rylan",
  "license": "MIT"
}
```

In the `package.json` we define two new properties, `main` and `module`. We use these to help package bundlers like Webpack understand how to consume our package and where our library's entry point is. This is the basic setup for a simple NPM package.

Now that we have the minimal setup, we can publish our component by running:

```
npm publish
```

[6] https://nodejs.org

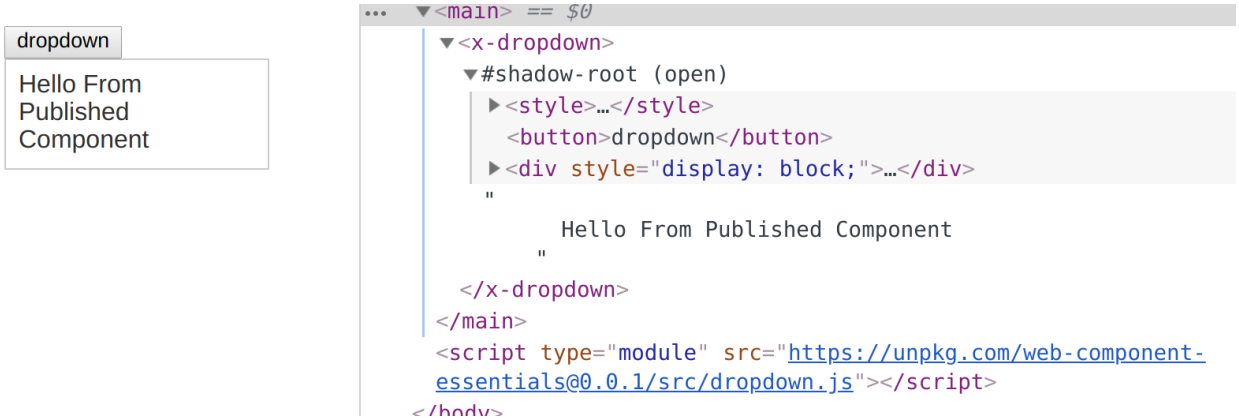
Assuming you have an NPM account and are logged in, your component should be successfully published to the NPM registry. Here is where our demo dropdown component is located https://www.npmjs.com/package/web-component-essentials[7].

To use our component in a web project, we have a couple of options. Typically we would have some build system or asset bundler like Webpack. For this first demo, we are going to keep it simple with a good old script tag. That's right, no fancy Webpack config or build step, just a single script tag in an HTML page. Let's take a look.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
</head>

<body>
  <x-dropdown>
    Hello From Published Component
  </x-dropdown>
  <script type="module" src="https://unpkg.com/web-component-essentials@0.0.1/src/dr\
opdown.js"></script>
</body>

</html>
```

Our `index.html` we have a reference to the `x-dropdown` and a single script tag. We can use the unpkg.com[8] CDN to use our published component without any build steps easily. If we take a look at our running web page, we see the following:

[7] https://www.npmjs.com/package/web-component-essentials
[8] unpkg.com




**Publish Web Component**

Congratulations, you have created and published your first Web Component! Even though we have gotten this far, there is still a lot to learn. In our following sections, we will learn about advanced tools to handle more complex Web Components and get the optimal browser support needed for our components.

# Browser Support

Web Components are now supported in all modern browsers. Only with IE11 will you need to compile down to ES5 and handle the polyfills.

There are two challenges with shipping Web Components to browsers that do not yet support them. The first is the lack of ES2015 JavaScript support. Older browsers, yes, looking at you IE, don't support some of the latest JavaScript features like Classes that our Custom Element definitions depend on. To get the optimal browser support, we need to add a build step to our codebase to transpile our code. Transpiling code is similar to a compiler; we take our ES2015 code and convert it to an older ES5 code that all browsers support, even IE.

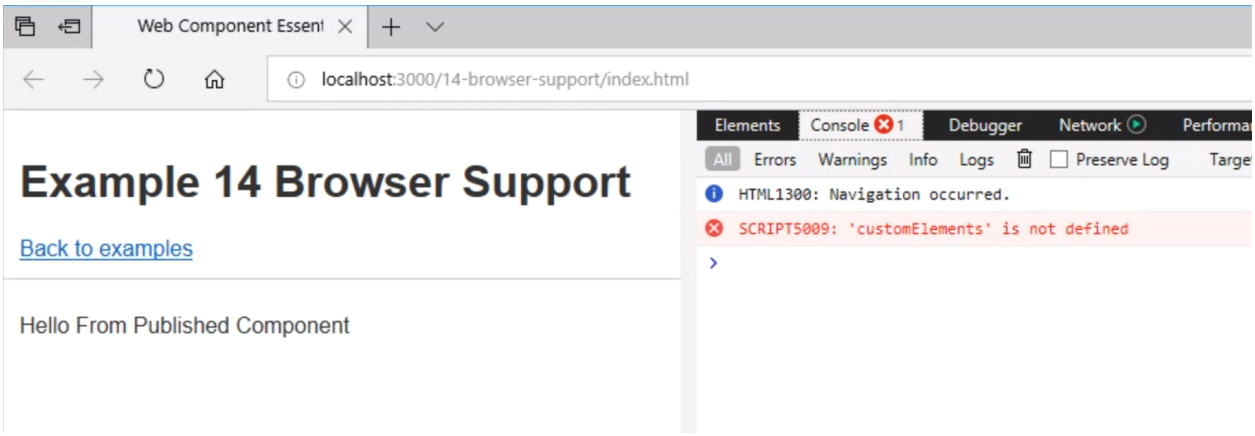
There are a couple of common ways to transpile our JavaScript code from ES2015 to ES5 code. The two most common ways to achieve this are via Babel and TypeScript. Babel takes the ES2015 code and can transpile it down to ES5. TypeScript has similar transpiling support as Babel and adds static typing to our JavaScript, making it much more maintainable and testable. We will go into TypeScript in a later chapter. This section will stick with vanilla JavaScript and use Babel to change our ES2015 code to ES5.

# Polyfills

All modern evergreen browsers have ES2015 JavaScript support. This allows us to use new language features, such as Classes and Modules. While almost all browsers excluding IE support ES2015, not all browsers support the Web Component APIs. Some of these APIs include Custom Elements,

Templates, Shadow DOM, and CSS Custom Properties. Fortunately, we can polyfill some of these features to still create Web Components and get cross-browser support.

We will use the Web Component JS Polyfills[9]. These polyfills include the essential Web Component API features we need. Going back to the dropdown component we published in our previous section worked just fine in Chrome. Let's go ahead and open the example in Edge (version 17).




**Broken Web Component**

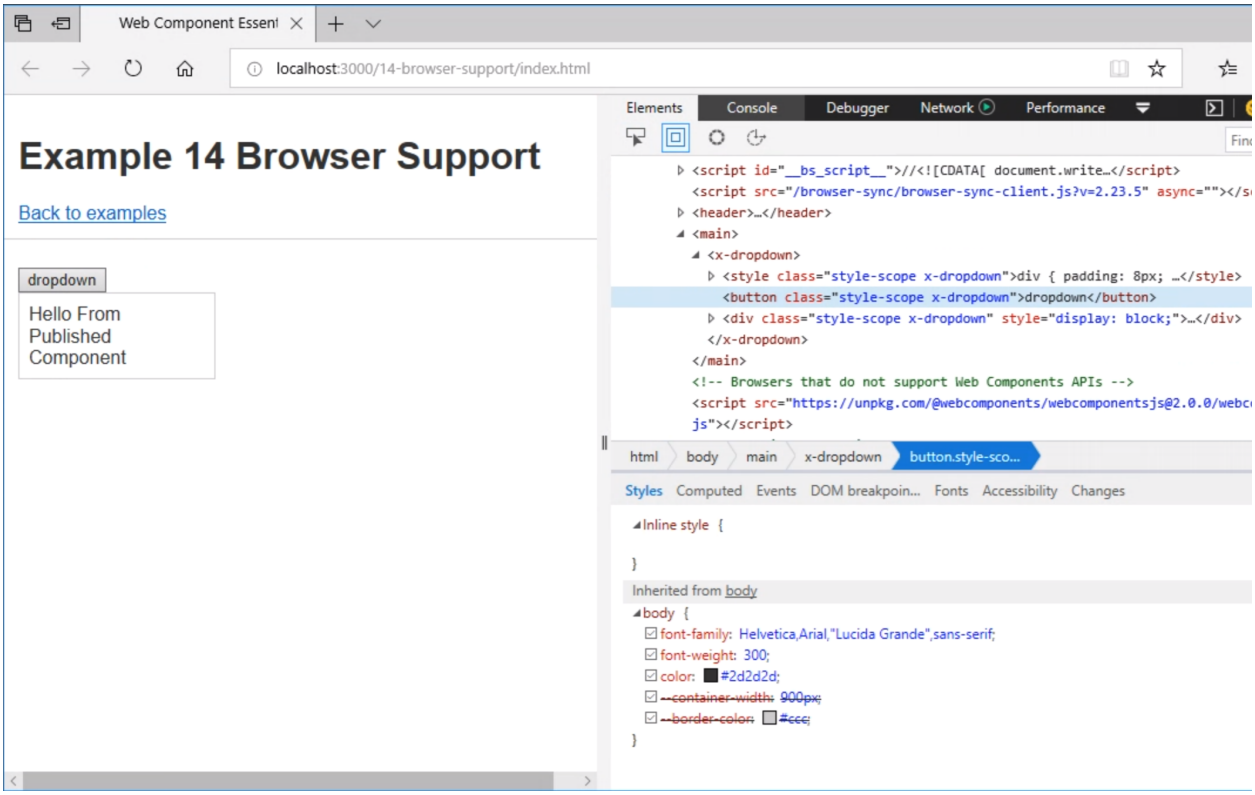
As you can see, our component fails to be created in this version of Edge. This is because this version of Edge (version 17) does not support some Web Component APIs. To fix this, let's add the following code to our `index.js` file.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Web Component Essentials</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="/index.css" rel="stylesheet" />
</head>

<body>
  <header>
    <h1>Example 14 Browser Support</h1>
    <a href="/">Back to examples</a>
  </header>

  <main>
    <x-dropdown>
```

[9] https://github.com/webcomponents/webcomponentsjs

```
19        Hello From Published Component
20      </x-dropdown>
21    </main>
22    <!-- Browsers that do not support Web Components APIs -->
23    <script src="https://unpkg.com/@webcomponents/webcomponentsjs@2.0.0/webcomponents-\
24  bundle.js"></script>
25
26    <!-- ES2015+ version of our library for evergreen browsers -->
27    <script type="module" src="https://unpkg.com/web-component-essentials@0.0.4/src/dr\
28  opdown.js"></script>
29  </body>
30
31  </html>
```

We add the `@webcomponents` polyfill bundle, which fills in the missing features for our component to work. If we save and go back to Edge, we should see our component working.




**Polyfilled Web Component working in Edge**

Polyfills can help us get the cross-browser support we need, but many of us still need to worry about older browsers, specifically IE11. To support IE11, we will need the same polyfills and another build step to convert our ES2015 code to ES5 code that IE can understand.

In our library containing the dropdown component, we will add some tools to our project to compile

our ES2015 code to ES5. The two primary tools we will be using in this section are Webpack and Babel. Webpack is a tool that packages dependencies together for web applications. Babel is a compiler that can take our ES2015 JavaScript and compile it out to ES5 code that we need for IE11.

This section will be a brief introduction to these tools, but by no means is a comprehensive overview. One could easily have multiple books written about the in-depth technical topics of Babel and Webpack. We will focus on the minimal work to achieve our IE11 support then, in a later section, cover high-level component authoring tools to make this work even more straightforward.

# Installing Webpack and Babel

The first step to transpiling our code is to install both Webpack[10] and Babel[11]. To install in our previous project, we will run the following command.

```
npm install rimraf webpack webpack-cli babel-core babel-loader babel-preset-env --sa\
ve-dev
```

This will install several packages needed for our build step. We need to make a few changes to our `package.json` Let's go ahead and look at that now.

```
{
  "name": "web-component-essentials",
  "version": "0.0.4",
  "description": "",
  "main": "dist/index.js",
  "module": "src/index.js",
  "scripts": {
    "clean": "rimraf dist",
    "build": "npm run clean && webpack --mode production"
  },
  "directories": {
    "src": "src"
  },
  "author": "Cory Rylan",
  "license": "MIT",
  "devDependencies": {
    "rimraf": "^2.6.2",
    "webpack": "^4.16.5",
    "webpack-cli": "^3.1.0",
    "babel-core": "^6.26.3",
```

[10] https://webpack.js.org/
[11] https://babeljs.io/

```
    "babel-loader": "^7.1.5",
    "babel-preset-env": "^1.7.0"
  },
  "dependencies": {}
}
```

In our `package.json` we see we have our dependencies for our build process. We also have the following two entries:

```
"main": "dist/index.js",
"module": "src/index.js",
```

The main entry will point build tools to the compiled distributed ES5 code. The `module` entry will point build tools to where the ES2015 module code is located. The JavaScript situated in the `/dist` directory will be compiled from the source code we write in the `/src` directory. We also added two new npm scripts to the `package.json`.

```
"scripts": {
  "clean": "rimraf dist",
  "build": "npm run clean && webpack --mode production"
}
```

The `clean` script will remove any generated code from the previous build. The `build` script will run our clean script and then call the webpack command to build our library code. To run the build command in the command line, we run `npm run build`. Before we can run the command, we need to set up some more configuration.

# Webpack

Webpack[12] is a module/dependency packaging tool for web applications. Webpack makes it easy to take a large tree of dependencies in our application and package them up to be easy to distribute and version to a production environment. Webpack can package many different kinds of assets via plugins. For example, if we wanted to use a CSS preprocessor like Less or Sass, we could use a Webpack plugin to keep track of all our Sass files and compile them. Compiling the Sass would be enabled by a Sass Webpack plugin that will allow Webpack to compile, version, and package our CSS for production. For our use case, we are using Babel to compile and package up our ES2015 code to ES5.

First, we need to create a `webpack.config.js` file. This file is where we will configure Webpack to build and package our component library. Let's go ahead and jump into the configuration code.

[12] https://webpack.js.org/

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'index.js',
    library: 'webComponentEssentials',
    libraryTarget: 'umd'
  }
};
```

The file above is our minimal Webpack configuration. The `entry` property defines the entry point or main of our library. This file is where Webpack will start and resolve any sub-dependencies (import statements). The `output` property defines where the generated/compiled JavaScript should be located. The `output` property also defines what the output file name should be and what kind of JavaScript module to compile to.

Before JavaScript modules existed in browsers, the JavaScript ecosystem had to create makeshift modules. With this arose many different module formats. Nowadays, we can use native JavaScript modules for our code. For browsers that don't support native JavaScript modules (IE), we typically compile them to umd (Universal Module Definition) modules or globals.

Now that we have our config defined, we can run our npm build script.

```
npm run build
```

If we look in the dist folder, you will see an `index.js` file that is compiled to the older ES5 JavaScript that will be compatible with older browsers like IE11.

We have not yet addressed issues, such as dynamically serving different polyfills or bundles based on browser feature detection or support. This chapter was a brief introduction to the JavaScript tooling ecosystem for transpiling and adding browser support. In later chapters, we will introduce Web Component-specific tooling that makes Web Component authoring and publishing more straightforward. Before we dig into more tooling in our next chapter, we will take the dropdown component that we have built and integrate it into an Angular and VueJS application.

# Chapter 9 - Using Web Components in Angular and VueJS

One of the big reasons to adopt Web Components is the ability to reuse components across a wide variety of JavaScript frameworks and libraries. This chapter will see an example Angular and VueJS application that will use our dropdown component.

## Angular

Angular[13] has been designed from the ground up to work with Web Components. Angular can consume Web Components and publish Angular components as Web Components via the Angular Elements API. For our example, we will be showing how to install the dropdown component into an Angular CLI project. You can learn more about the Angular CLI at cli.angular.io[14]. Let's get started.

First, we will create an Angular project using the Angular CLI. We will need to install the Angular CLI by running the following command:

```
npm install -g @angular/cli
```

This command will install the Angular CLI tooling to our terminal/command line. Once installed, we can run the following command to create our CLI project.

```
ng new my-app
```

This command will create a CLI project and install all the necessary NPM packages. Once completed in our CLI project, we can run:

```
ng serve
```

The `ng serve` command will run our Angular application locally at `localhost:4200`. Now that we have our Angular project up and running, we need to install our dropdown component. Remember, in our previous chapters, we published our component to NPM. In our Angular project, we can now install that component by running:

[13] https://angular.io
[14] https://cli.angular.io

```
npm install web-component-essentials
```

This command will install our component to our Angular project and add an entry into the `package.json`. Once installed in our `app.module.ts` we can import the component.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import 'web-component-essentials';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  schemas: [
    CUSTOM_ELEMENTS_SCHEMA // Tells Angular we will have custom tags in our templates
  ]
})
export class AppModule { }
```

Once imported, we need to add `CUSTOM_ELEMENTS_SCHEMA` from `@angular/core` to the application module. The `CUSTOM_ELEMENTS_SCHEMA` tells Angular that we will be using custom elements that are not registered Angular components in our application.

Now that our component is installed, we can use it in our Angular application. Let's take a look at the `app.component.ts` file.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myTitle = 'project-angular';
  open = false;
```

```
11
12    toggle(event) {
13      console.log(event);
14      this.open = event.detail;
15    }
16  }
```

The App component is the root component of our Angular application. On our component, we will have two properties. The `myTitle` will be passed to the dropdown and the `open` property to track if the dropdown is open or closed.

The App component also has a single method, `toggle()` called whenever the dropdown has opened or closed. Next, let's look at the `app.component.html` template.

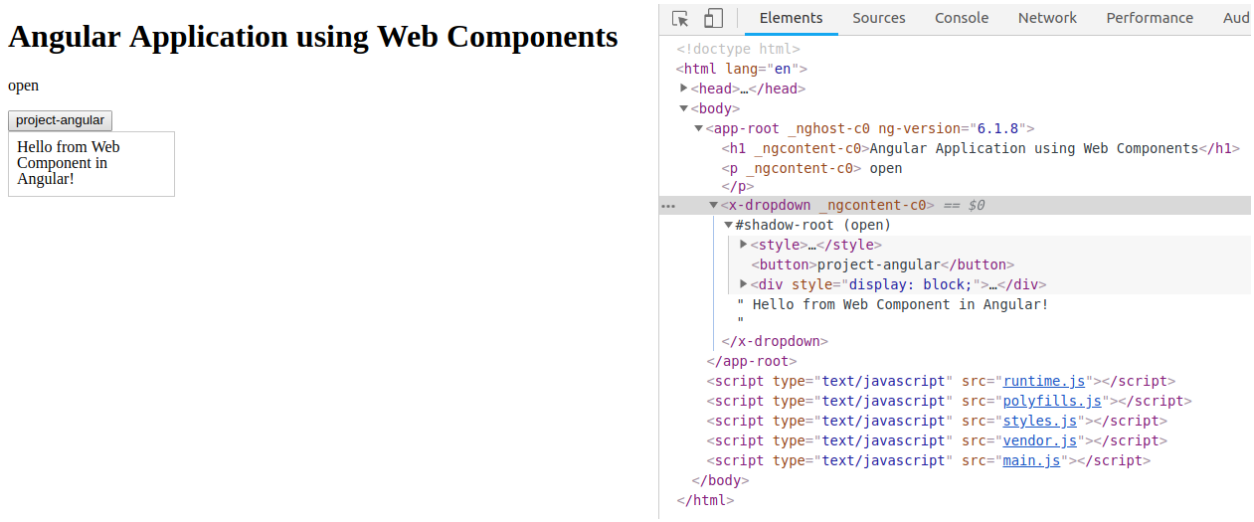
```
1  <h1>Angular Application using Web Components</h1>
2
3  <p>
4    {{open ? 'open' : 'closed'}}
5  </p>
6
7  <x-dropdown [title]="myTitle" (show)="toggle($event)">
8    Hello from Web Component in Angular!
9  </x-dropdown>
```

Our template has an Angular expression that displays the message `open` or `closed` based on the value of the `open` property. Angular has two different pieces of syntax for binding to properties and events. This syntax not only works for Angular components but also Web Components.

The first binding is the property binding syntax. This syntax uses the square braces `[title]="myTitle"` to tell Angular what property on the component should be set. In our example, we take the `myTitle` property value and set the `[title]` property of the dropdown component.

The second binding syntax is the event syntax. Angular components can listen to DOM events as well as Angular and Web Component events with this syntax. To bind to a event we use the parentheses `(show)="toggle($event)"`. In the parentheses, we pass the name of the event we want to listen to. On the right hand of the binding, we pass a method we want to be executed whenever the event occurs. If we want to pass the event value to the method, we use the `$event` keyword to tell Angular to pass the event value onto the `log` method.

With everything hooked up, we should see an output similar to this:




**Dropdown Web Component in Angular**

Angular is an excellent option for client-side applications as it has a robust API that works well for large enterprise applications while also adding fantastic Web Component support. Our next section will look into how to add our dropdown component to a VueJS application.

# VueJS

VueJS[15] is a new JavaScript framework that has recently gained a lot of popularity for its simple API and easier learning curve. This section will create a VueJS CLI project and add our dropdown Web Component to the project.

For this example, we will use the Vue CLI tool to generate a Vue project. The Vue CLI will provide all the tooling we need to get started building and running our application. You can learn more about the Vue CLI at cli.vuejs.org[16].

First, we need to install the Vue CLI tool. We can install the Vue CLI by running the following command:

```
1  npm install -g @vue/cli
```

Once installed, we can create our project by running the following:

```
1  vue create my-app
```

This command will create a basic Vue project as well as installing any dependencies. Once installed, we can install our dropdown by running the following:

[15] https://vuejs.org
[16] https://cli.vuejs.org/

```
npm install web-component-essentials --save
```

This command installs the dropdown package that contains the web component we will be using. Once installed, we can now import our dropdown into our Vue application. In the `main.js` we can add the following:

```
import Vue from 'vue'
import App from './App.vue'
import 'web-component-essentials'

Vue.config.productionTip = false

new Vue({
  render: h => h(App)
}).$mount('#app')
```

To run our Vue application, we can run the following command:

```
npm run serve
```

This command will start up our Vue app at `localhost:8080`. Let's take a look at the `HelloWorld.vue` component. Vue components use a single file style of organization. For example, Angular components have a TypeScript, HTML, and CSS file. Vue components have a single file that contains all three parts of the component. We will start with the template first.

```
// HelloWorld.vue
<template>
  <div>
    <h1>VueJS Application using Web Components</h1>

    <p>
      {{show ? 'open' : 'closed'}}
    </p>

    <x-dropdown :title="myTitle" @show="log">
      Hello from Web Component in Vue!
    </x-dropdown>
  </div>
</template>
```

The template of our Vue component has a similar element-binding syntax to Angular. We can see an expression that shows if the dropdown is open or closed, `{{show ? 'open' : 'closed'}}`. On

the dropdown component, we are using Vue's binding syntax. This binding syntax works with all HTML elements as well as custom elements from using Web Components.

To bind to a property, we use the `:` character. To bind a property to the dropdown title property, we write `:title="myTitle"`. Our Vue component has a `myTitle` property that has its value assigned to the `title` of the dropdown component.

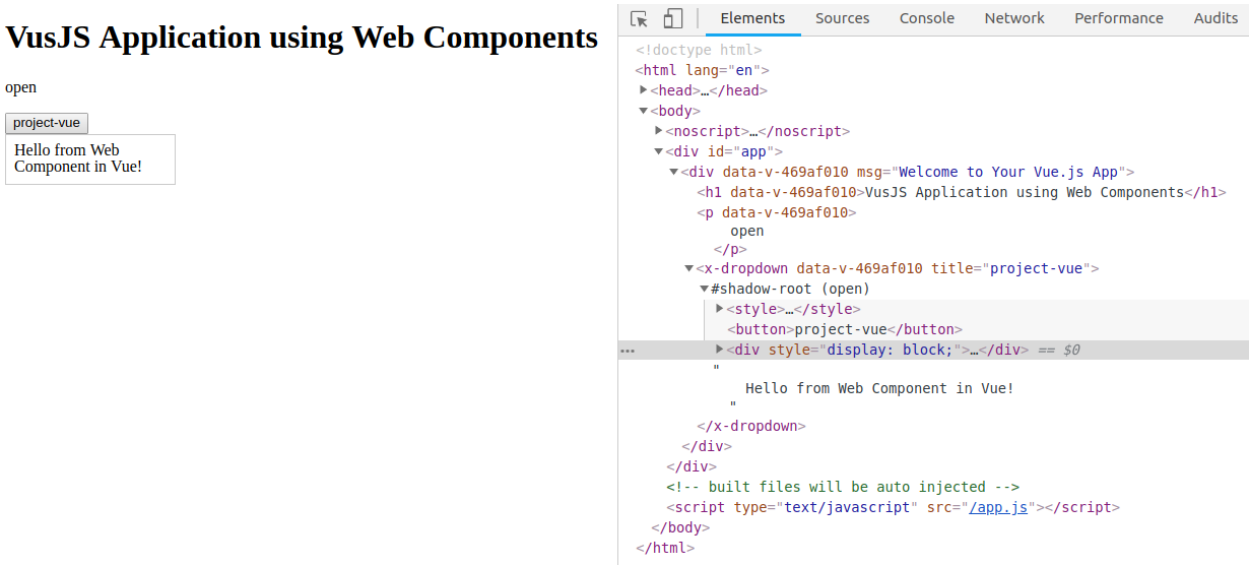
To listen to events, we use the `@` character. Our dropdown has a single event `show`. To listen to this event, we write `@show="log"`. This event binding will call the log method on our Vue component whenever the show event occurs.

Next, let's look at the Vue component JavaScript.

```
<script>
export default {
  name: 'HelloWorld',
  data: function () {
    return {
      myTitle: 'project-vue',
      show: false
    }
  },
  methods: {
    log: function (event) {
      console.log(event);
      this.show = event.detail;
    }
  }
}
</script>
```

The Vue component definition has data and method properties we want to bind on our Vue template. In our example, we have the two data properties, `myTitle` and `show`. We have a single method, `log`, which we saw being bound to the `@show` event.

If everything is hooked up correctly, we should see something similar to this in the browser:




**Dropdown Web Component in VueJS**

VueJS is a great lightweight option to build JavaScript applications that work well with Web Components. Our following few chapters will cover Web Component-specific tooling that makes Web Component authoring and publishing more straightforward and more manageable.

# Chapter 10 - Using Web Components in React

In our previous chapter, we saw how easy it is to reuse Web Components in JavaScript frameworks like Angular and VueJS. This chapter will cover how to integrate a Web Component into the React component library.

React is a JavaScript library made by Facebook that allows developers to compose UIs with components. React was the first JavaScript library/framework to popularize component-driven architecture. React was also created before the Web Components APIs were standardized. Thus, React does not have broad support for Web Components like the majority of other JavaScript libraries and frameworks.

## React Compatibility

React uses a similar mechanism for component communication by passing properties and functions as events between components. Unfortunately, the React event system is a synthetic system that does not use built-in browser events. This synthetic system means Web Component events cannot communicate with React components. React and the JSX templating syntax it uses treat all custom element properties as attributes incorrectly, forcing React users to use string values.

To overcome these shortcomings in our example, we will show how we can create thin React wrapper components around our Web Components. Wrapper components will allow React to be able to become compatible with our Web Components.

## Create React App

To demonstrate Web Components in React, we will use the Create React App[17] CLI tool to easily create a React application. To create our app, we run the following commands:

```
npx create-react-app my-app
cd my-app
npm start
```

Once created, we will have a complete running React application. Now we need to install from NPM our Web Component just like in our previous examples.

[17] https://facebook.github.io/create-react-app/

```
npm install web-component-essentials --save
```

In our React application, we will need to create a React `Dropdown` component to wrap our existing `x-dropdown` component.

```
import React, { Component } from 'react';
import 'web-component-essentials';

export class Dropdown extends Component {
  render() {
    return (
      <x-dropdown>
        {this.props.children}
      </x-dropdown>
    )
  }
}
```

To use our `x-dropdown`, we import the package into the `Dropdown.js` React component. In the render function, we add `{this.props.children}` to pass child elements into our content slot.

## Properties and Events

We need to map the Web Component properties and events to our React version of the component. We need to use the `componentDidMount()` lifecycle hook.

```
import React, { Component } from 'react';
import 'web-component-essentials';

export class Dropdown extends Component {
  constructor(props) {
    super(props);
    this.dropdownRef = React.createRef();
  }

  componentDidMount() {
    this.dropdownRef.current.title = this.props.title;

    if (this.props.onShow) {
      this.dropdownRef.current.addEventListener('show', (e) => this.props.onShow(e));
    }
```

```
  }

  render() {
    return (
      <x-dropdown ref={this.dropdownRef}>
        {this.props.children}
      </x-dropdown>
    )
  }
}
```

Using the Refs API[18], we can grab a DOM reference to our `x-dropdown`. Using this reference, we can create our event listener. In our event listener, we can call any passed functions to our `onShow` prop for our react component. This will allow our Web Component to be able to communicate with other React components. We also assign the `title` prop of our React dropdown to our Web Component property.

```
// current gets the current DOM element attached to the ref
this.dropdownRef.current.title = this.props.title;
```

## Prop Updates

Next, we need to add additional code whenever one of the props on our React dropdown changes. To listen for prop updates we can use the `componentDidUpdate()` lifecycle hook.

```
import React, { Component } from 'react';
import 'web-component-essentials';

export class Dropdown extends Component {
  constructor(props) {
    super(props);
    this.dropdownRef = React.createRef();
  }

  componentDidMount() {
    this.dropdownRef.current.title = this.props.title;

    if (this.props.onShow) {
      this.dropdownRef.current.addEventListener('show', (e) => this.props.onShow(e));
```

[18] https://reactjs.org/docs/refs-and-the-dom.html

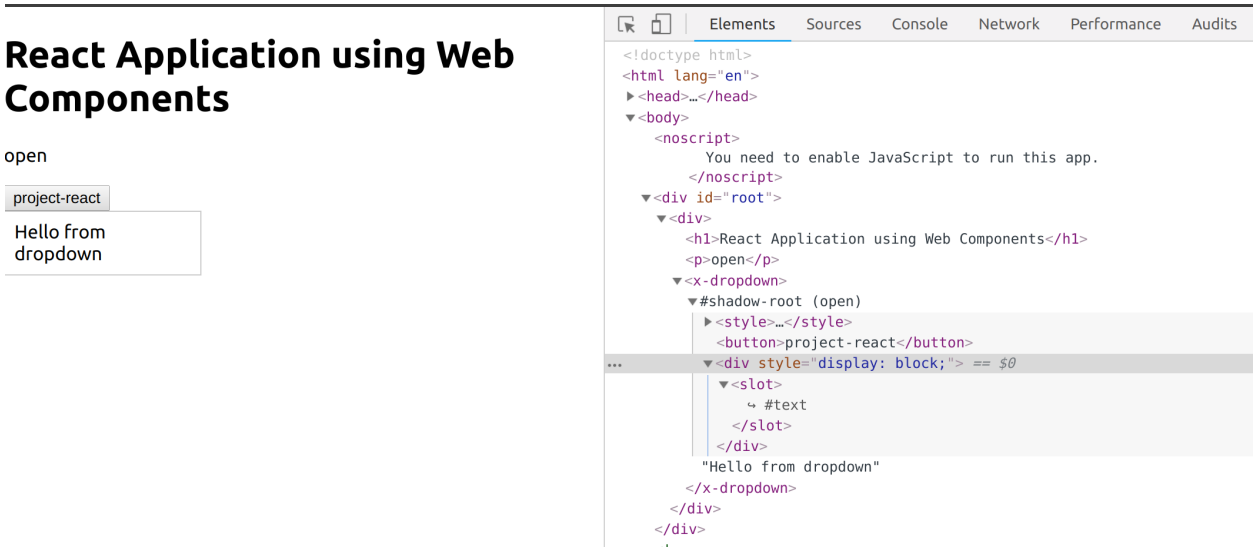
```
    }
  }

  componentDidUpdate(prevProps) {
    if (this.props.title !== prevProps.title) {
      this.dropdownRef.current.title = this.props.title;
    }

    if (this.props.show !== prevProps.show) {
      this.dropdownRef.current.show = this.props.show;
    }
  }

  render() {
    return (
      <x-dropdown ref={this.dropdownRef}>
        {this.props.children}
      </x-dropdown>
    )
  }
}
```

Using `componentDidUpdate()`, we can check when props are updated and efficiently update our Web Component properties. Now that we have mapped React props to our Web Component properties and events, we can use the `Dropdown` React component.

```
import React, { Component } from 'react';
import './App.css';
import { Dropdown } from './dropdown.js';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      show: false,
      title: 'project-react'
    };

    this.handleShow = this.handleShow.bind(this);
  }

  render() {
    return (
```

```
18        <div>
19          <h1>React Application using Web Components</h1>
20
21          <p>
22            {this.state.show ? 'open' : 'closed'}
23          </p>
24
25          <Dropdown title={this.state.title} onShow={this.handleShow}>
26            Hello from dropdown
27          </Dropdown>
28        </div>
29      );
30    }
31
32    handleShow(e) {
33      this.setState({ show: e.detail });
34    }
35  }
36
37  export default App;
```

Now we should see our rendered Web Component working in a React application.




**Dropdown Web Component in React**

In our `App` component, you can see the syntax is not much different than our Angular and Vue examples. Unfortunately, due to the incompatibility of React with the custom elements API, we have to add a thin compatibility layer between our components.

Hopefully, soon React will be able to adapt and become compatible with the custom elements API. To

follow the status of the open React issues related to Web Components, check out custom-elements-everywhere.com[19].

[19] https://custom-elements-everywhere.com/

# Chapter 11 - Lit

So far in this book, we have been exclusively working with the low-level Web Component APIs. These APIs are designed to be low level and for Framework and library authors to quickly build on top of. In the following few chapters, we will cover a few different options when it comes to authoring our Web Components. In this chapter, we are going to look into an advanced templating solution called Lit.

Lit is a lightweight library that makes it easier to write and use Web Components. Lit was created and is maintained by the Polymer[20] team at Google. Lit is not the Polymer Web Component library but a new stand-alone library. Let's get started!

## Lit and Template Literal Strings

Lit includes a lightweight templating library built on top of tagged template literals. Tagged templates are special functions attached to JavaScript template literal strings (the backtick symbol `). Template literals enable us to create a powerful template syntax right in the browser. Alongside the templating features, Lit also includes a base class to make it easier to build Web Components.

In this example, we will take our dropdown component we created in the previous chapters and convert it to use Lit. Let's start with a minimal amount of code to create our component.

```
1  import { LitElement, html } from 'lit';
2
3  class XDropdown extends LitElement {
4    render() {
5      return html`
6        Hello from Lit
7      `;
8    }
9  }
10
11 customElements.define('x-dropdown', XDropdown);
```

We import the `LitElement` base class from the `lit` package to create our dropdown element. Our dropdown extends the LitElement class and registers it just like a standard Custom Element. Lit expects a `render()` method to be implemented. This render method returns our template for our component. Our template is created using the tagged template function `html` from the lit package. The `html` template function will provide additional templating features that we will see next.

[20] https://www.polymer-project.org/

## Templates and Event Listeners

Let's go ahead and add the template needed to render our dropdown component.

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

class XDropdown extends LitElement {
  render() {
    return html`
      <style>
        .dropdown div {
          border: 1px solid #ccc;
          padding: 12px;
        }
      </style>
      <div class="dropdown">
        <button @click="${() => this.toggle()}">${this.title}</button>
        ${this.visible ?
          html`
            <div>
              <slot></slot>
            </div>`
          : '' }
      </div>
    `;
  }

  toggle() {
    this.visible = !this.visible;
  }
}

customElements.define('x-dropdown', XDropdown);
```

In the dropdown template, we have the button and slot element that will dynamically toggle our dropdown content, just like our previous chapter examples. Lit will automatically create our template using the HTML Template API and Shadow DOM. Using these APIs will allow us to use CSS Encapsulation.

Lit has a declarative API for listening to events. As you can see in the template we have the following binding:

```
`<button @click="${() => this.toggle()}">${this.title}</button>`
```

Using the `@` character, we can tell Lit what event to bind to and trigger expressions to be called. Our use case is when the click event occurs, call the `toggle()` method on the component class.

If we try to run the code as-is, the dropdown will not function correctly. We still need to add a few more things before our component is ready to use.

## Properties and Decorators

For Lit to correctly render our template when a property changes, we need to tell Lit which properties can change or be set explicitly. To do this, we have two options. The first option allows us to declare a list of properties that can be set on the component.

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

class XDropdown extends LitElement {
  static get properties() {
    return {
      visible: { type: Number },
      title: { type: String },
    }
  }

  constructor() {
    super();
    this.visible = false;
    this.title = 'dropdown';
  }

  render() {
    return html`
      <style>
        .dropdown div {
          border: 1px solid #ccc;
          padding: 12px;
        }
      </style>
      <div class="dropdown">
        <button @click=${(e) => this.toggle(e)}>${this.title}</button>
        ${this.visible ?
          html`
```

```
            <div>
              <slot></slot>
            </div>`
          : '' }
      </div>
    `;
  }

  toggle() {
    this.visible = !this.visible;
  }
}

customElements.define('x-dropdown', XDropdown);
```

The `static get properties()` will allow Lit to track when the properties are set and if it should re-render the component. The other optional API is that you can use experimental decorators and property initializers instead of using a static list of properties. Decorators and property initializers are proposed language features for JavaScript. If you are using Babel or TypeScript, you can have the following code instead,

```
import { LitElement, html } from 'lit';
import { property } from 'lit/decorators.js';

class XDropdown extends LitElement {
  @property()
  visible = false;

  @property()
  title = 'dropdown';

  render() {
    return html`
      <style>
        .dropdown div {
          border: 1px solid #ccc;
          padding: 12px;
        }
      </style>
      <div class="dropdown">
        <button @click=${(e) => this.toggle(e)}>${this.title}</button>
        ${this.visible ?
          html`
```

```
            <div>
              <slot></slot>
            </div>`
          : '' }
      </div>
    `;
  }

  toggle() {
    this.visible = !this.visible;
  }
}

customElements.define('x-dropdown', XDropdown);
```

Using decorators, you can have a much more declarative syntax. The tradeoff is you will need a build step to create your components which we lightly covered in the previous chapter.

## Custom Events

Events in Lit components are no different than standard custom element events.

```
import { LitElement, html } from 'lit';
import { property } from 'lit/decorators.js';

class XDropdown extends LitElement {
  static get properties() {
    return {
      visible: { type: Number },
      title: { type: String },
    }
  }

  constructor() {
    super();
    this.visible = false;
    this.title = 'dropdown';
  }

  render() {
    return html`
      <style>
```

```
        .dropdown div {
          border: 1px solid #ccc;
          padding: 12px;
        }
      </style>
      <div class="dropdown">
        <button @click=${(e) => this.toggle(e)}>${this.title}</button>
        ${this.visible ?
          html`
            <div>
              <slot></slot>
            </div>`
          : '' }
      </div>
    `;
  }

  toggle() {
    this.visible = !this.visible;

    // trigger custom event
    this.dispatchEvent(new CustomEvent('visibleChange', { detail: this.visible }));
  }
}

customElements.define('x-dropdown', XDropdown);
```

Now we have our properties and events set up, we can interact with our component just like any other Web Component.

```
const dropdown = document.querySelector('x-dropdown');
dropdown.title = 'custom dropdown';
dropdown.addEventListener('visibleChange', (e) => console.log(e));
```

## Binding to other Web Components with Lit

Lit can also be used to interact with other existing Web Components in our applications. In our previous example, we imperatively created a reference and an event listener for our dropdown. In the following example, we will use Lit to interact with our dropdown. First, we will create a top-level app component.

```
import { LitElement, html } from 'lit';
import { property } from 'lit/decorators.js';
import 'dropdown';

class XApp extends LitElement {
  render() {
    return html`
      <x-dropdown>
        Hello From Lit!
      </x-dropdown>
    `;
  }
}

customElements.define('x-app', XApp);
```

With our app component, we can use Lit's declarative binding to set properties and listen to events.

```
import { LitElement, html } from 'lit';
import { property } from 'lit/decorators.js';
import 'dropdown';

class XApp extends LitElement {
  render() {
    return html`
      <x-dropdown @visibleChange=${(e) => this.log(e)}>
        Hello From Lit!
      </x-dropdown>
    `;
  }

  log(e) {
    console.log(e);
  }
}

customElements.define('x-app', XApp);
```

In the app component template, we use the same `@` binding to listen to events. This binding works not only with native DOM events but custom events like the `visibleChange` event. We can also set the properties of our components with Lit.

```
import { LitElement, html } from 'lit';
import { property } from 'lit/decorators.js';
import 'dropdown';

class XApp extends LitElement {
  constructor() {
    super();
    this.customTitle = 'Custom Title!';
  }

  render() {
    return html`
      <x-dropdown
        @visibleChange=${(e) => this.log(e)}
        .title=${this.customTitle}>
        Hello From Lit!
      </x-dropdown>
    `;
  }

  log(e) {
    console.log(e);
  }
}

customElements.define('x-app', XApp);
```

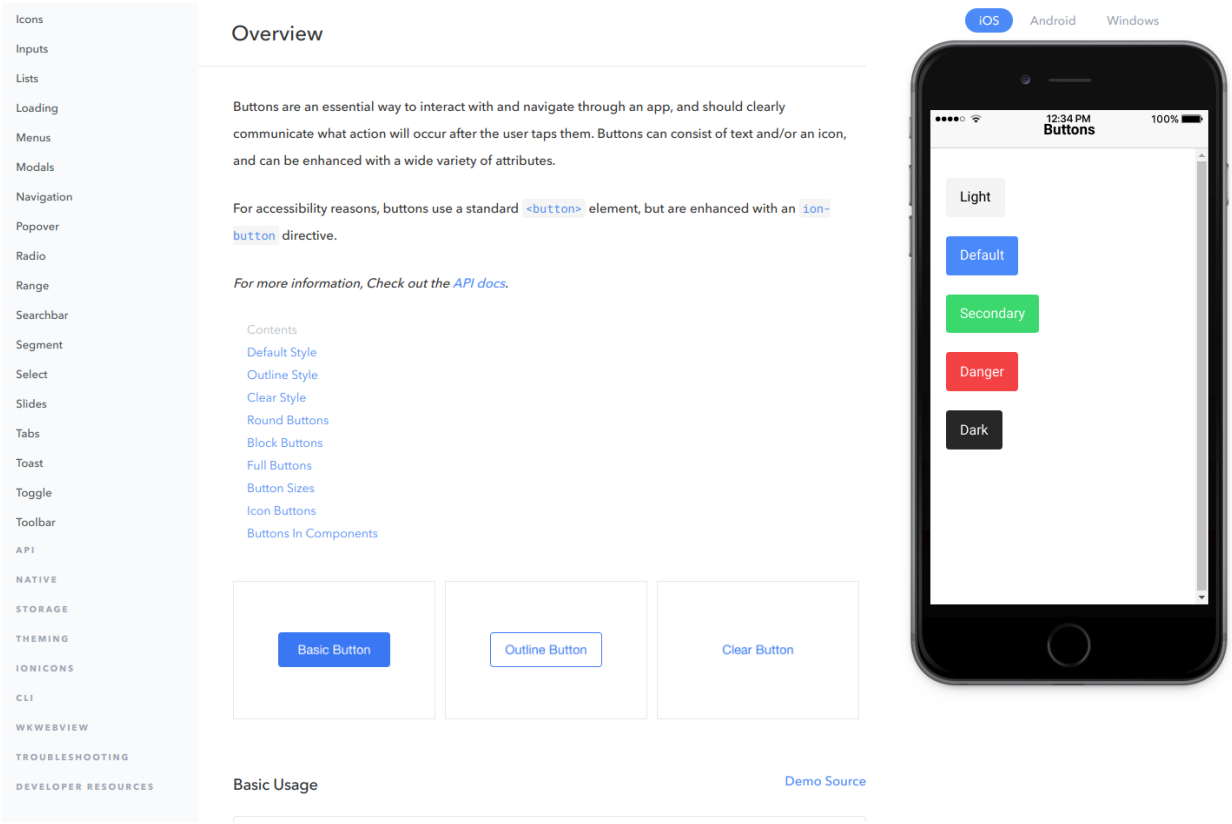
With `.title=${this.customTitle}` we use the `.` to bind to the `title` property of the dropdown. This binding allows us to pass data down to the component.

Using Lit, we can create Web Components with a nice declarative syntax while still keeping excellent performance. In our next chapter, we will look at a new tool called StencilJS that takes the Web Component abstraction one step further as a Web Component Compiler.

# Chapter 12 - Stencil JS

In the previous chapter, we learned about Lit and how it gives us an excellent templating system for our Web Components. In this chapter, we will cover another Web Component authoring tool called StencilJS[21]. Stencil is referred to as a Web Component compiler.

Stencil, unlike Lit, adds a build process to create its Web Components. Stencil is more opinionated but provides a higher-level API to create Web Components. Stencil leverages technologies like TypeScript and JSX. Stencil was made and is maintained by Ionic. Ionic is a company specializing in developing native mobile apps and Progressive Web Apps with Web technologies. Ionic has a rich suite of UI components built as Web components allowing developers to make native mobile apps on IOS and Android but use standard Web technologies.




**Ionic Component Docs**

The Ionic UI kit was built initially with Angular components, but recently Ionic created Stencil to develop their components as Web Components. Doing so has allowed anyone to use Ionic components and not restrict them to only being used in Angular apps.

[21] https://stenciljs.com/

In this chapter, we are going to create our dropdown component with Stencil. We will cover the benefits that Stencil provides as a top-down solution for authoring Web Components.

# The Stencil CLI

The Stencil CLI is a command-line tool that makes it easy to create and build Stencil Component libraries and Progressive Web Apps. The Stencil CLI allows us to write our components with TypeScript and JSX. The CLI also provides a mechanism to write CSS and Sass in stand-alone files. Lastly, the CLI also provides built-in unit testing so you can quickly write automated tests for your Web Components. In our example, we will use the Stencil CLI to create a component library for our dropdown component. Let's get started!

First, we need to run the Stencil CLI by running the following command:

```
npm init Stencil
```

This command will run the Stencil CLI and prompt you to choose a project to create. We will want to select the component library option.

Once completed, you will have a Stencil project to start authoring your components in. In the Stencil project, there will be an `src/components` directory. The project will start out with a single component `my-component.tsx`. Stencil is written with TypeScript and JSX. If you have used React or Angular, this will feel very familiar. Stencil looks almost like a hybrid of Angular and React.

We will create an `x-dropdown.tsx` file for our dropdown component in the components directory. Let's go ahead and look at the component code.

```
import { Component, Event, EventEmitter, Prop, State } from '@stencil/core';

@Component({
  tag: 'x-dropdown',
  styleUrl: 'x-dropdown.css',
  shadow: true
})
export class XDropdown {
  @Prop() title = 'dropdown';
  @State() show = false;
  @Event() showChange: EventEmitter;

  render() {
    return (
      <div>
        <button onClick={() => this.toggle()}>{this.title}</button>
```

```
17          {this.show
18            ? <div class="x-dropdown__content"><slot /></div>
19            : <div></div>
20          }
21        </div>
22      );
23    }
24
25    toggle() {
26      this.show = !this.show;
27      this.showChange.emit(this.show);
28    }
29  }
```

## Decorators

Stencil uses a combination of decorators and JSX to make a declarative and straightforward way to create Web Components. First, let's start with the `@Component` decorator.

```
1  @Component({
2    tag: 'x-dropdown',
3    styleUrl: 'x-dropdown.css',
4    shadow: true
5  })
```

If you have used Angular, this decorator will look very familiar. Decorators are built into TypeScript and in the proposal stages for JavaScript. Decorators allow developers to add additional metadata to classes and properties. Stencil uses these decorators to help create and compile your component into a Web Component.

The `@Component` decorator describes our dropdown component. The `tag` property defines what the element tag name should be. The `styleUrl` allows us to define a stand-alone stylesheet for component level styles. The `shadow` property will enable us to tell Stencil if we want to enable or disable the Shadow DOM for our component.

Next, in our class definition, we have three properties with decorators.

```
1  @Prop() title = 'dropdown';
2  @State() show = false;
3  @Event() showChange: EventEmitter;
```

Each decorator is provided by Stencil and adds behavior to our component. The first decorator, `@Prop` allows us to define public properties on our component for others to use and pass data around. For our single property, we have the `title` to set the dropdown button text.

The second decorator, `@State` allows Stencil to know which properties, when updated, should trigger the template to re-render. For our use case, whenever the button is clicked, we update the `show` property, and because of the `@State` decorator, the template will be re-rendered.

The last decorator, `@Event` allows us to create custom events for our component. By applying the `@Event` decorator we can call `this.showChange.emit(this.show);` to trigger our `showChange` event. The name of the property will be the name of the event emitted.

## JSX Templates

Next, in our component class definition, we have the `render()` method. The `render()` method expects us to return a JSX template.

```
render() {
  return (
    <div>
      <button onClick={() => this.toggle()}>{this.title}</button>
      {this.show
        ? <div class="x-dropdown__content"><slot /></div>
        : <div></div>
      }
    </div>
  );
}
```

In the template, we can listen to DOM events and trigger method calls. In our example `onClick` of the button, we call the `toggle()` method.

```
<button onClick={() => this.toggle()}>{this.title}</button>
```

To conditionally render parts of our template, we can use a ternary operator.

```
{this.show
  ? <div class="x-dropdown__content"><slot /></div>
  : <div></div>
}
```

Lastly, in our dropdown, we have the `toggle()` method.

```
toggle() {
  this.show = !this.show;
  this.showChange.emit(this.show);
}
```

The toggle method inverts the `show` property and then triggers the custom event to emit. Now that we have recreated our dropdown in Stencil, let's look at using our component in another Stencil template.

## JSX Component Bindings

In the `MyComponent` that the Stencil CLI created, we will update it to use our `XDropdown` component.

```
import { Component } from '@stencil/core';

@Component({
  tag: 'my-component',
  styleUrl: 'my-component.css',
  shadow: true
})
export class MyComponent {
  private myTitle = 'StencilJS';

  render() {
    return (
      <div>
        Hello, World! I'm a Web Component built with Stencil!

        <div>
          <x-dropdown title={this.myTitle} onShowChange={(e) => this.log(e)}>
            Hello from Stencil Dropdown!
          </x-dropdown>
        </div>
      </div>
    );
  }

  private log(event: any) {
    console.log(event);
  }
}
```

In the `MyComponent` file, we have a simple template that demonstrates how Stencil can bind to Web Components. Just like we saw in Angular and Vue, Stencil can bind to properties and events.

```
<x-dropdown title={this.myTitle} onShowChange={(e) => this.log(e)}>
  Hello from Stencil Dropdown!
</x-dropdown>
```

With Stencil, by default, binds to properties by naming the property on the component tag. To listen to custom events, we prefix the event name with `on`.

## Building your Stencil Components

Stencil can build entire applications, but it is designed to allow you to write and quickly publish your Web Components. To create your Web Components, in the root of your Stencil project, run the following command:

```
npm run build
```

This command will compile our Stencil components into plain Web Components that we can use anywhere. You can configure where the components are compiled in the `stencil.config.ts` file. You can find more options on how to distribute your components in the distribution documentation found here stenciljs.com/docs/distribution[22]

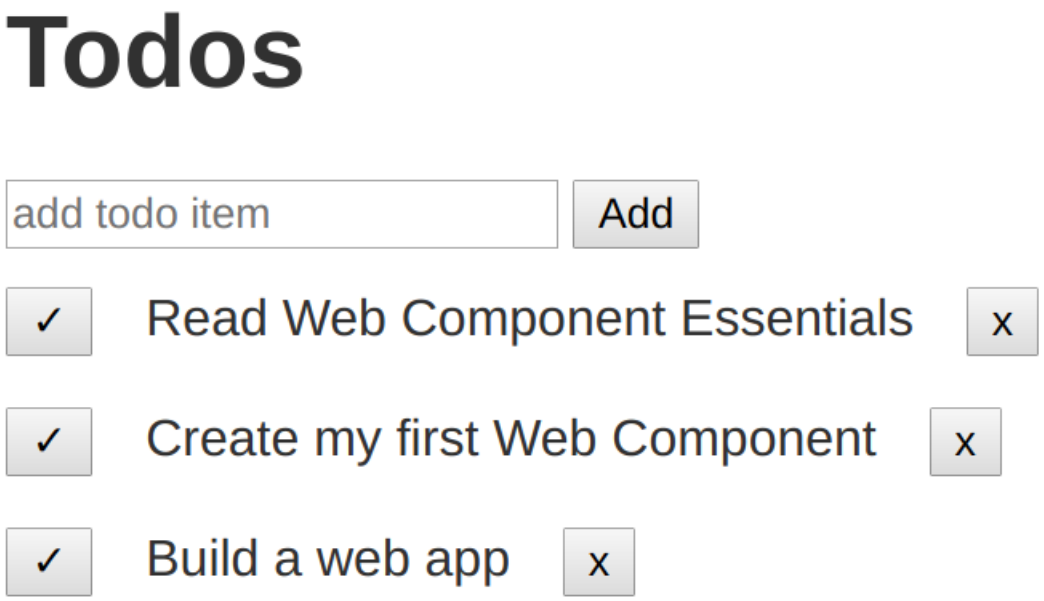
Stencil is an excellent choice for authoring Web Components as it gives you the convenience of a framework and all the benefits of Web Components.

[22] https://stenciljs.com/docs/distribution

# Chapter 13 - Building a Todo App with Lit

In this chapter, we are going to expand on how to build web apps with Web Components. For our example, we will make a simple todo app that can create, delete, and update a todo list using Lit and local storage. This example will also reinforce some of the best practices with component architecture.

Let's first take a look at what our todo app will look like and define the requirements.




**Todo App**

This todo app will save our todos into local storage on the user's device. We should be able to add, edit, and delete todos. When we double click and existing todo, we will switch the text with input to update the todo item.

# Implementation

In this example, we will use Lit for our Web Component templates. We will also use a simple Webpack build to compile our application with TypeScript.

Let's start with the `index.html` file.

```
<!DOCTYPE HTML>

<html>

<head>
  <title>Todo App with Lit</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="/index.css" rel="stylesheet" />
</head>

<body>
  <main>
    <x-app></x-app>
  </main>
  <script src="dist/bundle.js"></script>
</body>

</html>
```

The `index.html` will be pretty straightforward. Like most component is driven Web apps, we will have a single root entry point component. Many JavaScript frameworks follow similar conventions and name the component "root" or "app". For our entry component, we will call it `x-app`. Our components will be bundled into a single JavaScript file by Webpack and included at the bottom script tag.

The first thing we need to do is define our root component `x-app`. Let's take a look at the component in `index.ts`.

```
// todos.ts
import { LitElement, html } from 'lit';
import { customElement } from 'lit/decorators.js';
import './todos';

@customElement('x-app')
export class XApp extends LitElement {
  constructor() {
    super();
  }

  render() {
    return html`
      <header>
        <h1>Todos</h1>
      </header>
      <main>
        <x-todos></x-todos>
      </main>
      <footer>
        2018
      </footer>
    `;
  }
}
```

Our root component defines the basic structure of our application. If our app had a more elaborate header or navigation, it would likely go here. In the `main` of our app, we have a single component, `x-todos`. We import this component at the top of our file. The `x-todos` component will be responsible for managing the todos in our application.

Note we can simplify registering our Web Component by using the `customElement` decorator from `lit/decorators.js`. The `customElement` decorator takes in the tag name we would like to use for our component and automatically registers the element for us.

## Todos Data Service

Before we jump into the `x-todos` component, we need to make a service class that handles saving and retrieving our todo items from local storage. We will create a file called `todo.service.ts`; this file will be responsible for any changes to our todos and persisting those in local storage for us. Let's go ahead and take a look.

```
// todo.service.ts
import { Todo } from "./interfaces";

export class TodoService {
  getTodos() {
    // we use JSON.parse as local storage only allow key value string pairs
    const todos: Todo[] = JSON.parse(localStorage.getItem('todos'));
    return todos ? todos : [];
  }

  // TypeScript interfaces allow us to define what types we expect
  updateTodo(todo: Todo, index: number) {
    const todos =  this.getTodos();
    todos[index] = todo;
    return this.saveTodos(todos);
  }

  deleteTodo(todo: Todo) {
    const todos = this.getTodos();
    const updatedTodos = todos.filter(t => t.value !== todo.value);
    return this.saveTodos(updatedTodos);
  }

  createTodo(value: string) {
    const todos =  this.getTodos();
    const updatedTodos = [...todos, { completed: false, value: value }];
    return this.saveTodos(updatedTodos);
  }

  private saveTodos(updatedTodos: Todo[]) {
    localStorage.setItem('todos', JSON.stringify(updatedTodos));
    return updatedTodos;
  }
}
```

Our todo service abstracts all the saving and updating logic into a standalone class; that way, we can keep our components focused on just rendering the data. Best practice with Web apps is to have a clean separation of data logic and render logic between components. This separation typically makes components and business logic easier to unit test.

Our service was also using a TypeScript interface, `Todo`. This interface allows us to define a contract of how our todo data is shaped.

```ts
// interfaces.ts
export interface Todo {
  completed: boolean;
  value: string;
}
```

Now that we have the update logic defined for our todo app, let's take a look at the `x-todos` component.

## Todos List

The `x-todos` component will communicate directly with our todos service to save and update our data.

```ts
// todos.ts
import { LitElement, html } from 'lit';
import { customElement, property } from 'lit/decorators.js';
import { TodoService } from './todo.service';
import { Todo } from './interfaces';
import './todo-item';

@customElement('x-todos')
export class XTodos extends LitElement {
  @property()
  todos: Todo[] = [];

  todoService = new TodoService();

  constructor() {
    super();
    this.todos = this.todoService.getTodos();
  }
}
```

We have created our component, which has a `todos` property using the `@property` decorator from Lit. The `@property` decorator will allow lit to know to re-render the list if it changes. We also created an instance of our todo service to use within our component. In the constructor, we assign any existing todos from local storage to the `todos` property. Next, we can add the template of the component.

```
import { LitElement, html } from 'lit';
import { customElement, property } from 'lit/decorators.js';
import { TodoService } from './todo.service';
import { Todo } from './interfaces';
import './todo-item';

@customElement('x-todos')
export class XTodos extends LitElement {
  @property()
  todos: Todo[] = [];

  todoService = new TodoService();

  constructor() {
    super();
    this.todos = this.todoService.getTodos();
  }

  render() {
    return html`
      <form @submit="${(e: Event) => this.createTodo(e)}">
        <input placeholder="add todo item" aria-label="add todo item" />
        <button>Add</button>
      </form>

      <ul>
        ${this.todos.map((t, i) => html`
        <li>
          <x-todo-item
            .todo="${t}"
            @update="${(e: CustomEvent) => this.updateTodo(e.detail, i)}"
            @delete="${(e: CustomEvent) => this.deleteTodo(e.detail)}">
          </x-todo-item>
        </li>`)}
      </ul>
    `;
  }
}
```

The first part of our template defines an HTML form element for us to create our todos with. Using a form element, we can trigger submit events with a button element and use the enter key. On submit of the form, we call the `createTodo()` method, which we will see in a bit.

The second part of our template renders the list of todos. A single `x-todo-item` component renders

each todo. The `x-todo-item` contains the logic for when the user clicks done, delete, or updates the todo value. Whenever one of these user actions occurs, the `x-todo-item` will emit the `@update` or `@delete` events so our list component can adequately update the todo with the todo service.

```
<x-todo-item
  .todo="${t}"
  @update="${(e: CustomEvent) => this.updateTodo(e.detail, i)}"
  @delete="${(e: CustomEvent) => this.deleteTodo(e.detail)}">
</x-todo-item>
```

Our `x-todo-item` is a pure rendering component. It takes in a todo object by binding it to the `.todo` property. If anything changes with that object, it emits the updates to the parent list. This isolation makes the logic easier to maintain and makes the `x-todo-item` more generic and reusable. This component is the similar pattern we discussed in our earlier chapters with component architecture best practices.

Next are the methods that are called when the template events occur.

```
// todos.ts
import { LitElement, html } from 'lit';
import { customElement, property } from 'lit/decorators.js';
import { TodoService } from './todo.service';
import { Todo } from './interfaces';
import './todo-item';

@customElement('x-todos')
export class XTodos extends LitElement {
  @property()
  todos: Todo[] = [];
  todoService = new TodoService();

  constructor() {
    super();
    this.todos = this.todoService.getTodos();
  }

  render() {
    return html`
      <style>
        ul {
          margin: 0;
          padding: 0;
        }
```

```
26
27          li {
28            list-style: none;
29            margin: 0 0 16px 0;
30          }
31
32          input {
33            margin-bottom: 12px;
34          }
35        </style>
36
37        <form @submit="${(e: Event) => this.createTodo(e)}">
38          <input placeholder="add todo item" aria-label="add todo item" />
39          <button>Add</button>
40        </form>
41
42        <ul>
43          ${this.todos.map((t, i) => html`
44          <li>
45            <x-todo-item
46              .todo="${t}"
47              @update="${(e: CustomEvent) => this.updateTodo(e.detail, i)}"
48              @delete="${(e: CustomEvent) => this.deleteTodo(e.detail)}">
49            </x-todo-item>
50          </li>`)}
51        </ul>
52      `;
53    }
54
55    updateTodo(todo: Todo, index: number) {
56      this.todos = this.todoService.updateTodo(todo, index);
57    }
58
59    deleteTodo(todo: Todo) {
60      this.todos = this.todoService.deleteTodo(todo);
61    }
62
63    createTodo(e: any) {
64      // prevent the submit event from triggering a POST
65      e.preventDefault();
66
67      const input = e.target.querySelector('input');
68      const todo = input.value;
```

```
69
70      if (todo.length > 0) {
71        this.todos = this.todoService.createTodo(todo);
72        input.value = '';
73      }
74    }
75  }
```

As you can see, the list component methods, for the most part, pass the information along to the todo service. Using the todo service, the list is now, and the todo service could be used elsewhere in a more complex application. Next, let's look at the implementation of the `x-todo-item` component.

# Todo Item

The `x-todo-item` component is responsible for rendering a single todo item and emitting and updates to that particular item. First, we will look at the component definition and template.

```
1  // todo-item.ts
2  import { LitElement, html } from 'lit';
3  import { customElement, property } from 'lit/decorators.js';
4  import { Todo } from './interfaces';
5
6  @customElement('x-todo-item')
7  export class XTodoItem extends LitElement {
8    @property()
9    todo: Todo;
10
11   @property()
12   editing = false;
13
14   render() {
15     return html`
16       <button @click="${() => this.completeTodo()}" aria-label="mark done">
17         âœ“
18       </button>
19
20       ${this.editing ?
21         html`
22         <form @submit="${(e: any) => this.editTodo(e)}">
23           <input .value="${this.todo.value}" aria-label="create todo" placeholder="t\
24 odo" />
25         </form>
```

```
        `:
        html`
        <span class="${this.todo.completed ? 'completed' : ''}" @dblclick="${() => t\
his.toggleForm()}">
          ${this.todo.value}
        </span>
        `}

      <button @click="${() => this.deleteTodo()}" aria-label="delete todo">
        x
      </button>
    `;
  }
}
```

For this component, we will have two properties. One property will be for the todo object, and the second property is for tracking if the todo is being edited. Our template has two buttons, one to mark the todo complete and one to delete the todo. We have a ternary condition in the template that will show a form to edit the todo if the todo has been double-clicked. Note we can bind to CSS classes as well. When a todo is marked complete, we can add a CSS class to add a strikethrough property on the todo.

```
import { LitElement, html } from 'lit';
import { customElement, property } from 'lit/decorators.js';
import { Todo } from './interfaces';

@customElement('x-todo-item')
export class XTodoItem extends LitElement {
  @property()
  todo: Todo;

  @property()
  editing = false;

  render() {
    return html`
      <style>
        .completed {
          text-decoration: line-through;
        }

        span {
          padding: 0 12px;
```

```
        }

        form {
          display: inline-block;
        }
      </style>

      <button @click="${() => this.completeTodo()}" aria-label="mark done">
        âœ“
      </button>

      ${this.editing ?
        html`
        <form @submit="${(e: any) => this.editTodo(e)}">
          <input .value="${this.todo.value}" aria-label="create todo" placeholder="t\
odo" />
        </form>
        `:
        html`
        <span class="${this.todo.completed ? 'completed' : ''}" @dblclick="${() => t\
his.toggleForm()}">
          ${this.todo.value}
        </span>
        `}

      <button @click="${() => this.deleteTodo()}" aria-label="delete todo">
        x
      </button>
    `;
  }

  completeTodo() {
    this.todo = { ...this.todo, completed: !this.todo.completed };
    this.emitUpdate();
  }

  deleteTodo() {
    this.dispatchEvent(new CustomEvent('delete', { detail: this.todo }));
  }

  editTodo(e: any) {
    e.preventDefault();

```

```
65      const input = e.target.querySelector('input');
66      const todo = input.value;
67
68      if (todo.length > 0) {
69        this.todo.value = todo;
70        this.toggleForm();
71        this.emitUpdate();
72      }
73    }
74
75    toggleForm() {
76      this.editing = !this.editing;
77    }
78
79    private emitUpdate() {
80      this.dispatchEvent(new CustomEvent('update', { detail: this.todo }));
81    }
82  }
```

The methods on the todo item update our todo object. Note we create a new object when updating a todo as our data must be immutable for Lit to re-enter our templates. Whenever an update or a todo gets deleted, we emit a custom event to the todos list.

# Chapter 14 - Unit Testing Basics

Unit testing can significantly improve the quality of our codebase. Unit tests ensure our code is correctly running even after making changes. Because Web Components work natively in any browser, we can write unit tests using almost any JavaScript testing framework.

For our first unit test, we will use the dropdown component that we have used in previous chapters. The Open Web Component (@open-wc)[23] and modern web[24] projects provide a suite of helpful utilities for testing Web Components and Web Components built with Lit.

## Setting up the Test Runner

With our dropdown, we can write a few different test cases, but first, let's walk through the initial setup of our test suite. To run our tests, we will need a test runner. We will use `@web/test-runner` to run our tests natively in the browser. Next, we will need a testing library to run the assertions for our code. The `@open-wc` has some great default options and utilities out of the box.

```
{
  "name": "testing example",
  "version": "1.0.0",
  "description": "Basic unit testing example",
  "scripts": {
    "start": "web-dev-server --node-resolve --open",
    "test": "web-test-runner ./**/*.spec.ts --node-resolve",
  },
  "dependencies": {
    "lit": "^2.0.0"
  },
  "devDependencies": {
    "@open-wc/testing": "^2.5.33",
    "@web/dev-server": "^0.1.24",
    "@web/dev-server-esbuild": "^0.2.14",
    "@web/test-runner": "^0.13.18",
    "typescript": "^4.4.3"
  }
}
```

[23] https://open-wc.org/
[24] https://modern-web.dev/

Above is our basic `package.json` with our dependencies. The `@open-wc` package provides our testing utilities. The `@web/dev-server` and `@web/test-server` run both our localhost and test runners. And lastly, the `@web/dev-server-esbuild` and `typescript` packages will allow us to write not only our components in TypeScript but our unit tests as well.

Let's look at the `web-test-runner.config.mjs` at the root of the `/code` examples direcotry.

```
import { esbuildPlugin } from '@web/dev-server-esbuild';

export default {
  nodeResolve: true,
  rootDir: './',
  plugins: [esbuildPlugin({ ts: true, target: 'esnext' })]
};
```

This config file allows us to configure and add any plugins needed for running our unit tests. For example, since we are writing in Lit and TypeScript, we can use the `esBuildPlugin` to compile our TypeScript code for the unit tests.

## Unit Test Structure

We will start with the dropdown component to test what we created in our previous chapters.

```
import { LitElement, html, css } from 'lit';
import { property } from 'lit/decorators/property.js';
import { customElement } from 'lit/decorators/custom-element.js';

@customElement('x-dropdown')
export class XDropdown extends LitElement {
  @property({ type: Boolean }) open = false;
  @property({ type: String }) title = 'dropdown';

  render() {
    return html`
      <button @click=${() => this.toggle()}>${this.title}</button>
      ${this.open  ? html`<div><slot></slot></div>` : '' }
    `;
  }

  private toggle() {
    this.open = !this.open;
    this.dispatchEvent(new CustomEvent('openChange', { detail: this.open }));
```

```
  }
}
```

We can test for a few different behaviors but let's walk through the basic unit test setup.

```
import { expect, fixture, html } from '@open-wc/testing';
import { XDropdown } from './dropdown.element.js';
import './dropdown.element.js';

describe('x-dropdown', () => {
  it('creates component', async () => {
    const element = await fixture(html`<x-dropdown></x-dropdown> `);
    expect(element.tagName).to.equal('X-DROPDOWN')
  });
});
```

The `@open-wc/testing` package provides utilities for testing Lit-based Web Components. The unit tests are organized by using `describe` and `it` function blocks. The `describe` function defines what we are testing, and the `it` function runs our `expect` test for each particular behavior we want to test.

For our first example, we will create a basic test that checks to see if the component was created successfully. Since our component is built with Lit, the template renders asynchronously. We can use the `fixture` function to wait until the component has been rendered to get a reference to it. This ensures the component template has been rendered and is available for us to access within our unit test.

## First Unit Test

For our first test, we can check that when the `open` property is set to `false`, the dropdown is hidden, and `true` the dropdown is visible.

```
it('should open and close dropdown when open property is set', async () => {
  const element = await fixture<XDropdown>(html`<x-dropdown></x-dropdown> `);
  await element.updateComplete;
  expect(element.open).to.equal(false);
  expect(element.shadowRoot.querySelector('div')).to.equal(null);
});
```

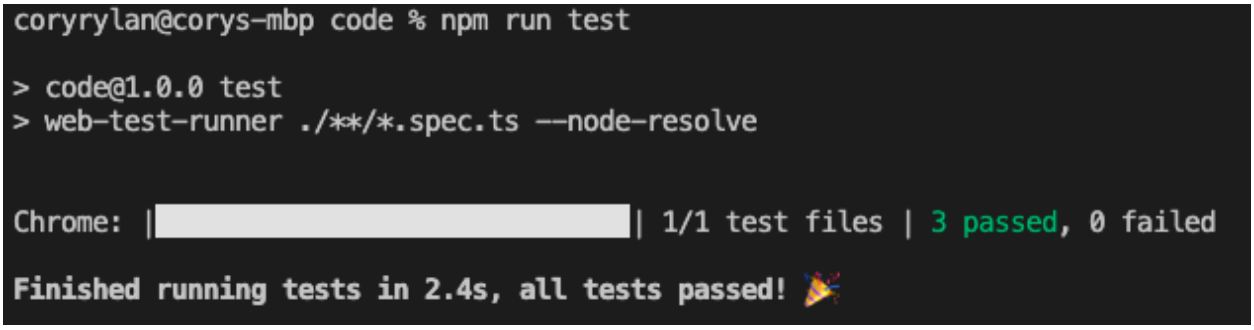
Using `await` and `fixture`, we can reference our component once it's completed rendering. We expect the default for `open` to be `false`. We then can query the `shadowRoot` of our component and ensure the `div` dropdown was not rendered. To run our tests, we can run the following command,

```
web-test-runner ./**/*.spec.ts --node-resolve
```

or as our npm script and run,

```
npm run test
```

If everything is installed and runs correctly, we should see the following output.




**Successfull Unit Test**

Now let's add another test to check when `open` is set to `true` our dropdown is shown as expected.

```
it('should open and close dropdown when open property is set', async () => {
  const element = await fixture<XDropdown>(html`<x-dropdown></x-dropdown> `);
  expect(element.open).to.equal(false);
  expect(element.shadowRoot.querySelector('div')).to.equal(null);

  element.open = true;
  await element.updateComplete;
  expect(element.open).to.equal(true);
  expect(element.shadowRoot.querySelector('div').tagName).to.equal('DIV');
});
```

Because we are setting the `open` property to `true`, we need to use the following line, `await element.updateComplete`. This ensures the test waits until Lit has finished updating the component template from our change.

If we rerun our tests, we should see that they all pass as expected. When starting the test command, you can run the tests in a watch mode by adding the `--watch` flag. To learn more about `@web/test-runner` check out the documentation at modern-web.dev[25]. Modern Web provides a fantastic suite of web development tools worth checking out even if you are not building with Web Components.

[25] https://modern-web.dev/

# Chapter 15 - Conclusion

Web Components are still a new technology but promise a lot of potential for the future of the Web. This book will be continually updated with new chapters and code examples. Please check back often for updates. Keep up to date with new content about Web Components and subscribe at webcomponent.dev[26].

[26] https://webcomponent.dev/

# Chapter 16 - Code Examples

Thank you for purchasing Web Component Essentials! Your download should have included a zip file with all code examples. If not, visit https://webcomponent.dev/purchased/code-examples.zip to download the latest available examples.