Web Components Tutorial for Beginners [2019]

JUNE 10, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

This tutorial teaches you how to build your first Web Components and how to use them in your applications. Before we get started, let's take a moment to learn more about Web Components in general: In recent years, Web Components, also called **Custom Elements**, have become a standard API for several browsers which allow developers to implement reusable components with only HTML, CSS and JavaScript. No React, Angular or Vue needed here. Instead, Custom Elements offer you encapsulation of all the structure (HTML), styling (CSS), and behavior (JavaScript) in one custom HTML element. For instance, imagine you could have a HTML dropdown component like the one in the following code snippet:

```
<my-dropdown
  label="Dropdown"
  option="option2"
  options='{ "option1": { "label": "Option 1" }, "option2": { "label": "Opti
```

```
></my-dropdown>
```

In this tutorial, we will implement this dropdown component step by step from scratch with Web Components. Afterward, you can continue using it across your application, make it an open source web component to install it somewhere else, or use a framework like React to build upon a solid foundation of Web Components for your React application.

WHY WEB COMPONENTS?

A personal story to illustrate how to benefit from Web Components: I picked up Web Components when a client of mine with many cross functional teams wanted to create a UI library based on a style guide. Two teams started to implement components based on the style guide, but each team used a different framework: React and Angular. Even though both implementations shared *kinda* the same structure (HTML) and style (CSS) from the style guide, the implementation of the behavior (e.g. opening/closing a dropdown, selecting an item in a dropdown) with JavaScript was up to each team to implement with their desired framework. In addition, if the style guide made mistakes with the style or structure of the components, each team fixed these mistakes individually without adapting the style guide afterward. Soonish both UI libraries diverged in their appearance and behavior.

Note: Independent from Web Components, this is a common flaw in style guides, if they are not used pro actively (e.g. living style guide) in code, but only as documentation on the side which gets outdated eventually.

Eventually both teams came together and discussed how to approach the problem. They asked me to look into Web Components to find out whether their problem could be solved with them. And indeed Web Components offered a compelling solution: Both teams could use implement common Web Components based on the style guide. Components like Dropdown, Button and Table would be implemented with only HTML, CSS, and JavaScript. Moreover, they weren't forced to use explicitly Web Components for their individual applications later on, but would be able to consume the components in their React or Angular applications. If the requirements of the style guide change, or a component needs to get fixed, both teams could collaborate on their shared Web Component UI library.

GETTING STARTED WITH WEB COMPONENTS

If you need a starter project for the following tutorial, you can clone this one from GitHub. You should look into the *dist/* and *src/* folders to make your adjustments from the tutorial along the way. The finished project from the tutorial can be found here on GitHub.

Let's get started with our first web component. We will not start to implement the dropdown component from the beginning, but rather a simple button component which is used later on in the dropdown component. Implementing a simple button component with a Web Component doesn't make much sense, because you could use a `<button>` element with some CSS, however, for the sake of learning about Web Components, we will start out with this button component. Thus, the following code block is sufficient to create a Web Component for an individual button with custom structure and style:

```
const template = document.createElement('template');

template.innerHTML = `
  <style>
    .container {
      padding: 8px;
    }

    button {
      display: block;
      overflow: hidden;
      position: relative;
      padding: 0 16px;
      font-size: 16px;
      font-weight: bold;
      text-overflow: ellipsis;
      white-space: nowrap;
      cursor: pointer;
      outline: none;

      width: 100%;
      height: 40px;

      box-sizing: border-box;
      border: 1px solid #a1a1a1;
      background: #ffffff;
      box-shadow: 0 2px 4px 0 rgba(0,0,0, 0.05), 0 2px 8px 0 rgba(161,161,16
      color: #363636;
      cursor: pointer;
    }
  </style>

  <div class="container">
    <button>Label</button>
  </div>
`;

class Button extends HTMLElement {
  constructor() {
    super();

    this._shadowRoot = this.attachShadow({ mode: 'open' });
```

```
      this._shadowRoot.appendChild(template.content.cloneNode(true));
    }
  }

  window.customElements.define('my-button', Button);
```

Let's go through everything step by step. The definition of your Custom Element (Web Component) happens with a JavaScript class that extends from HTMLElement which helps you to implement *any* custom HTML element. By extending from it, you will have access to various class methods - for instance, **lifecycle callbacks** (lifecycle methods) of the component - which help you to implement your Web Component. You will see later how we make use of these class methods.

In addition, Web Components are using Shadow DOM which shouldn't be mistaken for Virtual DOM (performance optimization). The Shadow DOM is used to encapsulate CSS, HTML, and JavaScript which ought to be hidden for the outside components/HTML that are using the Web Component. You can set a mode for your Shadow DOM, which is set to true in our case, to make the Shadow DOM kinda accessible to the outside world. Anyway, you can think of the Shadow DOM as its own subtree inside your custom element that encapsulates structure and style.

There is another statement in the constructor which appends a child to our Shadow DOM by cloning the declared template from above. Templates are usually used to make HTML reusable. However, templates also play a crucial role in Web Components for defining the structure and style of it. At the top of our custom element, we defined the structure and style with the help of such template which is used in the constructor of our custom element.

The last line of our code snippet defines the custom element as valid element for our HTML by defining it on the window. Whereas the first argument is the name of our reusable custom element as HTML -- which has to have a hyphen -- and the second argument the definition of our custom element including the rendered template. Afterward, we can use our new custom element somewhere in our HTML with `<my-button></my-button>`. Note that custom elements cannot/shouldn't be used as self closing tags.

HOW TO PASS ATTRIBUTES TO WEB COMPONENTS?

So far, our custom element isn't doing much except for having its own structure and style. We could have achieved the same thing by using a button element with some CSS. However, for the sake of learning about Web Components, let's continue with the custom button element. As for now, we cannot alter what's displayed by it. For instance, what about passing a label to it as HTML attribute:

```
<my-button label="Click Me"></my-button>
```

The rendered output would still show the internal custom element's template which uses a `Label` string. In order to make the custom element react to this new attribute, you can observe it, and do something with it by using class methods coming from the extended HTMLElement class:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

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

  attributeChangedCallback(name, oldVal, newVal) {
    this[name] = newVal;
  }
}
```

Every time the label attribute changes, the `attributeChangedCallback()` function gets called, because we defined the label as observable attribute in the `observedAttributes()` function. In our case, the callback function doesn't do much except for setting the label on our Web Component's class instance (here: `this.label = 'Click Me'`). However, the custom element still isn't rendering this label yet. In order to adjust the rendered output, you have to grab the actual HTML button and set its HTML:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');
  }

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

  attributeChangedCallback(name, oldVal, newVal) {
    this[name] = newVal;

    this.render();
  }
  render() {
    this.$button.innerHTML = this.label;
  }
}
```

Now, the initial label attribute is set within the button. In addition, the custom element will react to changes of the attribute as well. You can implement other attributes the same way. However, you will notice that non JavaScript primitives such as objects and arrays need to be passed as string in JSON format. We will see this later when implementing the dropdown component.

REFLECTING PROPERTIES TO ATTRIBUTES

So far, we have used **attributes to pass information to our Custom Element**. Every time an attribute changes, we set this attribute as property on our Web Component's instance in the callback function. Afterward, we do all necessary changes for the rendering imperatively. However, we can also use a get method to **reflect the attribute to a property**. Doing it this way, we make sure that we always get the latest value without assigning it in our callback function ourselves. Then, `this.label` always returns the recent attribute from our getter function:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');
  }

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

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

  attributeChangedCallback(name, oldVal, newVal) {
    this.render();
  }

  render() {
    this.$button.innerHTML = this.label;
  }
}
```

That's it for reflecting an attribute to a property. However, the other way around, you can also **pass information to an custom element with properties**. For instance, instead of rendering our button with an attribute `<my-button label="Click Me"></my-button>`, we can also set the information as property for the element. Usually this way is used when assigning information like objects and arrays to our element:

```
<my-button></my-button>

<script>
  const element = document.querySelector('my-button');
  element.label = 'Click Me';
</script>
```

Unfortunately our callback function for the changed attributes isn't called anymore when using a property instead of an attribute, because it only reacts for attribute changes doesn't handle properties. That's where a set method on our class comes neatly into play:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');
  }

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

  set label(value) {
    this.setAttribute('label', value);
  }

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

  attributeChangedCallback(name, oldVal, newVal) {
    this.render();
  }

  render() {
    this.$button.innerHTML = this.label;
  }
}
```

Now, since we *set the property* from the outside on our element, our custom element's setter method makes sure to **reflect the property to an attribute**, by setting the element's attribute to the reflected property value. Afterward, our attribute callback runs again, because the attribute has changed and thus we get the rendering mechanism back.

You can add console logs for each method of this class to understand the order on when each method happens. The whole reflection can also be witnessed in the DOM by opening the browser's developer tools: the attribute should appear on the element even though it is set as property.

Finally, after having getter and setter methods for our information in place, we can pass information as attributes and as properties to our custom element. The whole process is called **reflecting properties to attributes** and vice versa.

HOW TO PASS A FUNCTION TO A WEB COMPONENT?

Last but not least, we need to make our custom element work when clicking it. First, the custom element could register an event listener to react on an user's interaction. For instance, we can take the button and add an event listener to it:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');

    this.$button.addEventListener('click', () => {
      // do something
    });
  }

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

  set label(value) {
    this.setAttribute('label', value);
  }

  static get observedAttributes() {
    return ['label'];
  }
  attributeChangedCallback(name, oldVal, newVal) {
    this.render();
  }

  render() {
    this.$button.innerHTML = this.label;
  }
}
```

Note: It would be possible to add this listener simply from the outside on the element -- without bothering about it in the custom element -- however, defining it inside of the custom element gives you more control of what should be passed to the listener that is registered on the outside.

What's missing is a callback function given from the outside that can be called within this listener. There are various ways to solve this task. First, we could **pass the function as attribute**. However, since we have learned that passing non primitives to HTML elements is cumbersome, we would like to avoid this case. Second, we could **pass the function as property**. Let's see how this would look like when using our custom element:

```
<my-button label="Click Me"></my-button>

<script>
  document.querySelector('my-button').onClick = value =>
    console.log(value);
</script>
```

We just defined an `onClick` handler as function to our element. Next, we could call this function property in our custom element's listener:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');

    this.$button.addEventListener('click', () => {
      this.onClick('Hello from within the Custom Element');
    });
  }

  ...

}
```

See how you are in charge what's passed to the callback function. If you wouldn't have the listener inside the custom element, you would simply receive the event. Try it yourself. Now, even though this works as expected, I would rather use the built-in event system provided by the DOM API. Therefore, let's register an event listener from the outside instead without assigning the function as property to the element:

```
<my-button label="Click Me"></my-button>

<script>
  document
    .querySelector('my-button')
    .addEventListener('click', value => console.log(value));
</script>
```

The output when clicking the button is identical to the previous one, but this time with an event listener for the click interaction. That way, the custom element is still able to send information to the outside world by using the click event, because our message from the inner workings of the custom element is still send and can be seen in the logging of the browser. Doing it this way, you can also leave out the definition of the event listener within the custom element, if no special behavior is needed, as mentioned before.

There is one caveat by leaving everything this way though: We can only use the built-in events for our custom element. However, if you would later on use your Web Component in a different environment (e.g. React), you may want to offer custom events (e.g. onClick) as API for your component as well. Of course, we could also map manually the `click` event from the custom element to the `onClick` function from our framework, but it would be less a hassle if we could simply use the same naming convention there. Let's see how we can take our previous implementation one step further to support custom events too:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$button = this._shadowRoot.querySelector('button');

    this.$button.addEventListener('click', () => {
      this.dispatchEvent(
        new CustomEvent('onClick', {
          detail: 'Hello from within the Custom Element',
        })
      );
    });
  }
```

```
    ...

  }
```

Now we are exposing a custom event as API to the outside called `onClick` whereas the information is passed through the optional `detail` property. Next, we can listen to this new custom event instead:

```
<my-button label="Click Me"></my-button>

<script>
  document
    .querySelector('my-button')
    .addEventListener('onClick', value => console.log(value));
</script>
```

This last refactoring from a built-in event to a custom event is optional though. It's only there to show you the possibilities of custom events and perhaps to give you an easier time for using Web Components later in your favorite framework if that's what you are looking for.

WEB COMPONENTS LIFECYCLE CALLBACKS

We have almost finished our custom button. Before we can continue with the custom dropdown element -- which will use our custom button element -- let's add one last finishing touch. At the moment, the button defines an inner container element with a padding. That's useful for using these custom buttons side by side with a natural margin to each other. However, when using the button in another context, for instance a dropdown component, you may want to remove this padding from the container. Therefore, you can use one of the lifecycle callbacks of a Web Component called `connectedCallback`:

```
class Button extends HTMLElement {
  constructor() {
    super();

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

    this.$container = this._shadowRoot.querySelector('.container');
    this.$button = this._shadowRoot.querySelector('button');

    ...
  }

  connectedCallback() {
    if (this.hasAttribute('as-atom')) {
      this.$container.style.padding = '0px';
    }
  }

  ...

}
```

In our case, if there is an existent attribute called `as-atom` set on the element, it will reset our button container's padding to zero. That, by the way, is how you can create a great UI library with atomic design principles in mind whereas the custom button element is an atom and the custom dropdown element a molecule. Maybe both end up with another element later in a greater organism. Now our button can be used without padding in our dropdown element the following way: `<my-button as-atom></my-button>`. The label of the button will be later set by using a property.

But what about the lifecycle callback? The `connectedCallback` runs once the Web Component got appended to the DOM. That's why you can do all the things that need to be done once the component gets rendered. There exists an equivalent lifecycle callback for when the component gets removed called `disconnectedCallback`. Also you have already used a lifecycle method in your custom element before called `attributeChangedCallback` to react on attribute changes. There are various lifecycle callbacks available for Web Components, so make sure to check them out in detail.

WEB COMPONENTS WITHIN WEB COMPONENT

Last but not least, we want to use our finished Button Web Component within another Web Component. Therefore, we will implement a custom dropdown element which should be used the following way:

```
<my-dropdown
  label="Dropdown"
  option="option2"
  options='{ "option1": { "label": "Option 1" }, "option2": { "label": "Opti
></my-dropdown>
```

Note that the options -- which are an object -- are passed as JSON formatted attribute to the custom element. As we have learned, it would be more convenient to pass objects and arrays as properties instead:

```
<my-dropdown
  label="Dropdown"
  option="option2"
></my-dropdown>

<script>
  document.querySelector('my-dropdown').options = {
    option1: { label: 'Option 1' },
    option2: { label: 'Option 2' },
  };
</script>
```

Let's dive into implementation of the custom dropdown element. We will start with a straightforward foundation that defines our structure, style, and boilerplate code for the class that defines our Web Component. The latter is used for setting the mode of the Shadow DOM, attaching the template to our Custom Element, defining getter and setter methods for our attributes/properties, observing our attribute changes and reacting to them:

```
const template = document.createElement('template');

template.innerHTML = `
  <style>
    :host {
      font-family: sans-serif;
    }

    .dropdown {
      padding: 3px 8px 8px;
    }

    .label {
      display: block;
      margin-bottom: 5px;
      color: #000000;
      font-size: 16px;
      font-weight: normal;
      line-height: 16px;
    }

    .dropdown-list-container {
      position: relative;
    }

    .dropdown-list {
      position: absolute;
      width: 100%;
      display: none;
      max-height: 192px;
      overflow-y: auto;
      margin: 4px 0 0;
      padding: 0;
      background-color: #ffffff;
      border: 1px solid #a1a1a1;
      box-shadow: 0 2px 4px 0 rgba(0,0,0, 0.05), 0 2px 8px 0 rgba(161,161,16
      list-style: none;
    }

    .dropdown-list li {
      display: flex;
      align-items: center;
      margin: 4px 0;
      padding: 0 7px;
      font-size: 16px;
      height: 40px;
      cursor: pointer;
    }
  </style>

  <div class="dropdown">
    <span class="label">Label</span>

    <my-button as-atom>Content</my-button>

    <div class="dropdown-list-container">
      <ul class="dropdown-list"></ul>
    </div>
  </div>
`;

class Dropdown extends HTMLElement {
  constructor() {
    super();

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

  static get observedAttributes() {
    return ['label', 'option', 'options'];
  }

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

  set label(value) {
    this.setAttribute('label', value);
  }

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

  set option(value) {
    this.setAttribute('option', value);
  }

  get options() {
    return JSON.parse(this.getAttribute('options'));
  }
  set options(value) {
    this.setAttribute('options', JSON.stringify(value));
  }

  static get observedAttributes() {
    return ['label', 'option', 'options'];
  }

  attributeChangedCallback(name, oldVal, newVal) {
    this.render();
  }

  render() {

  }
}

window.customElements.define('my-dropdown', Dropdown);
```

There are several things to note here: First, in our style we can set a *global style* for our custom element with the `:host` selector. Second, the template uses our custom button element, but doesn't give it a label attribute yet. And third, there are getters and setters for each attribute/property, however, the getter and setter for the `options` attribute/property reflection are parsing the object from/to JSON.

Note: Except for all the mentioned things, you may also notice lots of boilerplate for all of our getter and setter methods for the property/attribute reflection. Also the lifecycle callback for our attributes looks repetitive and the constructor is the same as the one in our custom button element. You may learn later that there exist various lightweight libraries (e.g. LitElement with LitHTML) to be used on top of Web Components to remove this kind of repetitiveness for us.

So far, all the passed properties and attributes are not used yet. We are only reacting to them with an empty render method. Let's make use of them by assigning them to the dropdown and button elements:

```
class Dropdown extends HTMLElement {
  constructor() {
    super();

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

    this.$label = this._sR.querySelector('.label');
    this.$button = this._sR.querySelector('my-button');
  }

  ...
  static get observedAttributes() {
    return ['label', 'option', 'options'];
  }

  attributeChangedCallback(name, oldVal, newVal) {
    this.render();
  }

  render() {
    this.$label.innerHTML = this.label;

    this.$button.setAttribute('label', 'Select Option');
  }
}

window.customElements.define('my-dropdown', Dropdown);
```

Whereas the dropdown gets its label from the outside as attribute to be set as inner HTML, the button sets an arbitrary label as attribute for now. We will set this label later based on the selected option from the dropdown. Also, we can make use of the options to render the actual selectable items for our dropdown:

```
class Dropdown extends HTMLElement {
  constructor() {
    super();

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

    this.$label = this._sR.querySelector('.label');
    this.$button = this._sR.querySelector('my-button');
    this.$dropdownList = this._sR.querySelector('.dropdown-list');
  }

  ...

  render() {
    this.$label.innerHTML = this.label;

    this.$button.setAttribute('label', 'Select Option');

    this.$dropdownList.innerHTML = '';

    Object.keys(this.options || {}).forEach(key => {
      let option = this.options[key];
      let $option = document.createElement('li');
      $option.innerHTML = option.label;

      this.$dropdownList.appendChild($option);
    });
  }
}
```

```
window.customElements.define('my-dropdown', Dropdown);
```

In this case, on every render we wipe the inner HTML of our dropdown list, because the options could have been changed. Then, we dynamically create a list element for each `option` in our `options` object and append it to our list element with the `option` property's `label`. If the `properties` are undefined, we use a default empty object to avoid running into an exception here, because there exists a race condition between incoming attributes and properties. However, even though the list gets rendered, our style defines the CSS `display` property as none. That's why we cannot see the list yet, but we will see it in the next step after we added some more JavaScript for the custom element's behavior.

BEHAVIOR OF WEB COMPONENTS WITH JAVASCRIPT

So far, we have mainly structured and styled our custom elements. We also reacted on changed attributes, but didn't do much in the rendering step yet. Now we are going to add behavior with more JavaScript to our Web Component. Only this way it is really different from a simple HTML element styled with CSS. You will see how all the behavior will be encapsulated in the custom dropdown element without any doings from the outside.

Let's start by opening and closing the dropdown with our button element which should make our dropdown list visible. First, define a new style for rendering the dropdown list with an open class. Remember that we have used `display: none;` for our dropdown list as default styling before.

```
const template = document.createElement('template');

template.innerHTML = `
  <style>
    :host {
      font-family: sans-serif;
    }

    ...

    .dropdown.open .dropdown-list {
      display: flex;
      flex-direction: column;
    }

    ...
  </style>

  ...
```

```
`;
```

In the next step, we define a class method which toggles the internal state of our custom element. Also, when this class method is called, the new class is added or removed to our dropdown element based on the new `open` state.

```
class Dropdown extends HTMLElement {
  constructor() {
    super();

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

    this.open = false;

    this.$label = this._sR.querySelector('.label');
    this.$button = this._sR.querySelector('my-button');
    this.$dropdown = this._sR.querySelector('.dropdown');
    this.$dropdownList = this._sR.querySelector('.dropdown-list');
  }

  toggleOpen(event) {
    this.open = !this.open;

    this.open
      ? this.$dropdown.classList.add('open')
      : this.$dropdown.classList.remove('open');
  }

  ...
}
```

Last but not least, we need to add an event listener for our custom button element's event to toggle the dropdown's internal state from open to close and vice versa. Don't forget to bind `this` to our new class method when using it, because otherwise it wouldn't have access to `this` for setting the new internal state or accessing the assigned `$dropdown` element.

```
class Dropdown extends HTMLElement {
  constructor() {
    super();

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

    this.open = false;

    this.$label = this._sR.querySelector('.label');
    this.$button = this._sR.querySelector('my-button');
    this.$dropdown = this._sR.querySelector('.dropdown');
    this.$dropdownList = this._sR.querySelector('.dropdown-list');

    this.$button.addEventListener(
      'onClick',
      this.toggleOpen.bind(this)
    );
  }

  toggleOpen(event) {
    this.open = !this.open;

    this.open
      ? this.$dropdown.classList.add('open')
      : this.$dropdown.classList.remove('open');
  }

  ...
}
```

Try your Web Component yourself now. It should be possible to open and close the custom dropdown element by clicking our custom button. That's our first real internal behavior of our custom element which would have been implemented in a framework like React or Angular otherwise. Now your framework can simply use this Web Component and expect this behavior from it. Let's continue with selecting one of the items from the opened list when clicking it:

```
class Dropdown extends HTMLElement {

  ...

  render() {
    ...

    Object.keys(this.options || {}).forEach(key => {
      let option = this.options[key];
      let $option = document.createElement('li');
      $option.innerHTML = option.label;

      $option.addEventListener('click', () => {
        this.option = key;

        this.toggleOpen();

        this.render();
      });

      this.$dropdownList.appendChild($option);
    });
  }
}
```

Each rendered option in the list gets an event listener for the click event. When clicking the option, the option is set as property, the dropdown toggles to `close`, and the component renders again. However, in order to see what's happening, let's visualize the selected option item in the dropdown list:

```
const template = document.createElement('template');

template.innerHTML = `
  <style>
    ...

    .dropdown-list li.selected {
      font-weight: 600;
    }
  </style>

  <div class="dropdown">
    <span class="label">Label</span>

    <my-button as-atom>Content</my-button>

    <div class="dropdown-list-container">
      <ul class="dropdown-list"></ul>
    </div>
  </div>
`;
```

Next we can set this new class in our render method whenever the option property matches the option from the list. With this new styling in place, and setting the styling dynamically on one of our options from the dropdown list, we can see that the feature actually works:

```
class Dropdown extends HTMLElement {

  ...

  render() {
    ...

    Object.keys(this.options || {}).forEach(key => {
      let option = this.options[key];
      let $option = document.createElement('li');
      $option.innerHTML = option.label;

      if (this.option && this.option === key) {
        $option.classList.add('selected');
      }

      $option.addEventListener('click', () => {
        this.option = key;
```

```
          this.toggleOpen();

          this.render();
        });

        this.$dropdownList.appendChild($option);
      });
    }
  }
```

Let's show the current selected option in our custom button element instead of setting an arbitrary value:

```
class Dropdown extends HTMLElement {

  ...

  render() {
    this.$label.innerHTML = this.label;

    if (this.options) {
      this.$button.setAttribute(
        'label',
        this.options[this.option].label
      );
    }

    this.$dropdownList.innerHTML = '';

    Object.keys(this.options || {}).forEach(key => {
      ...
    });
  }
}
```

Our internal behavior for the custom dropdown element works. We are able to open and close it and we are able to set a new option by selecting one from the dropdown list. One crucial thing is missing: We need to offer again an API (e.g. custom event) to the outside world to notify them about a changed option. Therefore, dispatch a custom event for each list item click, but give each custom event a key to identify which one of the items got clicked:

```
class Dropdown extends HTMLElement {

  ...

  render() {
    ...

    Object.keys(this.options || {}).forEach(key => {
        let option = this.options[key];
        let $option = document.createElement('li');
        $option.innerHTML = option.label;

        if (this.option && this.option === key) {
          $option.classList.add('selected');
        }

        $option.addEventListener('click', () => {
          this.option = key;

          this.toggleOpen();

          this.dispatchEvent(
            new CustomEvent('onChange', { detail: key })
          );

          this.render();
        });

        this.$dropdownList.appendChild($option);
      });
    }
  }
```

Last, when using the dropdown as Web Component, you can add an event listener for the custom event to get notified about changes:

```
<my-dropdown label="Dropdown" option="option2"></my-dropdown>

<script>
  document.querySelector('my-dropdown').options = {
    option1: { label: 'Option 1' },
    option2: { label: 'Option 2' },
  };

  document
    .querySelector('my-dropdown')
    .addEventListener('onChange', event => console.log(event.detail));
</script>
```

That's it. You have create a fully encapsulated dropdown component as Web Component with its own structure, style and behavior. The latter is the crucial part for a Web Component, because otherwise you could have simply used a HTML element with some CSS as style. Now, you have also the behvaior encapsulated in your new custom HTML element. Congratulations!

. . .

The implementation of the dropdown and button element as Web Components can be found in this GitHub project with a few helpful extensions. As I said before, the custom button element is a bit unessential for the dropdown component, because it doesn't implement any special behavior. You could have used a normal HTML button element with CSS styling. However, the custom button element has helped us to grasp the concept of Web Components with a simple example. That's why I think it was a good thought to start with the button component which is used later in the dropdown component. If you want to continue to use your Web Components in React, check out this neat React hook or this Web Components for React tutorial. In the end, I hope you have learned a lot from this Web Components tutorial. Leave a comment if you have feedback or simply liked it :-)

Show Comments

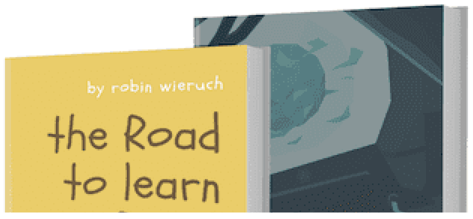
KEEP READING ABOUT REACT

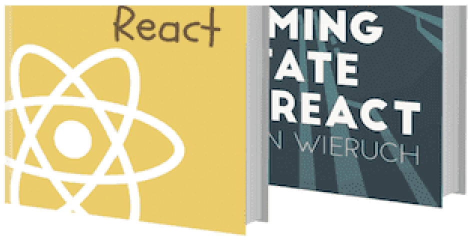
HOW TO USE WEB COMPONENTS IN REACT

In this tutorial, you will learn how to use Web Components, alias Custom Elements, in React . If you want to get started to build your own Web Components before, check out this tutorial: Web...

A FIREBASE IN REACT TUTORIAL FOR BEGINNERS [2019]

Interested in reading this tutorial as one of many chapters in my advanced React with Firebase book? Checkout the entire The Road to Firebase book that teaches you to create business web...




React
MING
ATE
REACT
N WIERUCH

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms