The Hidden Power of Custom States For Web Components

Danny Moerkerke on 2022-10-11

A crucial step in the evolution of Custom Elements

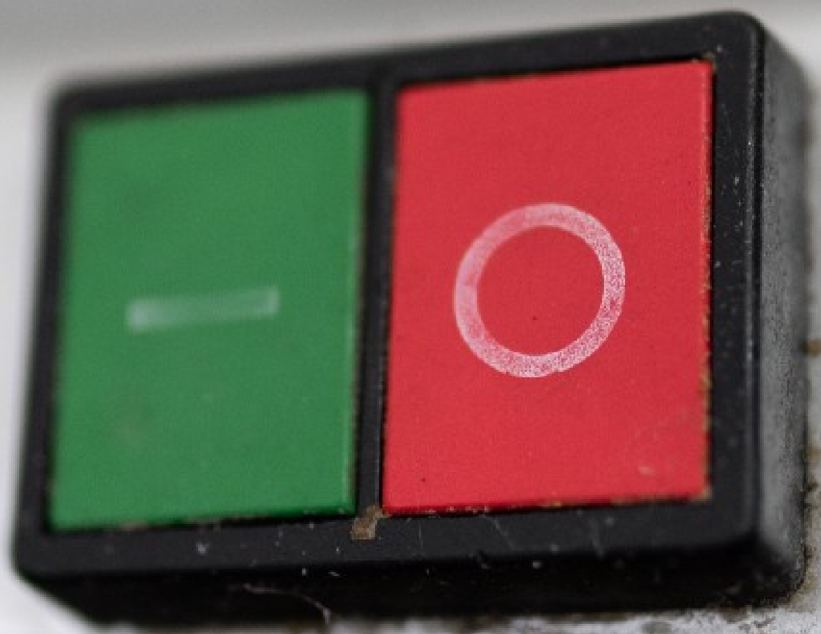


Photo by [Justus Menke](#) on [Unsplash](#)

In my previous articles “[Web Components Can Now Be Native Form Elements](#)” and “[Native Form Validation Of Web Components](#)”, I wrote about the `ElementInternals` property that enables Custom Elements to be associated with a form.

This interface also enables developers to associate custom states with Custom Elements and style them based on these states.

The `states` property of `ElementInternals` returns a `CustomStateSet` that stores a list of possible states for a Custom Element to be in, and allows states to be added and removed from the set.

Each state in the set is represented by a string that has the same form as a CSS Custom Property, namely`--mystate`.

These states can then be accessed from CSS with the custom state pseudo-class in the same way that built-in states can be accessed.

For example, a checkbox that is checked can be accessed from CSS using the built-in `:checked` pseudo-class:

```
input[type=”checkbox”]:checked {
 outline: solid green;
}
```

Another example is a disabled `button` that can be accessed from CSS using the `:disabled` pseudo-class:

```
button:disabled {
 cursor: not-allowed;
}
```

In the same way, an element containing the custom state `--mystate` can be accessed from CSS like this:

```
my-element: - mystate {
 color: red;
}
```

A use case for Custom States

Custom states unlock a powerful feature.

They enable Web Components to be styled based on internal states without having to add attributes or classes to the component to reflect these states, so they stay fully internal.

For example, let’s say you have a `<video-player>` component that shows a play button to play a video.

When the play button is clicked and the video starts playing, you want this play button to be hidden and a pause button to be shown.

Then, when the pause button is clicked, it will be hidden and the play button will be shown again.

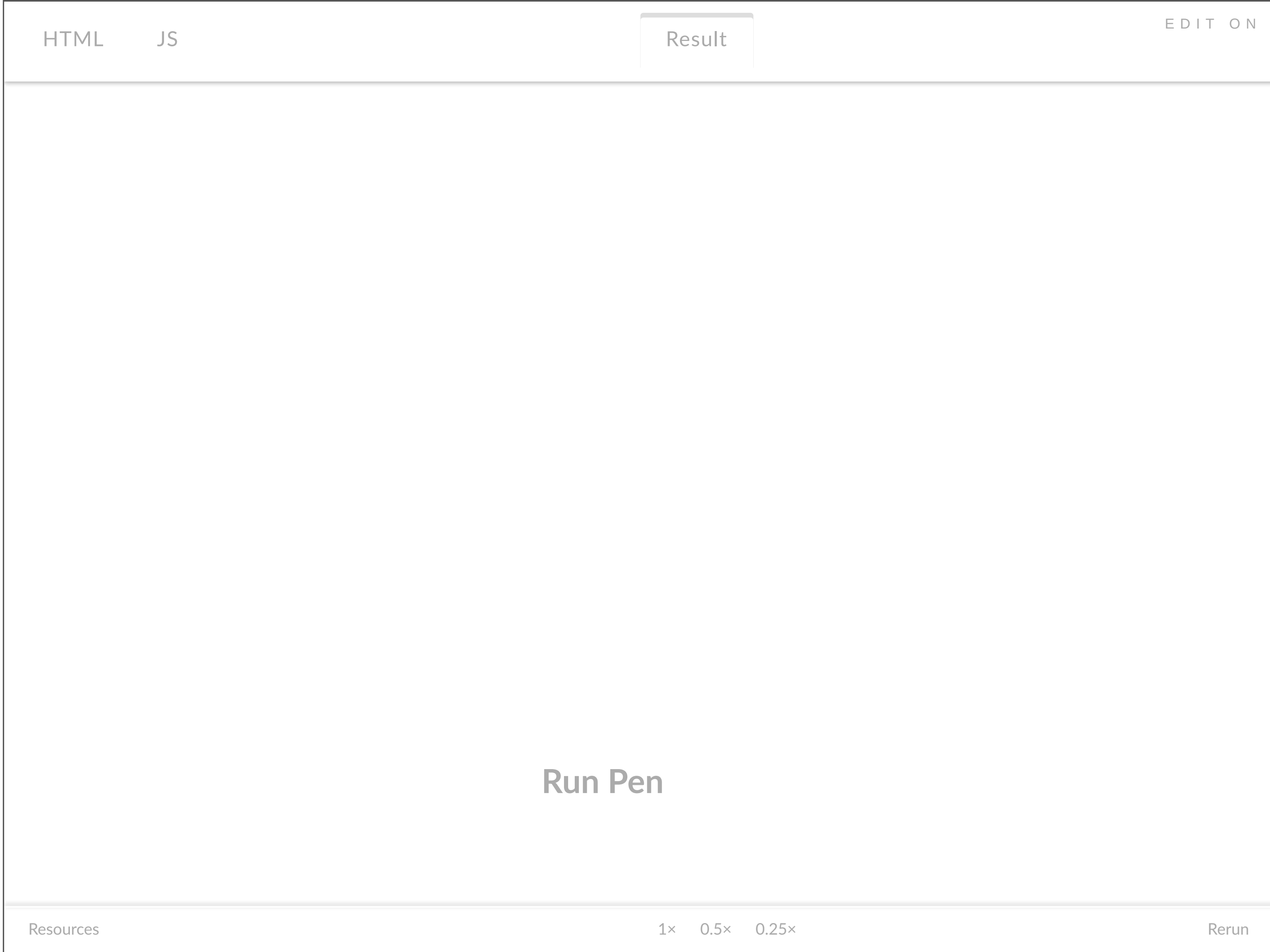
A simple way to do this is to introduce a `playing` property and reflect it to a `playing` attribute and use the `:host` pseudo-class to show and hide the buttons:

```
class VideoPlayer extends HTMLElement {

  constructor() {
    super();

    const shadowRoot = this.attachShadow({mode: 'open'});

    shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          width: 300px;
          height: 300px;
          border: 2px solid red;
          display: flex;
          justify-content: center;
```

```
          align-items: center;
          background-color: transparent;
        }

        #pause {
          display: none;
        }

        :host([playing]) #play {
          display: none;
        }

        :host([playing]) #pause {
          display: block;
        }
      </style>

      <button id="play" type="button">Play</button>
      <button id="pause" type="button">Pause</button
    `;
  }

  connectedCallback() {
    const playButton = this.shadowRoot.querySelector('#play');
    const pauseButton = this.shadowRoot.querySelector('#pause');

    playButton.addEventListener('click', () => {
      this.playing = true;
    });

    pauseButton.addEventListener('click', () => {
      this.playing = false;
    });
  }

  get playing() {
    return this.hasAttribute('playing');
  }

  set playing(isPlaying) {
    if(isPlaying) {
      this.setAttribute('playing', '');
    }
    else {
      this.removeAttribute('playing');
    }
  }
}
```

By default, the play button will be shown. A setter has been defined for the `playing` property that either sets or removes the `playing` attribute and the CSS rules take care of showing and hiding the buttons using the `:host` pseudo-class.

Below is a working example:



While this works fine, there is a potential problem with this implementation.

Exposing internal properties as attributes like this may not always be desirable and breaks encapsulation.

In this case, exposing a `playing` property may not be a bad idea but it does give users the ability to manually set the component in a `playing` state by just adding the attribute, but it won't actually start playing the video.

Exposing this property may even raise the expectation that the video can be played by just adding the `playing` attribute.

In fact, adding an attribute to put a Web Component in a certain state doesn't really put it in that state because it doesn't set the corresponding *property:* just adding the `playing` attribute does not set the `playing` property to `true`.

While in this case, it may not cause real harm, there will always be cases where exposing internal properties is not a good idea.

This is a perfect use case for custom states: no properties will be exposed but the component can still be styled using CSS based on these states.

Adding and removing custom states

As mentioned, all custom states are stored in a `CustomStateSet` object that is stored in the `states` property of the `ElementInternals` interface.

It has the methods `add` and `delete` to add and remove states and the `has` method to check if the element has a certain state.

Other notable methods are `clear` to clear all states and `forEach` to iterate over all states of an element:

```
// attach the internals
this.internals = this.attachInternals();

// add states
this.internals.states.add(' - foo');
this.internals.states.add(' - bar');

// iterate over states
this.internals.states.forEach(state => {
 console.log(state); // foo bar
});

// remove states
this.internals.states.delete(' - bar');

// check for existence of states
this.internals.states.has(' - foo'); // true
this.internals.states.has(' - bar'); // false
```

When you try to add a state that doesn't start with `--` an error will be thrown:

```
this.internals = this.attachInternals();
this.internals.states.add('foo'); // error, does not start with '--'
```

To make the previous example work with custom states, the getter and setter for the `playing` property are changed to work with the states:

```
get playing() {
  return this.internals.states.has('--playing');
}

set playing(isPlaying) {
  if(isPlaying) {
    this.internals.states.add('--playing');
  }
  else {
    this.internals.states.delete('--playing');
  }
}
```

and the `:host()` pseudo-class now takes the `--playing` selector instead of `[playing]`:

```
host(:--playing) #play {
  display: none;
}

:host(:--playing) #pause {
  display: block;
}
```

While this makes sure no internal properties are exposed as attributes, it's still possible for consumers to access `states` through the `internals` property and add or remove states by calling the `add` and `delete` methods:

```
const player = document.querySelector('video-player');
player.internals.states.add('--playing');
```

Even worse, consumers can just call the setter of `playing` to change the internal state.

You can fix this by making both the getter and setter and the `internals` property private by prefixing them with `#`:

```
// internals is now private
this.#internals = this.attachInternals();

get #playing() {
  return this.#internals.states.has('--playing');
}

set #playing(isPlaying) {
  if(isPlaying) {
    this.#internals.states.add(' - playing');
```

```
  }
  else {
    this.#internals.states.delete(' - playing');
  }
}
```

It may feel counter-intuitive to write a getter and setter pair for a private property but this actually works.

Even though `playing` has a getter and setter defined, it is still private and only accessible from within the class.

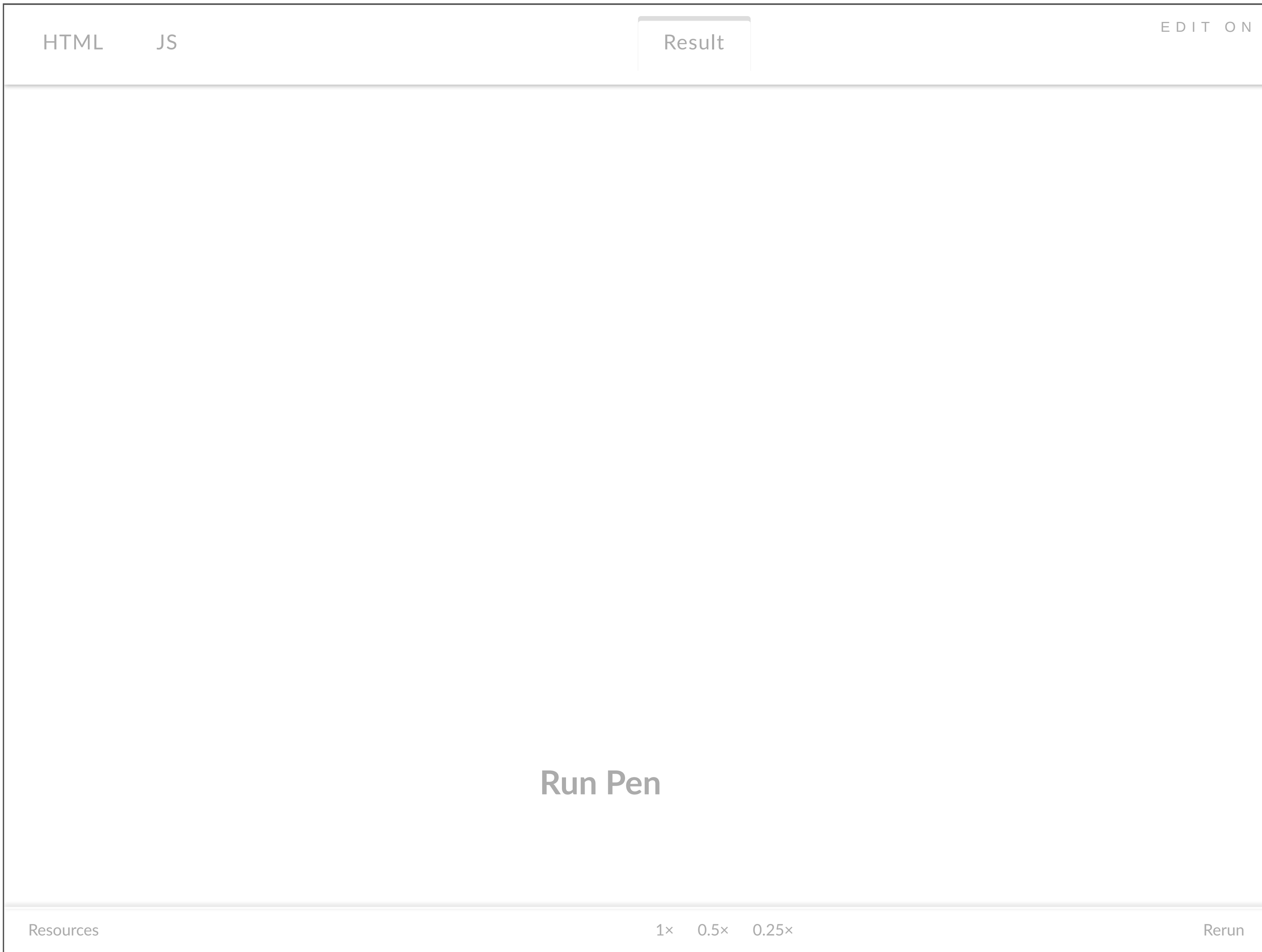
Assigning a value to it will call the setter and reading its value will call the getter.

Here's the full code:

```
class VideoPlayer extends HTMLElement {
  #internals;  // class field needed for private property

  constructor() {
    super();

    const shadowRoot = this.attachShadow({mode: 'open'});

    this.#internals = this.attachInternals();

    shadowRoot.innerHTML = `
      <style>
        :host {
          width: 300px;
          height: 300px;
          border: 2px solid red;
          display: flex;
          justify-content: center;
          align-items: center;
          background-color: transparent;
        }

        #pause {
          display: none;
        }

        :host(:--playing) #play {
          display: none;
        }

        :host(:--playing) #pause {
          display: block;
        }
      </style>

      <button id="play" type="button">Play</button>
      <button id="pause" type="button">Pause</button
    `;
  }

  connectedCallback() {
    const playButton = this.shadowRoot.querySelector('#play');
    const pauseButton = this.shadowRoot.querySelector('#pause');


    playButton.addEventListener('click', () => {
      this.#playing = true;
    });

    pauseButton.addEventListener('click', () => {
      this.#playing = false;
    });
  }

  get #playing() {
    return this.#internals.states.has('--playing');
  }
```

```
  set #playing(isPlaying) {
    if(isPlaying) {
      this.#internals.states.add('--playing')
    }
    else {
      this.#internals.states.delete('--playing');
    }
  }
}
```

And here's a working example that, at the time of writing only works in Chrome 90+:



These examples show how a Custom Element can be styled based on its custom states from *inside* the component using the `:host` pseudo-class.

A Custom Element can also be styled from the *outside* based on custom states.

This styling has the same form as styling components based on built-in states like `:checked` and `:hover`:

```
video-player: - playing {
  border: 1px solid red;
}
```

When styling based on a custom state is defined for the same CSS property from the inside and the outside, the styling defined on the outside takes precedence.

In the following example, the component will get a green border when it is in the `--playing` custom state.

The blue border defined inside the component for the `--playing` state will be overwritten:

```
// styling defined outside the component
// this will have precedence so the component will get a green border
video-player: - playing {
  border: 2px solid green;
}

// styling defined inside the component
// this will be overwritten by the styling defined outside of it
:host(: - playing) {
  border: 2px solid blue;
}
```

Conclusion

Custom States are a crucial step in the evolution of Web Components.

They enable styling of components without having to add classes or attributes which allows the state of a component to remain read-only so it can never be manipulated from the outside.

Support for `ElementInternals` has not yet landed in Safari but has already been partially implemented in Safari Tech Preview.

Follow me on [Twitter] where I write about what the modern web can do, PWAs, and Web Components.