

[Accessibility](#)[UX](#)[CSS](#)[JavaScript](#)[Performance](#)[Design](#)[Figur](#)[Nathan Smith](#) ([↗ /author/nathan-smith/](#)) / APR 28, 2022 / [12](#) ([↗ #comments-cta-modal-l](#))


CTA Modal: How To Build A Web Component

QUICK SUMMARY ↗ In this article, Nathan Smith explains how to create modal dialog windows with rich interaction that will only require authoring HTML in order to be used. They are based on Web Components that are currently supported by every major browser.

I have a confession to make — I am not overly fond of modal dialogs (or just “modals” for short). “Hate” would be too strong a word to use, but let’s say that nothing is more of a turnoff when starting to read an article than being “slapped in the face” with a modal window before I have even begun to comprehend what I am looking at.


Or, if I could [quote Andy Budd](#) ([↗ https://twitter.com/andybudd/status/1477634654429663237](https://twitter.com/andybudd/status/1477634654429663237)):

Andy Budd
@andybudd · [Follow](#)





A typical website visit in 2022


1. Figure out how to decline all but essential cookies
2. Close the support widget asking if I need help
3. Stop the auto-playing video
4. Close the “subscribe to our newsletter” pop-up
5. Try and remember why I came here in the first place

8:35 PM · Jan 2, 2022 

[Read the full conversation on Twitter](#)

 41.8K

 Reply

 Copy link

[Read 400 replies](#)

That said, modals are **everywhere** among us. They are a user interface paradigm that we cannot simply disinvent. When used *tastefully* and *wisely*, I dare say they can even help add more context to a document or to an app.

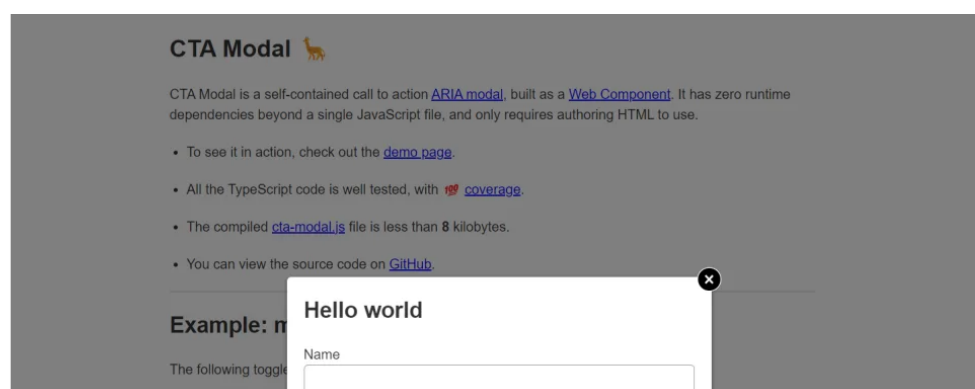
Throughout my career, I have written my fair share of modals. I have built bespoke implementations using vanilla JavaScript, jQuery, and more recently — [React](https://reactjs.org/) ([↪https://reactjs.org/](https://reactjs.org/)). If you have ever struggled to build a modal, then you will know what I mean when I say: It is easy to get them wrong. Not only from a visual standpoint but there are plenty of tricky user interactions that need to be accounted for as well.

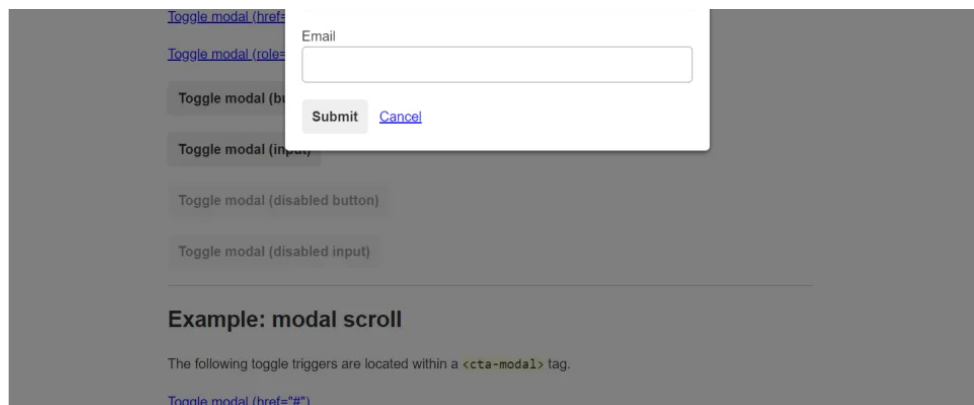
I am the type of person who likes to “go deep” on topics that vex me — especially if I find the topic resurfacing — hopefully in an effort to avoid revisiting them ever again. When I started to get more into [Web Components](https://developer.mozilla.org/en-US/docs/Web/Web_Components) ([↪https://developer.mozilla.org/en-US/docs/Web/Web_Components](https://developer.mozilla.org/en-US/docs/Web/Web_Components)), I had an “a-ha!” moment. Now that Web Components are widely supported by every major browser (RIP, [IE11](https://en.wikipedia.org/wiki/Internet_Explorer_11) ([↪https://en.wikipedia.org/wiki/Internet_Explorer_11](https://en.wikipedia.org/wiki/Internet_Explorer_11))), this opens up a whole new door of opportunity. I thought to myself:

“What if it were possible to build a modal that, as a developer authoring a page or app, I would not have to fuss with any additional JavaScript config?”

Write once and run everywhere, so to speak, or at least that was my lofty aspiration. Good news. It is indeed possible to build a modal with rich interaction that only requires authoring HTML to use.

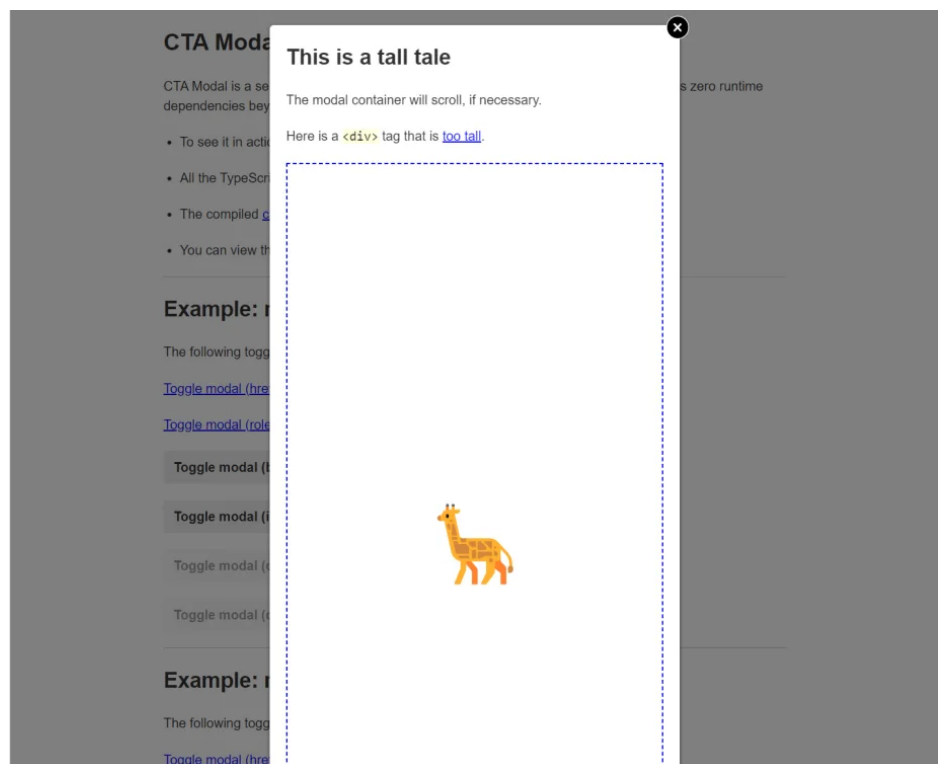
Note: *In order to benefit from this article and code examples you will need some basic familiarity with HTML, CSS, and JavaScript.*





(<https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/cc511cb3-91ff-4aac-8034-859479befd27/3-cta-modal-build-web-component.png>)

📷 CTA Modal – displaying a form. ([Large preview \(https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/cc511cb3-91ff-4aac-8034-859479befd27/3-cta-modal-build-web-component.png\)](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/cc511cb3-91ff-4aac-8034-859479befd27/3-cta-modal-build-web-component.png))



(<https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/401ad39a-7403-4b76-83ec-cc303494ae27/1-cta-modal-build-web-component.png>)

📷 CTA Modal – scrollable content. ([Large preview \(https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/401ad39a-7403-4b76-83ec-cc303494ae27/1-cta-modal-build-web-component.png\)](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eadd629/401ad39a-7403-4b76-83ec-cc303494ae27/1-cta-modal-build-web-component.png))

BEFORE WE EVEN BEGIN # ([https://www.smashingmagazine.com/2022/04/cta-moda...](#))

If you are tight on time and just want to see the finished product, check it out here:

- CTA Modal [Demo page](https://host.sonspring.com/cta-modal) (↪ <https://host.sonspring.com/cta-modal>)
- CTA Modal [Git repo](https://github.com/nathansmith/cta-modal) (↪ <https://github.com/nathansmith/cta-modal>)

Use The Platform [# \(↪ #Use-the-platform\)](#)

Now that we have covered the “why” of scratching this particular itch, throughout the rest of this article I will explain the “how” of building it.

First, a quick crash course on Web Components. They are bundled snippets of HTML, CSS, and JavaScript that encapsulate scope. Meaning, no styles from outside of a component will affect within, nor vice versa. Think of it like a hermetically sealed “[clean room](https://en.wikipedia.org/wiki/Cleanroom) (↪ <https://en.wikipedia.org/wiki/Cleanroom>)” of UI design.

At first blush, this may seem nonsensical. Why would we want a chunk of UI that we cannot control externally via CSS? Hang onto that thought, because we will come back to it soon.

The best explanation is reusability. Building a component in this manner means we are not beholden to any particular JS framework *du jour*. One common phrase that gets bandied about in conversations around web standards is “[use the platform](https://timkadlec.com/remembers/2019-10-21-using-the-platform/) (↪ <https://timkadlec.com/remembers/2019-10-21-using-the-platform/>).” Now more than ever, the platform itself has superb [cross-browser support](https://www.ravedigital.agency/blog/cross-browser-compatibility/) (↪ <https://www.ravedigital.agency/blog/cross-browser-compatibility/>).

Deep Dive [# \(↪ #Deep-dive\)](#)

For reference, I will be referring to this code example — `cta-modal.ts` (↪ <https://github.com/nathansmith/cta-modal/blob/main/src/assets/ts/cta-modal.ts>).

Note: I am using [TypeScript](https://www.typescriptlang.org/) (↪ <https://www.typescriptlang.org/>) here, but you absolutely do

not need any additional tooling to create a Web Component. In fact, I wrote my initial proof-of-concept in *vanilla JS*. I added TypeScript later, to bolster confidence in others using it as an NPM package.

The `cta-modal.ts` file is chunked apart into several sections:

01 [Conditional wrapper](#) ([↪ #conditional-wrapper](#));

02 Constants:

- [Reusable variables](#) ([↪ #reusable-variables](#)),
- [Component styles](#) ([↪ #component-styles](#)),
- [Component markup](#) ([↪ #component-markup](#));

03 `CtaModal` class:

- [Constructor](#) ([↪ #constructor](#)),
- [Binding `this` context](#) ([↪ #binding-this-context](#)),
- [Lifecycle methods](#) ([↪ #lifecycle-methods](#)),
- [Adding and removing events](#) ([↪ #adding-and-removing-events](#)),
- [Detecting attribute changes](#) ([↪ #detecting-attribute-changes](#)),
- [Focusing specific elements](#) ([↪ #focusing-specific-elements](#)),
- [Detecting “outside” modal](#) ([↪ #detecting-outside-modal](#)),
- [Detecting motion preference](#) ([↪ #detecting-motion-preference](#)),
- [Toggling modal show/hide](#) ([↪ #toggling-modal-show-hide](#)),
- Handle event: [click overlay](#) ([↪ #handle-event-click-overlay](#)),
- Handle event: [click toggle](#) ([↪ #handle-event-click-toggle](#)),
- Handle event: [focus element](#) ([↪ #handle-event-focus-element](#)),
- Handle event: [keyboard](#) ([↪ #handle-event-keyboard](#));

04 [DOM loaded callback](#) ([↪ #dom-loaded-callback](#)):

- Waits for the page to be ready,
- Registers the `<cta-modal>` tag.

More after jump! Continue reading below ↓

CONDITIONAL WRAPPER [#\(↪ #CONDITIONAL-WRAPPER\)](#)

There is a single, top level `if` that wraps the entirety of the file's code:

```
// =====
// START: if "customElements".
// =====

if ('customElements' in window) {
  /* NOTE: LINES REMOVED, FOR BREVITY. */
}

// =====
// END: if "customElements".
// =====
```

The reason for this is twofold. We want to ensure that there is [browser support \(↪ https://caniuse.com/custom-elementsv1\)](#) for `window.customElements`. If so, this gives us a handy way to maintain variable scope. Meaning, that when declaring variables via `const` or `let`, they do not “leak” outside of the `if {...}` block. Whereas using an old school `var` would be problematic, inadvertently creating several global variables.

REUSABLE VARIABLES [#\(↪ #REUSABLE-VARIABLES\)](#)

Note: A JavaScript class `Foo {...}` differs from an HTML or CSS `class="foo"`.

Think of it simply as: “A group of functions, bundled together.”

This section of the file contains [primitive](https://developer.mozilla.org/en-US/docs/Glossary/Primitive) ([↗https://developer.mozilla.org/en-US/docs/Glossary/Primitive](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)) values that I intend to reuse throughout my JS [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) ([↗https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)) declaration. I will call out a few of them as being particularly interesting.

```
// =====
// Constants.
// =====

/* NOTE: LINES REMOVED, FOR BREVITY. */

const ANIMATION_DURATION = 250;
const DATA_HIDE = 'data-cta-modal-hide';
const DATA_SHOW = 'data-cta-modal-show';
const PREFERS_REduced_MOTION = '(prefers-reduced-motion: reduce)';

const FOCUSABLE_SELECTORS = [
  '[contenteditable]',
  '[tabindex="0"]:not([disabled])',
  'a[href]',
  'audio[controls]',
  'button:not([disabled])',
  'iframe',
  'input:not([disabled]):not([type="hidden"])',
  'select:not([disabled])',
  'summary',
  'textarea:not([disabled])',
  'video[controls]',
].join(',');
```

- **ANIMATION_DURATION**

Specifies how long my CSS animations will take. I also reuse this later

within a `setTimeout` to keep my CSS and JS in sync. It is set to 250 milliseconds, which is a quarter of a second.

While CSS allows us to specify `animation-duration` in whole seconds (or milliseconds), JS uses increments of milliseconds. Going with this value allows me to use it for both.

- `DATA_SHOW` and `DATA_HIDE`

These are strings for the HTML data attributes `'data-cta-modal-show'` and `'data-cta-modal-hide'` that are used to control the show/hide of modal, as well as adjust animation timing in CSS. They are used later in conjunction with `ANIMATION_DURATION`.

- `PREFERS_REDUCED_MOTION`

A media query that determines whether or not a user has set their operating system's preference to `reduce` for `prefers-reduced-motion`. I look at this value in both CSS and JS to determine whether to turn off animations.

- `FOCUSABLE_SELECTORS`

Contains CSS selectors for all elements that could be considered focusable within a modal. It is used later more than once, via `querySelectorAll`. I have declared it here to help with readability, rather than adding clutter to a function body.

It equates to this string:

```
[contenteditable], [tabindex="0"]:not([disabled]), a[href],
audio[controls], button:not([disabled]), iframe,
input:not([disabled]):not([type='hidden']), select:not([disabled]),
summary, textarea:not([disabled]), video[controls]
```

Yuck, right!? You can see why I wanted to break that into multiple lines.

As an astute reader, you may have noticed `type='hidden'` and `tabindex="0"` are using different quotation marks. That is purposeful, and we will revisit the reasoning later on.

COMPONENT STYLES [# \(↪ #COMPONENT-STYLES\)](#)

This section contains a multiline string with a `<style>` tag. As mentioned before, styles contained within a Web Component do not affect the rest of the page. It is worth noting how I am using embedded variables `${etc}` via string interpolation.

- We reference our variable `PREFERS_REDUCED_MOTION` to forcibly set animations to `none` for users who prefer reduced motion.
- We reference `DATA_SHOW` and `DATA_HIDE` along with `ANIMATION_DURATION` to allow shared control over CSS animations. Note the use of the `ms` suffix for milliseconds, since that is the lingua franca of CSS and JS.

```
// =====  
// Style.  
// =====  
  
const STYLE = `  
  <style>  
    /* NOTE: LINES REMOVED, FOR BREVITY. */  
  
    @media ${PREFERS_REDUCED_MOTION} {  
      *,  
      *:after,  
      *:before {  
        animation: none !important;  
        transition: none !important;  
      }  
    }  
  
    [${DATA_SHOW}='true'] .cta-modal__overlay {  
      animation-duration: ${ANIMATION_DURATION}ms;  
      animation-name: SHOW-OVERLAY;  
    }  
  
    [${DATA_SHOW}='true'] .cta-modal__dialog {
```

```

        animation-duration: ${ANIMATION_DURATION}ms;
        animation-name: SHOW-DIALOG;
    }

    [${DATA_HIDE}='true'] .cta-modal__overlay {
        animation-duration: ${ANIMATION_DURATION}ms;
        animation-name: HIDE-OVERLAY;
        opacity: 0;
    }

    [${DATA_HIDE}='true'] .cta-modal__dialog {
        animation-duration: ${ANIMATION_DURATION}ms;
        animation-name: HIDE-DIALOG;
        transform: scale(0.95);
    }
</style>
`;

```

COMPONENT MARKUP [# \(↪ #COMPONENT-MARKUP\)](#)

The markup for the modal is the most straightforward part. These are the essential aspects that make up the modal:

- slots,
- scrollable area,
- focus traps,
- semi-transparent overlay,
- dialog window,
- close button.

When making use of a `<cta-modal>` tag in one's page, there are two insertion

points for content. Placing elements inside these areas cause them to appear as part of the modal:

- `<div slot="button">` maps to `<slot name='button'>`,
- `<div slot="modal">` maps to `<slot name='modal'>`.

You might be wondering what “focus traps” are, and why we need them. These exist to snag focus when a user attempts to tab forwards (or backwards) outside of the modal dialog. If either of these receives focus, they will place the browser’s focus back inside.

Additionally, we give these attributes to the div we want to serve as our modal dialog element. This tells the browser that the `<div>` is semantically significant. It also allows us to place focus on the element via JS:

- `aria-modal='true'`,
- `role='dialog'`,
- `tabindex='-1'`.

```
// =====  
// Template.  
// =====  
  
const FOCUS_TRAP = `  
  <span  
    aria-hidden='true'  
    class='cta-modal__focus-trap'  
    tabindex='0'  
  ></span>  
`;  
  
const MODAL = `  
  <slot name='button'></slot>  
  
  <div class='cta-modal__scroll' style='display:none'>
```

```

    ${FOCUS_TRAP}

    <div class='cta-modal__overlay'>
      <div
        aria-modal='true'
        class='cta-modal__dialog'
        role='dialog'
        tabindex='-1'
      >
        <button
          class='cta-modal__close'
          type='button'
        >x</button>

        <slot name='modal'></slot>
      </div>
    </div>

    ${FOCUS_TRAP}
  </div>
`;

// Get markup.
const markup = [STYLE,
MODAL].join(EMPTY_STRING).trim().replace(SPACE_REGEX, SPACE);

// Get template.
const template = document.createElement(TEMPLATE);
template.innerHTML = markup;

```

You may be wondering: “Why not use the `dialog` tag?” Good question. At the time of this writing, it still has some cross-browser quirks. For more on that, read this article by [Scott O’hara](https://www.scottohara.me/blog/2019/03/05/open-dialog.html) (↪ <https://www.scottohara.me/blog/2019/03/05/open-dialog.html>). Also, according to the [Mozilla documentation](https://developer.mozilla.org/en-US/docs) (↪ <https://developer.mozilla.org/en-US/docs>

[/Web/HTML/Element/dialog](#)), `dialog` is not allowed to have a `tabindex` attribute, which we need to put focus on our modal.

CONSTRUCTOR [#\(↪ #CONSTRUCTOR\)](#)

Whenever a JS class is instantiated, its `constructor` function is called. That is just a fancy term that means an *instance* of the `CtaModal` class is being created. In the case of our Web Component, this instantiation happens automatically whenever a `<cta-modal>` is encountered in a page's HTML.

Within the `constructor` we call `super` which tells the `HTMLElement` class (which we are `extend`-ing) to call its own `constructor`. Think of it like glue code, to make sure we tap into some of the default lifecycle methods.

Next, we call `this._bind()` which we will cover a bit more later. Then we attach the “shadow DOM” to our class instance and add the markup that we created as a multiline string earlier.

After that, we get all the elements — from within the aforementioned *component markup* section — for use in later function calls. Lastly, we call a few helper methods that read attributes from the corresponding `<cta-modal>` tag.

```
// =====
// Lifecycle: constructor.
// =====

constructor() {
  // Parent constructor.
  super();

  // Bind context.
  this._bind();

  // Shadow DOM.
  this._shadow = this.attachShadow({ mode: 'closed' });

  // Add template.
```

```
this._shadow.appendChild(  
  // Clone node.  
  template.content.cloneNode(true)  
);  
  
// Get slots.  
this._slotForButton = this.querySelector("[slot='button']");  
this._slotForModal = this.querySelector("[slot='modal']");  
  
// Get elements.  
this._heading = this.querySelector('h1, h2, h3, h4, h5, h6');  
  
// Get shadow elements.  
this._buttonClose = this._shadow.querySelector('.cta-modal__close') as  
HTMLElement;  
this._focusTrapList = this._shadow.querySelectorAll('.cta-modal__focus-  
trap');  
this._modal = this._shadow.querySelector('.cta-modal__dialog') as  
HTMLElement;  
this._modalOverlay = this._shadow.querySelector('.cta-modal__overlay')  
as HTMLElement;  
this._modalScroll = this._shadow.querySelector('.cta-modal__scroll') as  
HTMLElement;  
  
// Missing slot?  
if (!this._slotForModal) {  
  window.console.error('Required [slot="modal"] not found inside cta-  
modal.');
```

```

// Set modal label.
this._setModalLabel();

// Set static flag.
this._setStaticFlag();

/*
=====
NOTE:
=====

    We set this flag last because the UI visuals within
    are contingent on some of the other flags being set.
*/

// Set active flag.
this._setActiveFlag();
}

```

BINDING `this` CONTEXT [#\(↪ #BINDING-THIS-CONTEXT\)](#)

This is a bit of JS wizardry that saves us from having to type tedious code needlessly elsewhere. When working with [DOM events](https://en.wikipedia.org/wiki/DOM_events) (↪ https://en.wikipedia.org/wiki/DOM_events) the context of `this` can change, depending on what element is being interacted with within the page.

One way to ensure that `this` always means the instance of our class is to specifically call `bind`. Essentially, this function makes it, so that it is handled automatically. That means we do not have to type things like this everywhere.

```

/* NOTE: Just an example, we don't need this. */
this.someFunctionName1 = this.someFunctionName1.bind(this);
this.someFunctionName2 = this.someFunctionName2.bind(this);

```

Instead of typing that snippet above, every time we add a new function, a handy `this._bind()` call in the constructor takes care of any/all functions we might have. This loop grabs every class property that is a function and binds it automatically.

```
// =====  
// Helper: bind `this` context.  
// =====  
  
_bind() {  
  // Get property names.  
  const propertyNames = Object.getOwnPropertyNames(  
    // Get prototype.  
    Object.getPrototypeOf(this)  
  ) as (keyof CtaModal)[];  
  
  // Loop through.  
  propertyNames.forEach((name) => {  
    // Bind functions.  
    if (typeof this[name] === FUNCTION) {  
      /*  
      =====  
      NOTE:  
      =====  
      */  
    }  
  })  
}
```

Why use "@ts-expect-error" here?

Calling ``.bind(this)`` is a standard practice when using JavaScript classes. It is necessary for functions that might change context because they are interacting directly with DOM elements.

Basically, I am telling TypeScript:


```

        "Let me live my life!"

        🧐

    */

    // @ts-expect-error bind
    this[name] = this[name].bind(this);
  }
});
}

```

LIFECYCLE METHODS [#\(↪ #LIFECYCLE-METHODS\)](#)

By nature of this line, where we extend from `HTMLElement`, we get a few built-in function calls for “free.” As long as we name our functions by these names they will be called at the appropriate time within the lifecycle of our `<cta-modal>` component.

```

// =====
// Component.
// =====

class CtaModal extends HTMLElement {
  /* NOTE: LINES REMOVED, FOR BREVITY. */
}

```

- `observedAttributes`

This tells the browser which attributes we are watching for changes.

- `attributeChangedCallback`

If any of those attributes change, this callback will be invoked. Depending on which attribute changed, we call a function to read the attribute.

- `connectedCallback`

This is called when a `<cta-modal>` tag is registered with the page. We use

this opportunity to add all our event handlers.

If you are familiar with [React](https://reactjs.org/) ([↪https://reactjs.org/](https://reactjs.org/)), this is similar to the `componentDidMount` lifecycle event.

- `disconnectedCallback`

This is called when a `<cta-modal>` tag is removed from the page. Likewise, we remove all obsolete event handlers when/if this occurs.

It is similar to the `componentWillUnmount` lifecycle event in React.

Note: *It is worth pointing out that these are the only functions within our class that are not prefixed by an underscore (`_`). Though not strictly necessary, the reason for this is twofold. One, it makes it obvious which functions we have created for our new `<cta-modal>` and which are native lifecycle events of the `HTMLElement` class. Two, when we minify our code later the prefix denotes they can be mangled. Whereas the native lifecycle methods need to retain their names verbatim.*

```
// =====
// Lifecycle: watch attributes.
// =====

static get observedAttributes() {
  return [ACTIVE, ANIMATED, CLOSE, STATIC];
}

// =====
// Lifecycle: attributes changed.
// =====

attributeChangedCallback(name: string, oldValue: string, newValue:
string) {
  // Different old/new values?
  if (oldValue !== newValue) {
    // Changed [active="..."] value?
    if (name === ACTIVE) {
      this._setActiveFlag();
    }
  }
}
```

```
,

// Changed [animated="..."] value?
if (name === ANIMATED) {
  this._setAnimationFlag();
}

// Changed [close="..."] value?
if (name === CLOSE) {
  this._setCloseTitle();
}

// Changed [static="..."] value?
if (name === STATIC) {
  this._setStaticFlag();
}
}
}

// =====
// Lifecycle: component mount.
// =====

connectedCallback() {
  this._addEvents();
}

// =====
// Lifecycle: component unmount.
// =====

disconnectedCallback() {
  this._removeEvents();
}
```

ADDING AND REMOVING EVENTS [# \(↪ #ADDING-AND-REMOVING-EVENTS\)](#)

These functions register (and remove) callbacks for various element and page-level events:

- buttons clicked,
- elements focused,
- keyboard pressed,
- overlay clicked.

```
// =====
// Helper: add events.
// =====

_addEvents() {
  // Prevent doubles.
  this._removeEvents();

  document.addEventListener(FOCUSIN, this._handleFocusIn);
  document.addEventListener(KEYDOWN, this._handleKeyDown);

  this._buttonClose.addEventListener(CLICK, this._handleClickToggle);
  this._modalOverlay.addEventListener(CLICK, this._handleClickOverlay);

  if (this._slotForButton) {
    this._slotForButton.addEventListener(CLICK, this._handleClickToggle);
    this._slotForButton.addEventListener(KEYDOWN,
this._handleClickToggle);
  }

  if (this._slotForModal) {
    this._slotForModal.addEventListener(CLICK, this._handleClickToggle);
```

```
        this._slotForModal.addEventListener(KEYDOWN, this._handleClickToggle);
    }
}

// =====
// Helper: remove events.
// =====

_removeEvents() {
    document.removeEventListener(FOCUSIN, this._handleFocusIn);
    document.removeEventListener(KEYDOWN, this._handleKeyDown);

    this._buttonClose.removeEventListener(CLICK, this._handleClickToggle);
    this._modalOverlay.removeEventListener(CLICK, this._handleClickOverlay);

    if (this._slotForButton) {
        this._slotForButton.removeEventListener(CLICK,
this._handleClickToggle);
        this._slotForButton.removeEventListener(KEYDOWN,
this._handleClickToggle);
    }

    if (this._slotForModal) {
        this._slotForModal.removeEventListener(CLICK,
this._handleClickToggle);
        this._slotForModal.removeEventListener(KEYDOWN,
this._handleClickToggle);
    }
}
```

DETECTING ATTRIBUTE CHANGES [#\(↪ #DETECTING-ATTRIBUTE-CHANGES\)](#)

These functions handle reading attributes from a `<cta-modal>` tag and setting

various flags as a result:

- Setting an `_isAnimated` boolean on our class instance.
- Setting `title` and `aria-label` attributes on our close button.
- Setting an `aria-label` for our modal dialog, based on heading text.
- Setting an `_isActive` boolean on our class instance.
- Setting an `_isStatic` boolean on our class instance.

You may be wondering why we are using `aria-label` to relate the modal to its heading text (if it exists). At the time of this writing, browsers are not currently able to correlate an `aria-labelledby="..."` attribute — within the shadow DOM — to an `id="..."` that is located in the standard (aka “light”) DOM.

I will not go into great detail about that, but you can read more here:

- [W3C: cross-root ARIA](https://w3c.github.io/webcomponents-cg/#cross-root-aria) (↪ <https://w3c.github.io/webcomponents-cg/#cross-root-aria>)
- [WHATWG: element reflection ticket](https://github.com/whatwg/html/issues/6063) (↪ <https://github.com/whatwg/html/issues/6063>)

```
// =====
// Helper: set animation flag.
// =====

_setAnimationFlag() {
  this._isAnimated = this.getAttribute(ANIMATED) !== FALSE;
}

// =====
// Helper: add close text.
// =====

_setCloseTitle() {
  // Get title.
```

```
const title = this.getAttribute(CLOSE) || CLOSE_TITLE;

// Set title.
this._buttonClose.title = title;
this._buttonClose.setAttribute(ARIA_LABEL, title);
}

// =====
// Helper: add modal label.
// =====

_setModalLabel() {
  // Set later.
  let label = MODAL_LABEL_FALLBACK;

  // Heading exists?
  if (this._heading) {
    // Get text.
    label = this._heading.textContent || label;
    label = label.trim().replace(SPACE_REGEX, SPACE);
  }

  // Set label.
  this._modal.setAttribute(ARIA_LABEL, label);
}

// =====
// Helper: set active flag.
// =====

_setActiveFlag() {
  // Get flag.
  const isActive = this.getAttribute(ACTIVE) === TRUE;

  // Set flag.
```

```

    this._isActive = isActive;

    // Set display.
    this._toggleModalDisplay(() => {
      // Focus modal?
      if (this._isActive) {
        this._focusModal();
      }
    });
  }

  // =====
  // Helper: set static flag.
  // =====

  _setStaticFlag() {
    this._isStatic = this.getAttribute(STATIC) === TRUE;
  }

```

FOCUSING SPECIFIC ELEMENTS [\(#FOCUSING-SPECIFIC-ELEMENTS\)](#)

The `_focusElement` function allows us to focus an element that may have been active before a modal became active. Whereas the `_focusModal` function will place focus on the modal dialog itself and will ensure that the modal backdrop is scrolled to the top.

```

  // =====
  // Helper: focus element.
  // =====

  _focusElement(element: HTMLElement) {
    window.requestAnimationFrame(() => {
      if (typeof element.focus === FUNCTION) {

```


DETECTING “OUTSIDE” MODAL [#\(↪ #DETECTING-OUTSIDE-MODAL\)](#)

This function is handy to know if an element resides outside the parent `<cta-modal>` tag. It returns a boolean, which we can use to take appropriate action. Namely, tab trapping navigation inside the modal while it is active.

```
// =====  
// Helper: detect outside modal.  
// =====  
  
_isOutsideModal(element?: HTMLElement) {  
  // Early exit.  
  if (!this._isActive || !element) {  
    return false;  
  }  
  
  // Has element?  
  const hasElement = this.contains(element) ||  
    this._modal.contains(element);  
  
  // Get boolean.  
  const bool = !hasElement;  
  
  // Expose boolean.  
  return bool;  
}
```

DETECTING MOTION PREFERENCE [#\(↪ #DETECTING-MOTION-PREFERENCE\)](#)

Here, we reuse our variable from before (also used in our CSS) to detect if a user is

okay with motion. That is, they have not explicitly set `prefers-reduced-motion` to `reduce` via their operating system preferences.

The returned boolean is a combination of that check, plus the `animated="false"` flag not being set on `<cta-modal>`.

```
// =====
// Helper: detect motion pref.
// =====

_isMotionOkay() {
  // Get pref.
  const { matches } = window.matchMedia(PREFERS_REDUCED_MOTION);

  // Expose boolean.
  return this._isAnimated && !matches;
}
```

TOGGLING MODAL SHOW/HIDE [\(#\(↪ #TOGGLING-MODAL-SHOW-HIDE\)\)](#)

There is quite a bit going on in this function, but in essence, it is pretty simple.

- **If the modal is not active, show it.** If animation is allowed, animate it into place.
- **If the modal is active, hide it.** If animation is allowed, animate it disappearing.

We also cache the currently active element, so that when the modal closes we can restore focus.

The variables used in our CSS earlier are also used here:

- `ANIMATION_DURATION`,
- `DATA_SHOW`,

- DATA_HIDE.

```
// =====  
// Helper: toggle modal.  
// =====  
  
_toggleModalDisplay(callback: () => void) {  
  // @ts-expect-error boolean  
  this.setAttribute(ACTIVE, this._isActive);  
  
  // Get booleans.  
  const isModalVisible = this._modalScroll.style.display === BLOCK;  
  const isMotionOkay = this._isMotionOkay();  
  
  // Get delay.  
  const delay = isMotionOkay ? ANIMATION_DURATION : 0;  
  
  // Get scrollbar width.  
  const scrollbarWidth = window.innerWidth -  
document.documentElement.clientWidth;  
  
  // Get active element.  
  const activeElement = document.activeElement as HTMLElement;  
  
  // Cache active element?  
  if (this._isActive && activeElement) {  
    this._activeElement = activeElement;  
  }  
  
  // =====  
  // Modal active?  
  // =====  
  
  if (this._isActive) {
```

```
// Show modal.

this._modalScroll.style.display = BLOCK;

// Hide scrollbar.

document.documentElement.style.overflow = HIDDEN;

// Add placeholder?

if (scrollbarWidth) {

    document.documentElement.style.paddingRight = `${scrollbarWidth}px`;

}

// Set flag.

if (isMotionOkay) {

    this._isHideShow = true;

    this._modalScroll.setAttribute(DATA_SHOW, TRUE);

}

// Fire callback.

callback();

// Await CSS animation.

this._timerForShow = window.setTimeout(() => {

    // Clear.

    clearTimeout(this._timerForShow);

    // Remove flag.

    this._isHideShow = false;

    this._modalScroll.removeAttribute(DATA_SHOW);

    // Delay.

}, delay);

/*

=====

NOTE:
```

=====

We want to ensure that the modal is currently visible because we do not want to put scroll back on the `` element unnecessarily.

The reason is that another `` in the page might have been pre-rendered with an `[active="true"]` attribute. If so, we want to leave the page's overflow value alone.

```
*/  
} else if (isModalVisible) {  
  // Set flag.  
  if (isMotionOkay) {  
    this._isHideShow = true;  
    this._modalScroll.setAttribute(DATA_HIDE, TRUE);  
  }  
  
  // Fire callback?  
  callback();  
  
  // Await CSS animation.  
  this._timerForHide = window.setTimeout(() => {  
    // Clear.  
    clearTimeout(this._timerForHide);  
  
    // Remove flag.  
    this._isHideShow = false;  
    this._modalScroll.removeAttribute(DATA_HIDE);  
  
    // Hide modal.  
    this._modalScroll.style.display = NONE;  
  
    // Show scrollbar.  
    document.documentElement.style.overflow = EMPTY_STRING;
```

```

        // Remove placeholder.

        document.documentElement.style.paddingRight = EMPTY_STRING;

        // Delay.
      }, delay);
    }
  }
}

```

HANDLE EVENT: CLICK OVERLAY [# \(↪ #HANDLE-EVENT-CLICK-OVERLAY\)](#)

When clicking on the semi-transparent overlay, assuming that `static="true"` is not set on the `<cta-modal>` tag, we close the modal.

```

// =====
// Event: overlay click.
// =====

_handleClickOverlay(event: MouseEvent) {
  // Early exit.
  if (this._isHideShow || this._isStatic) {
    return;
  }

  // Get layer.
  const target = event.target as HTMLElement;

  // Outside modal?
  if (target.classList.contains('cta-modal__overlay')) {
    this._handleClickToggle();
  }
}

```

HANDLE EVENT: CLICK TOGGLE [#\(↪ #HANDLE-EVENT-CLICK-TOGGLE\)](#)

This function uses [event delegation](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#event_delegation) (↪ https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#event_delegation) on the `<div slot="button">` and `<div slot="modal">` elements. Whenever a child element with the class `cta-modal-toggle` is triggered, it will cause the active state of the modal to change.

This includes listening for various events that are considered activating a button:

- mouse clicks,
- pressing the enter key,
- pressing the spacebar key.

```
// =====
// Event: toggle modal.
// =====

_handleClickToggle(event?: MouseEvent | KeyboardEvent) {
  // Set later.
  let key = EMPTY_STRING;
  let target = null;

  // Event exists?
  if (event) {
    if (event.target) {
      target = event.target as HTMLElement;
    }

    // Get key.
    if ((event as KeyboardEvent).key) {
      key = (event as KeyboardEvent).key;
    }
  }
}
```

```
        key = key.toLowerCase();
    }
}

// Set later.
let button;

// Target exists?
if (target) {
    // Direct click.
    if (target.classList.contains('cta-modal__close')) {
        button = target as HTMLButtonElement;

        // Delegated click.
    } else if (typeof target.closest === FUNCTION) {
        button = target.closest('.cta-modal-toggle') as HTMLButtonElement;
    }
}

// Get booleans.
const isValidEvent = event && typeof event.preventDefault === FUNCTION;
const isValidClick = button && isValidEvent && !key;
const isValidKey = button && isValidEvent && [ENTER,
SPACE].includes(key);

const isButtonDisabled = button && button.disabled;
const isButtonMissing = isValidEvent && !button;
const isWrongKeyEvent = key && !isValidKey;

// Early exit.
if (isButtonDisabled || isButtonMissing || isWrongKeyEvent) {
    return;
}

// Prevent default?
```



```

    if (isValidKey || isValidClick) {
      event.preventDefault();
    }

    // Set flag.
    this._isActive = !this._isActive;

    // Set display.
    this._toggleModalDisplay(() => {
      // Focus modal?
      if (this._isActive) {
        this._focusModal();

        // Return focus?
      } else if (this._activeElement) {
        this._focusElement(this._activeElement);
      }
    });
  }
}

```

HANDLE EVENT: FOCUS ELEMENT [# \(↪ #HANDLE-EVENT-FOCUS-ELEMENT\)](#)

This function is triggered whenever an element receives `focus` on the page. Depending on the state of the modal, and which element was focused, we can trap tab navigation within the modal dialog. This is where our `FOCUSABLE_SELECTORS` from early comes into play.

```

// =====
// Event: focus in document.
// =====

_handleFocusIn() {
  // Early exit.

```

```
if (!this._isActive) {
  return;
}

// prettier-ignore
const activeElement = (
  // Get active element.
  this._shadow.activeElement ||
  document.activeElement
) as HTMLElement;

// Get booleans.
const isFocusTrap1 = activeElement === this._focusTrapList[0];
const isFocusTrap2 = activeElement === this._focusTrapList[1];

// Set later.
let focusListReal: HTMLElement[] = [];

// Slot exists?
if (this._slotForModal) {
  // Get "real" elements.
  focusListReal = Array.from(
    this._slotForModal.querySelectorAll(FOCUSABLE_SELECTORS)
  ) as HTMLElement[];
}

// Get "shadow" elements.
const focusListShadow = Array.from(
  this._modal.querySelectorAll(FOCUSABLE_SELECTORS)
) as HTMLElement[];

// Get "total" elements.
```

HANDLE EVENT: KEYBOARD [#\(↪ #HANDLE-EVENT-KEYBOARD\)](#)

If a modal is active when the `escape` key is pressed, it will be closed. If the `tab` key is pressed, we evaluate whether or not we need to adjust which element is focused.

```
// =====
// Event: key press.
// =====

_handleKeyDown({ key }: KeyboardEvent) {
  // Early exit.
  if (!this._isActive) {
    return;
  }

  // Get key.
  key = key.toLowerCase();

  // Escape key?
  if (key === ESCAPE && !this._isHideShow && !this._isStatic) {
    this._handleClickToggle();
  }

  // Tab key?
  if (key === TAB) {
    this._handleFocusIn();
  }
}
```

DOM LOADED CALLBACK [#\(↪ #DOM-LOADED-CALLBACK\)](#)

This event listener tells the window to wait for the DOM (HTML page) to be loaded, and then parses it for any instances of `<cta-modal>` and attaches our JS interactivity to it. Essentially, we have created a new HTML tag and now the

browser knows how to use it.

```
// =====
// Define element.
// =====

window.addEventListener('DOMContentLoaded', () => {
  window.customElements.define('cta-modal', CtaModal);
});
```

Build Time Optimization [#\(↪ #Build-time-optimization\)](#)

I will not go into great detail about this aspect, but I think it is worth calling out.

After transpiling from TypeScript to JavaScript, I run [Terser](https://terser.org/) (↪ <https://terser.org/>) against the JS output. All the aforementioned functions that begin with an underscore (`_`) are marked as safe to mangle. That is, they go from being named `_bind` and `_addEvents` to single letters instead.

That step brings the file size down considerably. Then I run the minified output through a [minifyWebComponent.js](https://github.com/nathansmith/cta-modal/blob/main/scripts/minifyWebComponent.js) (↪ <https://github.com/nathansmith/cta-modal/blob/main/scripts/minifyWebComponent.js>) process that I created, which compresses the embedded `<style>` and markup even further.

For example, class names and other attributes (and selectors) are minified. This happens in the CSS and HTML.

- `class='cta-modal__overlay'` becomes `class=o`. The quotes are removed as well because the browser does not technically need them to understand the intent.
- The one CSS selector that is left untouched is `[tabindex="0"]`, because removing the quotes from around the `0` seemingly makes it invalid when parsed by `querySelectorAll`. However, it is safe to minify within HTML from `tabindex='0'` to `tabindex=0`.

When it is all said and done, the file size reduction looks like this (in bytes):

- un-minified: 16,849,
- terser minify: 10,230,
- and my script: 7,689.

To put that into perspective, the `favicon.ico` file on Smashing Magazine is 4,286 bytes. So, we are not really adding much overhead at all, for a lot of functionality that only requires writing HTML to use.

Conclusion [#\(↪#Conclusion\)](#)

If you have read this far, thanks for sticking with me. I hope that I have at least piqued your interest in Web Components!

I know we covered quite a bit, but the good news is: That is all there is to it. There are no frameworks to learn unless you want to. Realistically, you can get started writing your own Web Components using vanilla JS without a build process.

There really has never been a better time to `#UseThePlatform`. I look forward to seeing what you imagine.

FURTHER READING [#\(↪#FURTHER-READING\)](#)

I would be remiss if I did not mention that there are a myriad of other modal options out there.

While I am biased and feel my approach brings something unique to the table — otherwise I would not have tried to “reinvent the wheel” — you may find that one of these will better suit your needs.

The following examples differ from CTA Modal in that they all require at least **some** additional JavaScript to be written by the end-user developer. Whereas with CTA Modal, all you have to author is the HTML code.

Flat HTML & JS:

- [a11y-dialog](https://a11y-dialog.netlify.app) (↪ <https://a11y-dialog.netlify.app>)
- [Bootstrap modal](https://getbootstrap.com/docs/5.1/components/modal) (↪ <https://getbootstrap.com/docs/5.1/components/modal>)
- [Micromodal](https://micromodal.vercel.app) (↪ <https://micromodal.vercel.app>)

Web Components:

- [aria-modal](https://keiyao1.github.io/aria-modal-doc/top) (↪ <https://keiyao1.github.io/aria-modal-doc/top>)
- [web-dialog](https://github.com/andreasbm/web-dialog) (↪ <https://github.com/andreasbm/web-dialog>) with [@a11y/focus-trap](https://appnest-demo.firebaseio.com/focus-trap) (↪ <https://appnest-demo.firebaseio.com/focus-trap>)

jQuery:

- [jQuery Modal](https://jquerymodal.com) (↪ <https://jquerymodal.com>)
- [Lightbox](https://lokeshdhakar.com/projects/lightbox2) (↪ <https://lokeshdhakar.com/projects/lightbox2>)
- [Thickbox](http://codylindley.com/thickbox) (↪ <http://codylindley.com/thickbox>)

React:

- [React Modal](https://reactcommunity.org/react-modal) (↪ <https://reactcommunity.org/react-modal>)

Vue:

- [Vue.js Modal](http://vue-js-modal.yev.io) (↪ <http://vue-js-modal.yev.io>)



(mb, vf, yk, il)

Explore more on

[Apps](/category/apps) (↪ </category/apps>)

[HTML](/category/html) (↪ </category/html>)

[Browsers](/category/browsers) (↪ </category/browsers>)

[JavaScript](/category/javascript) (↪ </category/javascript>)



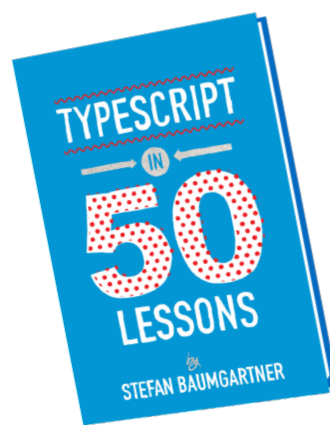
Smashing Newsletter

*Tips on front-end & UX,
delivered weekly in your
inbox. Just the things you can
actually use.*



Front-End & UX Workshops, Online

*With practical takeaways, live
sessions, video recordings and
a friendly Q&A.*



TypeScript in 50 Lessons

*Everything TypeScript, with
code walkthroughs and
examples. And other printed
books.*



With a commitment to quality content for the design community.
Founded by Vitaly Friedman and Sven Lennartz. 2006–2022.
Smashing is proudly running on [Netlify](#).
Fonts by [Latinotype](#).

Cats can be forgetful, but we are not.
Thanks for being truly smashing — yet again.
www.smashingmagazine.com