Piccalilli Front-end education for the real world. *Since 2018.*

From **set.studio**

Articles **Links** **Newsletter** **Login**

Build a fully-responsive, progressively enhanced burger menu

Published: 15 January 2021 **Written by:** Andy Bell

Take your CSS skills beyond the next level, joining over 750 others who are taking our Complete CSS course

This is a *long* tutorial, so make sure you have plenty of time to get though it. Let's dive in.

Getting started

We're keeping things on The Platform™ in this tutorial, so no need to worry about build processes. We're going to have a single HTML page, a CSS file and a couple of JavaScript files.

First up, grab these starter files that will give you the right structure.

DOWNLOAD STARTER FILES

Extract those into the folder you want to work out of. It should look like this:

TEXT COPY TO CLIPBOARD

```
├── css
│   └── global.css
```

```
├── images
│   └── logo.svg
├── index.html
├── js
│   ├── burger-menu.js
│   └── get-focusable-elements.js
```

These files are all empty (apart from the logo), so let’s start filling them up.

Adding our HTML

We’ll start with HTML, so open up `index.html` and add the following to it:

HTML COPY TO CLIPBOARD

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0'
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Build a fully-responsive, progressively enhanced burger menu</
    <link rel="stylesheet" href="https://unpkg.com/modern-css-reset/dist/
    <link rel="stylesheet" href="css/global.css" />
    <link rel="preconnect" href="https://fonts.gstatic.com" />
    <link
      href="https://fonts.googleapis.com/css2?family=Halant:wght@600&fam
      rel="stylesheet"
    />
  </head>
  <body>
    <script src="js/burger-menu.js" type="module" defer></script>
  </body>
</html>
```

This is our HTML shell and features all the relevant CSS, fonts and JavaScript files, pre-linked and ready to go. We're using this modern CSS reset to give us some sensible defaults that we'll build on with our custom CSS.

Let's add some more HTML, first. Open up `index.html` again, and after the opening

<body> tag, add the following:

HTML COPY TO CLIPBOARD

```html
<header class="site-head" role="banner">
  <a href="#main-content" class="skip-link">Skip to content</a>
  <div class="wrapper">
    <div class="site-head__inner">
      <a href="/" aria-label="ACME home" class="site-head__brand">
        <img src="images/logo.svg" alt="ACME logo" />
      </a>
      <burger-menu max-width="600">
        <nav class="navigation" aria-label="primary">
          <ul role="list">
            <li>
              <a href="#">Home</a>
            </li>
            <li>
              <a href="#">About</a>
            </li>
            <li>
              <a href="#">Our Work</a>
            </li>
            <li>
              <a href="#">Contact Us</a>
            </li>
            <li>
              <a href="#">Your account</a>
            </li>
          </ul>
        </nav>
      </burger-menu>
    </div>
  </div>
</header>
```

This is our main site header and it’s got a few bits that we’ll break down. First up, we have a skip link. A skip link allows a user to skip past the header and navigation and jump straight to the `<main>` element—which in our case—contains a simple `<article>`. It’ll be visually invisible by default and show on focus, when we get around to writing some CSS.

Next up, we have the brand element, which contains our placeholder logo. We’re using the `aria-label` attribute to provide the text to mainly assist screen readers. Very importantly, the `alt` on the image describes the image as “ACME logo”, with “ACME” being the name of the company.

Lastly, we have our main navigation—a classic unordered list of links in a `<nav>` element. It’s wrapped in a `<burger-menu>` element which is a Custom Element. This is a great example of HTML being an incredibly smart programming language because even though we’ve not defined what this element is or does yet, but the browser doesn’t care—it just continues doing what it’s doing, without any fuss. This capability helps us build this project *progressively*, which we’ll get into more, shortly.

We have an `aria-label` on our `<nav>` element. It’s not required in our case, but if you have more than one `<nav>` on a page, you must label them to help assistive technology.

Let’s wrap up our HTML by adding the last bit. Still inside `index.html`, add the following **after** the closing `</header>`:

HTML COPY TO CLIPBOARD

```
<main id="main-content" tabindex="-1" class="wrapper">
  <article class="post flow">
```

```
    <h1>A responsive, progressively enhanced burger menu</h1>
    <p>
      Burger menus are a relic of responsive design that no matter what y
      them is, they continue to be a dominant design pattern. They’re ver
      preserving often-limited horizontal space, but they also, more ofte
      built in a user-hostile, non-accessible fashion.
    </p>
    <p>
      <a
        href="https://piccalil.li/premium/build-a-fully-responsive-progre
        >In this premium tutorial</a
      >, we’re going to build a burger menu from the ground up, using pro
      enhancement, `ResizeObserver`, `Proxy` state and of course, super-s
      that pull from the CUBE CSS principles.
    </p>
    <p>
      This whole page is what you’re building in the tutorial 👆.
      <a
        href="https://piccalil.li/premium/build-a-fully-responsive-progre
        >Let’s dive in</a
      >
    </p>
  </article>
```

```
</main>
```

There's not much to say about this as it's a pretty straightforward `<article>`. The only bit to make you aware of is that the `<main>` element should only feature once on the page, being the **main content**. We've got an `id` of `main-content` on there, too, which is what our skip link links to.

We've got all of our HTML now and guess what: we've got a fully functional web page (if you ignore the `#` links). The key to progressive enhancement is building up with a principle of least power approach. We know now that as long as this very small HTML page lands in the user's browser, they can immediately and effectively use it.

Minimum Viable Experience CSS

We're going to approach our CSS progressively too, so using principles of CUBE CSS: we're going to start right at the top with some global CSS.

FYI

> If you haven't read up on CUBE CSS yet, I would recommend reading this high-level overview post to give yourself some background knowledge.

Open up `css/global.css` and add the following to it:

CSS COPY TO CLIPBOARD

```css
:root {
  --color-light: #ffffff;
  --color-light-shade: #fafffd;
  --color-dark: #062726;
  --color-primary: #d81159;
  --color-primary-shade: #b90f4c;
}
```

These are our site colours, neatly organised as some root-level CSS Custom Properties. The `:root` pseudo-class is just a posh way of using the `<html>` element. Keep in mind, though, that a pseudo-class (not pseudo-elements) has a higher specificity than a HTML element (`<html>` in this case). They have the same specificity as a `class`.

Let’s add some core globals. Still in `global.css`, add the following:

CSS COPY TO CLIPBOARD

```css
body {
  background: var(--color-light-shade);
  color: var(--color-dark);
  line-height: 1.5;
  font-family: 'Hind', 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-
  font-weight: 400;
}

h1,
h2 {
  font-family: 'Halant', Georgia, 'Times New Roman', Times, serif;
  font-weight: 600;
  line-height: 1.1;
  max-width: 30ch;
}
```

```
h1 {
  font-size: 2rem;
}

h2 {
  font-size: 1.8rem;
}

a {
  color: currentColor;
}

:focus {
  outline: 1px dotted currentColor;
  outline-offset: 0.2rem;
}

p,
li,
dl {
  max-width: 70ch;
}
```

```
article {
  margin-top: 2.5rem;
  font-size: 1.25rem;
}

main:focus {
  outline: none;
}

@media (min-width: 40em) {
  h1 {
    font-size: 3rem;
  }

  h2 {
    font-size: 2.5rem;
  }
}
```

These are high-level, global HTML-element styles. We're setting the basics, mainly, but again, with progressive enhancement in mind, we're keeping things as simple as possible.

Some key points:

1. We set a `max-width` on headings, paragraphs, lists elements and description lists using a `ch` unit. This really helps with readability and a `ch` unit is equal to the width of a `0` character in your chosen font and size. You can read more about why we do this here.
2. We set `:focus` styles globally by modifying how the `outline` looks. This means that any element that can receive focus, such as `<a>` and `<button>` will have a consistent focus style. The `outline-offset` pulls the outline away from the content a bit, which in my opinion, makes it more user-friendly.
3. We remove focus styles from the `<main>` element because when someone activates the skip link from before, it programatically focuses the `<main>` because it's the `:target`. The focus ring is unnecessary though, because making the `<main>` focusable, programatically, is purely for making tabbing on the keyboard more predictable for users who want to skip navigation. If we didn't move focus, they could end up in a situation where hitting the tab key sends them back up the the navigation!

FYI

Did you know that setting a `tabindex` HTML attribute value of `-1` allows you to focus it programatically? What this means is that you can—just like the skip link—pass focus when it's targetted, but you can also use JavaScript to pass focus, using the `focus()` function.

What's handy with this approach is that using `tabindex="-1"` prevents the user from being able to tab to the element with their keyboard too, so it's really helpful with focus management of interactive elements, which we'll get on to later in this tutorial.

CSS Utilities

We've got some sensible, global CSS, so let's now add some utilities.

In the CUBE-CSS documentation, I describe utilities like so:

> *A utility, in the context of CUBE CSS, is a CSS class that does one job and does that one job well.*

With that in mind, we're going to add 3 utilities: a skip link, a wrapper and finally a flow utility.

Let's add the skip link first. Open up `global.css` and add the following to it:

CSS COPY TO CLIPBOARD

```css
.skip-link {
  display: inline-block;
  padding: 0.7rem 1rem 0.5rem 1rem;
  background: var(--color-light);
  color: var(--color-primary-shade);
  text-decoration: none;
  font-weight: 700;
  text-transform: uppercase;
  position: absolute;
  top: 1rem;
  left: 1rem;
}
```

The skip link is styled to look like a button. It’s good to give it plenty of contrast against it’s context—which in our case, is the site header—and making it look like a button really helps with that. Other than that, there’s not much to discuss about this, so let’s add the clever stuff.

FYI

You might be wondering why we’re lumping all of the CSS into `global.css`. It’s to keep this tutorial as simple as possible. I would recommend breaking global CSS, utilities, blocks and composition styles (both coming up, shortly) into their own areas, in a proper project, though.

Still inside `global.css`, add the following:

CSS COPY TO CLIPBOARD

```css
.skip-link:hover {
  background: var(--color-dark);
```

```css
    color: var(--color-light-shade);
  }

  .skip-link:not(:focus) {
    border: 0;
    clip: rect(0 0 0 0);
    height: auto;
    margin: 0;
    overflow: hidden;
    padding: 0;
    position: absolute;
    width: 1px;
    white-space: nowrap;
  }
```

The first part is some good ol' `:hover` styles. The important part, though, is that when the skip link is **not focused**, we **visually hide it**. The CSS in the `.skip-link:not(:focus)` block is the same as this visually hidden utility, which allows screen readers and parsers to "see" it, while visually, it isn't present. Because we use the `:not(:focus)` pseudo-selector: when the skip link is focused, it shows, visually.

Right, that's our main utility sorted. This one very much blurs the line between a CUBE CSS block and a utility, but I'm pretty happy putting it where we have it for this one.

Next up, let's add those remaining utilities. We'll add the `wrapper` which is a simple container that helps keep our sections aligned with each other.

Open up `global.css` and add the following to it:

CSS COPY TO CLIPBOARD

```css
.wrapper {
  max-width: 65rem;
  margin-left: auto;
  margin-right: auto;
  padding-left: 1.25rem;
  padding-right: 1.25rem;
}
```

This does exactly what it says on the tin, really. The only part to mention is that we specifically add left/right padding/margin so other compositional CSS can comfortably manage vertical space, if needed.

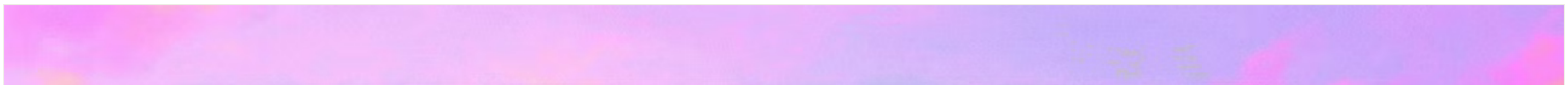
Let’s add the flow utility, too. Inside `global.css`, add the following:

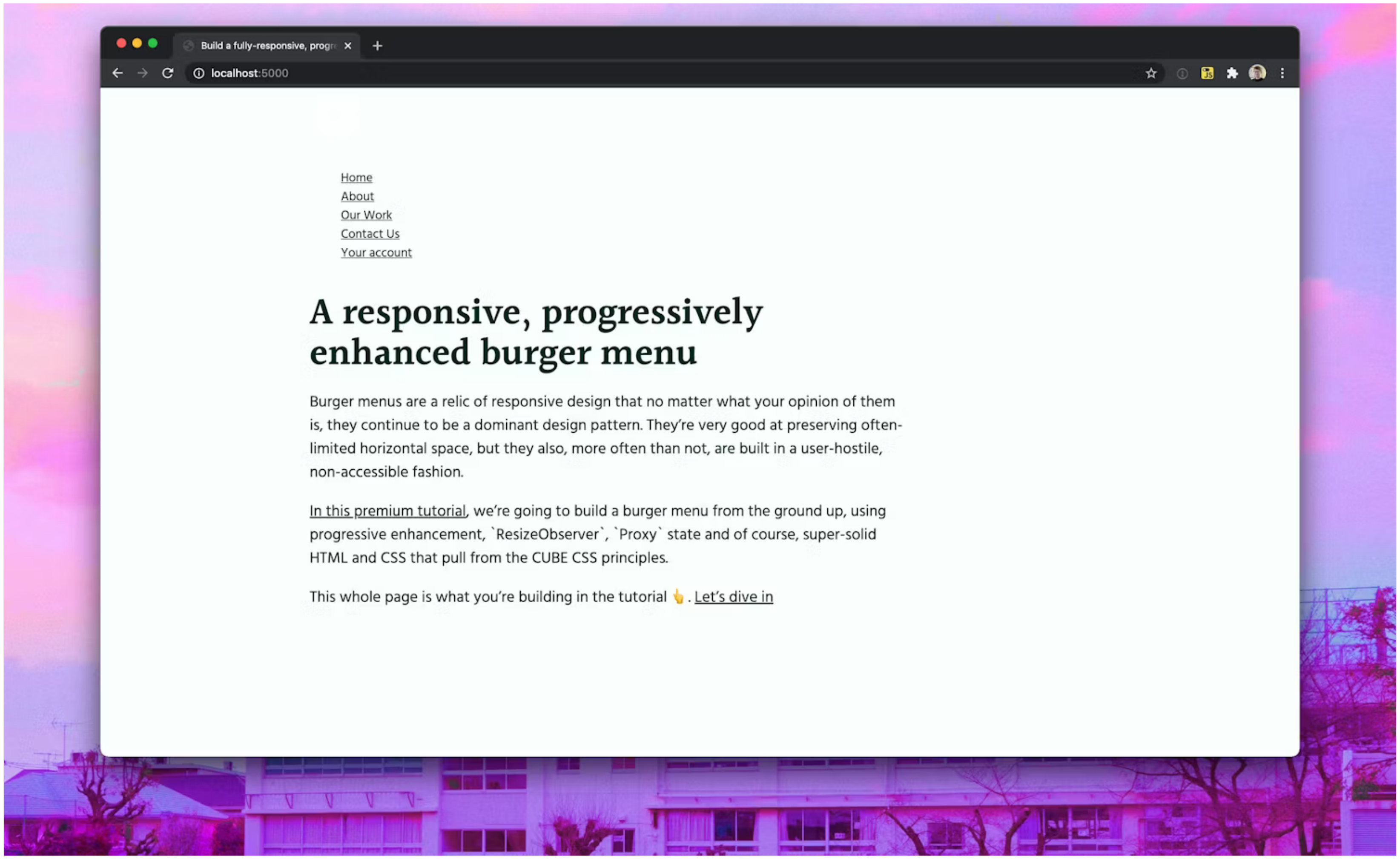
CSS   COPY TO CLIPBOARD

```css
.flow > * + * {
  margin-top: var(--flow-space, 1em);
}
```

This utility is explained in this quick tip, so go ahead and give it a read, but in short, it adds space to sibling elements automatically and I use it **all the time**.

With all of the HTML, global CSS and CSS utilities added, your page should look like this.





Home
About
Our Work
Contact Us
Your account

A responsive, progressively enhanced burger menu

Burger menus are a relic of responsive design that no matter what your opinion of them is, they continue to be a dominant design pattern. They're very good at preserving often-limited horizontal space, but they also, more often than not, are built in a user-hostile, non-accessible fashion.

In this premium tutorial, we're going to build a burger menu from the ground up, using progressive enhancement, `ResizeObserver`, `Proxy` state and of course, super-solid HTML and CSS that pull from the CUBE CSS principles.

This whole page is what you're building in the tutorial 👆. Let's dive in

CSS Blocks

Let's dig into the details now—but we're only going to do our minimum viable experience CSS.

This is the default experience, for **when** JavaScript doesn't load for a user. We want to make sure that the experience is solid and that the user doesn't even notice that there is missing functionality.

We build this CSS first, but it's also worth continually testing this use-case *even when* all your main functionality—in our case, a burger menu—is fully implemented.

Let's start by styling up the site head. Open up `global.css` and add the following to it:

CSS COPY TO CLIPBOARD

```css
.site-head {
  padding: 0.6rem 0;
  background: var(--color-primary);
  border-top: 5px solid var(--color-primary);
  border-bottom: 5px solid var(--color-primary-shade);
  color: var(--color-light);
  line-height: 1.1;
}

.site-head :focus {
  outline-color: var(--color-light);
}
```

```
.site-head__inner {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
  align-items: center;
  gap: 0 1rem;
}

.site-head__brand {
  display: block;
  width: 3rem;
}
```

To start with, we add the colour to site head and a nice bottom border. Having just a bottom border puts things out of kilter, visually, so we add an optical adjustment, in the form of the same border style—*but* the same colour as the background. This border is essentially invisible, but it levels things out. Brains are weird, right?

Next, some good ol' flexible layout stuff. The `site-head__inner` element uses flexbox to *push* elements away from each other—importantly, only where there is space. We use `flex-wrap: wrap` to allow items to stack on top of each other where needed.

All mixed together, this means that because `justify-content` affects the **horizontal axis**—because we are using the default `flex-direction` value of `row`—it won't affect items that are not on the same axis anymore. This means we get responsive layout with no media queries. Handy.

Let's move on to a navigation block. Open up `global.css` again and add the following:

CSS COPY TO CLIPBOARD

```css
.navigation ul {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  gap: 0.3rem 0.8rem;
  padding: 0;
}

.navigation li {
  margin: 0.1rem;
}
```

```
.navigation a {
  font-weight: 600;
  text-transform: uppercase;
  text-decoration: none;
  color: currentColor;
}

.navigation a:hover {
  color: var(--color-dark);
}
```

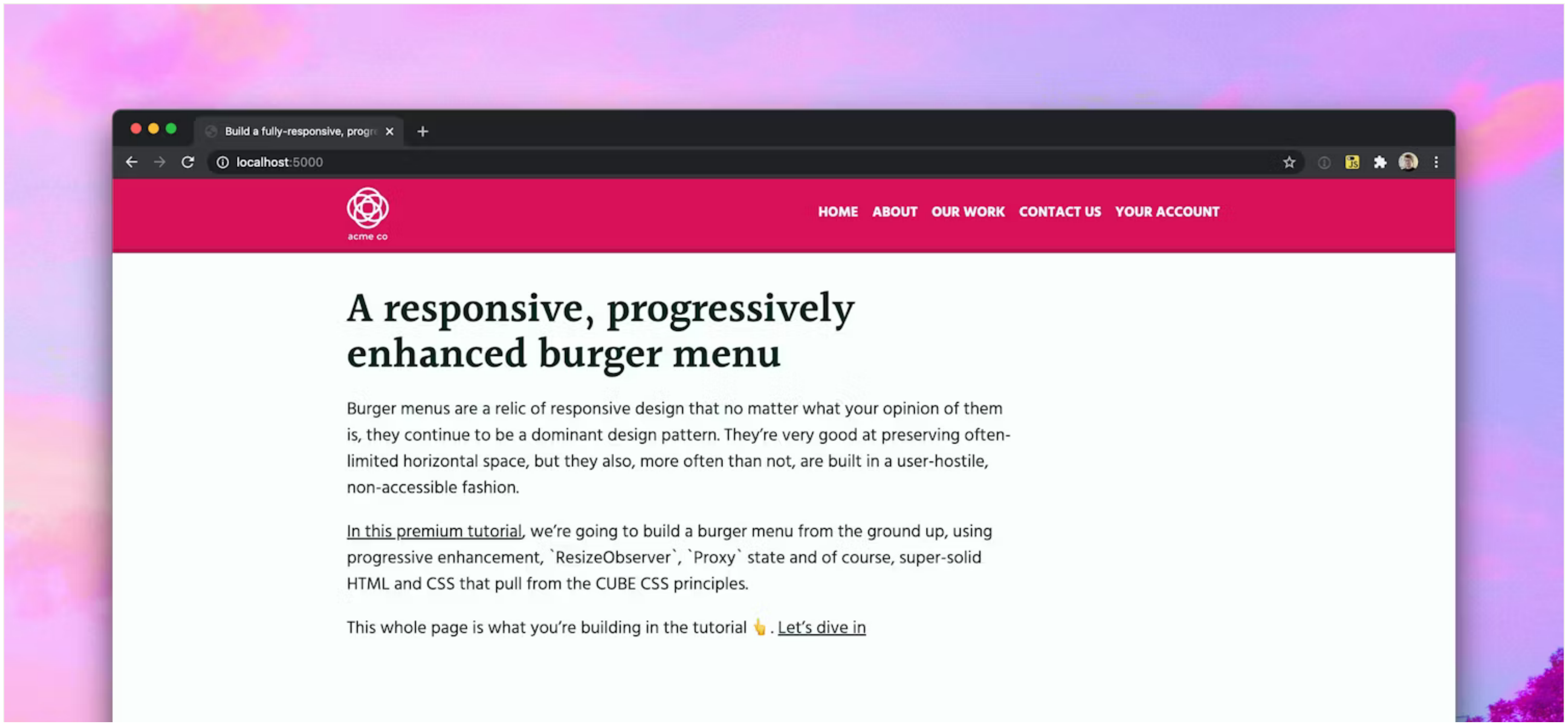
The first thing to note here is that we don’t have any specific styles for the `.navigation` block itself. This is because with CUBE CSS, a block is more of a namespace. Diving into the list element, though, we’re using the same sort of flexible layout approach as we did in the `site-head` block.

We’re using `gap` to space elements and again, allowing items to fall onto a new line if there’s no space. This gives us a handy, extremely acceptable, minimum viable experience, regardless of viewport size.

We do provide a cheeky little fallback for if `gap` isn't supported (at the time of writing, Safari, mainly). By adding a tiny, `0.1rem` margin on **all sides** of a list item. You could use `@supports` to remove this for `gap` support, but remember, **principal of least power**. Also, it's tricky to detect.

The rest of this block is pretty darn self-explanatory and we still have *a lot* to cover, so let's write some JavaScript.

First, take a look at your lush, Minimum Viable Experience:

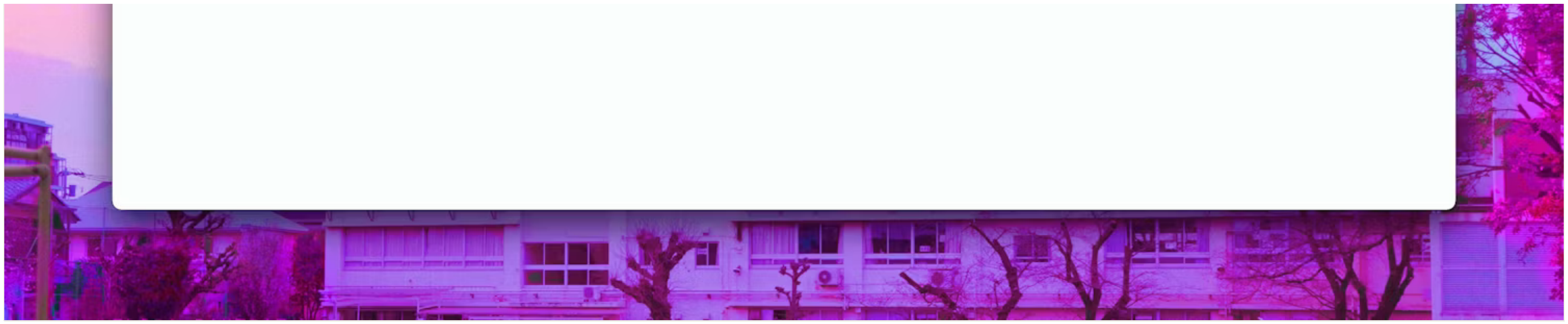


A responsive, progressively enhanced burger menu

Burger menus are a relic of responsive design that no matter what your opinion of them is, they continue to be a dominant design pattern. They're very good at preserving often-limited horizontal space, but they also, more often than not, are built in a user-hostile, non-accessible fashion.

In this premium tutorial, we're going to build a burger menu from the ground up, using progressive enhancement, `ResizeObserver`, `Proxy` state and of course, super-solid HTML and CSS that pull from the CUBE CSS principles.

This whole page is what you're building in the tutorial 👆. Let's dive in



JavaScript

Now to move on to the main course of this tutorial. Have a quick stretch, get a drink, then get comfy, because this is where we're going to really *dig in* to the details.

Burger menu web component

We'll start with the main component itself. Open up `js/burger-menu.js` and add the following.

JS COPY TO CLIPBOARD

```js
class BurgerMenu extends HTMLElement {
  constructor() {
    super();

    const self = this;

    this.state = new Proxy(
      {
        status: 'open',
        enabled: false
      },
      {
        set(state, key, value) {
          const oldValue = state[key];

          state[key] = value;
          if (oldValue !== value) {
            self.processStateChange();
          }
          return state;
        }
      }
    );
  }
  get maxWidth() {
    return parseInt(this.getAttribute('max-width') || 9999, 10);
  }

  connectedCallback() {
    this.initialMarkup = this.innerHTML;
    this.render();
  }

  render() {
    this.innerHTML = `
      <div class="burger-menu" data-element="burger-root">
        <button class="burger-menu__trigger" data-element="burger-menu-tı
          <span class="burger-menu__bar" aria-hidden="true"></span>
        </button>
        <div class="burger-menu__panel" data-element="burger-menu-panel">
          ${this.initialMarkup}
        </div>
      </div>
    `;

    this.postRender();
  }
```

```
}

if ('customElements' in window) {
  customElements.define('burger-menu', BurgerMenu);
}

export default BurgerMenu;
```

We've got a heck of a chunk of JavaScript here, but fear-not, we will go through what is happening.

First of all, we're creating a custom element—a web component. We instantiate a new class which extends `HTMLElement`, which is the basis of **all HTML elements.**

FYI

A custom element uses JavaScript classes. If you've not done much with these already, I definitely recommend that you have a quick break from this (don't worry, we'll wait) to brush up on how they work with this handy resource.

Inside our `BurgerMenu` class: we've got the constructor, which initialises everything when it's loaded. The first thing we do is call `super()`, which tells our extended `HTMLElement` to do the same.

The next part is one of my favourite features of modern JavaScript: Proxies. These are a handy type of object that let us do *loads* of cool stuff, but my favourite part is that we can **observe and manipulate changes**. I wrote up about them in this tutorial and we're using a similar setup to manage our component's `state`.

In a nutshell, we create a `set` method in our `Proxy` which gets fired every time a value of our `state` is changed. For example, if elsewhere in our code, we write `state.enabled = false`, the `set` method of our `Proxy` is fired and it's *in here* where we intercept and commit that change.

Alongside committing that change, we're comparing the old value to the new value. If that value has changed, we fire of a yet-to-exist, `processStateChange()` method. We've got some light, reactive state going on here and it's all baked into vanilla JavaScript. Cool, right?

Next up, we use a handy modern JavaScript feature, a getter to grab the `max-width` property from our `<burger-menu>` instance. This gives the component a `maxWidth` value, which we'll use later to determine wether or not to enable our burger menu.

After this getter, we come across a lifecycle callbacks. These—if defined in your component—fire off at various points in a component's lifecycle. In this instance, the `connectedCallback` fires when our `<burger-menu>` is appended to the document. Think of it as a "ready" state. Now we know our component is connected, we're going to tell our component to `render()`.

Before we do that, though, we store the markup that's inside the `<burger-menu>`, which in our case is the `<nav>` element and it's content. We store it for two reasons:

1. We're going to render that same markup inside our component markup
2. If all fails in this component, we can re-render the markup as if there was no burger menu whatsoever

Moving swiftly onto the `render()` method: we're using a template literal to write out some HTML. You'll notice inside that HTML, we call on our `initialMarkup`, which we just stored. The markup is pretty straightforward. It's a trigger button and an associated panel—similar to a disclosure element.

Lastly, we apply this component using the `customElements.define('burger-menu', BurgerMenu);` line *only* if `customElements` is available—another bit of progressive enhancement. We also `export` the class as a `default` export for if someone was to `import` this component.

Finally, after all this is set, we move on to `this.postRender()` which we will add now. Still in `burger-menu.js`, **under the `render()` method**, add the following:

JS COPY TO CLIPBOARD

```
postRender() {
  this.trigger = this.querySelector('[data-element="burger-menu-trigger"
  this.panel = this.querySelector('[data-element="burger-menu-panel"]');
  this.root = this.querySelector('[data-element="burger-root"]');
  this.focusableElements = getFocusableElements(this);

  if (this.trigger && this.panel) {
    this.toggle();

    this.trigger.addEventListener('click', evt => {
      evt.preventDefault();

      this.toggle();
    });
```

```
      document.addEventListener('focusin', () => {
        if (!this.contains(document.activeElement)) {
          this.toggle('closed');
        }
      });

      return;
    }

    this.innerHTML = this.initialMarkup;
  }
```

There’s quite a bit going on in this method, whose role is to handle the fresh new HTML that’s just been rendered.

The first thing we do is grab the elements we want. I personally like to use `[data-element]` attributes for selecting elements with JavaScript, but really, do whatever works for you and your team. I certainly don’t do it for any good reason other than it makes it more obvious what elements have JavaScript attached to them.

The next thing we do is test to see if the `trigger` and `panel` are **both** present. Without both of these, our burger menu is redundant. If they are both there, we fire off the yet-to-be-defined `toggle()` method and wire up a click event to our `trigger` element, which again, fires off the `toggle()` method.

The next part is an accessibility pro tip. The burger menu—when we finish applying all of the CSS—covers the entire viewport. This means that if the user is shifting focus with their tab key and focus escapes the burger menu itself: they will lose focus *visually*. This is a poor user experience, so this `focusin` event listener on the `document`, *outside* of this component tests to see if the currently focused element—`document.activeElement`—is inside our component. If it isn't: we force the menu closed, immediately.

Lastly, as a last-ditch fallback, we re-render the original markup. This is to make sure that if all fails, the user still gets the minimum viable experience. Don't you just *love* the smell of progressive enhancement?

Let's define that `toggle()` method. Still inside the `burger-menu.js` file, add the following **after** the `postRender()` method:

JS COPY TO CLIPBOARD

```js
toggle(forcedStatus) {
    if (forcedStatus) {
      if (this.state.status === forcedStatus) {
        return;
      }

      this.state.status = forcedStatus;
    } else {
      this.state.status = this.state.status === 'closed' ? 'open' : 'closec
    }
  }
```

In `toggle()`, we can pass an optional `forcedStatus` parameter which—just like in the above focus management—let's us force the component into a specific, finite state: `'open'` or `'closed'`. If that isn't defined, we set the current `state.status` to be open or closed, depending on what the current status is, using a ternary operator.

Now that we've got a state toggle, let's process that state. We'll add the method that is called in our `Proxy`: `processStateChange()`. Still in `burger-menu.js`, add the following **after** the `toggle()` method:

JS COPY TO CLIPBOARD

```
processStateChange() {
  this.root.setAttribute('status', this.state.status);
  this.root.setAttribute('enabled', this.state.enabled ? 'true' : 'false

  this.manageFocus();

  switch (this.state.status) {
    case 'closed':
      this.trigger.setAttribute('aria-expanded', 'false');
      this.trigger.setAttribute('aria-label', 'Open menu');
      break;
    case 'open':
    case 'initial':
      this.trigger.setAttribute('aria-expanded', 'true');
      this.trigger.setAttribute('aria-label', 'Close menu');
      break;
  }
}
```

This method is fired every time state changes, so its only job is to grab the current state of our component and reflect it where necessary. The first part of that is setting our root element's attributes. We're going to use this as style hooks later. Then, we set the `aria-expanded` attribute and the `aria-label` attribute on our trigger. We'll do the actual visual toggling of the panel with CSS.

FYI

We're using a disclosure pattern to do the visual toggling of this component. I wrote a tutorial on that a while ago that explains how the `aria-expanded` attribute works. Check it out!

We're getting close now, pals, hang in there. This is a *long* tutorial, but heck, we are creating something pretty darn resilient. Let's wrap up the JavaScript with some focus management, then get back to the comfort and warmth of CSS.

We referenced a `manageFocus()` method earlier that we need to write, so still in `burger-menu.js`, **after** the `processStateChange` method, add the following:

JS COPY TO CLIPBOARD

```
manageFocus() {
  if (!this.state.enabled) {
    this.focusableElements.forEach(element => element.removeAttribute('ta
    return;
  }

  switch (this.state.status) {
    case 'open':
      this.focusableElements.forEach(element => element.removeAttribute(
      break;
    case 'closed':
      [...this.focusableElements]
        .filter(
          element => element.getAttribute('data-element') !== 'burger-men
        )
        .forEach(element => element.setAttribute('tabindex', '-1'));
      break;
  }
}
```

Here, we look grab our focusable elements (we're doing that bit next) and then depending on wether or not we're in an open or closed, state, we add `tabindex="-1"` or remove it. We add it when we in a closed state because if you remember rightly, this prevents keyboard focus. For the same reason we automatically closed the menu when focus escaped in the open state, earlier, we are now preventing focus from leaking in if it is closed.

FYI

You might be thinking, "what the heck is Andy doing with `[...`??". This is called the spread syntax and what we are doing here is converting our `NodeList` into an `Array` so we can use the `filter()` method to filter out the `trigger` element.

We now need to add a mechanism that enables or disables the burger menu UI, based on the `maxWidth` property. To do that, we're going to use a `ResizeObserver` which does exactly what it says on the tin: observes resizing.

Go back to the `connectedCallback()` method and **inside it**, add the following:

JS COPY TO CLIPBOARD

```js
const observer = new ResizeObserver(observedItems => {
  const {contentRect} = observedItems[0];
  this.state.enabled = contentRect.width <= this.maxWidth;
});

// We want to watch the parent like a hawk
observer.observe(this.parentNode);
```

The `ResizeObserver` gives us a callback, every time the observed element changes size—in our case, the `<burger-menu>` parent, which happens to be `.site-head__inner` —we can monitor it and if needed, react to it.

We destructure the `contentRect` out of the first item in `observedItems` (our `.site-head__inner` element) which gives us its dimensions. We then set our `state.enabled` flag based on wether or not is is less than, or equal to, our `maxWidth` property.

You might think that doing this in JavaScript is daft, but here me out: this is a low-level Container Query! It means this burger menu could be put *anywhere* in a UI. The `ResizeObserver` is *super* performant too, so there's not much to worry about on that front.

Right, let's add the final piece of the JavaScript puzzle: the helper method that finds all focusable elements for us. You can read up on how it works, here.

First up, still inside the `buger-menu.js` file, add the following **right at the top of the file:**

JS COPY TO CLIPBOARD

```js
import getFocusableElements from './get-focusable-elements.js';
```

All we're doing here is importing our helper, so let's go ahead and write that method. Open up `get-focusable-elements.js` and add the following to it:

JS COPY TO CLIPBOARD

```
/**
 * Returns back a NodeList of focusable elements
 * that exist within the passed parnt HTMLElement
 *
 * @param {HTMLElement} parent HTML element
 * @returns {NodeList} The focusable elements that we can find
 */
export default parent => {
  if (!parent) {
    console.warn('You need to pass a parent HTMLElement');
    return [];
  }

  return parent.querySelectorAll(
    'button:not([disabled]), [href], input:not([disabled]), select:not([
  );
};
```

This is just like a CUBE CSS utility, really. It does one job really well for us!

Burger menu CSS

We've got the burger menu interactivity written now and you might be happy to know that we *are done* with JavaScript. If you refresh your browser, it will be a *mess*, so let's make it look good.

Open up `css/global.css` and add the following:

CSS COPY TO CLIPBOARD

```css
.burger-menu__trigger {
  display: none;
}
```

Why the heck are we hiding the trigger? Well, think back to our `state.enabled` flag. If the component is disabled—which is our default state—we don't want to present a trigger. Hiding it with `display: none` will hide it from screen readers, too.

Let’s build the actual hamburger icon. We’ll do it all with CSS, so still in `global.css`, add the following:

CSS COPY TO CLIPBOARD

```css
.burger-menu__bar,
.burger-menu__bar::before,
.burger-menu__bar::after {
  display: block;
  width: 24px;
  height: 3px;
  background: var(--color-light);
  border: 1px solid var(--color-light);
  position: absolute;
  border-radius: 3px;
  left: 50%;
  margin-left: -12px;
  transition: transform 350ms ease-in-out;
}

.burger-menu__bar {
  top: 50%;
  transform: translateY(-50%);
}

.burger-menu__bar::before,
.burger-menu__bar::after {
  content: '';
}

.burger-menu__bar::before {
  top: -8px;
}

.burger-menu__bar::after {
  bottom: -8px;
}
```

The first thing to note here is that we're using good ol' pixel sizes because we really want some control of how this thing is sized. The first thing we do is target the `.burger-menu__bar` (which lives inside the trigger) and both its `::before` and `::after` pseudo-elements and make them *all* look the same as each other: a bar.

After this, we break off and target specific parts—so positioning the `.burger-menu__bar` dead-center with absolute positioning, which allows us to comfortably animate it, knowing it won't affect layout. We then add `content: ''` to both the pseudo-elements so they render and push one up and one down. This gives us our hamburger!

We'll leave this hamburger for now and deal with our `enabled` state in CSS.

Handling the `enabled` state

Our `BurgerMenu` enables and disables itself based on its parent's width and its own `maxWidth` property. We need to handle this state with CSS.

We're going to use the CUBE CSS Exception principle to do this, which means hooking into data attribute values in our CSS. The `BurgerMenu` sets an `[enabled="true|false"]` attribute on our `.burger-menu` component, so let's deal with that.

Still in `global.css`, add the following:

CSS COPY TO CLIPBOARD

```css
.burger-menu[enabled='true'] .burger-menu__trigger {
  display: block;
  width: 2rem;
  height: 2rem; /* Nice big tap target */
  position: relative;
  z-index: 1;
  background: transparent;
  border: none;
  cursor: pointer;
}

.burger-menu[enabled='true'] .burger-menu__panel {
  position: absolute;
  top: 0;
  left: 0;
  padding: 5rem 1.5rem 2rem 1.5rem;
  width: 100%;
  height: 100%;
  visibility: hidden;
  opacity: 0;
  background: var(--color-primary-shade);
  overflow-y: auto;
```

```
    -webkit-overflow-scrolling: touch;
  }
```

Because the burger menu is enabled, we can style up the trigger and the panel. The trigger is a transparent button, because it houses the burger bars we just created. We do make the button considerably bigger than them, though, so there's a decent sized tap target.

For the panel, we make it fill the screen. We set the vertical overflow to be `auto` so long menus can be scrolled. Lastly, we make it hidden by using `opacity` and `visibility`.

FYI

Pro tip: if you set `visibility: hidden`, it will hide the element from a screen reader, so just be aware of that!

Let's add some more CSS. Inside `global.css`, add the following:

CSS COPY TO CLIPBOARD

```css
.burger-menu[enabled='true'] .navigation ul {
  display: block;
}

.burger-menu[enabled='true'] .navigation ul > * + * {
  margin-top: 2rem;
}

.burger-menu[enabled='true'] .navigation li {
  font-size: 1.5rem;
}
```

What we are doing here is converting our navigation into a stacked menu when the burger menu is enabled. This is where the enabled flag is super handy because we don't need to rely on viewport-wide media queries and instead, we have full control over our specific context, instead. Ah, just leave me to dream about Container Queries for a second, will you?

Right, back from my dreaming, let’s wrap up with our interactive states! Add the following to `global.css`:

CSS COPY TO CLIPBOARD

```
.burger-menu[enabled='true'][status='open'] .burger-menu__panel {
  visibility: visible;
  opacity: 1;
  transition: opacity 400ms ease;
}

.burger-menu[enabled='true'][status='closed'] .burger-menu__panel > * {
  opacity: 0;
  transform: translateY(5rem);
}

.burger-menu[enabled='true'][status='open'] .burger-menu__panel > * {
  transform: translateY(0);
  opacity: 1;
  transition: transform 500ms cubic-bezier(0.17, 0.67, 0, 0.87) 700ms, op
      800ms;
}
```

The panel's visibility can only be changed **if** the burger menu is enabled **and** the status is open. This is a great example of finite state giving our UI some real resilience and importantly, reducing the risk of presenting a broken or confusing state to our users.

When the panel is "open", we transition the `opacity` to `1` and set `visibility` to `visible` to show it. I like to only add transitions in the changed state like this. It makes the UI much snappier when elements revert *immediately*.

The last section is pretty cool, too. In the "closed" state, we visually hide the navigation items with `opacity` and push them down with `transform`. When the panel is in the "open" state, we transition them back to being visible with full opacity. It gives us a lovely transition effect.

Right, let's add the *last bit of code*. In `global.css`, add the following:

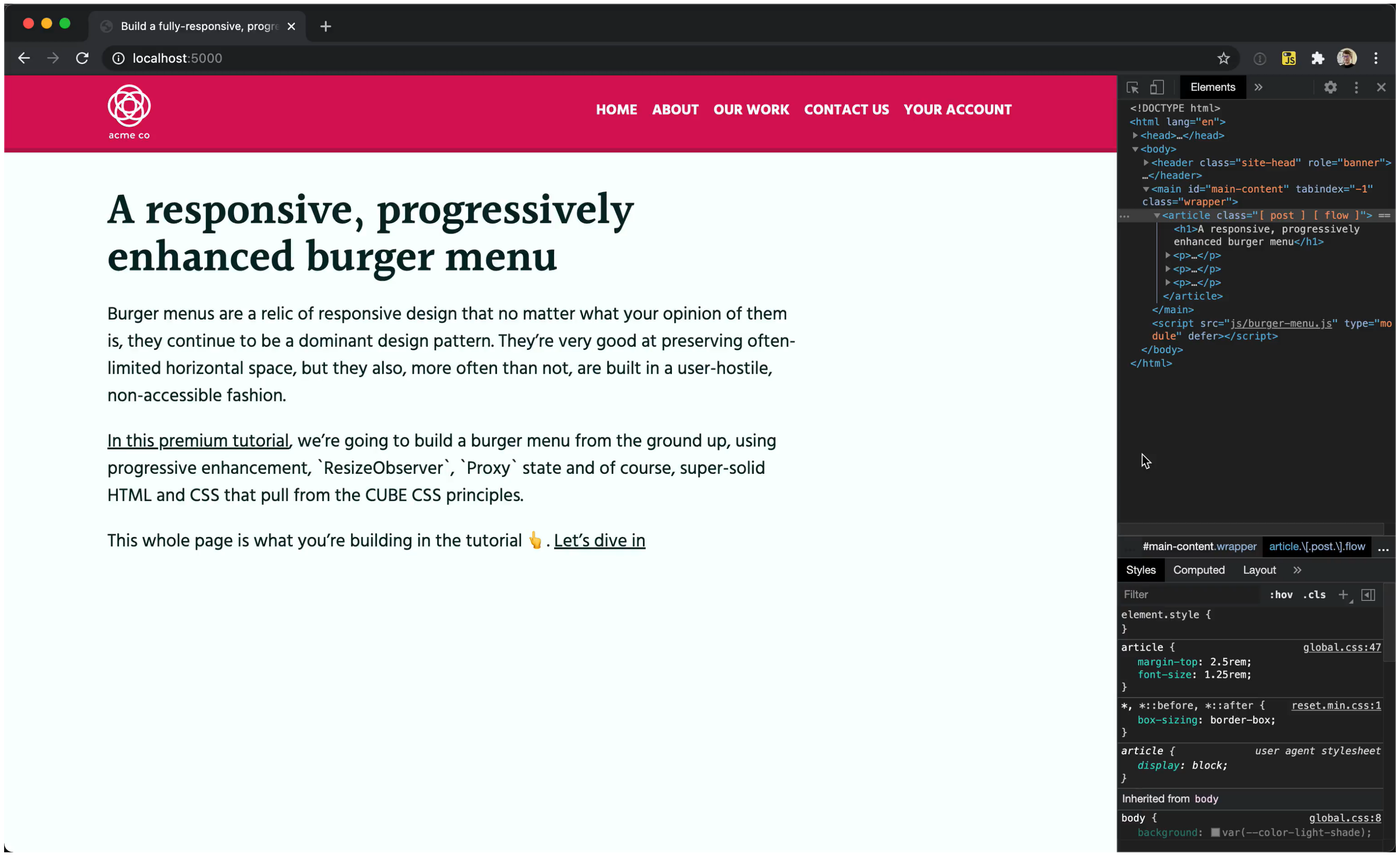
CSS COPY TO CLIPBOARD

```css
.burger-menu[enabled='true'][status='open'] .burger-menu__bar::before {
  top: 0;
  transform: rotate(45deg);
}

.burger-menu[enabled='true'][status='open'] .burger-menu__bar::after {
  top: 0;
  transform: rotate(-45deg);
}

.burger-menu[enabled='true'][status='open'] .burger-menu__bar {
  background: transparent;
  border-color: transparent;
  transform: rotate(180deg);
}
```

This is our burger bars converting themselves into a close icon when the menu is open. We achieve this by first, setting the background and border of the main (central) bar to be transparent, then we rotate the pseudo-elements in opposite directions to create a cross. Lastly, we spin it all around by transitioning the whole thing 180 degrees.

Again, we're using our state to determine this, which admittedly, creates some gnarly selectors, but it also provides resilience and solidity.

With that, **we are done**. If you refresh your browser, resize it and toggle the menu, it should all look like this.



A responsive, progressively enhanced burger menu

Burger menus are a relic of responsive design that no matter what your opinion of them is, they continue to be a dominant design pattern. They're very good at preserving often-limited horizontal space, but they also, more often than not, are built in a user-hostile, non-accessible fashion.

In this premium tutorial, we're going to build a burger menu from the ground up, using progressive enhancement, `ResizeObserver`, `Proxy` state and of course, super-solid HTML and CSS that pull from the CUBE CSS principles.

This whole page is what you're building in the tutorial 👇. Let's dive in

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▶<header class="site-head" role="banner">
   …</header>
   ▼<main id="main-content" tabindex="-1"
   class="wrapper">
     ▼<article class="[ post ] [ flow ]"> ==
        <h1>A responsive, progressively
        enhanced burger menu</h1>
       ▶<p>…</p>
       ▶<p>…</p>
       ▶<p>…</p>
      </article>
    </main>
    <script src="js/burger-menu.js" type="mo
    dule" defer></script>
  </body>
</html>
```

```
element.style {
}
article {                              global.css:47
   margin-top: 2.5rem;
   font-size: 1.25rem;
}
*, *::before, *::after {          reset.min.css:1
   box-sizing: border-box;
}
article {                    user agent stylesheet
   display: block;
}
Inherited from body
body {                                  global.css:8
   background: var(--color-light-shade);
```

Wrapping up

That was a hell of a long tutorial. Fair play to you for sticking it out! If your version is broken, you can go ahead and download a complete copy, here. You can also see a live version, here.

DOWNLOAD FINAL VERSION

I hope you've learned a lot in this tutorial, but the main takeaway I'd love you to go away with is that even seemingly simple interactive elements—like burger menus—get really complex when you make them fully inclusive. Luckily, progressive enhancement makes that inclusivity a little bit easier.

Until next time, take it easy 👋

By Andy Bell

Designer and developer. Founder of Piccalilli and Set Studio.

» More about Andy Bell

Topic(s). HTML CSS CUBE CSS JAVASCRIPT PREMIUM PROGRESSIVE ENHANCEMENT A11Y

Take your CSS skills beyond the next level, joining over 750 others who are taking our Complete CSS course

Advertise with Piccalilli

NEWSLETTER

The Index

Join over 2400 subscribers and discover our twice weekly newsletter, featuring high quality, curated design, dev and tech links.

Short.	*~5 links, twice weekly*
Digestible.	*Readable in ~1–2 mins*
Curated.	*Good links, curated by humans, not AI*
Free.	*Zero cost, and no spam, ever*

Enter your email

SUBSCRIBE

SUBSCRIBE VIA RSS

Powered by Buttondown and Postmark - Privacy policy

Piccalilli