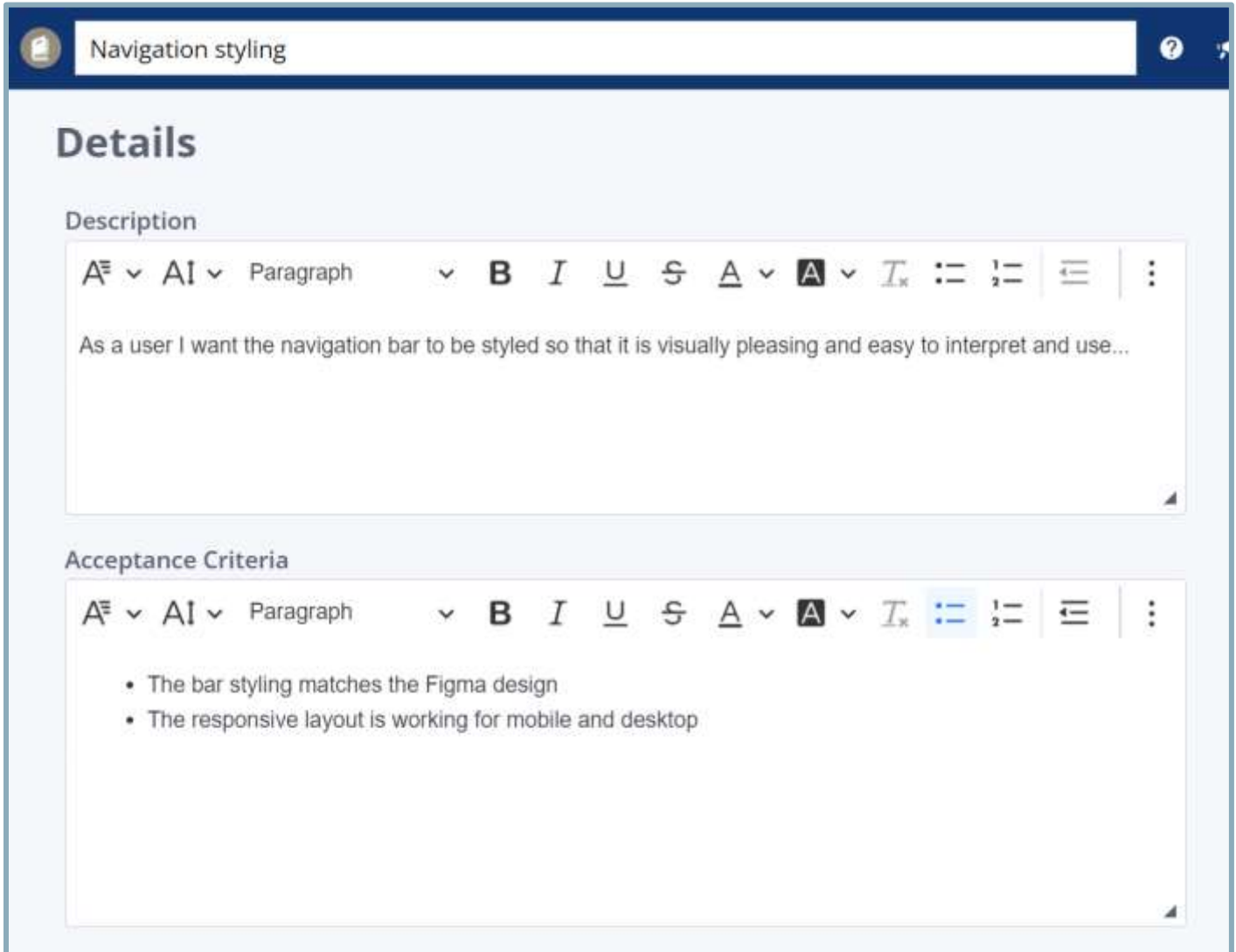


Build the Book Binge App - L002

Lesson Two: Add styles and responsive behavior

Here's our next user story for building the Book Binge app:



Navigation styling

Details

Description

As a user I want the navigation bar to be styled so that it is visually pleasing and easy to interpret and use...

Acceptance Criteria

- The bar styling matches the Figma design
- The responsive layout is working for mobile and desktop

According to our Figma, the Navbar should look like this:



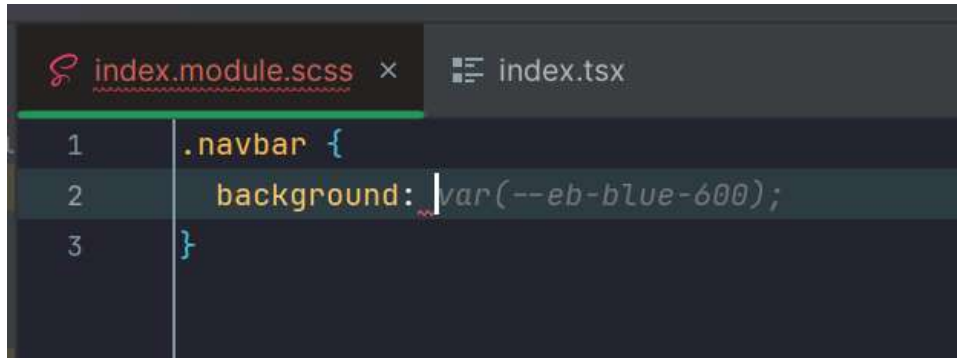
Our current navbar looks nothing like this.

We'll have to rely on the Figma design to fix it. If you haven't already found the **BookBinge.fig** file inside the Figma folder of the repo and uploaded it into your Figma client, do so now.

1. Make a new feature branch. You may wish to check out last lesson's branch (*001/lesson-001-navbar*) and branch off from that.
2. To begin, we'll make the navbar container that's green and stretches across the whole screen. See if you can figure out how to use Figma to identify the variable name of the green background color. You may have to click through the elements on the left-hand side of the Figma client.
3. Once you've found the color variable name (it should start with a double-dash (--), let's apply it as a background color. In the 'navbar' folder, create a file called **index.module.scss**.

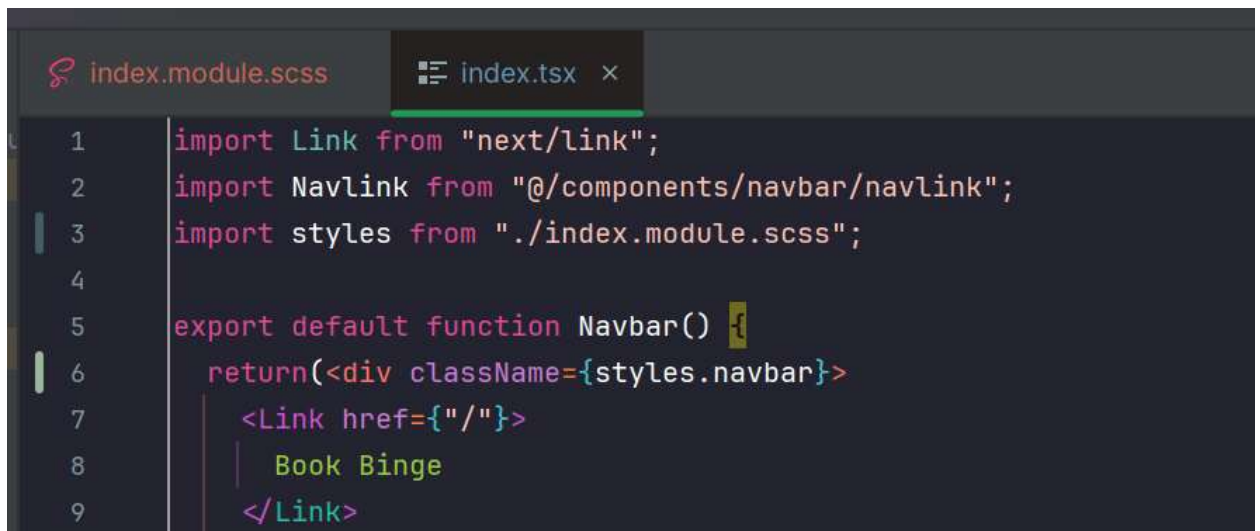
SCSS files always follow the pattern **[name-of-the-file-you're-styling].module.scss**. SCSS (Syntactically Awesome Style Sheet SCSS (Sassy CSS)) is a preprocessor for CSS that provides additional features not available in regular CSS. The syntax is very similar, but it allows for the use of variables, nesting and mixins. If you're feeling shaky on it, the official SASS guide has a good overview: <https://sass-lang.com/guide/>

4. Let's create our first style class **.navbar {}** and give it a **background** property.



```
1 .navbar {
2   background: var(--eb-blue-600);
3 }
```

5. All the variables for color are already predefined in the *global.scss* (which provides styles *globally*). Apply variables using the syntax, **var(--variable-name)**. Try to apply the name of the green color from the Figma design.
6. To see your work, you'll have to import your styles into the component and assign the correct *className* to the container div.



```
1 import Link from "next/link";
2 import Navlink from "@components/navbar/navlink";
3 import styles from "./index.module.scss";
4
5 export default function Navbar() {
6   return(<div className={styles.navbar}>
7     <Link href={"/"}>
8       Book Binge
9     </Link>
```

7. If you applied the variable correctly, you should see a nice green bar across the top of the page when you `npm run dev` and go to `localhost:4000`.
8. Now let's take care of the app logo, *Book Binge*.

We'll take advantage of SCSS's nesting capabilities and create an *appName* class inside of the *navbar* class. We're doing this mainly for our own semantic organizational purposes. It would work the same if we didn't nest it--in this situation. However, CSS uses the concept of "specificity" to determine the priority of styles, and sometimes if you are having trouble overriding a 3rd party library style, the easiest way to get your styles to take priority is to make them more specific by nesting them.

The properties we'll need to define on our *appName*, in no particular order, are:

- Color
 - Font-family
 - Font-size
 - Font-weight
9. Once again, you'll have to examine the Figma to figure out these values. You may also wish to view the UI library (`localhost:4000/ui-library`) to see examples of available font families. The color will again be a variable name, as with all color properties throughout the application. The font size appears in Figma's right-hand details panel as a pixel value. The font-weight is a semantic term (*ExtraBold* in this case). How to apply this to our SCSS?

A note about **rem**: Use rem to define font sizes. Rem stands for "root em" and when you use it, it means the font size will scale according to the root size of "em" (historically, this meant the width of an upper-case M). In most browsers, this is 16 pixels. In other words, 1 rem = 16 pixels. In addition to fonts, you may decide to use rem for padding and margins too—anything you want to scale with viewport size and still maintain proportion—on a case-by-case basis. But it is especially important for fonts to better allow users to control the size of fonts in their browsers.

Does this mean you need to constantly divide by 16 to figure out how many rems something should be?

Well, yes. Unless the design team provides you with the information in rem, you'll have to do the math. Luckily, plenty of tools exist to speed this up. Check out this site: <https://nekocalc.com/px-to-rem-converter>

You may also be able to use a `rem()` function, or create one yourself:

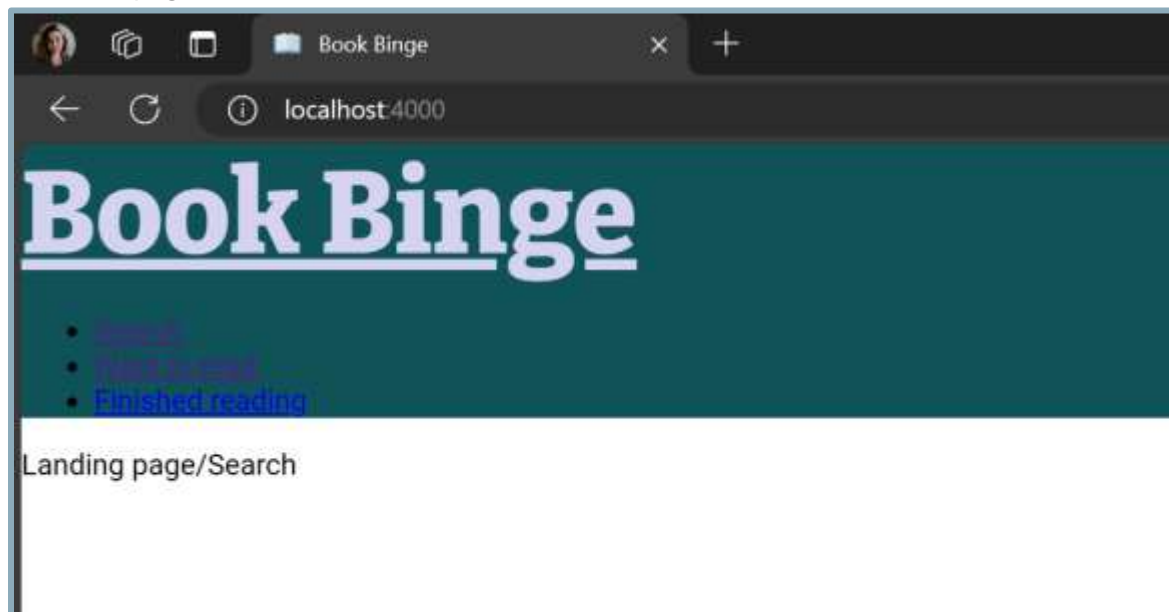
<https://medium.com/@bhargav3shah/scss-convert-pixel-values-to-rem-using-functions-f1cef575edfd>

The next issue you're likely to run into is the font-weight. If you try to put "ExtraBold" or "extra-bold" in the font-weight property, you'll get an error. What gives?

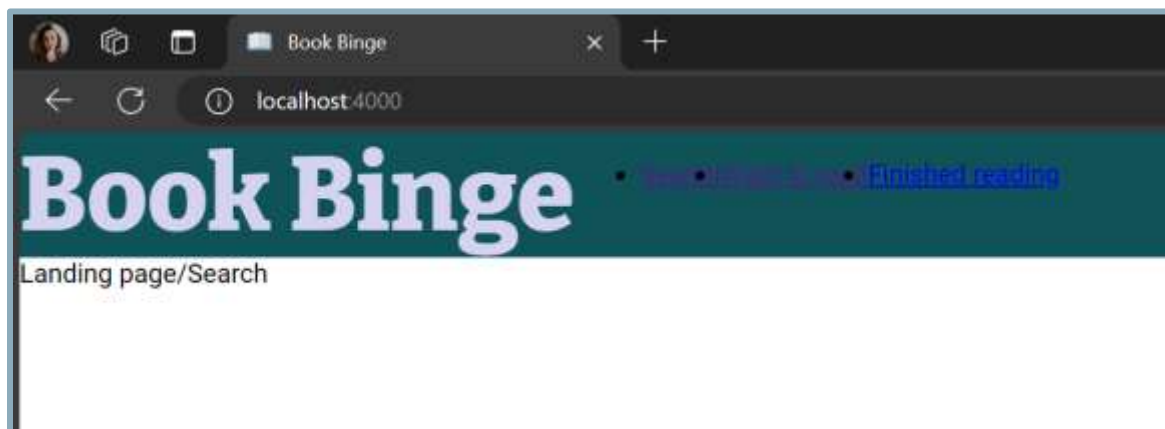
This is yet another thing you must translate. Use the MDM docs to find the solution:

<https://developer.mozilla.org/en-US/docs/Web/CSS/@font-face/font-weight>

10. OPTIONAL: Did you figure out how to set the font weight? It turns out that semantic weight names have numeric equivalents, which are what we really want to use. If you want to be proactive, you could add some variables to the `_variables.scss` to make assigning font weights easier for Future You.
11. Assuming you have correctly set the color, font-family, font-size, and font-weight, your home page should now look like this:



12. Wait a minute! Why is it underlined? We didn't set that! –It's because *Book Binge* is a **link**, and that's how browsers style links by default. Learning to override default styles is a crucial skill and you might as well start practicing now.
- No hints this time. Use your Google-fu and figure out how to make that line disappear.
13. For our next challenge, let's get those nav links to spread out in a nice row instead of a vertical list. The magic words are **display: flex**. 🙌 Add them to the style classes for two *different elements* in the Navbar and your result should be this:



If you've never heard of Flexbox, you might be wondering what's going on. For a deep dive, check out this wonderful guide from *CSS-Tricks*: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

Flexbox (and something we'll touch on in a future lesson, Grid) are important concepts to grasp. You're likely to need Flexbox more often than Grid (which, as the CSS-Tricks guide mentions, is more suited to large page-layout flows), but you will absolutely encounter both, frequently. If you feel confused about what Flexbox can accomplish or want to practice until using it feels second-nature, check out the classic game of **Flexbox Froggy**: <https://flexboxfroggy.com/>

So, did you figure out where to **display: flex**? 🐸 A key concept for making flex work is understanding the hierarchy between container and children. The flex goes on the container and affects its children (but *not* any nested children). By putting flex on the top-level div of the **Navbar**, we can make the *Book Binge* logo and the move into a row together. By adding it to the as well, we move the links into a row together. Your SCSS should look something like this:

```
1  .navbar {
2    background: var(--midnight-green);
3    display: flex;
4
5    .appName {
6      color: var(--lavender);
7      font-family: "Bitter", serif;
8      font-size: 4rem;
9      font-weight: 800;
10     text-decoration: none;
11   }
12
13   ul {
14     display: flex;
15   }
16 }
```

If you gave the a class name instead of accessing it by element role, that is fine as well.

14. Continuing with Flexbox, see if you can get the of links to move over to the right side of the screen, and align with the bottom of the navbar. HINT: You can do this by adding two properties to the **.navbar** class. If you're at a loss, it's time to play some **Flexbox Froggy**.



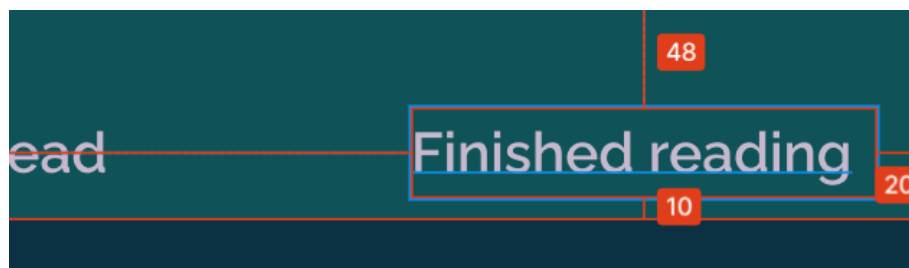
Here's another hint: The two properties you need to add are *justify-content* and *align-items*. Get comfortable with these and how they differ, as you will use them a lot.

15. Bust out your research skills once more and figure out how to get rid of the unordered list bullets.
16. You already know how to get rid of the underlines. To fix the Navlinks, it's time to add a new *index.module.scss* file to the 'navlink' directory. Create a style class for the navlink that removes the underline. Then import it into the **Navlink** component and apply the style to the `<Link>` element's `className` property.

What about the underline that shows in the Figma when the page is selected? Don't worry about that for now. We'll get to that in a later lesson.

17. Now that we have a class for our nav links, let's also style their color, font-family, font-weight, and font-size, just like we did for *Book Binge*. Once again, refer to the Figma.
18. Even though we're not doing the underline effect while a page is selected (yet), we should still add an effect so that when a user hovers their mouse over the link, the underline appears. You can achieve this by using the **:hover** pseudo-selector. In SCSS syntax, this is can be added by nesting **&:hover {}** within a style class block. Go ahead and add the underline text-decoration to the hover block.
19. We're getting close, but the spacing is still wrong. As you may know by now, Flexbox offers many tools to adjust how things are spaced—but in this case, while those properties can get us close, none are quite right. The link array clearly needs to stay towards the left, and they're spaced differently between each other than they are from the *Book Binge* logo. The easiest solution is simply to apply margins to the nav links to space them out from each other. Note that we also need a bit of margin to the left of the *Finished reading* link.

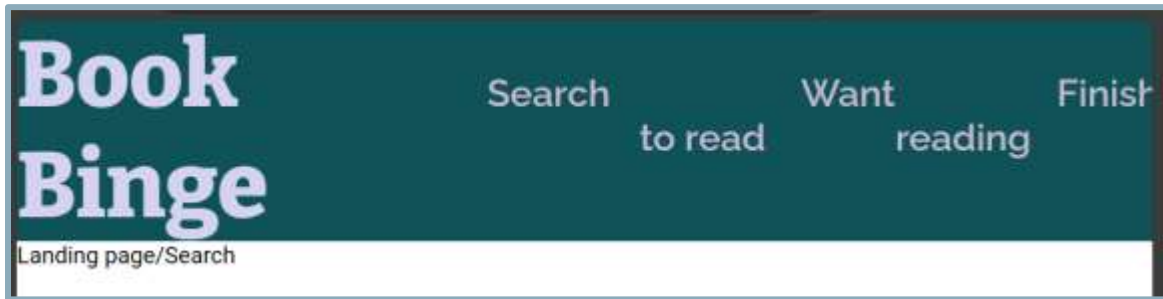
FIGMA TRICK: When you have an element selected in Figma, you can hold down the *alt* key to see spaces measured in pixels! Move your mouse around while holding *alt* to measure the distance to other elements.



20. To get the spaces correct, you may add both a left and a right margin to the Navlink style class—or you can use shorthand, e.g., **margin: top right bottom left**.

Another tip: We only need the *right* margin on the *Finished reading* link, so you might decide to use a CSS pseudo-selector for the last list element to add right margin to that link alone (**:last-child**), OR you could simply split the difference with the *left* margin and put it on all the elements (so that the final space between them is the same). Either approach is fine.

21. When you're done, the result should look a lot like the Figma. But... What happens when you make your browser window narrower? Go on, shrink it down:



Uh oh! That looks bad! What to do?

22. Flexbox exists for this very reason. We just need to tweak it a bit. Specifically, we need to tell it to allow the child elements to *wrap*. Add **flex-wrap: wrap;** to both your flex containers.

23. We now have a navbar that is *responsive* to the size of the viewport without breaking down.



24. This is pretty good, but it can be better. While we don't have a mobile design from the UX Team, we can take it upon ourselves to apply standard practices for mobile-friendly design. We'll do this by using the **@media** rule to tell our CSS to apply different properties up to the max width of a mobile device viewport (we'll use our *\$phone-max* variable, which equals 766px).

To get this to work, you'll need to import the variables into to your SCSS files. At the **Navbar** level, it looks like: **@import "../_variables.scss";**

You'll have to navigate up one directory higher from inside the **Navlink** style sheet.

Now you can use the *\$phone-max* variable to make a media query in your style block. For example, try adding this to your **.navbar** style class:


```
@media (max-width: $phone-max) {  
  justify-content: center;  
}
```

This tells the Flexbox to align the child elements along a *center* axis vertically (since we are wrapped into a column at small screen widths), as opposed to using *space-between* on the horizontal axis (when the screen is wide enough for elements to lie in a row).

Again, if you're feeling lost here, **Flexbox Froggy** is your friend!

25. Add **@media** rules to some of the other divs: Make the *Book Binge* logo appear at 3rem at mobile screen size; give the `` an explicit flex-direction of column so we avoid any weird wrapping and set its padding to 0 (zero) to override the default; finally, inside the **Navlink**'s style class, set margin to 0 (zero) and set the line-height to 3rem to make the list more spacious for finger-tapping.

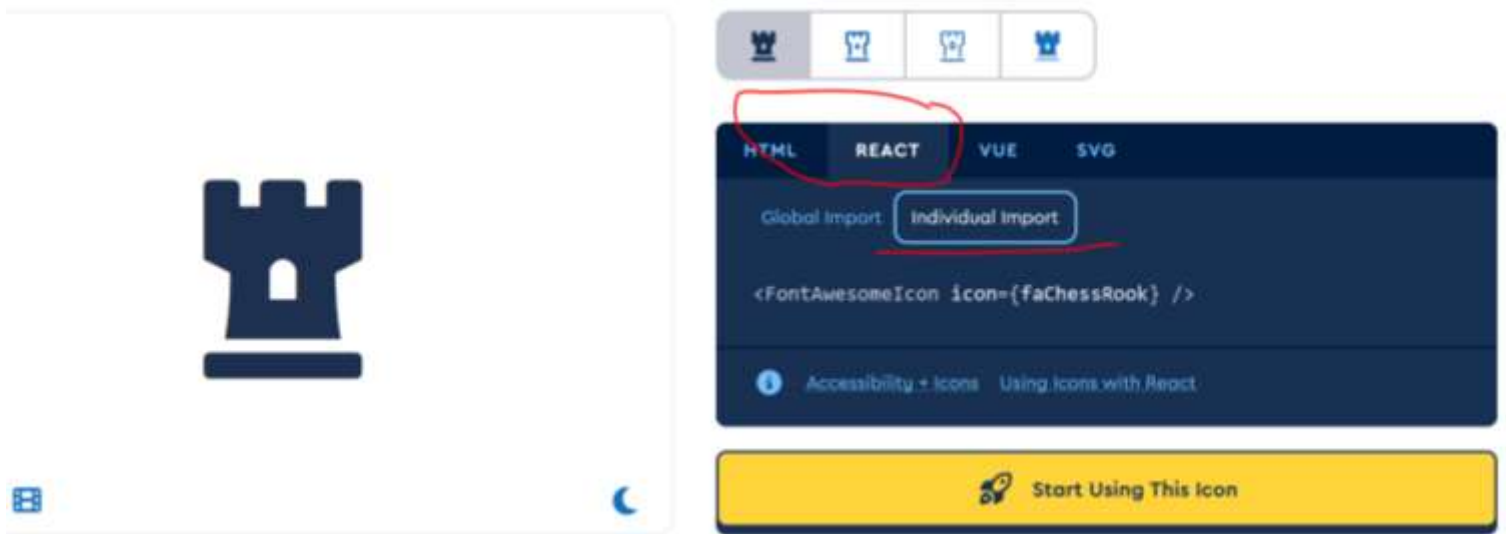
Use your browser's responsive viewer to check how the app looks on an iPhone or a Pixel. It should look something like this:



26. There is still one big thing missing. The *Book Binge* icon! The icon is from the free icon library, Font Awesome, and this project comes with Font Awesome installed already. In our case, the UX Team didn't leave any notes for us to identify the name of the icon we need, so we'll have to look it up. Navigate to the Fontawesome 5 free fonts page and search the icons for a "book" that looks like the one in the logo:

<https://fontawesome.com/v5/search?q=book&o=r&m=free>

Once you find it, click on it and open it's "React" import tab, and *individual import*:



27. To import a Font Awesome icon into a component, follow the most specific import path you can. For the "chess-rook" (faChessRook) icon pictured above, the import would be:

```
import {faChessRook} from "@fortawesome/free-solid-svg-icons/faChessRook";
```

This is because we only need a couple of icons for our application. If we stopped at `import {faChessRook} from "@fortawesome/free-solid-svg-icons";` the whole library of solid icons would load into our application instead of just the one we need. Keep the app lightweight!

28. Once you've imported the book icon into the **Navbar**, you can place it inside the component using the exact same syntax that Font Awesome shows you in the example code. You'll also need to import the `FontAwesomeIcon` component from `"@fortawesome/react-fontawesome."` In the case of the chess-rook, the code would look like this:

```
<FontAwesomeIcon icon={faChessRook} />
```

29. When you're done, your navbar should show a tiny black book:

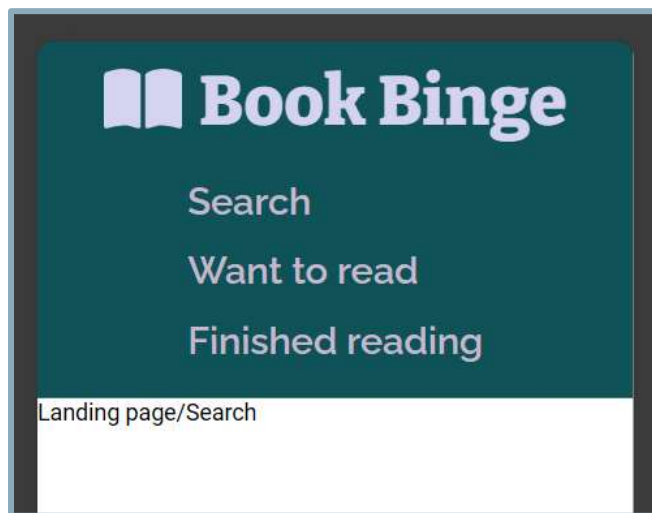


30. Add a class for the icon in the **Navbar** stylesheet and give it a color and a font-size (yes, Font Awesome icons behave like fonts) per the Figma design. Also give it some margin to space everything nicely (remember the Figma *alt* trick).

NOTE: The updated version of the Font Awesome library used in this project has an open book icon that looks slightly different from the one in the Figma. Don't sweat it.

31. Last but not least, let's give that icon some responsive styling to match the rest of the logo. Add an **@media** query to the icon's style class, set the font-size to match the *Book Binge* logo, and set the margin to `1rem .75rem 0 0`;

When you're done, you should have a navbar that matches the Figma very closely. Check out the lesson branch to see the final code.



EXTRA CREDIT:

If you have extra time, learn about the differences between SASS selector nesting and native CSS nesting: <https://sass-lang.com/blog/sass-and-native-nesting/> (5 min read)

Also take a moment to study up on best practices for SASS nesting: <https://fedmentor.dev/posts/sass-nesting/> (5 min read)