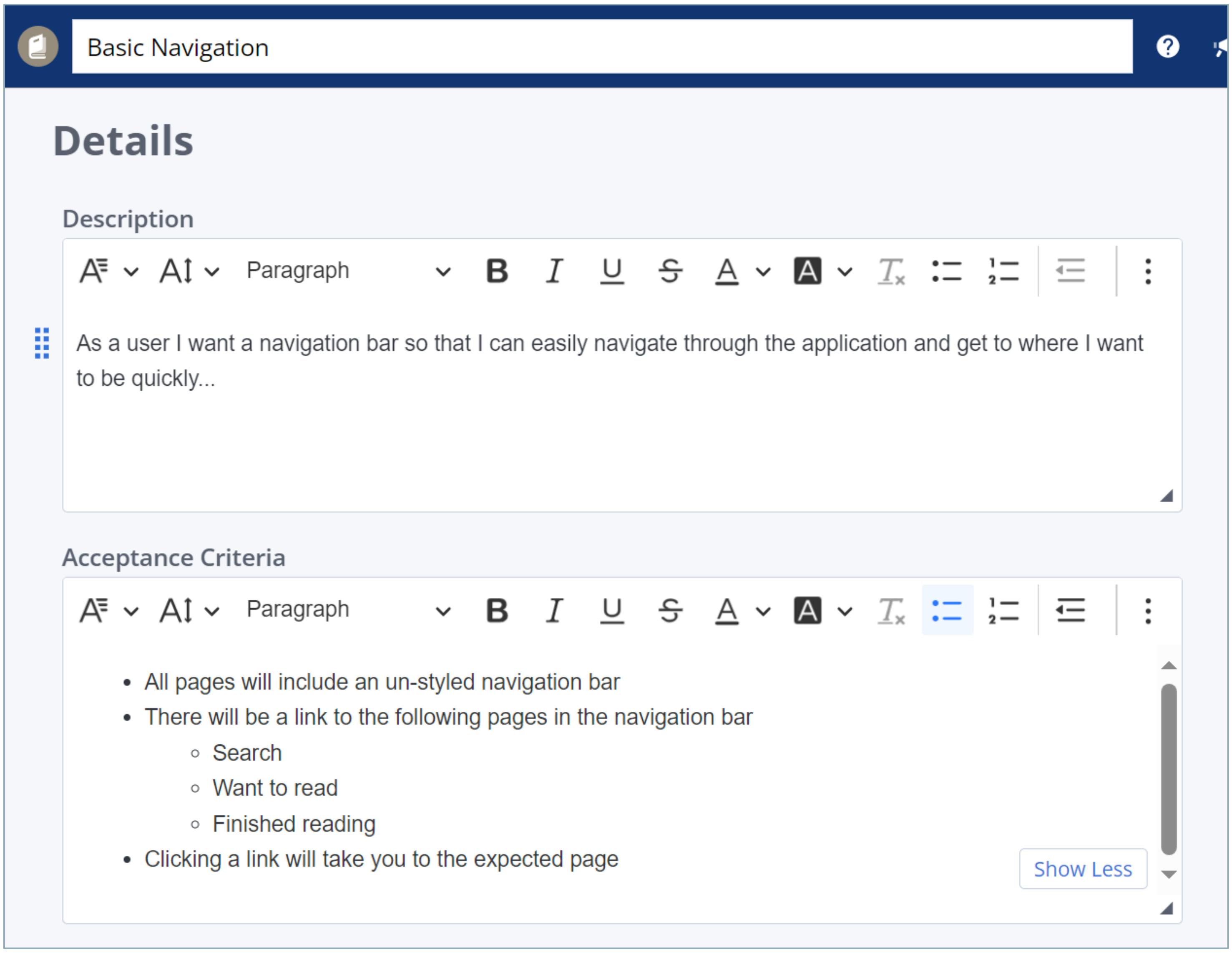
Build the Book Binge App – L001

Lesson One: Create a Navbar and add *Next* Pages

Let's pull in our first user story for the Book Binge app. It covers the basic Acceptance Criteria for a functioning navbar.



Basic Navigation

Details

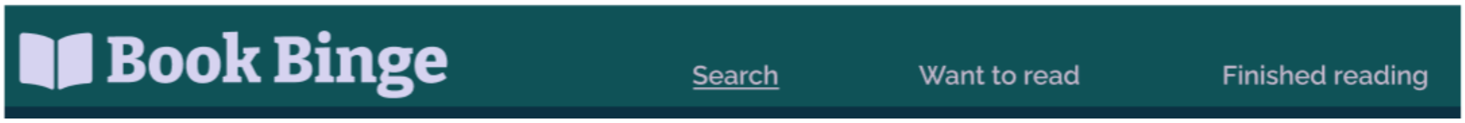
Description

As a user I want a navigation bar so that I can easily navigate through the application and get to where I want to be quickly...

Acceptance Criteria

- All pages will include an un-styled navigation bar
- There will be a link to the following pages in the navigation bar
    - Search
    - Want to read
    - Finished reading
- Clicking a link will take you to the expected page

For reference, the Figma design shows the final version of the navbar, though for our first story, we're making an un-styled version, focusing on functionality first:



It looks like we'll need a div at the top of the screen to contain the app name (Book Binge), along with three navigation links (*Search*, *Want to read*, and *Finished reading*).

The three repeated elements all have the same functionality: click them to show a different page. This is a good opportunity to practice making a reusable component, simple though it is.

We'll create two new components, one for the navbar, and one for the nav links.

1. Make a new feature branch.

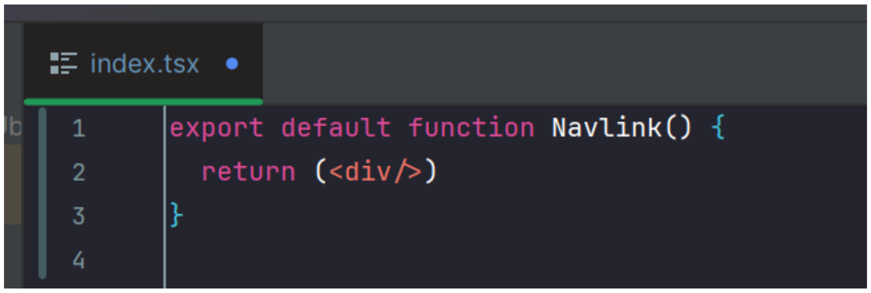
2. Inside the */components* directory, create a */navbar* directory and add an *index.tsx*.

Index files get automatically imported from a directory, which is why it's extremely common to use them. But there is nothing stopping you from naming this file *navbar.tsx* instead. The difference is ***`import Navbar from "../navbar"`*** versus ***`import Navbar from "../navabar/navbar.tsx"`***—not a big deal. But we'll follow the index convention.

3. Create a new React component in the index file and call the component **Navbar**. Export it using either *`export default function`*... or, if you created it as an arrow function, remember to export it at the bottom of the file.

4. Create a */navlink* directory inside the */navbar* directory and give it an *index.tsx* file. We're putting the **Navlink** component inside the */navbar* directory because **Navbar** is the only component that will use **Navlink** and it helps keep our */component* directories uncluttered. In the *index.tsx*, create a React component called **Navlink** and export it.

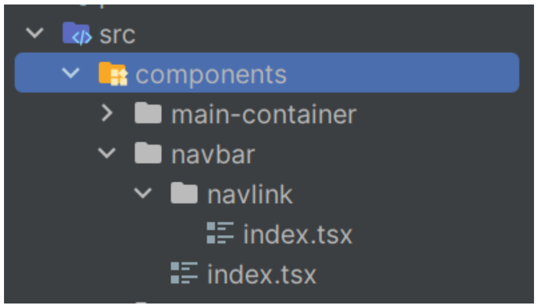
NOTE: Your linter may complain if your components are empty, so just return an empty div:



```
export default function Navlink() {
  return (<div/>)
}

```

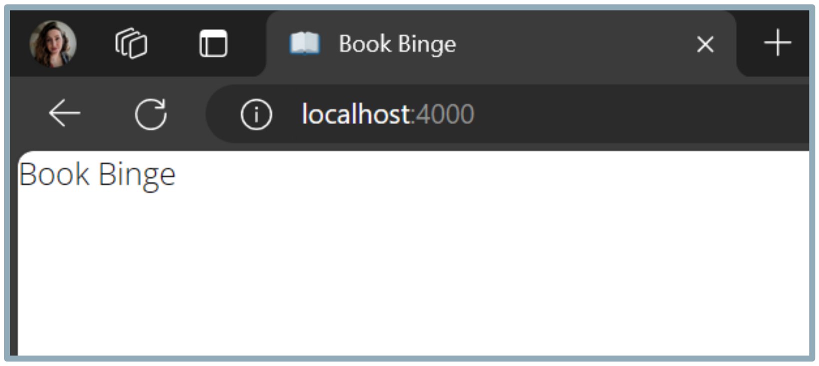
At this point, your file structure should look like this:



```
src
  components
    main-container
    navbar
      navlink
        index.tsx
      index.tsx
```

5. Try your hand at creating a rudimentary **Navbar** component. For now, this will be just a div parent with a div child inside it to hold the app logo text. It will not look like the Figma until we style it. That will happen in the next lesson.

6. To see your work, you'll have to import it into the **MainContainer**, which serves as the entry point into the root of the application. You can delete the other contents in the main container as we won't be needing them. Place the **Navbar** in the **MainContainer** and see if your app title shows at *localhost:4000* (`npm run dev`).



Time to add the links. We are planning to reuse the same component, **Navlink**, for all the links on the navbar. For that to work, **Navlink** will have variable properties.

Q: What might these variables be? What properties are changing?

A: *The links have two variables. One is the text the link displays, and the other is the location the link navigates to. Knowing this, we know the* ***props*** *we need to pass.*

Since we're making a component that takes props (properties), we'll need to provide a type for those props to satisfy TypeScript. We'll use TypeScript's *Type* syntax rather than *Interface*, though it wouldn't be wrong either way. *Type* is usually a stronger choice when working in React because it's more suited to functional programming and allows Union types, which are frequently useful for props. You can read up on the differences in this insightful *Log Rocket* post (10 min read): https://blog.logrocket.com/types-vs-interfaces-typescript/

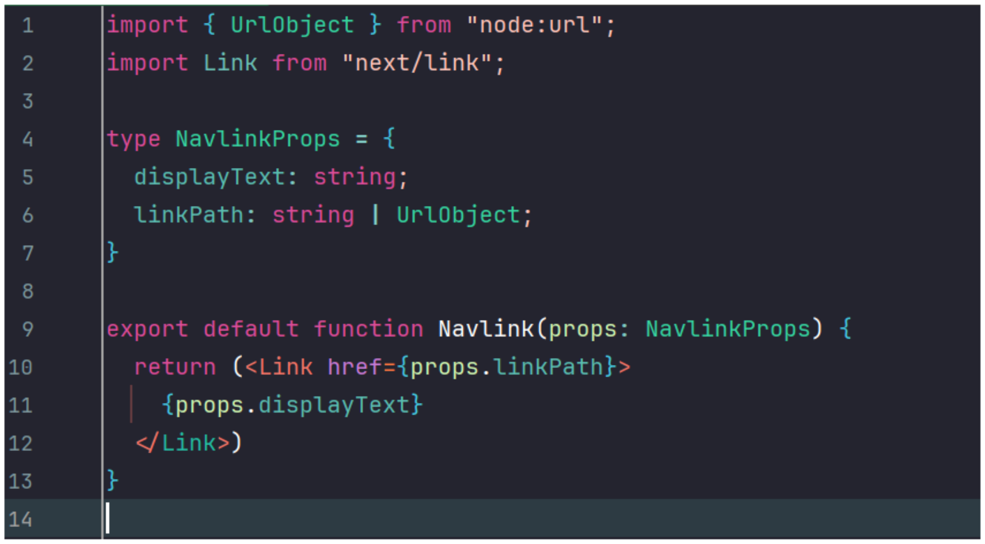
7. Try creating a type for the props and pass them into the component. It's probably easy to guess the type for the link's text, but what to do about the link's path? How do we even make a nav link?

   Next.js makes this easy. Consult the documentation for an example: https://nextjs.org/docs/pages/api-reference/components/link

When you're done, you should have a component (imported from the *Next* library) that accepts *displayText* and *linkPath* props (or whatever you decided to call them) and uses them in a simple component. The component should display the text and redirect to the path provided. To determine what type to use for *linkPath*, drill down into the type that the **<Link/>** component expects for the *href* property.

****A note on destructuring props:** *Props* are an Object that React creates to pass information from parent to child, like an argument passed to a function. It always uses the keyword "props," and some developers prefer to use that keyword to accept the data while others prefer to destructure the props as they come in. You will encounter both approaches as pictured below, and neither is "correct"—it's a stylistic choice.

Standard props argument



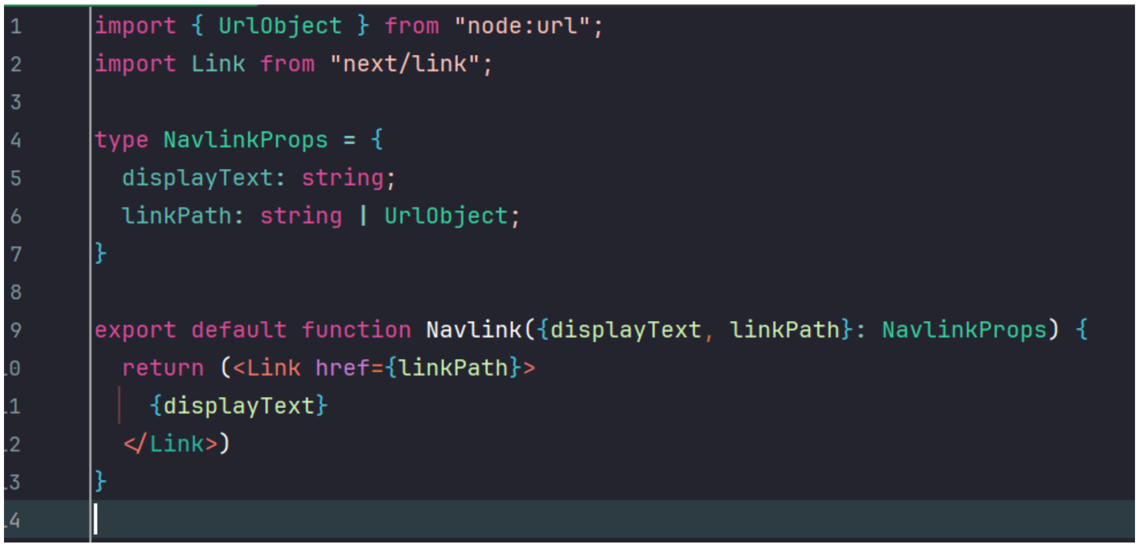
```
import { UrlObject } from "node:url";
import Link from "next/link";

type NavlinkProps = {
  displayText: string;
  linkPath: string | UrlObject;
}

export default function Navlink(props: NavlinkProps) {
  return (<Link href={props.linkPath}>
    {props.displayText}
  </Link>)
}

```

Destructured props argument



```
import { UrlObject } from "node:url";
import Link from "next/link";

type NavlinkProps = {
  displayText: string;
  linkPath: string | UrlObject;
}

export default function Navlink({displayText, linkPath}: NavlinkProps) {
  return (<Link href={linkPath}>
    {displayText}
  </Link>)
}

```

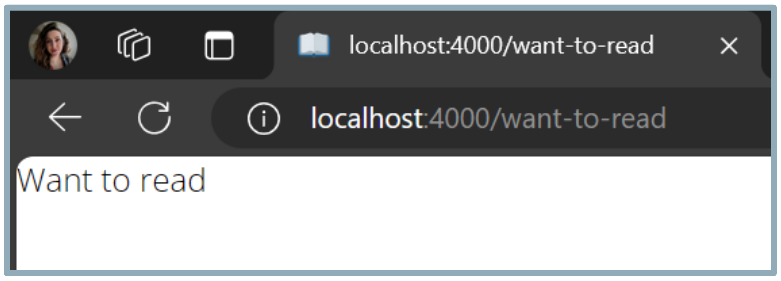
So you made a **Navlink** component. How can you test that it's working? It's time to add a new page.

There's more than one way to configure a *Next* app, but our implementation uses a directory called */pages* to organize—you guessed it—the application's pages. Simply add a new page and then navigate to it by appending the name of the page to your URL, easy as **localhost:4000/new-page**.

In a distributed micro-frontend architecture, each page might have its own repository. A container component remotely imports the child component's *exposed* contents or functionalities, appearing to the end user no different than a monolithic frontend with multiple pages. The details of micro-frontend architecture are beyond the scope of this curriculum, but suffice it to say, the */pages* configuration is only one way to set up a *Next* app's entry points. The application could just as easily be a constellation of components, each with an *exposes* configuration for remote entry, with the */pages* root used only for local development.

According to our ACs, we need a page called *Want to read*. Let's make that first.

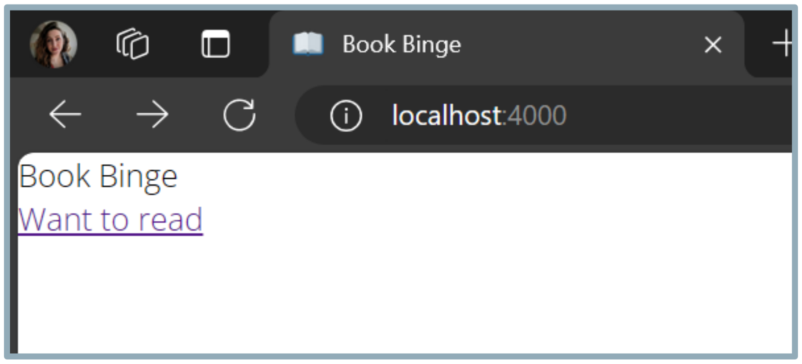
8. In the */pages* directory, create a new directory called */want-to-read* and inside that, create an *index.tsx* file. Create a component, **WantToRead**, that returns a div with the text "Want to read" inside it. Now, if you navigate to *localhost:4000/want-to-read,* you should see this:



Want to read

9. Let's wire it all together. Go to your **Navbar** component and add a new div under the one that contains the logo text. Both the logo text div and new div should be wrapped inside the outer container div. Import your **Navlink** component and add it inside the new div. See if you can figure out what values to pass in for its *displayText* and *linkPath* props.

   Clicking the *Want to read* link shows the new page as picoted above, while your home page should now look like this:



Book Binge
Want to read

10. However, there is a problem. According to our ACs, the navbar should show on every page, but when we click the *Want to read* link, the navbar disappears. How do we fix this? Should we add the Navbar to every page?

We could. And if the designers add a footer later, we'll just put that on every page too. If they add a chat button to message the help desk, we'll have to add that. And each time we create a new page, we must remember to add all these components to it… Argh! There must be a better way!

And there is. We'll change the **MainContainer** component to be a wrapper for our pages. For this to work, we'll need to use the *children* prop. This is covered in the *Learn React* training linked in the Prerequisites Lesson 000. Brush up on the children prop here: https://react.dev/learn/passing-props-to-a-component#passing-jsx-as-children

11. Go to the **MainContainer** component and `import { PropsWithChildren } from "react";` then use it to type the props. Put the children in (using curly braces) right under your **<Navbar />** component. If your <Navbar /> wasn't inside a div or a *Fragment* (<> … </>), you'll need to wrap the <Navbar /> and {children} in a parent div because a React component needs to return a single root element (as opposed to sibling elements).

12. To use your new wrapper, go back to **WantToRead** and "wrap" it around the component's contents. For example:

```
<WrapperComponent>
    <div>
       Child stuff
    </div>
</WrapperComponent>
```

    When you're done, you should be able to click the *Want to read* navbar link and still see the Navbar, along with the *Want to read* page content.

We can take this a step further though. The pages we'll be making (other than the landing page) all have a similar structure. They display the page title, followed by the page's contents. Instead of repeating that structure across multiple pages, let's create a reusable page wrapper too.

13. Back in the */components* directory, add a new directory called */page-container*, and add an *index.tsx* to it.

Since this wrapper will be reusable across different pages, what props might we need to pass in?

We'll certainly use the *children* prop so it can behave as a wrapper. But in the interest of only having to style the layout once, let's pass the page title in as a prop too.

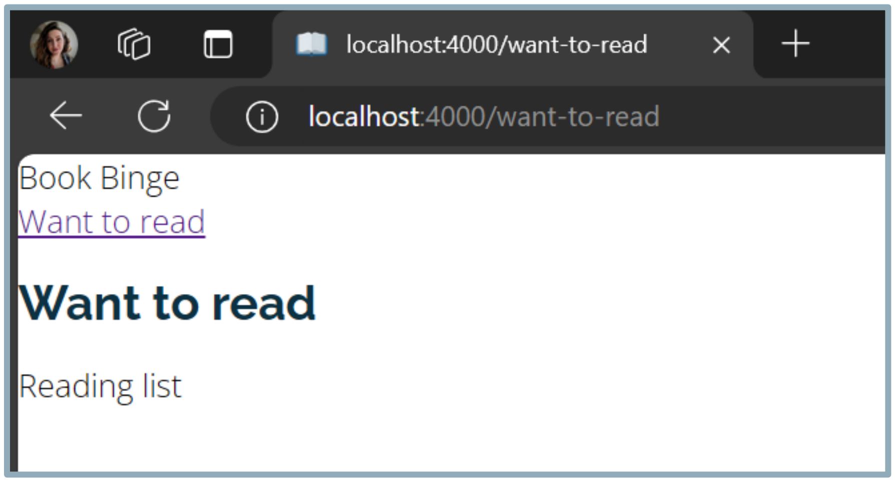
14. Create the **PageContainer** with props typed as follows:

```
type PageContainerProps = {
  title: string;
}
type PageContainerPropsWithChildren = PropsWithChildren & PageContainerProps;
```

You'll need a parent div with the title and children inside. What type of html tag do you think you should use for the title?

A general rule for accessible design is that the page title (the one that changes, not the app logo in the navbar) should always be inside an **h1** tag. This helps people relying on assistive technologies to quickly identify which page they're on. Slap that title in an `<h1>`!

15. When you're done with the **PageContainer**, import it into your **WantToRead** page and nest it inside the **<MainContainer>** wrapper. Since we are now passing the title, "Want to read" into the page props, we don't need include it in our JSX here. Instead, place some placeholder text in the inner-most nested div to represent the reading list that will eventually render on this page. Now when you click the *Want to read* link, you should see this:



Book Binge
Want to read

Want to read

Reading list

16. When you're happy with your **WantToRead** page, go ahead and make the *Finished reading* page using the same pattern.

    What about the *Search* page, you ask?

    We don't need a separate page for *Search* because it will render at root ("/"), based on the status of search results. To distinguish it from the other pages, go ahead and add a snippet of placeholder text inside a **<MainContainer>** wrapper on the */pages/index.tsx* (pages/index.tsx is what renders at root). Something like "Landing page/Search" will suffice.

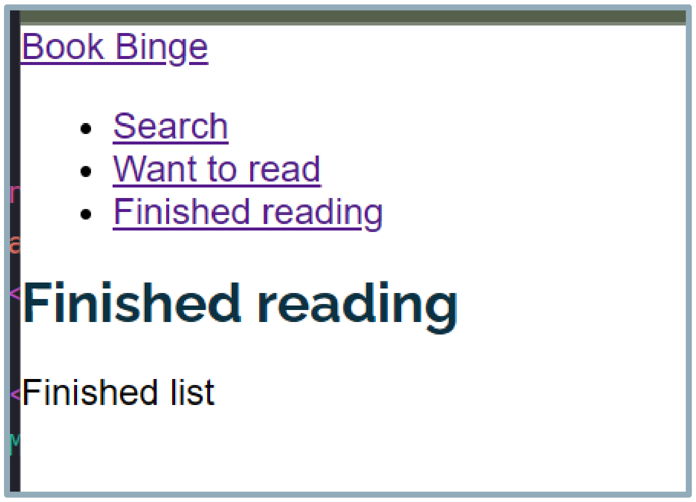
17. Now let's update the **Navbar** to have all the links. The convention is for navbar links to be encoded as an unordered list (**<ul>**). This, as with the h1 tag, is helpful to screen readers. Find the parent div that's holding your *Want to read* **<Navlink/>** and change it into an **<ul>.** Then wrap the **<Navlink />** inside a **<li>.** Add another **<li></li>** above it and another one below. Place the **<Navlink />** for *Search* and *Finished reading* inside them respectively.

    Finished reading should redirect to your new **FinishedReading** component page, while *Search* should redirect back to the home page ("/").

18. Almost done! There's one more link to add. Think about the expected behavior in any application when you click on the app name or logo inside the navbar.

    Refactor your Navbar so that clicking on "Book Binge" also redirects to the home page.

    Congratulations! You just made a navbar! In the next lesson, we'll make it pretty.



Book Binge

- Search
- Want to read
- Finished reading

Finished reading

Finished list