Build the Book Binge App

Lesson One: Create a Navbar and add Next Pages

If you get stuck, check out the **001/lesson-001-navbar** branch to see the final solution.

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

We need a div that will stretch across the full width of the top of the screen, and it needs to contain the app name (Book Binge), along with three navigation links (Search, Want to read, and Finished reading).

Three repeated elements that will all have a similar functionality: Click them to show a different page. This is a good opportunity to practice making a reusable component, simple though it is.

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

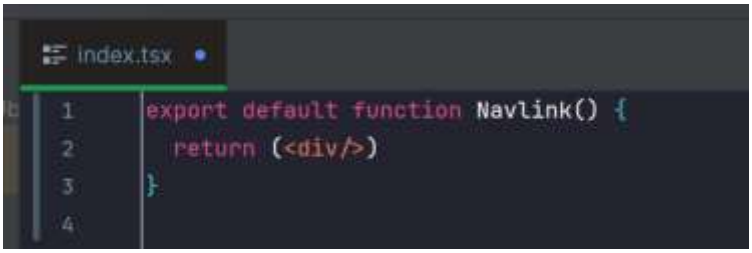
1. Make a new feature branch.
2. Create the 'navbar' directory and add an index.tsx.

Index files get automatically imported when importing a component 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.

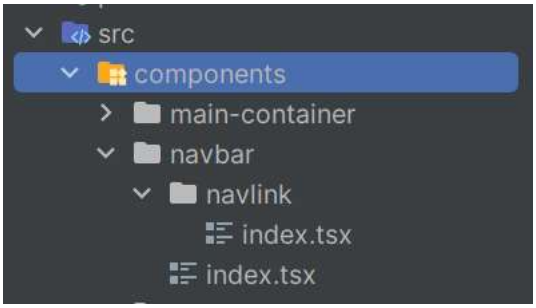
3. Create a new React component in the index file and call the component **Navbar**. Export it using either "export default function…" or, if you created it as an arrow function, remember to export it at the bottom of the file.

4. Create a 'navlink' directory inside the 'navbar' directory and add an index.tsx file to it as well. 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:

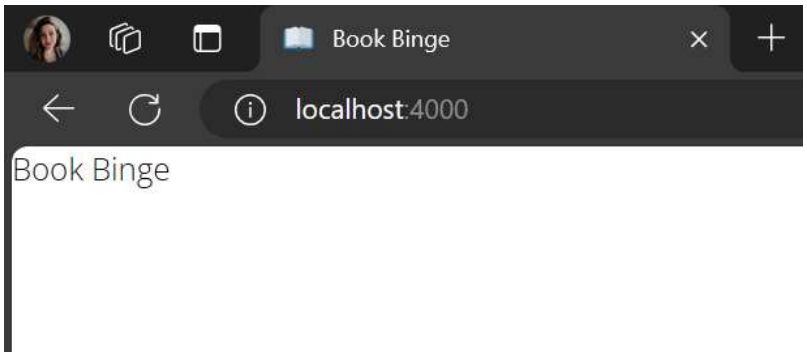


At this point, your file structure should look like this:



5. Try your hand at creating a rudimentary Navbar component. This will be just a div parent with a div child inside it to hold the app title text. It will not look like the Figma until we style it. That will happen in an upcoming 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 stuff in the main container, as we won't be needing it. Add the Navbar and see if you can get your app title to show when you are running (`npm run dev`) at localhost:4000.



Time to add the links. Since we are planning to reuse this component, what variables will have to change for that work?
Two things: the text the link displays, and the location the link navigates to. Knowing this, we know what props we'll need to pass.
Since we're making a component that will take props, we'll need to provide a type for those props to satisfy typescript. I generally prefer to use typescript's *Type* syntax rather than *Interface*, though it wouldn't be wrong either way. But *Type* is usually what you'll want while working with 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 great *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 that displays the text and redirects 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.

****A note on destructuring props:** Props are an object React creates to pass information from parent to child, like an argument passed to 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

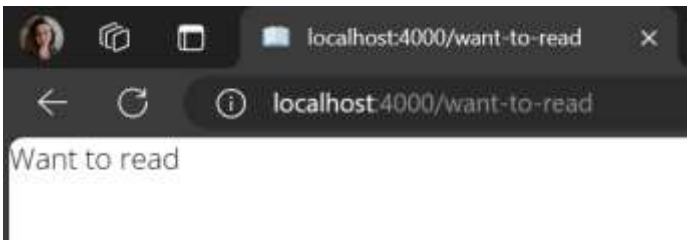
```
import { UrlObject } from "node:url";
import Link from "next/link";

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 to test it? 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. Just add a new page and you can navigate to it by appending the name of the page to your URL, easy as **localhost:4000/new-page**. In other words, Next automates routing for us. Pretty nifty!

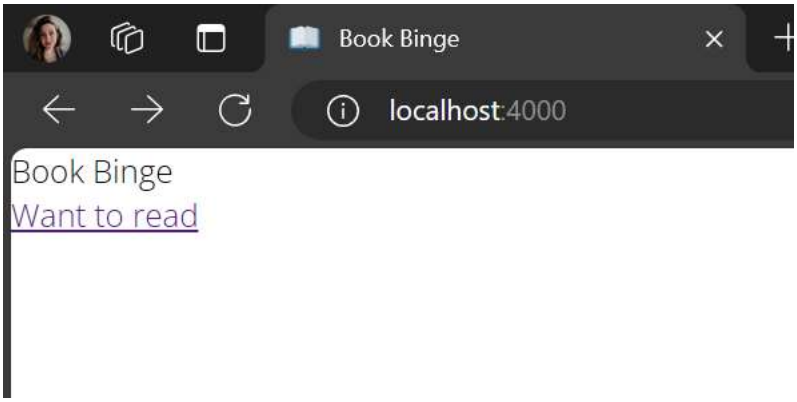
According to the Figma, there'll be a page called "Want to read." Let's make that one first.

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



9. Let's wire it all together. Go to your **Navbar** component and add a new div under the one that contains the title text. Both the title 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 to pass in for its props (its *displayText* and *linkPath*).

   Your home page should now look like this, and clicking on the "Want to read" link should show the new page as pictured above:



10. However, there is a problem. According to the Figma, the navbar should show on every page, but when we click on the "want to read" link, it disappears. How do we fix this? Do we add the Navbar to every page?

We could. And if the designers add a footer later, we'd just put that to every page too. And when they add a chat button to message the help desk, we add that too. 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 mentioned in the Prerequisites section of the Readme. 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 React needs to return a single parent element (as opposed to siblings).
12. To use your new wrapper, simply 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 on your "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 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 want 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 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 title 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. So slap that title in an <h1>!

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

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, like with the h1 tag, is helpful to screen readers. Find the div that's holding your "Want to read" <Navlink/> and change it into an <ul>. Then wrap the <Navlink /> inside a <li>. Add another above it and another one below. Place a <Navlink /> for "Search" and "Finished reading" inside them.

    "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.