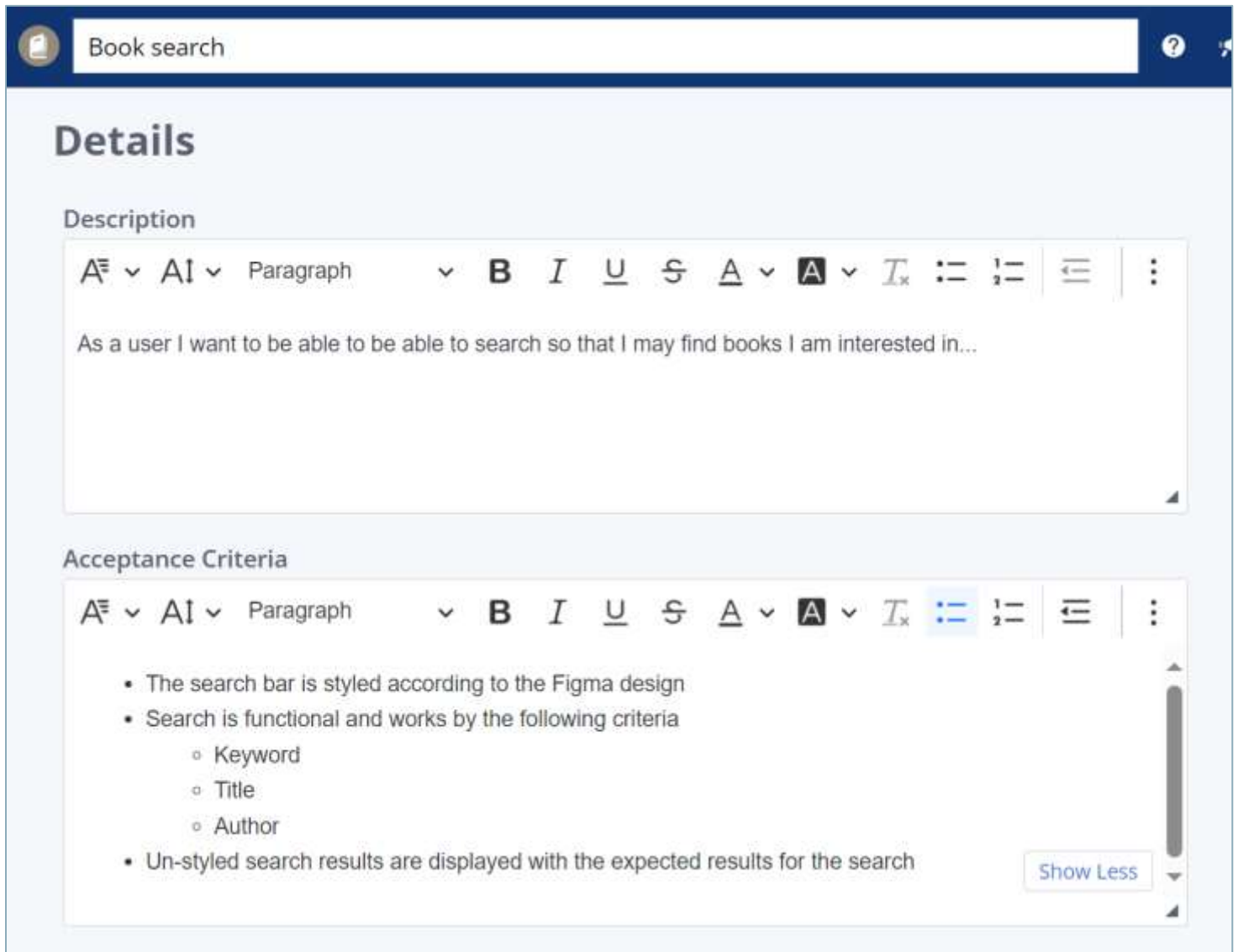


Build the Book Binge App - L003

Lesson Three: Create search functionality and use Composition

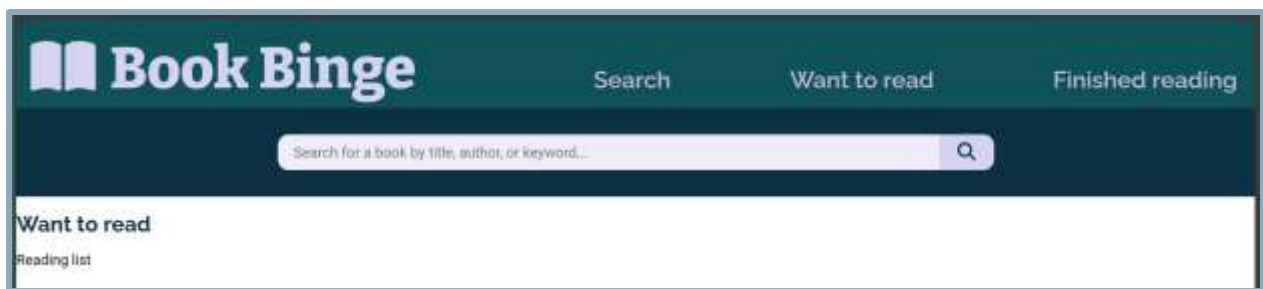


According to our Figma, the search bar should appear below the navbar, like this:



If you skipped Lesson 002, be sure to find the **BookBinge.fig** file inside the /Figma folder and upload it into your Figma client.

1. Make a new feature branch. You may wish to check out last lesson's branch (*002/lesson-002-nav-styles*) and branch off from that.
2. Create a new directory in */components* for the search banner and add your component. Add an SCSS file as well.
3. Use the skills you practiced in the last two modules to create and style the search banner. NOTE: the **SearchBar** widget itself already exists in the UI Library, so you can simply import it. You can also check its (brief) documentation by visiting *localhost:4000/ui-library*. --For the **SearchBar**'s callback function, just put in a *console.log* for now.
4. Where should you add your new search banner to the component tree to see it in the application? Go ahead and place it where you think makes the most sense. Adjust your styles as needed. When you're done, you should see it at *localhost:4000*, as well as on all the pages you can click from the navbar:



If you decided to place the search banner inside the MainContainer, you made the right choice. It would still work if you put in the PageContainer, but that's less ideal because we want the PageContainer to be a simple wrapper for the page that shows beneath the search bar. The search banner, like the navbar, remains static across all pages, so it makes the most sense to keep them both together in the MainContainer.

Did you remember to give the SearchBar placeholder text that matches the Figma? If not, do so now.

Did you check your styles in responsive mode? Does it look like this?



How did you decide to style the container to keep the search bar centered and of the correct (yet responsive) width? Feel free to check the solution branch to see how we did it (hint: we actually have a second container around the bar!). But like most things on the front end, there are many paths to the desired result.

5. Time to fetch some data! We must do a little housekeeping first though: creating an API service using Axios and adding data models to keep TypeScript happy. Move up to the `/src` directory and add a new directory called `/services`. In `/services`, add a file called `openLibraryService.ts` and one more directory, `/models`.
6. We're using the Open Library API for this project. For the search bar's data fetch, we'll specifically use the `/search` API—and even more specifically, the keyword query (as opposed to a dedicated title or author search). This will keep things versatile for our users, as they can enter any metadata as a keyword query, including titles, authors, and subjects. Go to the API docs: <https://openlibrary.org/dev/docs/api/search> and click on the first example to see the data structure delivered by the search request (<https://openlibrary.org/search.json?q=the+lord+of+the+rings>).
7. Compare what you find there with the Figma design showing search results.



Looks like we'll need the title, author, first publishing date, average rating, a page count, how many people are reading the book right now, and a list of subjects. We'll also want to show a cover image. Can you find the fields from the API request that correspond to the data we need?

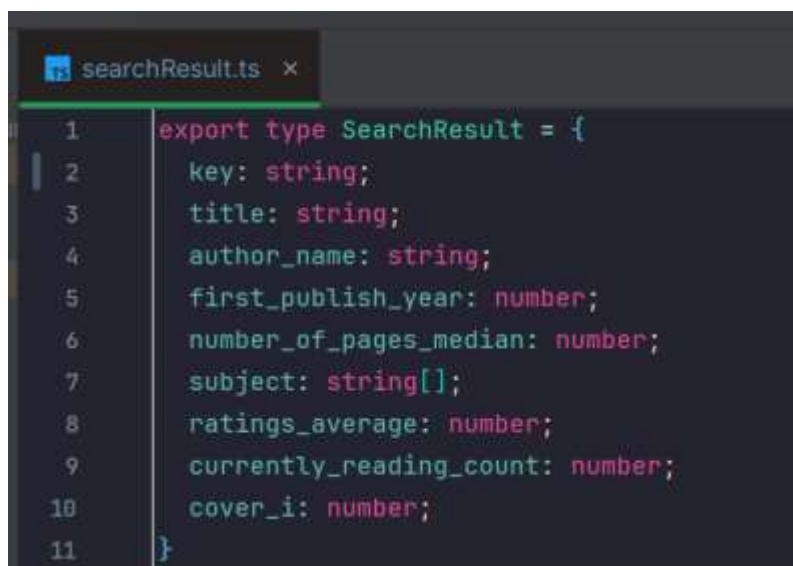
NOTE: We will not fetch the cover image just yet (that will be a future lesson), but we do need information from the query to make that possible.

ANOTHER NOTE: While our UI does not need to display a UUID, it is often useful to have one when working with data. The document model appears to offer a “key” for each item, so let’s add that too.

8. Add a new file to the `/models` directory called **`searchResult.ts`** and try crafting a model that has only the fields the client needs, and the `key`. The API sends back far more data than we care about—keep the model concise and ignore data we won’t use.

As with typing component props, your default choice for creating a model should be *Type*. If you need divergent models to extend a base model, then *Interface* is a better choice. If you need to instance empty objects using your model, then you’ll want to make a *Class*. But unless you require the special functionalities of Interfaces and Classes (and sometimes you will), save yourself from decision fatigue and just make a *Type*.

When you’re done, you should have a model similar to this:



```
1 export type SearchResult = {
2   key: string;
3   title: string;
4   author_name: string;
5   first_publish_year: number;
6   number_of_pages_median: number;
7   subject: string[];
8   ratings_average: number;
9   currently_reading_count: number;
10  cover_i: number;
11 }
```

9. You may have noticed that the API returns results inside an array with the key “docs.” There’s also metadata concerning the number of documents and the starting index. This will come in handy if we decide to paginate our results, but for now we will keep things simple and ignore those fields. But we still need to accurately type the full response. Create a new Type for the response object, called *SearchResponse*, with a “docs” key, the value being an array of *SearchResult*.
10. Now that we have our models, we can create the API service. We’ll use the **`openLibraryService.ts`** file as a central point to manage all our API calls. For now, this will be one call, but we’ll add to it over time.

What is **Axios** anyway? It’s basically a wrapper around the web’s native Fetch API. It comes with quality-of-life improvements to simplify development and is more commonly used these days than Fetch itself.

You may encounter projects with an Axios service that exports itself as a Class, its API calls as methods. That’s fine, but we don’t need to leverage OOP’s strengths (inheritance, encapsulation,

complex entity relationships...), and in fact, the functional programming spirit of React runs roughshod over a service imported as a class, re-instancing it every time the page renders.

Instead of creating an *OpenLibraryService* Class, we will instead create a suite of pure functions.

11. In **openLibraryService.ts** import 'axios,' its *AxiosResponse* type for TypeScript, and the *SearchResponse* type you just created.

```
import axios, { type AxiosResponse } from "axios";
import { SearchResponse } from "../models/searchResult";
```

12. Add a constant variable, `BASE_URL`, to hold the base url ("http://openlibrary.org").
13. Create an Axios client to use our base url. I'll give you the boilerplate here, but if you want to learn more about it, the Axios documentation contains basic examples: <https://axios-http.com/docs/instance>

```
const apiClient = axios.create({
  baseURL: BASE_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});
```

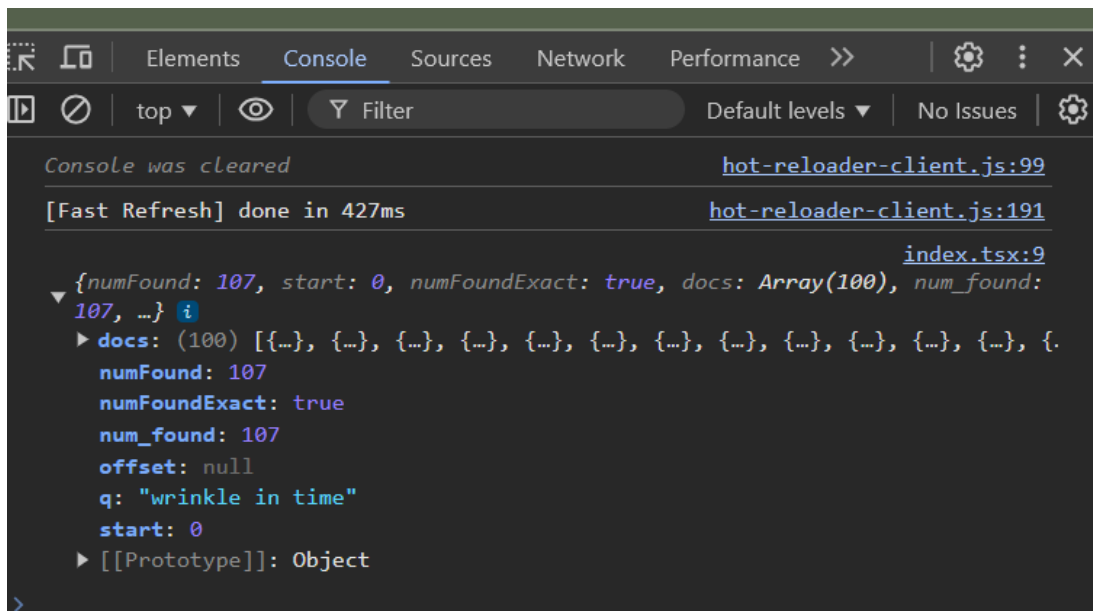
14. Finally, create an async function called **getSearchResults** that takes a single argument, **query**, and returns a **Promise<AxiosResponse<SearchResponse>>**. In the body of the function, await and return an **apiClient.get()** request to the Open Book API /search endpoint.

Can you figure out how to use the **query** parameter? Try string interpolation.

15. Export the function at the bottom of the file—don't use *export default* here because we will be exporting multiple functions from this file.
16. Let's test our work. Return to the **SearchBanner** component and import **getSearchResults**. Inside the component, above the return statement, add a new function called **searchCallback**. It takes a string (and returns void, but you don't have to explicitly state that; TypeScript handles implicit return types just fine).

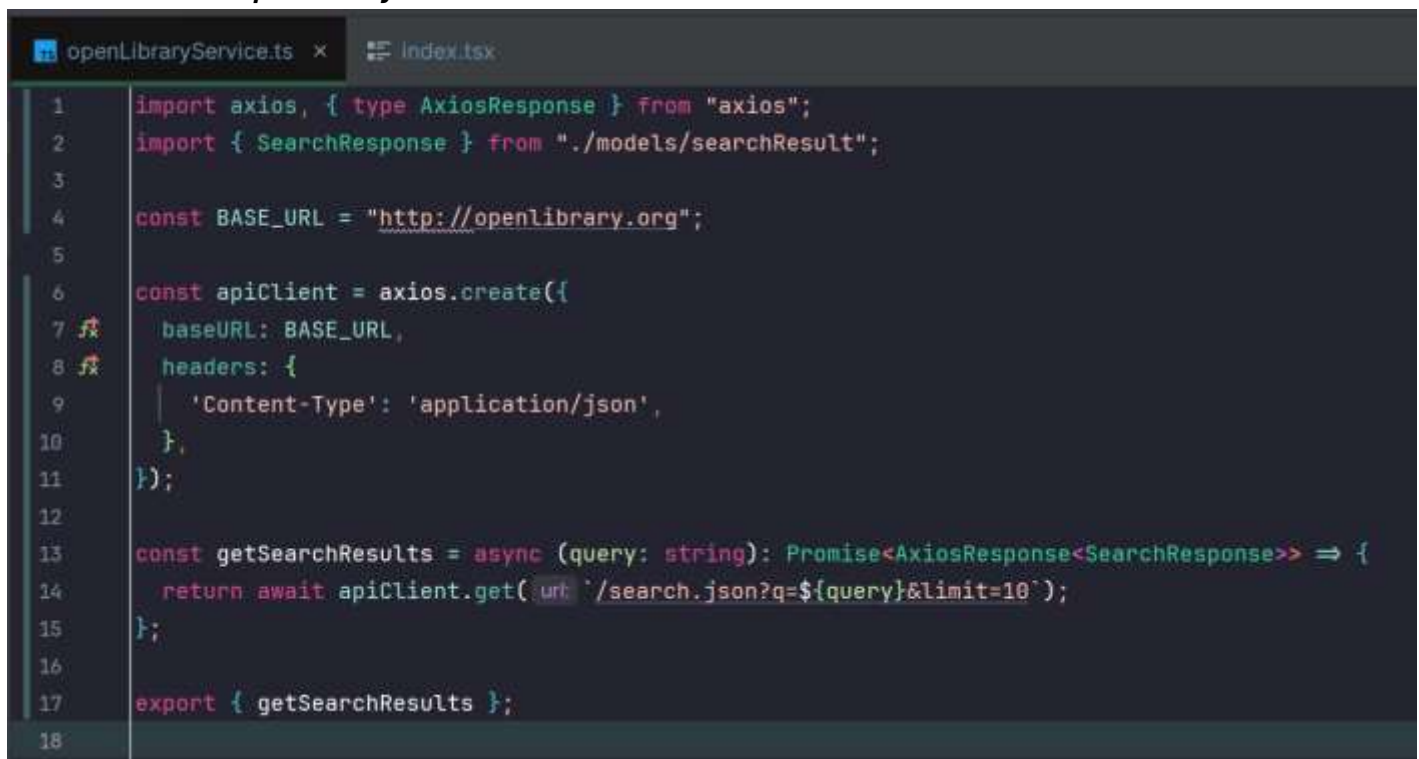
In the body of the function, add a try-catch block. In the try block, call **getSearchResults** and pass it the string from the callback's arguments. Log the *response.data*. In the catch block, use *console.error* to print the error.

17. The moment of truth! Go to *localhost:4000* and enter a search. Use your browser's dev tools to inspect the network calls and the console logs. You should see a log of the response data and a successful 200 response on the network.



18. Depending on what you searched, you may have received hundreds of items in the “docs” array. Rendering so many books in our search results might cause a laggy experience for end users, and we don’t want that! Long term, we’ll want to lazy-load or paginate the results. But for now, while we’re still developing the basic UI, we’ll use a cap on the number of responses. – Go back to your `getSearchResults` function and try adding `&limit=10` to the end of the request URL.

Your `openLibraryService.ts` file should now look like this:



19. Now all that remains is to get this data into a list of search results. For that, we'll need a component to render the list. Go up to the `/components` directory and add a `/search-results` folder. Create a stub component and corresponding SCSS file.
20. How will our new component get the search results data? Will it take props from the parent? Well, the parent is going to be a **PageContainer**, and passing props to `props.children` is not straightforward. Also, **SearchBanner** is not in a direct hierarchy with **SearchResults**, so it would have to pass the data up to the **MainContainer** first, which would then pass it down to **PageContainer**, and then to `children`—but only if the `children` is **SearchResults**!

Phew, this is getting complicated.

We *could* try using the Context API, which is a powerful way to provide state across the tree without passing props, but it has some drawbacks. We'll explore the Context Provider in a later lesson. Luckily, solving our problem is not as difficult as it seems. We'll use a different powerful tool known as Composition. Composition is not an API. It's a design pattern.

Let's get refactoring!

21. Find the `index.tsx` file inside the `/pages` directory. Remember, this is the content that renders at root ("`/`"). Remove anything nested inside the **MainContainer** and render a stand-alone `<MainContainer />`.

```
<main>
|  <MainContainer />
</main>
```

22. Go to the **MainContainer** component. Here we'll use local state to track the search results. Import `{ useState }` from "react" and invoke the `useState` hook to create a getter and setter for type **SearchResult[]**.
23. The goal is to return our new **SearchResults** component (inside a **PageContainer**) if the `searchResults` on state contains any items. If not, we'll show the landing page.

What landing page, you ask? –Go ahead and make a stub component inside `/components` (not `/pages`!) called **LandingPage**. It can return placeholder text for now.

24. Back in **MainContainer**, beneath your `useState` call, create a new function called `renderHomePage`. Here is where we'll return either the **SearchResults** or the **LandingPage** depending on the value of `searchResults`. You can use a ternary or an if-else block.

Remember to wrap `<SearchResults />` inside `<PageContainer>`. This is because it follows the same UX design as "Want to read" and "Finished reading," and the **PageContainer** component was created to share styles across our pages.

25. When you're done with the function, it's time to use it inside the **MainContainer** JSX. Should we use it to replace the `{ children }`?

Not quite. We still want to show the “Want to read” and “Finished reading” pages, which get wrapped in **MainContainer** and need to pass through. Instead, we’ll create yet another conditional render! I would use an inline ternary here, but you can create another function if you find that clearer to read: `{ children ? children : renderHomePage() }`

If you navigate to `localhost:4000`, you should see your **LandingPage** stub (because we don’t have any searchResults on state). And your nav links should still work as expected.

26. Time to wire up the state. First, we need to move our API call out of the **SearchBanner** and into the **MainContainer**. This is only because the search bar is available on every page, but when a search is entered, we need to re-direct to the search results. This will cause the page to refresh before the data can finish rendering. So the fetch needs to happen at the parent level in order to maintain the response.

Confused? Don’t worry. It will all make sense soon.

Copy the try-catch block out of the callback function in **SearchBanner** and paste it into a new function in **MainContainer** called *fetchSearchResults*.

The **SearchBanner** still needs to receive a *searchCallback*. What should happen when a user enters a query? As mentioned, we need to redirect to the search results page so that the results will appear. For this, we’ll leverage Next.js routing capabilities.

```
import { useRouter } from "next/router";
```

Then destructure the *push* command from the *useRouter()* hook. The *push* command allows us to push a new route onto our browsing history. (Browsing history is an array, so it uses methods like “push” and “pop.”)

We’ll use the *push* command to go back to the root (“/”) because that’s where our search results live.

What about the search terms though?

Don’t worry, the URL can help with that. We’ll mimic the API call and use a query string. Update your *searchCallback* in the **SearchBanner** to do the following, where *query* is the parameter the callback receives from the search bar:

```
await push(`/?query=${query}`, undefined, { shallow: true })
```

Next.js is not the main focus of this project, but if you’re curious about all that, see the Next documentation: <https://nextjs.org/docs/app/building-your-application/routing/redirecting#userouter-hook>

27. Your **SearchBanner** should now look like this:

```
1 import styles from './index.module.scss';
2 import SearchBar from "@ui-library/search-bar";
3 import { useRouter } from "next/router";
4
5 export default function SearchBanner() {
6   const { push } = useRouter();
7
8   const searchCallback = async (query: string) => {
9     await push( url: `/?query=${query}`, as: undefined, options: { shallow: true } )
10  };
11
12  return(<div className={styles.searchBanner}>
13    <div className={styles.searchBarContainer}>
14      <SearchBar
15        onSearchCallback={searchCallback}
16        placeholderText={"Search for a book by title, author, or keyword..."}
17      />
18    </div>
19  </div>);
20 }
```

28. Return to the **MainContainer**. We must now figure out how to run our *fetchSearchResults* function when—and only when—we get redirected to root (“/”) with a query string in the URL. 🤔

We’re no longer triggering our API call with a user action (entering a search). Instead, we’re relying on an external event—the URL change. For this situation, we need the *useEffect* hook.

UseEffect allows us to trigger side effects when external information changes. It should be considered as an escape hatch. It has its place, absolutely—and data fetching on render is a common one—but it’s easy to fall into the trap of using it needlessly when more efficient and performant means exist. See the React documentation for more information:

<https://react.dev/learn/you-might-not-need-an-effect>

In our case, we want to trigger a fetch of search results when the query parameters in our URL change. To see what’s on the query params, we once again need Next.js’s *useRouter* hook.

```
import { useRouter } from "next/router";
```

```
And destructure query: const { query } = useRouter();
```

29. Import *useEffect* from “react” and create a *useEffect()*. If the *query* object (from Next’s router) contains any keys (**`Object.keys(query).length`**), we’ll call *fetchSearchResults* with the query string. Else, we’ll *setSearchResults* to an empty array.

That last part is more important than it may seem at first. If someone navigates back to the home page after making a search query, we want to make sure a clean page renders by clearing the cache of search results.

You should end up with a *useEffect* like this:

```
useEffect( effect: () => {  
  if (Object.keys(query).length) {  
    fetchSearchResults(query.query as string);  
  } else {  
    setSearchResults( value: []);  
  }  
}, deps: [fetchSearchResults, query]);
```

30. Before we move on, there’s one more (minor) improvement we can make, and that’s to memoize our fetch function with the *useCallback* hook. In a tiny (fake) application like this, it won’t make much difference. But it’s good to be aware that, every time a component renders it creates a new reference for function within its scope. That means every time we submit a new query, causing the *useEffect* to run, and by side-effect, causing the *searchResults* to re-populate, it triggers a render. Which re-runs everything in the component. But the *fetchSearchResults* function itself does not need to change. We could even put it outside of the component entirely if it didn’t need to update state with the *setSearchResults* setter.

Luckily, we can memoize the function—cause React to cache it and not re-create it unless the whole app re-renders.

Like `useEffect`, `useCallback` takes a dependency array. But in our case, we aren't dependent on anything to re-create this function:

```
const fetchSearchResults = useCallback( callback: async (query: string) => {  
  try {  
    const response = await getSearchResults(query);  
    setSearchResults(response.data.docs);  
  } catch (error) {  
    console.error(error);  
  }  
}, deps: []);
```

31. Moving on. Open the **SearchResults** component and wire it up to take props `searchResults` as type `SearchResult[]`. Update its JSX to iterate through the `searchResults` and print each book's title. (We'll make the fancy display cards in a later lesson.)

If you're not sure how best to handle list iteration in React, it would be a good idea to brush up with the *Learn React* courses. You'll frequently find yourself using Array methods in React development, so it might be a good idea to review those as well: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

For this situation, you want the `map` method.

****BE AWARE:** Whenever you iteratively map out divs, you need to assign a unique key to each one. Don't use the array index unless you have nothing else because the order of items in a JavaScript array can change. React needs keys to track the items in its Virtual DOM. Neglecting to provide keys can cause performance problems and even weird state side-effects.

You'll see an error in your console if you forget. Remember to check your console while developing!

What unique data point can we use as a key? Titles won't necessarily be unique, nor authors... Good thing we added the `key` property to our model! That will do nicely.

When you're done, you should have something like this:

```
return(  
  <div>  
    {searchResults.map((book) => <div key={book.key}>{book.title}</div>)}  
  </div>  
);
```

32. Back in the **MainContainer** you should see an angry linter. That's because **SearchResults** now requires props. Go ahead and pass them in.

Your **MainContainer** should look a bit like the image below:

```
11 export default function MainContainer({ children }: PropsWithChildren) {
12   const { query } = useRouter();
13   const [searchResults, setSearchResults] = useState<SearchResult[]>({ initialState: []});
14
15   const fetchSearchResults = useCallback(async (query: string) => {
16     try {
17       const response = await getSearchResults(query);
18       setSearchResults(response.data.docs);
19     } catch (error) {
20       console.error(error);
21     }
22   }, [deps: []]);
23
24   useEffect(effect: () => {
25     if (Object.keys(query).length) {
26       fetchSearchResults(query.query as string);
27     } else {
28       setSearchResults(value: []);
29     }
30   }, [deps: [fetchSearchResults, query]]);
31
32   const renderHomePage = () => searchResults?.length
33     ? (<PageContainer title={"Search results"}>
34       <SearchResults searchResults={searchResults} />
35     </PageContainer>)
36     : <LandingPage />;
37
38   return (<
39     <Navbar />
40     <SearchBanner />
41     {children ? children : renderHomePage()}
42   </>);
43 }
```

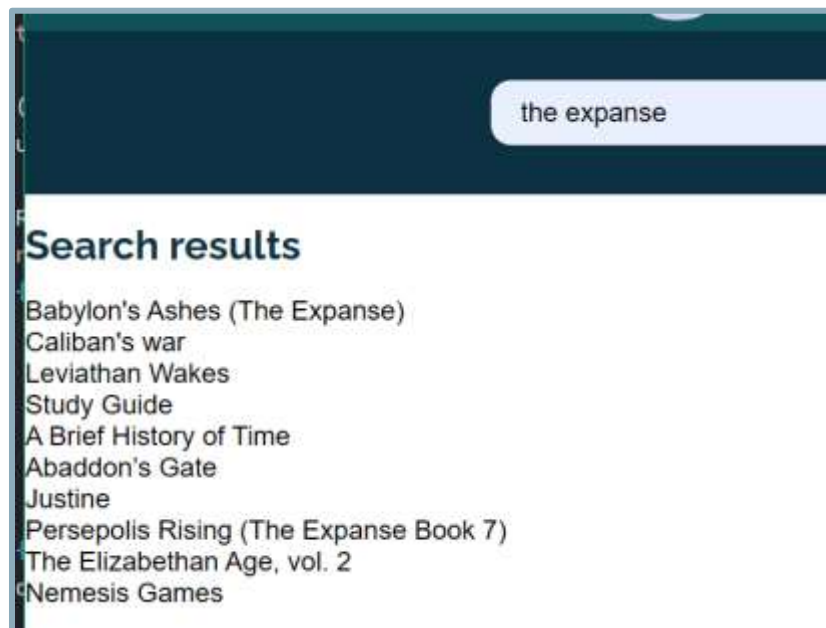
We see Composition shine with the Search Results page in our *renderHomePage* function (line 32). **PageContainer** is a wrapper that cannot easily pass props to abstract { children }, and even if we did set it up to do so with additional coding, the props would have to be the same for all the children even though **SearchResults** is the only one that needs them.

By *composing* our components this way, we can expose **SearchResults** to **MainContainer** directly and get the *searchResults* props without having to involve **PageContainer**.

We also *lifted up state* to the parent container (another React design pattern) so that we didn't have to rely on **SearchBanner** updating its parent in order for **SearchResults** to get the information it needs.

This is a clean solution to the state problem we encountered. If you find your code getting complex, it's worth asking whether you can compose the tree differently.

33. Test your work! Go ahead and search for a book. You should see a list of titles appear:



Better yet, you should be able to perform the search from any page and still get the search results (thanks to the *Next* router query!).

Congratulations! You have a working API service, a search bar, and a results page that conditionally renders. If you're running into trouble, check your work against the Lesson 003 branch (*003/lesson-003-search*).

Next up, we'll localize our application for international users and make additional UX improvements, including styles.

EXTRA CREDIT: Learn about the differences between *useCallback* and *useMemo* in this excellent interactive guide from *Log Rocket*: <https://blog.logrocket.com/react-usememo-vs-usecallback/>