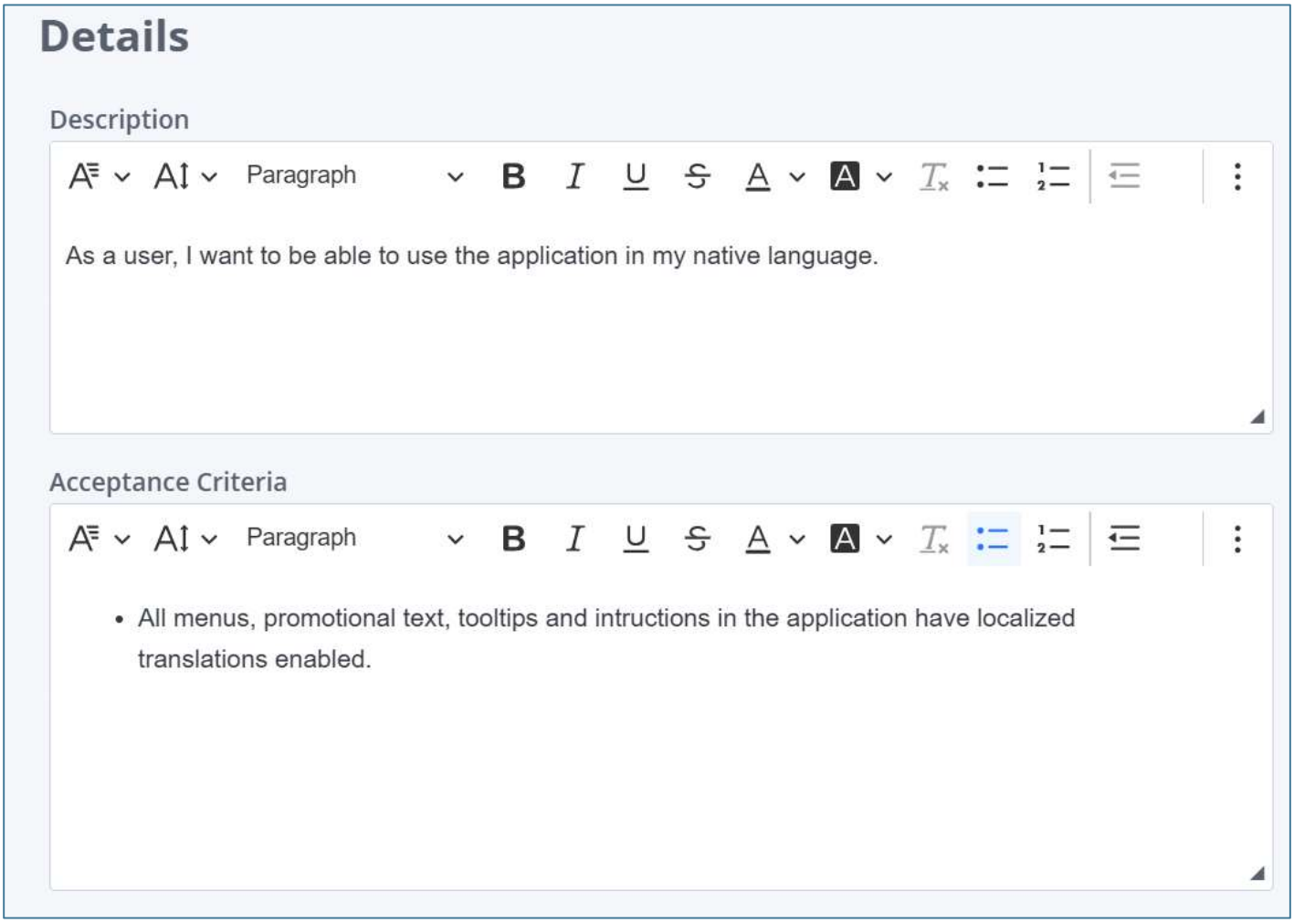
Build the Book Binge App - L004

Lesson Four: Localize the application and style the Landing and Search Results pages.

If you get stuck, check out the **004/lesson-004-localize-cards** branch to see the final solution.

The Book Binge company has exciting news! There's so much interest in our app that investors have decided to release the beta version to international audiences. The product team would like to make it possible for users across the globe to navigate the interface in their native language.



We won't actually create a translation pipeline (that is beyond the scope of this curriculum), but we will build a habit of localizing text as we code. The *React-Intl* library is already installed and configured, but you must add the functionality to inject translations. We'll start by refactoring the text that already exists, namely the nav bar links, page titles, and the search bar placeholder text.

NOTE: We won't translate "Book Binge," as that is a product logo, and we won't be able to translate search results (nor would we want to) because bibliographic data such as book titles should appear in the language in which they are published.

1. Make a new feature branch. You may wish to check out last lesson's branch (003/lesson-003-search) and branch off from that.

2. Start with the nav bar links. In /components/navbar, create a file called *messages.ts*. Import *defineMessages* from "react-intl." We're putting this file in the /navbar directory (instead of /navlinks) because the Navbar component contains the text that gets passed to the links.

3. Export the object returned from the *defineMessages* method.

```
1  import { defineMessages } from "react-intl";
2
3  export default defineMessages({});
4
```

4. Recall that our three navlinks are "Search," "Want to read," and "Finished reading." All of these will have to be translated. To add a translation to the *defineMessages* object, add a new key with a value of type MessageDescriptor, which contains a unique id for the message and the message itself (*defaultMessage*), which in our case, is the English text. For example, the message for "Search" should look something like this:

```
3  export default defineMessages({
4    search: {
5      id: "book.binge.navbar.navlink.search",
6      defaultMessage: "Search",
7    },
8  });
```

5. In the Navbar component, import the *useIntl* hook from "react-intl," and import *messages* from "./messages." Inside the component's code block, destructure *formatMessage* from the useIntl() hook:

```
6  import { useIntl } from "react-intl";
7  import messages from "./messages";
8
9  export default function Navbar() {
10   const { formatMessage } = useIntl();
11
12   return(<div className={styles.navbar}>
13     <div>
```

6. You can now use the *formatMessage* method inside your JSX by passing it the *messages* object and dotting into the desired translation:

   `formatMessage(messages.[yourKeyHere])`

7. Use it to replace the hard-coded text for "Search." Then test that everything is still working as expected by running **`npm run dev`** and checking *localhost:4000*. The "Search" navlink should still look the same.

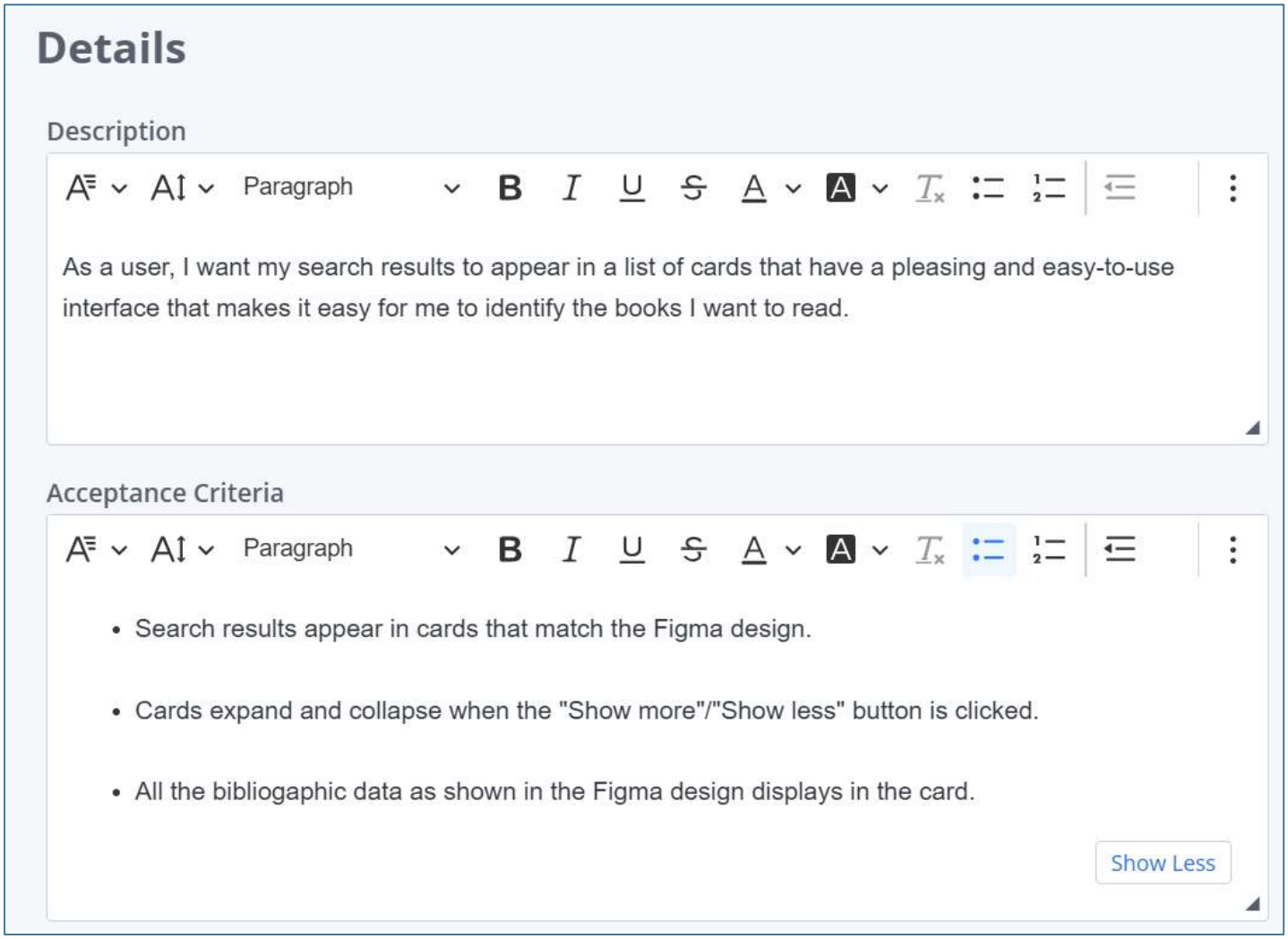
8. Go ahead and add messages for the rest of the navlinks ("Want to read" and "Finished reading") and replace the hard-coded text with formatted messages.

9. Now we'll translate the rest of the website. A good practice is to give each component that requires translations its own *messages.ts* file. This makes it easy to find and edit text associated with a component. In our case, we want to add a new messages file to /pages/finished, /pages/want-to-read, /components/main-container, and /components/search-banner.

   Go ahead and create the message objects following the same boilerplate we used for the navbar. Replace the hard-coded text for page titles, and the placeholder text in the search bar, with formatted messages.

   From this point forward, we will stop using hard-coded text and instead localize as we code. This is a good habit that will save us time and help avoid translation gaps.

10. Moving on, it's time to tackle our next User Story. Creating the book cards.



Looking across all the pages in our Figma design, we see that every view that shows a list of books displays them using the same type of card. There are some differences, but the general object is the same. This means we'll want a reusable, configurable component.

11. In /components, create a new directory called *book-card*, and add the standard files (*index.tsx, module.index.scss,* and *messages.ts*). Create a new React component, *BookCard*, in the index file. Add a props type.

    Q: What props should this component receive, based on changing data and features of the card?

    A: *The book data will be different for every book rendered. The buttons also change, depending on the page. Finally, the Finished page shows the user's rating at the top of the card. That means we'll have at least three props: the book data, a buttons array (because there can be more than one), and a Boolean to indicate whether the user rating should show.*

12. Create a simple div that uses book data to render the title and the author. For now, you can type the book data property as a *SearchResult* –we will have to change this later because the UI will be tracking additional data about a book besides what was retrieved from the API—namely, the user's interactions with the data. But for now, it works well.

    Q: How can we ensure that the book titles are easy to navigate with a screen reader?

    A: *Screen readers can easily jump between h-tags. We know our page title is the h1. As the next "heading" down, the book title should be enclosed in an h2 tag.*

13. To test your work, go to the SearchResults component and update it so that, instead of mapping out a simple div with the book title inside it, it maps out your BookCard. Check your results in your browser by entering a search.

14. Did you remember to give the mapped-out BookCards the key prop? (If you recall, ***key*** is a default prop that all React components can receive to help React track them in the virtual DOM; it doesn't need to be (and shouldn't be) explicitly added to our component's props.)

15. How did you handle the buttons array? Assuming you added it to your props type, you will need to pass something to the component. At this moment, since we are merely creating a prototype, you can just pass it an array with a button inside:

```
13          <BookCard
14            key={book.key}
15            book={book}
16            buttons={[<button key={book.key}>BTN</button>]}
17          />))}
```

The type for the button array should be **`ReactNode[ ]`**.

16. What about the Boolean to indicate whether to show the star rating? You would be correct to simply pass the property a **`false`** value, since we don't want the stars on our search results. However, since only one page uses the star rating (the "Finished reading" page), it might be cleaner and simpler to make the prop optional and give it a default value of **`false`**.

17. Did you remember to localize the card's contents? While we don't want to translate the book data itself, the byline includes the word "by," which *does* need to be translated.

    This is different from the navbar links, though. It needs to take a variable parameter (the author's name) and interpolate it in the message string. To do this, you need to write the

message like a template string using curly braces. The placeholder in the curly braces becomes the variable name:

```
defaultMessage: “Hello {yourName}, I am {myName}.”
```

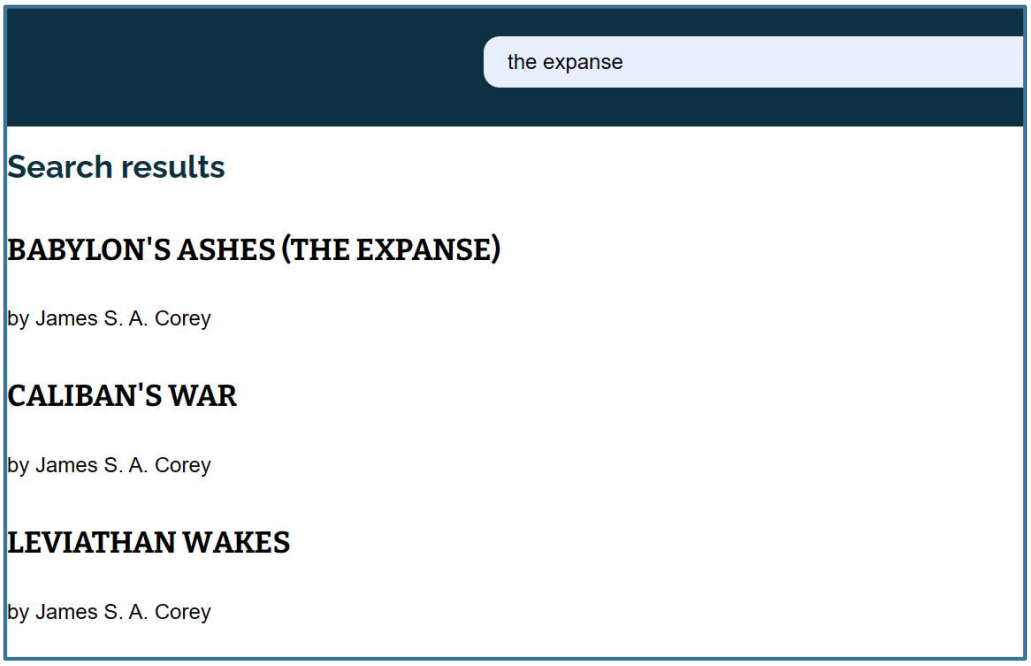
To assign this variable a value, you must pass a second optional parameter to the **`formatMessage()`** method, which takes a dictionary with keys that match the variables in your template string, and values that contain the dynamic data:

```
formatMessage(messages.greeting, { yourName: “World”, myName: “Amanda”}
```

The greeting message will render on screen as, “Hello World, I am Amanda.”

Go ahead and localize your BookCard to print “by” before the author’s name.

When you’re done, your search results should appear similar to this:



18. Great! Our search results are now wired up to our new card component! Time to get styling. Before we continue with the card, though, we’ll want the page itself to look more like the Figma, with contents centered against the correct background color.

    The best way to set the background color is to update the **`body`** CSS selector in the *global.scss* file. In a microfrontend environment, you wouldn’t have to set the app’s background color, as that would be handled at the container level. However, since *Book Binge* is not actually a microfrontend, it’s up to you.

    That said, sometimes a microfrontend can benefit from style changes made at the global

level, so it's good to keep in mind that you do have the power to modify them if needed. In this case, just add the **`background-color`** attribute to the **`body`** block in *global.scss*.

19. Next, let's ensure that all our pages align their contents with the left edge of the search bar and have the same padding around the edges, as per the Figma. To achieve this, we'll modify the MainContainer component because it is the root of all our pages. There is more than one way to achieve the appearance we want, but I'd recommend using two divs around the children—one to center the contents using Flex, and the other to control positioning of the children using the width of the div that holds them.

```
42    return (
43      <>
44        <Navbar />
45        <SearchBanner />
46        <div className={styles.pageRoot}>
47          <div className={styles.pageContents}>
48            {children ? children : renderHomePage()}
49          </div>
50        </div>
51      </>
52    );
```

20. See if you can achieve the desired result. If you're having trouble, remember that you can use your browser's dev tools to adjust styles on the fly. You can also check the lesson 004 branch if you get really stuck.

21. The landing page will need a little extra attention because it is the one page that has its contents arranged differently – there is a larger margin around it, and the text is all centered. First, give /landing-page a *messages.ts* and add the text from the Figma design.
NOTE: To best achieve the look of the centered text lines, you'll want to put each line in its own div, which means giving each one its own Message Descriptor.

22. Now style the div containing the lines of text to match the Figma. In addition to using the same styling approaches you practiced in Lesson 002, you'll also want to pay attention to line height, as well as the additional margin at the top, which you will have to guess, since the Figma isn't clear here. Sometimes you need to eyeball things and give it your best shot.

23. You should now have a finished landing page! Whoo hoo! Let's move on to Search Results.

    The page title should look good now, but we still need to add the main event: the cards. Begin by styling the card itself with the correct background color, margin to space the cards apart, rounded corners (**`border-radius`**) and a drop shadow (**`box-shadow`**). You'll have to

experiment to create a drop shadow that matches the Figma. Finally, the card should stretch to fill 100% of the available width (assuming the page contents are in a max-width parent div already, per step 19).

Box-shadow can be a tricky property because it has a lot of variables. Refer to the MDN documentation for good examples and a sandbox to experiment with:

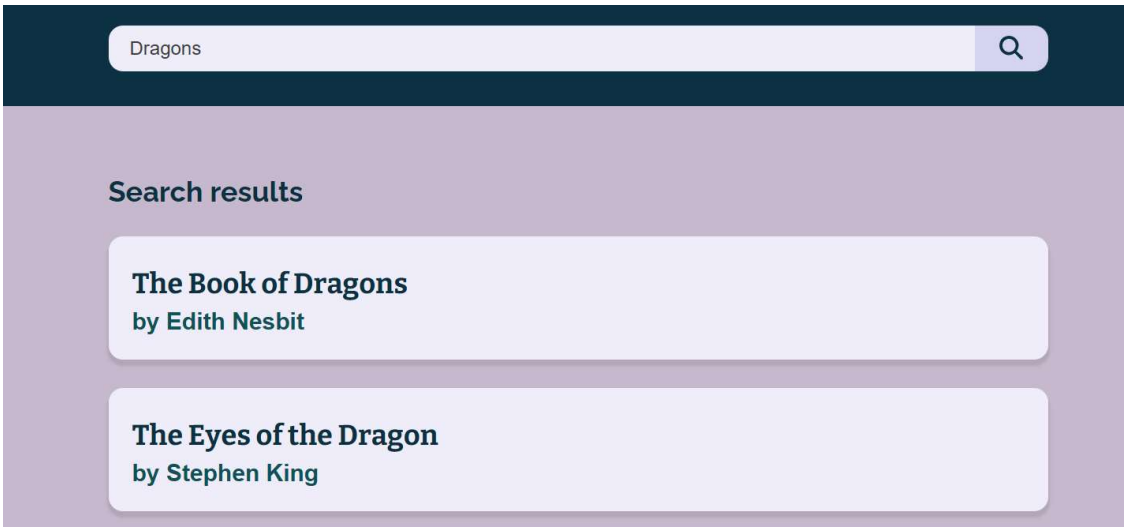
https://developer.mozilla.org/en-US/docs/Web/CSS/box-shadow

GitHub Copilot can also provide some tips if you get stuck.

24. Now style the book title and author byline to match the Figma. Recall that we assigned an **h2** tag to the book title. The CSS properties for **h2** tags are already set in the *global.scss* file, but it doesn't match our requirements.

    You could modify the global style since no other elements will be using the **h2** tag—but we never know what the future holds. The safest bet is to override the **h2** in your BookCard stylesheet or give the title a CSS class and assign different properties to that.

    When you're done, your search results should look like this:



25. We need to add the rest of the card's contents, but before we go wild, it's important to stop and think about how we want to lay out our divs. Each card has two different sections—the top, and the expandable bottom. Each of those sections have their own unique layout. The top of the card is relatively simple compared to the bottom half, so Flexbox is a good choice for the top.

    We could also manage the bottom with Flexbox alone, but it might get out of hand with lots of nested divs that become difficult to maintain. Instead, we'll use Grid to organize the bottom.

Regardless, our BookCard component will need two divs inside the top-level parent—one for the top half, and one for the bottom. For now, just use placeholder text to occupy the bottom half as we get the basic card functionality set up. Function first, then form.

Keep the book title and byline together in the top div.

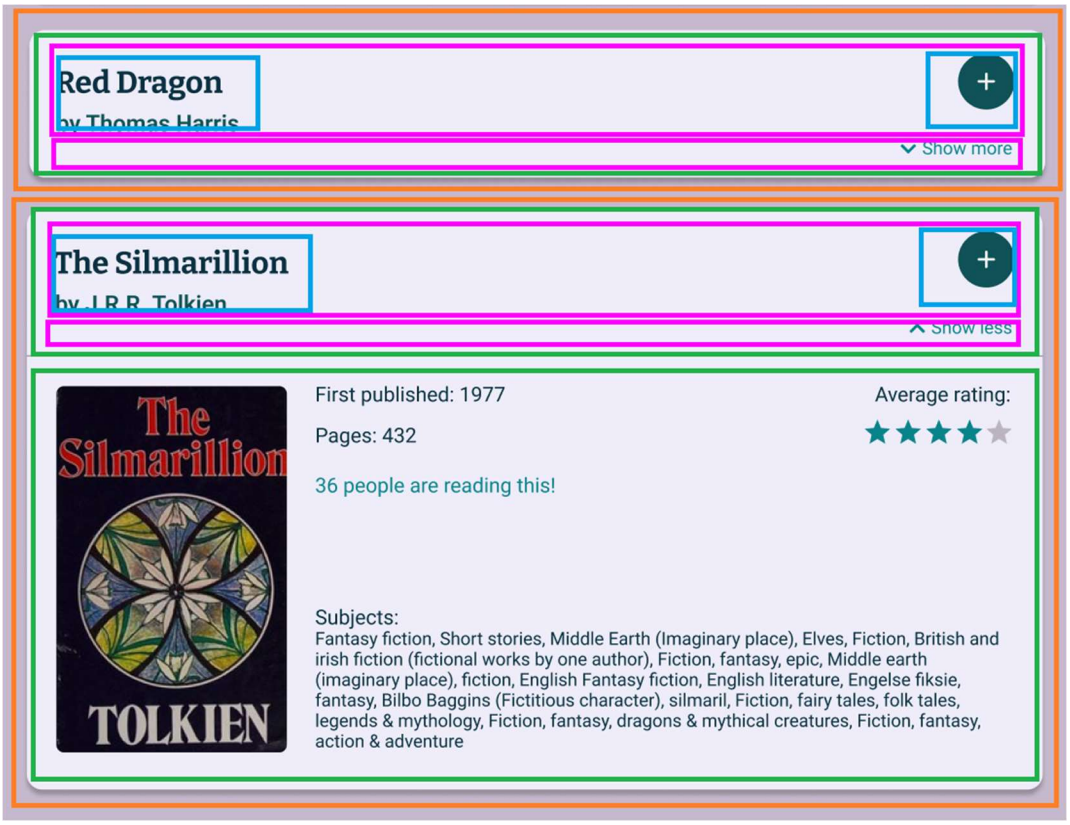
26. We need to add the “show more” button to implement the accordion behavior that will keep the “Book stuff” hidden until we click it.

    But where should the button go in our JSX to get it to match its placement in the Figma? Whenever you’re putting together a UI layout, take time to consider the DOM structure before you start coding.

    In the picture on the next page below, each colored box represents a div. Boxes of the same color are siblings within the hierarchy. You can see how even a relatively “simple” layout like the top of the book card can get a little complicated.

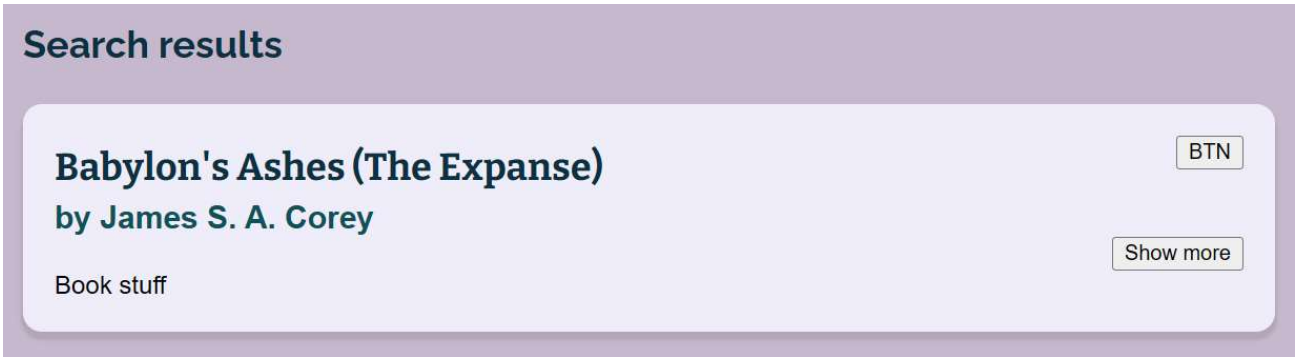
    The top-level parent is the orange box. Within that, we have the card top and card bottom divs in green.

    We’ll deal with the Grid in the bottom later, for now, let’s organize the divs in the top of the card so that we can Flex 💪 our way to success. Give it a shot!



27. While you're at it, go ahead and insert the dummy button as well, which is already passed in on props. For the "Show more" button, use a basic html button for now (and add localization for the "Show more" text!). Don't worry about fine-tuning the styling. Just try to get a rough approximation of the layout.

    You may have to re-arrange your divs and your stylesheet a few times and experiment. Can you get something like this?



28. As an extra hint, you can try arranging your divs as pictured below. Then Flex 💪!
    For the "Show more" button, try using the **`align-self`** property.

```
return (
  <div className={styles.cardRoot}>
    <div className={styles.cardTop}>
      <div className={styles.headerSection}>
        <div>
          <h2>{book.title}</h2>
          <div className={styles.byline}>
            {formatMessage(messages.byAuthor, { author: book.author_name })}
          </div>
        </div>
        <div className={styles.rightSideButtons}>
          {buttons.map((button : ReactNode ) : ReactNode  => button)}
        </div>
      </div>
      <div className={styles.showMoreButton}>
        <button>
          {formatMessage(messages.showMore)}
        </button>
      </div>
    </div>
    <div>
      Book stuff
    </div>
  </div>
)
```

29. Now that we have the top contents (more or less), let's add functionality. The "Show more" button needs a callback function that will toggle a Boolean on state to tell the bottom half of the card whether to render or not. We'll use conditional rendering along with conditional styling to achieve the effect of the card expanding and collapsing.

    Initialize a **`UseState`** const, create the callback to toggle its value, assign it to the button's **`onClick`**, and see if you can use this state to conditionally render the "Book stuff" div.

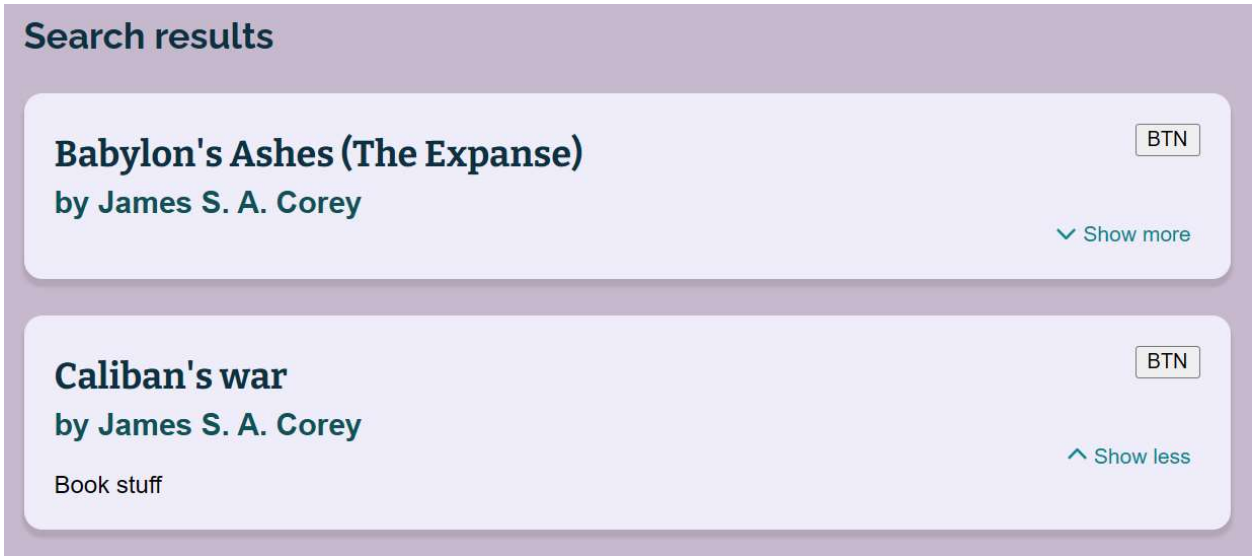
    When you click the "Show more" button, the card should "expand" and render *Book stuff*, and when you click it again, it should "collapse" as *Book stuff* disappears.

30. One more thing. The text of the button should change too, between "Show more" and "Show less." Add a conditional render to the button's (localized!) text content.

31. Now that we've got functionality, let's polish up the styles before we move on. Ignore the dummy "BTN" for now. We'll replace that when we implement the *Want to read* list in later work. Focus on the "Show more" button. You'll have to override the native HTML button styles *and* add an icon from the Fontawesome library—which will *also* have to conditionally render (pointing either up or down) depending on the state of the card's expansion toggle.

    If you've forgotten how to use the Fontawesome library, review Lesson 002. The icons you're looking for are called chevrons.

    When you're finished with all the bells, whistles, and style adjustments, your cards should resemble these:



32. That will do it for the top section (as far as this lesson is concerned). Time to do the bottom. Gird yourself for Grid!

    While Grid is a bit more involved than Flexbox, we'll be working with a simple implementation to get you familiar with the concept. Also, the *CSS Tricks* blog has an incredibly useful guide that you may wish to save for future reference:

    https://css-tricks.com/snippets/css/complete-guide-grid/

    Just as when planning our Flexbox layout, we need to plan our Grid. Take a moment to examine the Figma design and consider how you would try dividing the contents of the card's lower section into a grid layout. Keep in mind that items in a Grid can stretch across multiple rows and columns if we wish.

In this approach, we create a very basic grid of two rows and three columns. Four sections occupy different regions.

Let's create a new component for the card's lower section to avoid bloating BookCard. Inside the /book-card directory, create an /expanded-contents directory, and give it all the usual files.

33. Create the ExpandedContents component. Have it return a div with "Book stuff" inside it and replace the "Book stuff" div in BookCard with your new component.

34. What props will ExpandedContents take? Create a type for them and pass them in. For the cover image, we'll use a dummy placeholder for now. That will change in a future lesson when we implement the API call, but you can ignore it for now.

    Create the container div that will be the Grid. Add four divs inside it to represent each of the four regions.

    Let's get the layout of the grid working before we add the data so that we don't have to think about any additional nested children. For now, just add placeholder text in each div. Grid populates its children from left-to-right, top-to-bottom. Given this, decide which of the four divs should hold which type of data, and add your placeholder text accordingly.

For the cover image, you can use the function below. Just paste it in the ExpandedContents file above the React component and use it to inject a mock "image."

```
const mockCoverImage = () => (
  <div style={{
    backgroundColor: "gray",
    width: "11.25rem",
    height: "17.875rem",
    borderRadius: "6px"}}/>
)
```

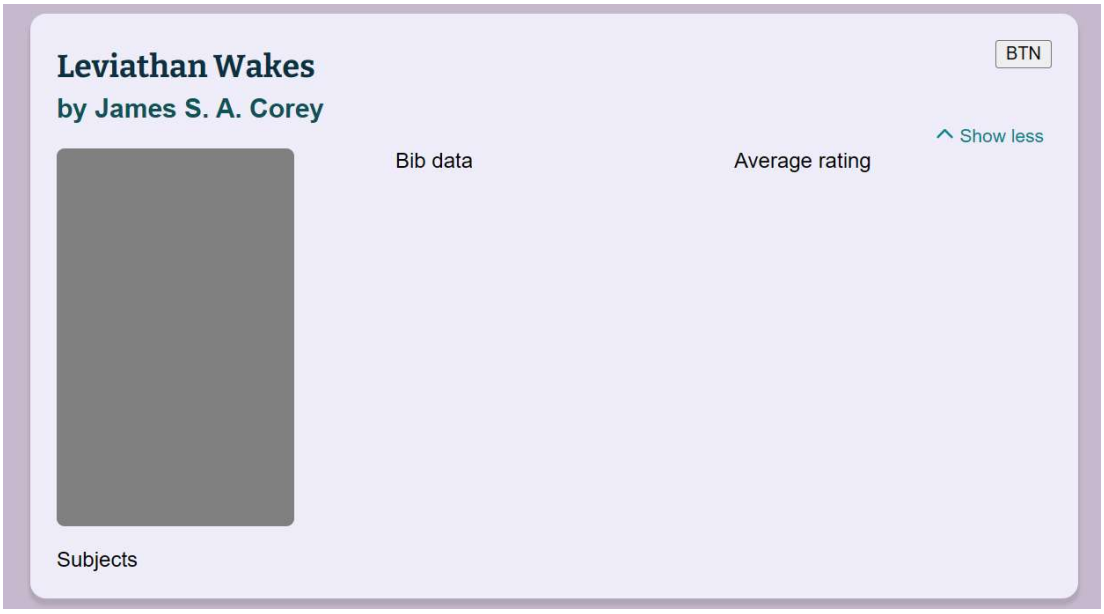
Your JSX should look something like this:

```
export default function ExpandedContents({ book }: ExpandedContentsProps) : Element {
  return (
    <div>
      <div>{mockCoverImage()}</div>
      <div>Bib data</div>
      <div>Average rating</div>
      <div>Subjects</div>
    </div>
  )
}
```

35. In the *scss* file, create a style class for the grid (**`display: grid`**) and use the **`grid-template-columns`** property to define the grid. Use the cover image width for the size of the first column and the generic fractional unit (**`fr`**) for the rest. Give the grid a 1.25rem **`gap`**.

    Here's a good tutorial on Grid on **`fr`** if you want to get a better handle on how it works: https://www.digitalocean.com/community/tutorials/css-css-grid-layout-fr-unit

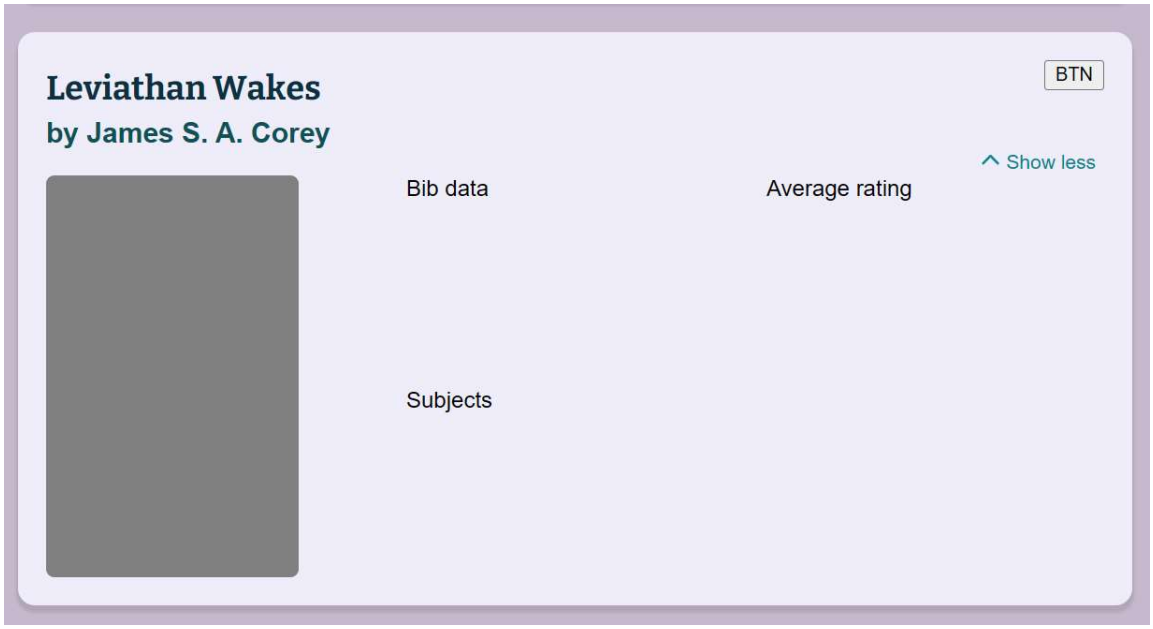
When you're done, apply the style class to your grid container and you should see this:



36. You'll notice that "Subjects" automatically wrapped onto the next row. While we do want it to do that, we don't want it to fall under the cover image. The cover image is supposed to take up two rows.

    Add a new class for the cover image and use the **`grid-column`** and **`grid-row`** properties to ensure it takes up one column and two rows.

    Now you should see this:



37. Great! The layout basically matches our plan. But before we do any more styling, lets plug in the real data. You'll need to add more divs inside the main children to get the layout to match the Figma. Go ahead and give it your best shot. Create the localized messages for translation while you're at it.

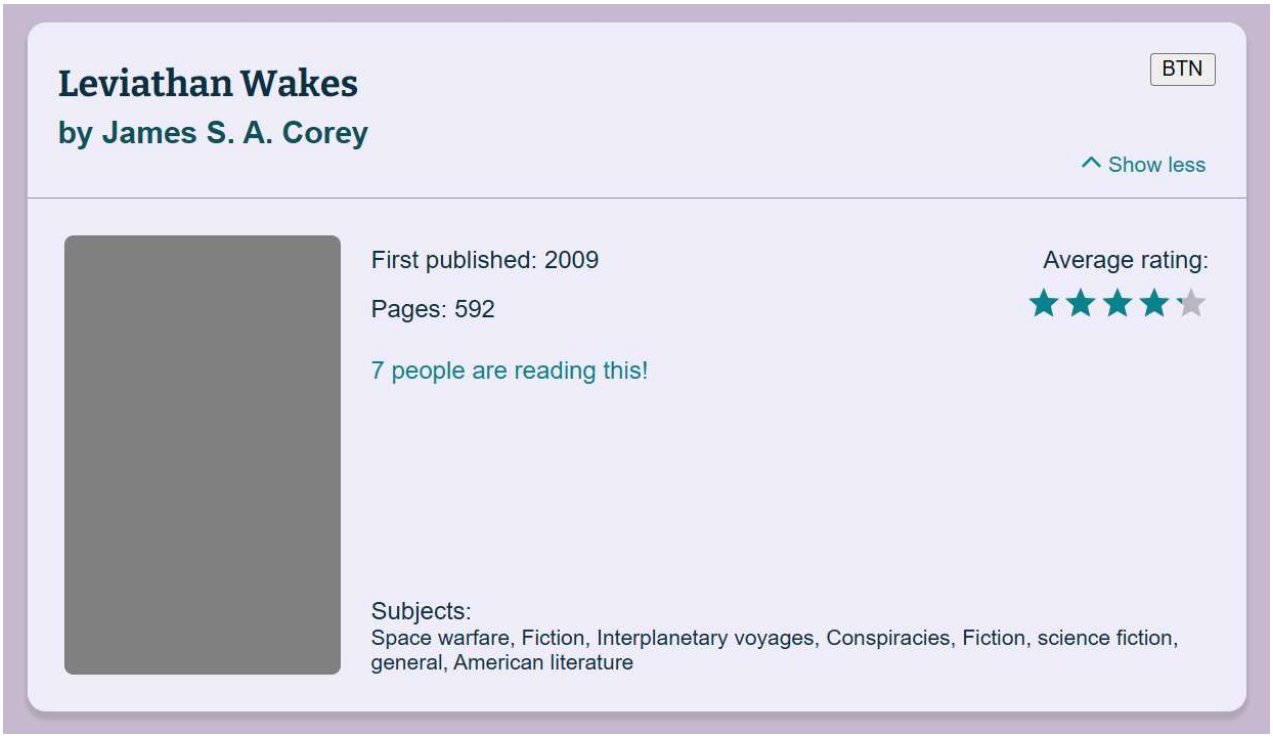
    To display the star rating, you'll need to use the Star Rating component from the UI Library.

    To get the subjects to display across two columns, you will have to use the **`grid-column`** property.

    Once you have it all in there, follow the Figma to style the rest. To get the contents to fall where you want inside their region in the Grid (left or right justified, for example), you can use the **`justify-self`** property on the Grid items, and **`text-align`** for text alignment

    Pay attention to the thin border that appears at the top of the expanded card, and the padding and margins throughout. You may have to adjust work you did on BookCard.

    When you're done, you should see the finished card contents:



EXTRA CREDIT: Add a transition animation to make the expansion smooth. You can refer to Stack Overflow for one way to achieve this – and while it is not necessarily the best solution, it does teach you how to create transition effects:
https://stackoverflow.com/questions/68465643/react-how-to-open-child-components-with-smooth-rendering