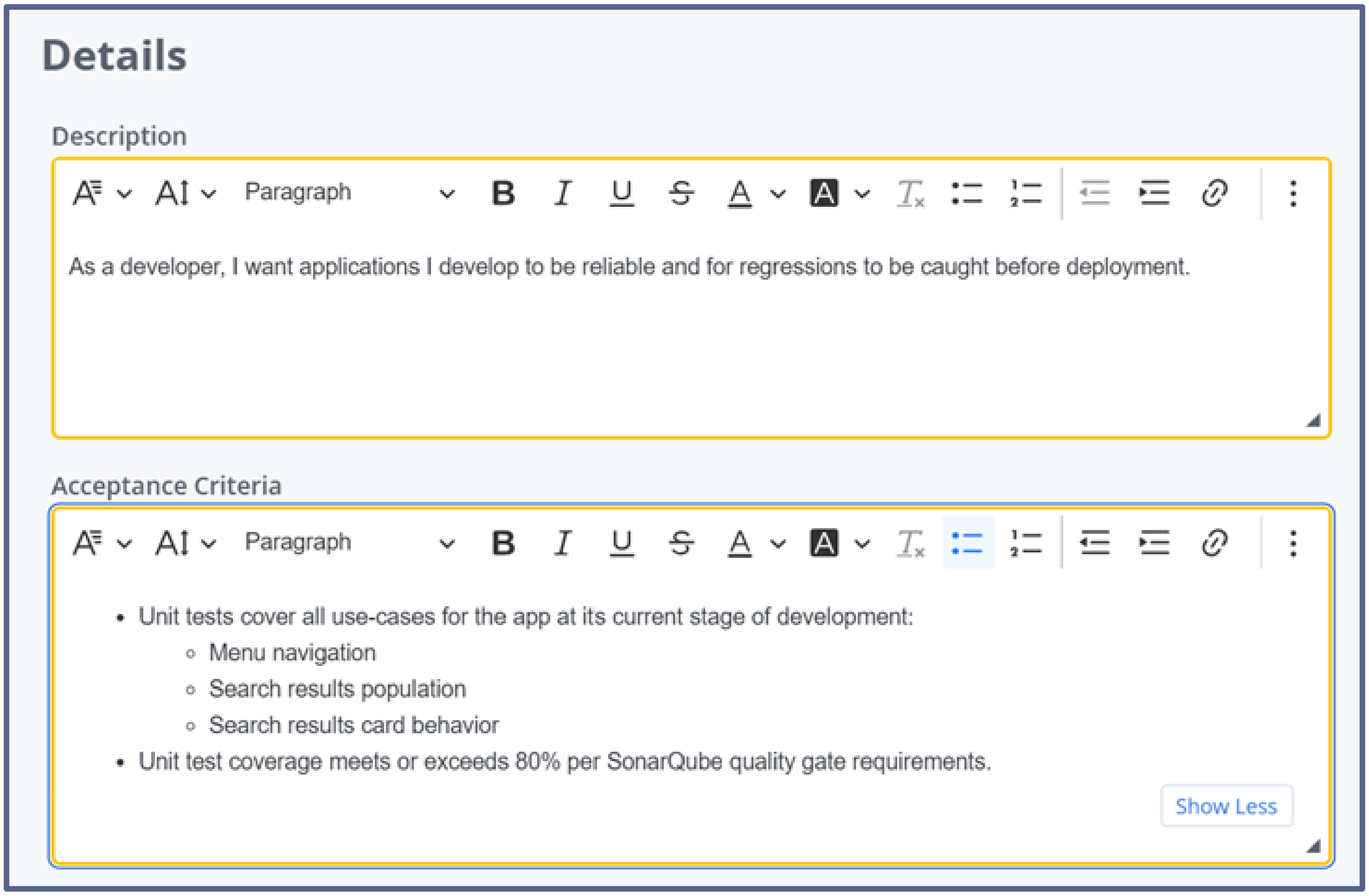
Build the Book Binge App - L005

Lesson Five: Add unit testing to make the Book Binge app more reliable.

If you get stuck, check out the **005/lesson-005-testing-complete** branch to see the final solution.

The Book Binge company has decided to implement new rules around testing to help ensure a high-quality application. It's up to us to implement test coverage.



As a developer, I want applications I develop to be reliable and for regressions to be caught before deployment.

- Unit tests cover all use-cases for the app at its current stage of development:
  - Menu navigation
  - Search results population
  - Search results card behavior
- Unit test coverage meets or exceeds 80% per SonarQube quality gate requirements.

We'll be using a popular combination of libraries to achieve this goal: *React Testing Library* with the *Jest* test runner.

React Testing Library is, as the name suggests, a library designed to test react components. It is also opinionated, intended to be used to mimic the ways users interact with applications while also encouraging developers to keep accessibility at the forefront of component design and testing.
https://testing-library.com/

Jest is a generic JavaScript test runner created and maintained by Meta and can support not only unit tests, but integration and end-to-end tests as well.
https://jestjs.io/

1. A new branch has been configured with *React Testing Library* and *Jest* : **005/lesson-005-testing-starter**. Branch off from that and run `npm install`.

2. Let's start with the nav bar, per the first use-case listed in our story's ACs. In /components/navbar, create a file called *index.unit.spec.tsx*.

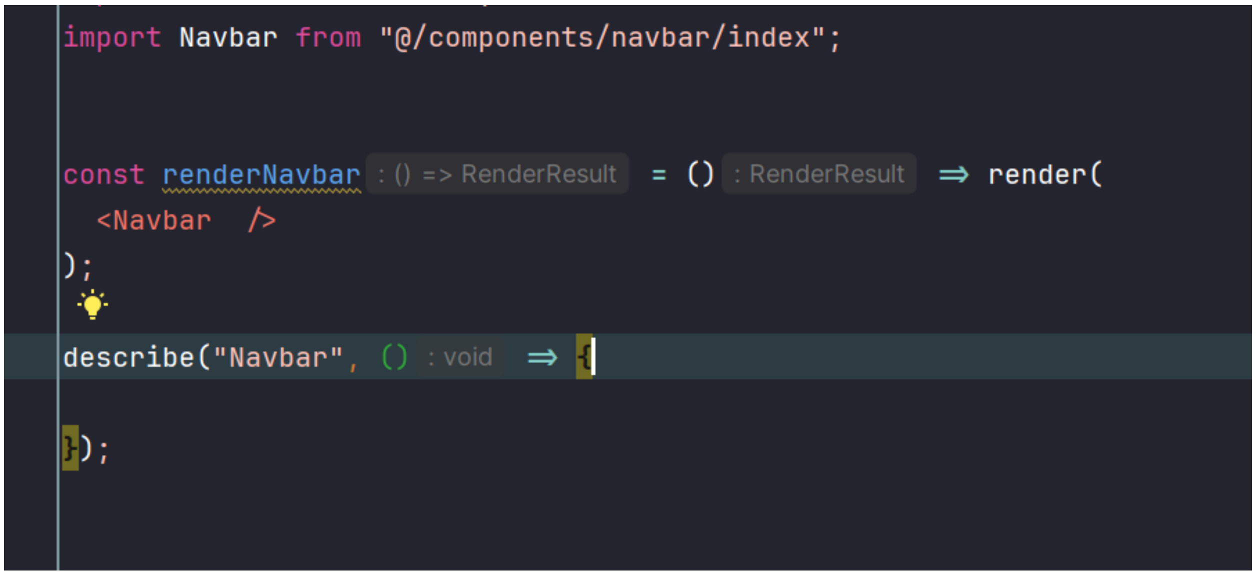
   The naming convention for the Jest test runner requires that we use certain key words—in our case, "spec"—for the test files to be identified. We'll use another convention of adding the keyword "unit" to indicate that the tests in question are unit tests. The first word in the scheme should follow the name of the file you are testing (this will usually be "index") and the file extension should match as well (typically this means it will be "tsx" since we are mainly testing React components).

3. A custom render method has been provided for you in a file called test-util-render, and you'll want to import this into every test file.

   ```
   import { render } from "@/test-util-render";
   ```

   This method is virtually the same as the render method from React Testing Library, but it adds a wrapper from React-Intl to make React-Intl hooks available within the testing environment.

4. To keep your tests organized and easy for other developers to understand, organize them into suites of tests using the "describe" block.



```
import Navbar from "@/components/navbar/index";


const renderNavbar : () => RenderResult  = () : RenderResult  ⇒ render(
  <Navbar  />
);

describe("Navbar", () : void  ⇒ {

});
```

   What exactly should we test inside the Navbar component?

   Your impulse might be to test that clicking on the nav links correctly redirects to the appropriate page. And that would be a great impulse! However, we cannot actually test such behavior using *react-testing-library* and *Jest* alone; due to how the Next router works under the hood, it would require a tool like Cypress, which uses a real browser.

However, the benefit vs time and cost of setting up such a test is probably not worth it. Let's consider why.

Go ahead and look at the code inside Navbar.

Go ahead and look. I'll wait.

What logic can you identify that we, the Book Binge developers, have written? We imported some Link components from the Next library and passed them text and an href.

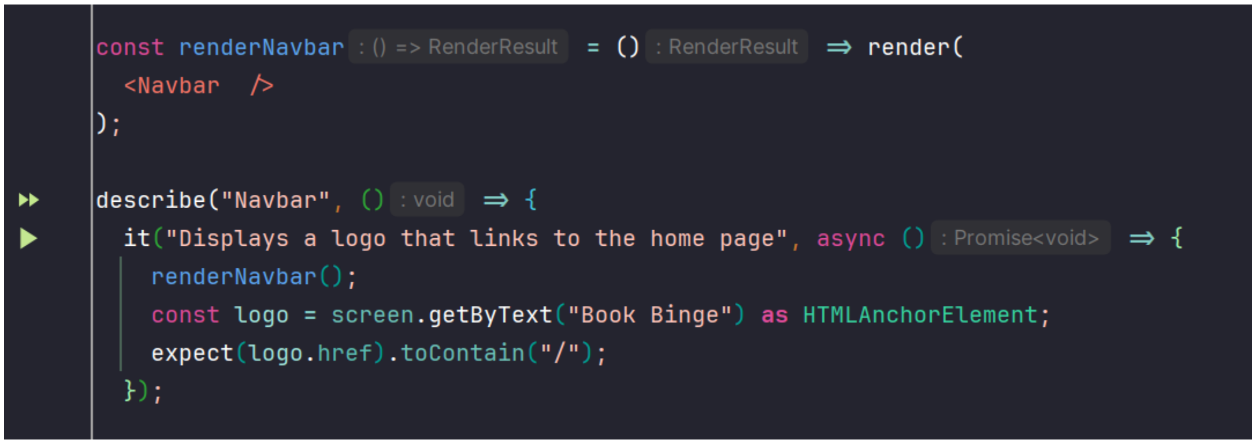
That's it.

We didn't write an *onClick* method; we didn't push a route onto the browser history. That's all handled by Next inside the Link component. And we can safely assume that Next thoroughly tests their own components.

Don't test third party logic. That's not your responsibility and your time is better spent testing the logic you yourself implemented.

On that note, what *should* we test here? –Don't overthink it. We just want to make sure that all the links we expect to be there are there.

5. Write a test for each link, ensuring that it can be found by the expected text, and that it will redirect to the expected path. Start with the logo. What if a future version of the app requires the *Book Binge* text to be swapped out with an image? What if we forget that the image also needs to be a link that points to the landing page? This test will catch such a regression:



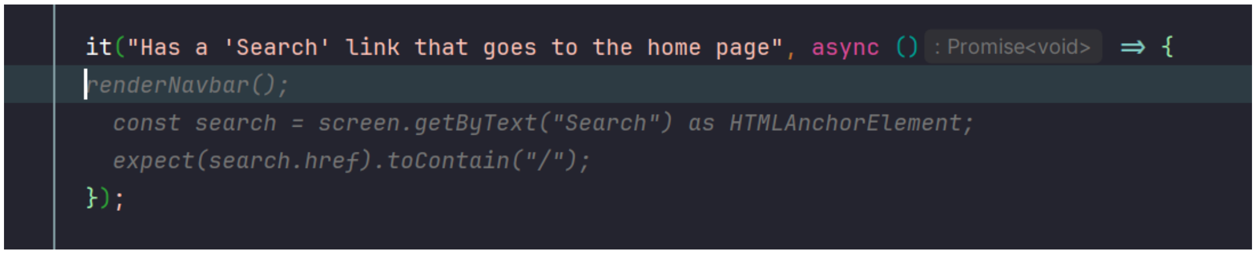
```
const renderNavbar : () => RenderResult  = () : RenderResult  => render(
  <Navbar  />
);

describe("Navbar", () : void  => {
  it("Displays a logo that links to the home page", async () : Promise<void>  => {
    renderNavbar();
    const logo = screen.getByText("Book Binge") as HTMLAnchorElement;
    expect(logo.href).toContain("/");
  });
```

** We need to use **`as HTMLAnchorElement`** to type the element we're accessing in order for TypeScript to allow us to reference the *.href* property in our *expect* statement.

6. Add similar tests for the rest of the links in the nav bar. And we're done with the first AC!

   If you are using GitHub Copilot, you may find that it will offer an autocomplete suggestion to write your unit test for you. It is especially accurate after you have written your "it" statement, and if you already have one or two similar tests written:



```
it("Has a 'Search' link that goes to the home page", async () : Promise<void> => {
renderNavbar();
  const search = screen.getByText("Search") as HTMLAnchorElement;
  expect(search.href).toContain("/");
});
```

   If you aren't using Copilot, I suggest giving it a whirl specifically in the unit testing context. As with all things Gen AI, check its work and verify that it is actually testing what you think it's testing, as it still gets things wrong. But overall, it can be a major time saver.

7. When you're all done, try **`npm run test`**. You should have four passing tests (five if you count the base test already included in the /__tests__ directory).

8. Moving on, let's test the book card behavior (the third AC). We'll save the testing of search results for last because it will be the most involved.

   Before you start writing tests, consider what you *want* to test. Namely, code logic. And we'll do this by having our test mimic the ways a user might interact with the application. Typically, by mimicking user behavior, including edge cases, we can easily achieve 100% coverage without having to isolate logic for testing in a vacuum.

   Furthermore, mimicking user interaction is a more useful way to test a UI because what we care about is *behavior* not implementation details. To quote the guiding principle of React Testing Library: "*The more your tests resemble the way your software is used, the more confidence they can give you.*"

   So, what do you want to test with the book card? Look through the BookCard component to refresh your memory. You may find it helpful to run the application and interact with it yourself to see all the ways a user might interact with the card.

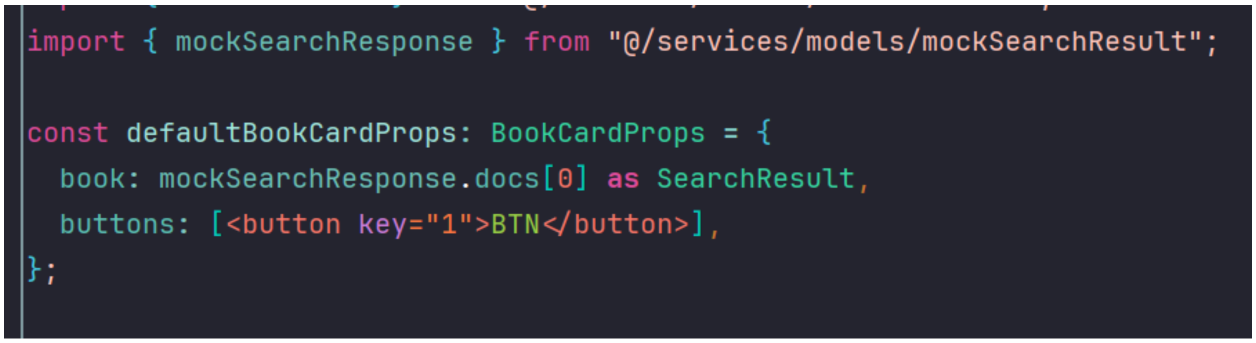
   When you think you have an idea of a test to write, create a test file in the /book-card directory.

9. Rendering the BookCard in our test will be a tiny bit more complex than rendering the Navbar component. This is because BookCard expects to receive props. Which means we need mock data.

   In fact, we are going to need mock data across other tests too, so we might as well make a universal object now. In the */services/models* directory, create a *mockSearchResult.ts* file. Export a *mockSearchResponse* that returns two dummy *SearchResults*. Feel free to copy the code below:

```
export const mockSearchResponse: SearchResponse = {
  docs: [
    {
      key: "1",
      title: "Title",
      author_name: "Author",
      first_publish_year: 2000,
      number_of_pages_median: 300,
      subject: ["Coolness", "Testing"],
      ratings_average: 3.5,
      currently_reading_count: 2,
      cover_i: 112233,
    },
    {
      key: "2",
      title: "Title 2",
      author_name: "Author 2",
      first_publish_year: 2002,
      number_of_pages_median: 302,
      subject: ["Testing", "React"],
      ratings_average: 4,
      currently_reading_count: 12,
      cover_i: 223344,
    },
  ],
};
```

10. Import the data into your BookCard test and set up some additional mocking:



```
import { mockSearchResponse } from "@/services/models/mockSearchResult";

const defaultBookCardProps: BookCardProps = {
  book: mockSearchResponse.docs[0] as SearchResult,
  buttons: [<button key="1">BTN</button>],
};
```

What is the point of this?

We want to make our BookCard render method dynamic so that we have the option to give it different props in our tests, allowing us to see different behavior. While we won't leverage this dynamism just yet, it will come in handy after we implement the other versions of the card in future lessons. Also, it's just a good pattern to know to make your testing life easier.

11. Create the render method, *renderBookCard*. Unlike *renderNavbar*, which took no arguments, we'll pass in *mockProps* with a default value equal to—you guessed it—*defaultBookCardProps*. Go ahead and set it up and pass on the props to BookCard. You can directly reference them by dotting into the object: *mockProps.book,* for example.

12. Time to write the first test! Where to start? How about starting simple by testing that the card renders all the data we expect to see? Go ahead and give it a shot.

    When you're done, try **`npm run test`** to see if your test passes.

    How did you decide to evaluate your test? Did you expect to "getByText" the title and the author? That's a perfect start, if so!

    Did you run into an issue that *getByText("Author")* is failing?

    What gives? Why would it find the title but not the author?

    The answer lies in how "getByText" behaves. It expects to find a full string match, not a partial one. You can see for yourself why this causes our test to fail by trying a little debug exercise...

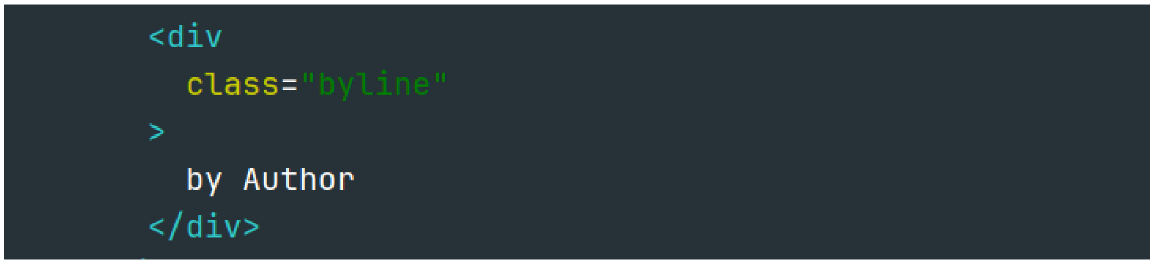
13. It's time to leverage React Testing Library's built-in *screen.debug()* method! Somewhere in your test (typically right before a failing line), enter the line:

    screen.debug(undefined, 600000);

    This tells the testing library to print everything in the DOM up to 600,000 characters. You can enter any number, but often the default value of 7,000 characters is not enough to see all the elements you need to see. 600,000 is a simple round number that covers virtually all use cases.

    The first argument, *undefined*, is also optional, but we need to pass it in to get to the second argument. The first argument can be used to specify a particular type of element to print out, but since we want everything, we'll just pass *undefined* instead.

14. Re-run your tests and you should see the DOM printed out. You may need to scroll up a bit to find it. Look for the "Author" field and how it gets rendered in the DOM.



```
    <div
      class="byline"
    >
      by Author
    </div>
```

Aha! *getByText* won't find "Author" because the string is actually "by Author."

Correct your test as needed.

**If you prefer working with regular expressions, you can pass them to getByText() instead.

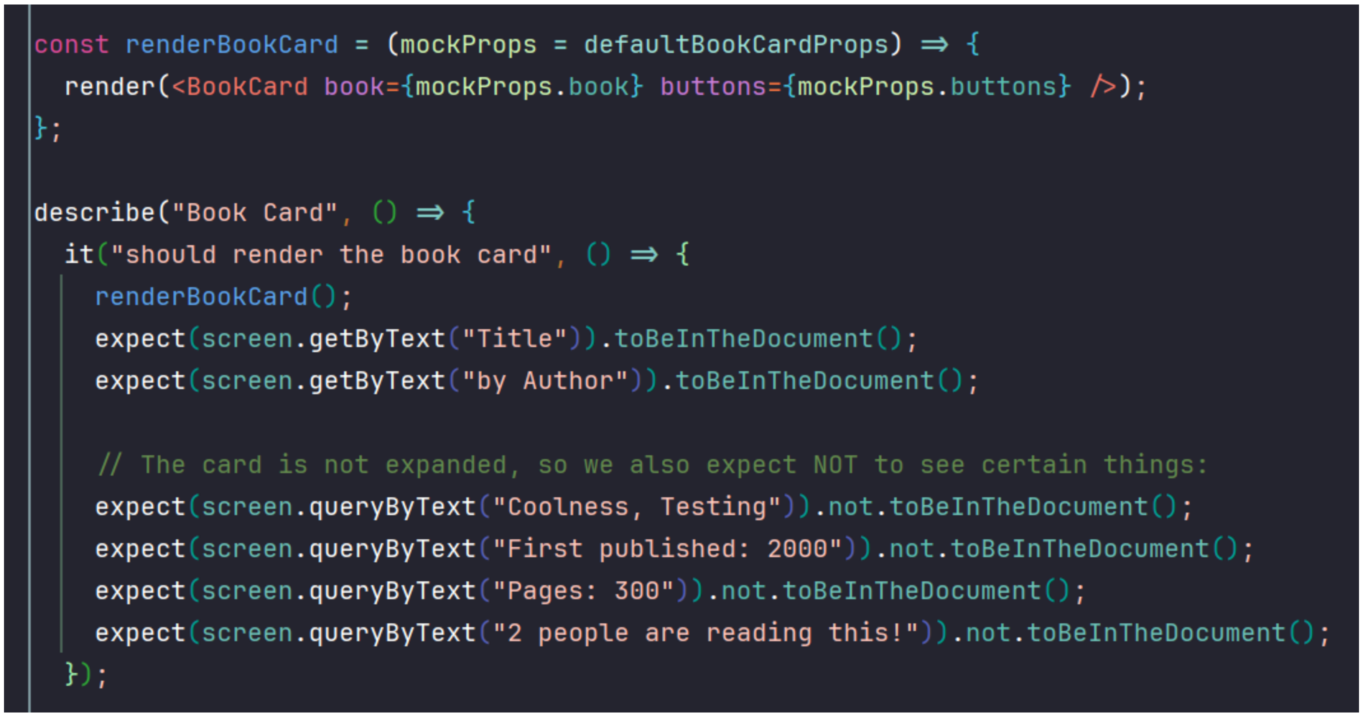
15. We have now tested that our card is rendering correctly by verifying that both the title and author are there, as expected. A great start! We can take it one step further, though. Since this is an expandable card, there are also several things we expect *not* to be there.

    Add a few lines following the format *expect(XXX).not.toBeInTheDocument()*.

    A word of caution though. If you use the "getByText" selector, the test will fail. When there is a possibility of returning *null*, which is what we hope in this case, you should instead use the "queryByText" selector.

    Learn all about queries here: https://testing-library.com/docs/queries/about/

    When you're finished, you might have something like this:



```
const renderBookCard = (mockProps = defaultBookCardProps) ⇒ {
  render(<BookCard book={mockProps.book} buttons={mockProps.buttons} />);
};

describe("Book Card", () ⇒ {
  it("should render the book card", () ⇒ {
    renderBookCard();
    expect(screen.getByText("Title")).toBeInTheDocument();
    expect(screen.getByText("by Author")).toBeInTheDocument();

    // The card is not expanded, so we also expect NOT to see certain things:
    expect(screen.queryByText("Coolness, Testing")).not.toBeInTheDocument();
    expect(screen.queryByText("First published: 2000")).not.toBeInTheDocument();
    expect(screen.queryByText("Pages: 300")).not.toBeInTheDocument();
    expect(screen.queryByText("2 people are reading this!")).not.toBeInTheDocument();
  });
```

16. Now for the real test. We need to simulate clicking the card to expand it, the same as a user would do, to see if we get the expected behavior.

    Start a new test. Because we are going to be testing asynchronous behavior (changes to the DOM based on user interaction), the test callback needs to be *async*.

    Get the “Show more" button and assign it to a const, *expandButton*. There are a couple of ways to do this. You could use good old reliable “getByText,” no problem there. But there is better selector that does double-duty: “getByRole.” In our case, it would look like this:

    ```
    const expandButton = screen.getByRole("button", { name: "Show more" });
    ```

    Why go to this trouble when “getByText” is shorter to write?

    It has to do with accessibility. Imagine we have a new junior dev excited to work on the Front End. The Design Team has come up with a whole new look for the “Show more” button. A perfect story for our new dev! The dev gets to work and realizes they don’t need to use that pesky HTML *button* element at all, with its ugly styles that have to be overridden. How much simpler to use a div, style it directly, and add on *onClick*?

    Our new dev tries to deploy their beautiful pixel-perfect button, but suddenly tests are failing in the PR scan!

    A ”getByText” test would have still passed, but because our test is looking for an element of role “button,” it now fails. Because a <div /> is not a < button /> no matter how much it looks and acts like one.

A sighted user wouldn’t notice the difference, but screen readers “see” the DOM differently. By using the “getByRole” query, we reinforce accessibility. You don’t have to use it if it won’t work in your situation, but always prefer it above other options.

17. Not that you have the button, go ahead and “click” it by using *fireEvent.click(expandButton).* This is preferred over using the lower-level .click() method directly on the element itself because it more closely mimics the event chain as experience by users in a browser.

    React Testing Library actually provides an even more accurate simulation of user mouse behavior (complete with hover side-effects) in their *userEvent* library, which must be separately installed, but it’s more than we need right now.

    NOTE: The click event is asynchronous. This means we need to wait for all side effects to complete before continuing with our test. Use this simple callback:

    ```
    await waitFor(() => fireEvent.click(expandButton));
    ```

18. Now “getByText” all the things you expected *not* to find in the first test. Check your work by running the tests.

19. With just those two tests we have already achieved 100% coverage on our Book Card. Go ahead and check by running **`npm run test:coverage`**.

    You may be surprised at how great our coverage is already looking! The truth is, there is not much complex logic in our app yet. The green and yellow you see on files we haven’t even touched with our tests is simply from rendering the app root in the base */__tests__* suite.

20. The coverage report looks great, but there is more to test. The only use case we haven’t tested is closing the card again and making sure everything we want to go away does go away. By testing that, we can also verify that our button text changes as we expect it to, from “Show more” to “Show less.”

    Go ahead and create this third and final test. Expand and collapse the card and verify that all is as it should be. If you get stuck, you can check out the **005/lesson-005-testing-complete** branch to compare your work.

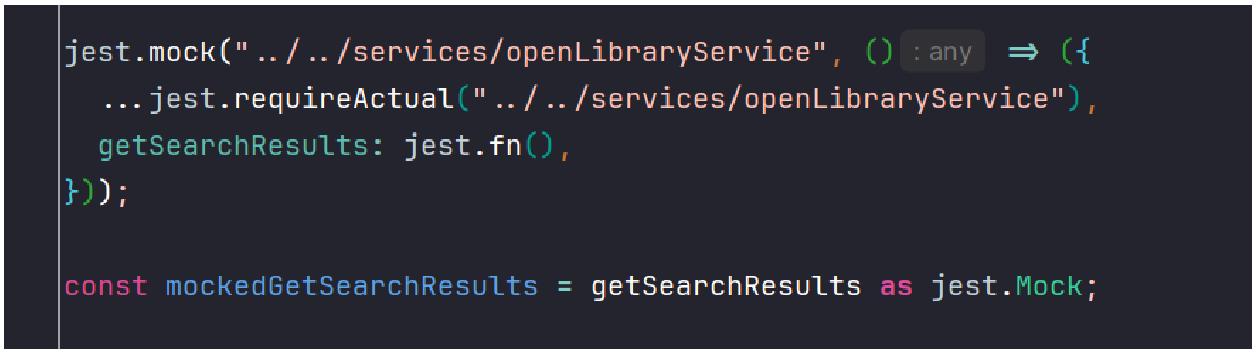
21. Time for our greatest challenge yet: testing search functionality. Because the search hits an API endpoint, we need to mock the Open Library service. Lucky for us, Jest comes bundled with a simple *mock()* method that allows us to isolate and control whole libraries and modules—or even just individual functions. However, the only thing you should ever need to mock, in 99% of cases, is API calls.

Do not mock down-tree React modules that you have written as part of your application. That only creates an artificial testing environment instead of running your code as it actually exists, making the tests less useful.

To get started, add a test file to /main-container. We want to test search here because the side-effects in the UI that we want to test are handled by the main container. We will not need to add separate tests to /search-banner. We're hitting it all here.

22. At the top of the test file, mock /services/openLibraryService and the *getSearchResults* function. This is the only function we want to mock from the API service at this time, so we'll require the actual module, and mock out the function so that we can control its output in our test.

    You can read more about module mocking and bypassing in the documentation: https://jestjs.io/docs/bypassing-module-mocks



```
jest.mock("../../services/openLibraryService", () : any => ({
  ...jest.requireActual("../../services/openLibraryService"),
  getSearchResults: jest.fn(),
}));

const mockedGetSearchResults = getSearchResults as jest.Mock;
```

23. Create a render function for Main Container and open a *describe* block. For the first test, we just want to make sure that the landing page renders as expected, when there is no search query entered. Go ahead and write the test using the strategies you've already practiced.

24. For the next test, we'll simulate a user entering a search. This will allow us to hit most of the logic throughout our application, not just the code inside Main Container. That's because we'll interact with the search bar the same way a user would and cause a render of search results, testing the expected behavior instead of implementation details.

    First, you'll want to create a mock resolved value for the *mockedGetSearchResults* function so that when it executes, it returns our mock data.

```
mockedGetSearchResults.mockResolvedValueOnce({
  status: 200,
  data: mockSearchResponse,
});
```

We’re not doing anything in our logic with the status code, so it’s not necessary to include it, but it better imitates the behavior of the API call.

25. Next, render the main container and select the search bar by using “getByPlaceholderText.” This approach not only gets us the element we want to interact with, it tests that the search bar has the expected placeholder text.

26. We’ll enter a search term by using the *fireEvent.change()* method:

    ```
    await waitFor(() => fireEvent.change(searchBar, { target: {
    value: "test" } }));
    ```

    This means we are entering “test” as the value in the field. You can use any term you like, since our API result is already a forgone conclusion thanks to our mock.

27. Now get the search button by role and name, and click it.

    We have just simulated a user’s actions to enter a search. What results do we expect?

    The page title should change to “Search results,” and the list of (two) results should appear. Use “getByText” to check and see.

28. Run your tests to verify that everything passes, and with that, the bulk of the work is done! There is one more logical branch we need to hit though. Our call to the API is handled inside a try-catch block, so we need to test that it can handle an exception.

    Once more, you’ll mock the response from *mockedGetSearchResults*. This time use the *mockRejectedValueOnce* method and have it return a **`new Error(“Error message”)`**. It doesn’t matter what message you pass as the error, since our app’s error handling is still very rudimentary.

29. Now execute the same user flow as in the previous test, but instead of expecting to see the Search results, we expect to still be on the landing page.

30. You should now have three passing tests in Main Container, and 100% code coverage on both the Main Container and the Search Banner.

    However, there is one pesky problem. Because our (very rudimentary) error handling logs a *console.error* whenever the API call throws an error, our tests spit out a patch of angry red.

    We would want the tests to log errors if it’s something we aren’t expecting, but in this case, it's an error we are intentionally creating. Having unhelpful errors log out in our test output makes the test results harder to peruse and true errors harder to identify.

Can anything be done?

31. Inside the *it* block of your test, at the top, use jest to mock the *console.error* method:

    ```
    const originalError = console.error;
    console.error = jest.fn();
    ```

    This will silence the error. At the bottom of the test, reset *console.error* back to its original function. If you miss this step, all console errors will be mocked for the remainder of the tests, and we will miss true errors that we want to log out!

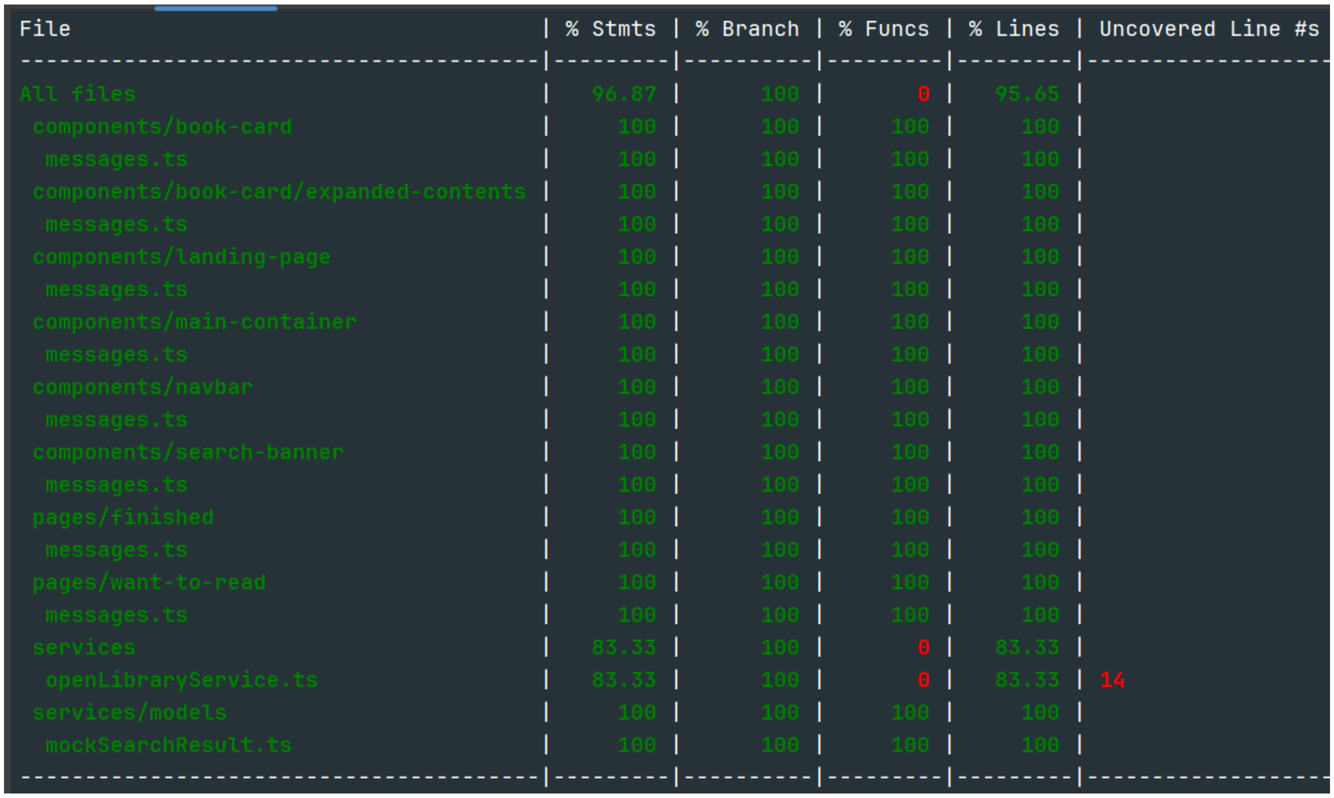
    ```
    console.error = originalError;
    ```

32. The only pages not covered by our tests are /want-to-read and /finished. This makes sense—we haven't rendered them in any of our tests.

    However, they also have no functionality, so there is little to test. As we build out the rest of the app, we'll be adding unit tests as part of the process.

    And so, you may now consider this lesson 100% covered!

EXTRA CREDIT: On second thought, keep going and achieve full coverage on the *Want to read* and *Finished reading* pages! It will only take one test per page.



```
File                                     | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------------------|---------|----------|---------|---------|------------------
All files                                |   96.87 |      100 |       0 |   95.65 |
 components/book-card                    |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 components/book-card/expanded-contents  |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 components/landing-page                 |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 components/main-container               |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 components/navbar                       |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 components/search-banner                |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 pages/finished                          |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 pages/want-to-read                      |     100 |      100 |     100 |     100 |
  messages.ts                            |     100 |      100 |     100 |     100 |
 services                                |   83.33 |      100 |       0 |   83.33 |
  openLibraryService.ts                  |   83.33 |      100 |       0 |   83.33 | 14
 services/models                         |     100 |      100 |     100 |     100 |
  mockSearchResult.ts                    |     100 |      100 |     100 |     100 |
-----------------------------------------|---------|----------|---------|---------|------------------
```