Practical: Testing React apps with Jest and React Testing Library

Initial Due Date: 2023-03-07 8:15AM
Final Due Date: 2023-03-31 4:15PM

Goals

- Learn some basic techniques for testing React apps
- Gain experience using Jest and the React esting Library
- Gain more TDD practice

Prerequisites

1. Create the git repository for your practical by accepting the assignment from GitHub Classroom. This will create a new repository for you with a skeleton application already setup for you.
2. Clone the repository to you computer with 💻 `git clone` (get the name of the repository from GitHub).
3. Open up the `package.json` file and add your name as the author of the package.
4. Install the module dependencies by executing 💻 `npm install` in the terminal.

You previously used Jest for unit testing JS code. Today we are going to use Jest in combination with the Testing Library library to test a React application. There are a number of items that need to be installed, but the project skeleton includes everything you need.

Regression Tests

Smoke test

The easiest form of testing we can perform is called a smoke test. Unlike the testing we saw earlier, we aren't going to assert anything, nor will we test anything explicitly. All we will do is try to render a component. If the process throws any errors, the test will fail, otherwise it succeeds. This kind of test is great for quick and dirty *regression testing*, where we are trying to make sure that adding new features or fixing bugs hasn't inadvertently added any errors in unexpected places. Note that it doesn't actually tell you that the *functionality* hasn't been broken, just that it didn't catch fire as it were (the name comes from the hardware side of the world, where a smoke test means "plug it in and see if smoke comes out").

For our smoke test, we will test the whole render tree of the `FilmExplorer` component. You will find the file `FilmExplorer.test.js` in the `src` directory. We have already provided the imports for you. All we need to do is render the component, so add the following to create your smoke test:

```
test("Smoke test",  () => {
  render(<FilmExplorer />);
});
```

Copy

Snapshots

Jest provides another quick regression testing tool, snapshots. You take a snapshot of the component at a time when you like the way it looks. Jest saves a representation of the component, and then every time you run the tests, Jest regenerates the component and compares it to the snapshot. If the component changes, the test will fail, which is a cue to either fix the offending code, or to take a new snapshot because the change was intentional. Note that the snapshot is not a literal picture, it is a JSON description of the component that can be quickly compared. Add the following to your `FilmExplorer` tests to create the snapshot:

```
test("Snapshot test",  () => {
  const { container } = render(<FilmExplorer />);
  expect(container.firstChild).toMatchSnapshot();
});
```

Copy

Here we are getting the `container` from the `render` function. The `container` is simply a `div` containing the rendered element. To get the actual root node, we ask for `container.firstChild`.

Note that we didn't write anything to generate the snapshot. Jest will do that automatically the first time the test is run. Go ahead and run the tests. You will find that Jest has created a new directory called `__snapshots__` in the same directory as your test file. Open this up and look at the snapshot that is stored in there. This should be committed with your code so that subsequent tests can use it.

So that you can see how the snapshot test works, go into `SearchBar.js` and find where the place where write the name of the app ("FilmExplorer") and change it to something else. If you run the test now, the snapshot test will fail. Notice that you are provided with a diff showing what has changed. Of course, sometimes you will be making a change and you *want* the page to be different. You can update your snapshot with `npm test -- -u` (or if you are running the test watcher, `npm test -- --watch`, you can just type `u`). Update your snapshot to acknowledge your change.

Can you use snapshots for TDD? Peek at the answer.

TDD with React

If you look carefully you will see that there is a new feature that has been added to Film Explorer. There is a small arrow next to the sort tool. If you click it, nothing happens, but we would like it to change the sort order of the films.

If you look in the code, you will see that the `FilmExplorer` component has a new piece of state called `ascending`, which is passed down to `SearchBar` to determine the direction of the arrow, but currently the state is not updated by clicking the arrow. You will now practice some Test Driven Development (TDD) by writing tests to reflect what the `ascending` prop *should* do, and then writing the code so that it *does* do it.

Testing state changes

We have a general pattern that we follow when writing tests around state changes.

1. Test that we are in the initial state
2. Initiate an action that should change state
3. Test that we are in the new state
4. Initiate action to return state to original

5. Test that we are in original state.

The first step is often overlooked, but important to establish that the state change is moving from a known baseline state. Without that, we can't know that a state change actually occurred. The last two steps are less important, but worth doing when the state is a binary toggle like our arrow.

Add a new test to `FilmExplorer.test.js` called `'Arrow changes direction'`. *Note that we are testing the `FilmExplorer` component, not the `SearchBar` component. While the indicator is in the `SearchBar`, the state is in `FilmExplorer`. We could simulate the clicks and make sure that the callback function was called with the correct value, but we are going to be satisfied with more of an integration test here.*

Start by copying the code from the "Rating changes" test to get the component mounted and initialized with the data.

We need to find the arrow component in order to test that it changes its display to reflect state changes, and also to simulate clicking it to initiate that change. To find the component, we will use `let arrow = screen.queryByText("▲")`. There are other options, but since the arrow is implemented as a single character, this is a pretty unique text string to look for. We are also using `queryBy` because it will allow us to query for things **not** being in the DOM rather than throwing an error. We will test if the component is visible using the matcher `.toBeInTheDocument()`. With jest, we can always invert a test by adding a `not` to the matcher (e.g., `.not.toBeInTheDocument()`). If the state changes, we can detect it, because our `"▲"` will be gone and replaced with `"▼"`. To simulate the click, we will use `fireEvent.click(arrow)`.

Put these together to write a test of this scenario:

Given that the page is rendered and the arrow defaults to `"▲"`, when the user clicks on the arrow it reverses direction to become `"▼"`. When the user clicks again, the `"▲"` is restored.

Note that for each changed state of the arrow, you need to run the query again.

The test should:

1. Render `FilmExplorer`
2. Check that `"▲"` is in the document
3. Simulate a click on the up arrow
4. Check that `"▲"` is no longer in the document
5. Check that `"▼"` is in the document
6. Simulates a click on the down arrow
7. Check that `"▼"` is no longer in the document
8. Check that `"▲"` is in the document1.

Run the test. It should fail (the arrow is currently just text).

Let's update `SearchBar` so that the test passes. Look through the code to find where the arrow is displayed. You will see that it is a simple `span` with a text arrow in it. The first thing to do is fix the arrow itself so the direction is determined by the `ascending` prop. Replace the arrow character with `{ascending ? "▲" : "▼"}`.

Next, we need to add an `onClick` handler so that the user can actually interact with it. Since the value is just a Boolean, we can simply invert it whenever the user clicks: `onClick={() => { setDirection((currAscending)`

`=> !currAscending); }}`. Note we are using a slightly different form of the setter where we supply a function instead of the value. This ensures that the update is applied to the current value of the state.

Run the test again, it should now pass!

Get sorting working

Clicking the arrow should now flip it back and forth, but it doesn't change the sort order, which it seems like it should. To make this happen, we need to turn our attention to `FilmTableContainer`, the other component rendered by `FilmExplorer`.

As its name suggests, `FilmTableContainer` is a "container component" (CC). It implements the film filtering and sorting. The actual presentation of the films is handled by the `FilmTable` (a "presentation component" or PC). `FilmTableContainer` works by transforming the Array of films its receives as a prop to create a new Array that is passed to `FilmTable` as a prop. `FilmExplorer` is also already providing the value of `ascending` to `FilmTableContainer` as a prop, so we just have to worry about what `FilmTableContainer` is doing with it.

Inside of the `components` directory, you will find `FilmTableContainer.test.js`, which already includes a collection of tests. We will walk through some of these before adding some new ones. In the first test we are making sure that with an empty search term we see all films. Since there is no filtering, we expect all of the films to be present in the DOM. Notice that we are passing in `jest.fn`, the jest mock function, as a placeholder function for the `setRating` prop. We don't care about the callback, but we also don't want PropTypes to complain, so we have to pass in a valid function.

```
test('Empty string does not filter films', () => {
  render(
    <FilmTableContainer
      films={films}
      searchTerm=""
      sortType="title"
      setRatingFor={jest.fn}
      ascending={true}
    />
  );

  films.forEach((film)=>{
    expect(screen.queryByText(film.title)).toBeInTheDocument();
  });
});
```

In the next test, we are looking at the filtering behavior of the component. In order to do that, we need to introduce a search term (in this case `"sub"` which only appears in the description of one film).

```
test('Any substring satisfies the filter', () => {
  render(
    <FilmTableContainer
      films={films}
      searchTerm="sub"
      sortType="title"
      setRatingFor={jest.fn}
```

```
        ascending={true}
      />
    );

    expect(screen.queryByText(films[0].title)).toBeInTheDocument();
    expect(screen.queryByText(films[1].title)).not.toBeInTheDocument();
    expect(screen.queryByText(films[2].title)).not.toBeInTheDocument();
  });
```

Take a moment to read through the remaining tests in filtering test suite and make sure you understand what they are doing.

For our new feature, we want to think about the sorting order. I have created another test suite to group the tests dealing with sort order. Inside you will find one test that tests that the films are sorted by `vote_average`. Let's break down this test as well. The challenge we have is figuring out what order items are displayed on the page (or more properly, in the DOM). The approach I took was to use `screen.queryAllByRole("heading")`. The "heading" role is grabbing the `<h2>` tags used to display the film titles, and by using the `getAllBy*` query, we are getting an array of the DOM elements that the Testing Library finds on the page in the order they appear. I then used `map` to extract the textual contents of those elements and put them in a new array.

The actual test part compares this new array of titles to an array of film titles assembled in the known correct order.

```
  test("Sorts by vote_average", () => {
    render(
      <FilmTableContainer
        films={testFilms}
        searchTerm=""
        sortType="vote_average"
        setRatingFor={jest.fn}
        ascending
      />
    );
    let items = screen
      .queryAllByRole("heading")
      .map((item) => item.textContent);

    expect(items).toEqual([films[0].title, films[1].title, films[2].title]);
  });
```

Of course, this test doesn't really test our new prop: `ascending`. Let's apply our "check, change, check" pattern here.

In order to do this tests, we need to be able to re-render the component and look for changes. For this, we need to have access to the `rerender` function. The `rerender` function is actually returned in the object returned by `render`. So, change `render` to `const { rerender } = render`.

After the assertion that checks if we have the films in the right order, call `rerender`, passing it the `FilmTableContainer` just as we did to `render`, but this time set `ascending` to `false`. Then, repeat the two steps

of gathering the list of film titles and comparing it to the known good ordering (which should be the reverse of the one we used in the first assertion).

Once you have that first sort ordering test written, repeat the process two more times to create tests `"Sorts by title"` and `"Sorts by release_date"`. Look at the sample films to figure out which order you expect them to be. This is one of the few moments when it will be okay to copy a block of code, paste it back in and change a couple of small values.

Run the tests

Again, we are practicing TDD, so when you run the tests they should fail. However, I want to also take this moment to remind you how to target specific tests.

By default, jest will look for tests in all files that end in `.test.js`, as well as any files in directories called `__tests__`. However, we can target a specific file by passing its name as an argument to the tool. In truth, it doesn't even need to be the full name of the file, jest will do case insensitive pattern matching. So, to just run the tests in `FilmTableContainer.test.js`, you can type `npm test table`. That will run all test files including the word table (which is just the one in this case).

We can get even more granular than that. For both test suites and individual tests, we can modify them with `only` or `skip` to focus in on the tests we are most interested in. So, to focus on a single test, you can write `test.only(name, test-func)` (see the docs for more details).

Fix the code

Okay, now that we have some failing tests, let's get the sort working properly. The way that Array.sort works is we pass it a comparator function `f(a,b)`, which is expected to return a value that is less than zero, equal to zero or greater than zero (corresponding to `a < b`, `a == b`, and `a > b` respectively). In `FilmTableContainer`, find the place where the films are sorted. Replace the comparator function with the following to adjust the sort ordering based on the `ascending` prop.

```
(m1, m2) => {
  if (m1[sortType] < m2[sortType]) {
    return ascending ? -1 : 1;
  } else if (m1[sortType] === m2[sortType]) {
    return 0;
  }
  return ascending ? 1 : -1;
}
```

Copy

Run the tests again, and they should all pass.

Finishing Up

1. Add and commit your changes and push those commit(s) to GitHub.
2. Submit your repository to Gradescope as described here

Grading

Required functionality:

- Smoke test
- Snapshot test
- Sort arrow tests
- Sort order tests
- Films are sorted properly
- Pass all tests
- Pass all ESLint checks

Recall that the Practical exercises are evaluated as “Satisfactory/Not yet satisfactory”. Your submission will need to implement all of the required functionality (i.e., pass all the tests) to be Satisfactory (2 points).
