

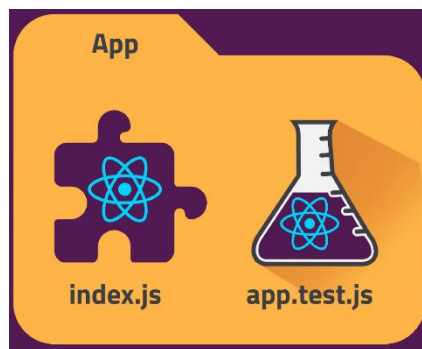
Testing in React Apps

“The more your tests resemble the way your software is used, the more confidence they can give you” – Kent C. Dodds. (We’ve seen the above quote before in the testing week 6.) (<https://kentcdodds.com/>)

As much as possible – run your tests in the same way as how a user would use your app. The whole point is to be confident the app works as expected.

We’re going to be looking at an npm package that can help us test our React apps, and its entire philosophy is based on the above quote.

Keep your tests for each file in its respective folder, like the App folder (This is a best practice pattern that you will see a lot!):



This is important, particularly as your app grows, so you can keep things organised. It also tells your testing library which tests belong with which file.

A lot of our Jest knowledge will transfer over!

Remember the 3 A pattern – Arrange, Act and Assert. This transfers over to testing in React.

Create-react-app comes with a lot of testing bundled in already, such as `@testing-library/jest-dom`, `/react` and `/user-event` – that’s why there is a npm test script in package.json. This means you don’t need to go about installing them separately – in fact Arshi suggests you don’t want to install Jest separately as it will cause conflicts with the libraries that come with create-react-app!

React Testing Library

- READ THE DOCS!
- Npm package, so can install it in the usual ways
- Lightweight solution for testing React components
- Uses actual DOM nodes, querying the DOM in the same way the user would (this is good as it means as long as your elements appear, the tests will still run, even if the behaviour has changed)

- Works with Jest but includes its own React-specific matchers as well, so can actually be used with any testing framework you choose

Testing Implementation vs. Behaviour – the trend used to be that people used to test implementation, but behavioural testing is more common and easier to do:
e.g. A function that adds $a + b$ could be implemented in many different ways. An example of testing implementation would be a test that checks for the $+$ operator within the function, so a valid function may fail even though it is providing the correct result. Behavioural testing is not looking “inside” the function to see what it is doing – just pass to the function and know what you’re expecting to get back. Whilst it may be argued that implementation testing is better for testing performance (as it is looking for the most efficient, “correct” code), you can still test performance in behavioural testing by measuring amount of time it is taking to return the result for example.

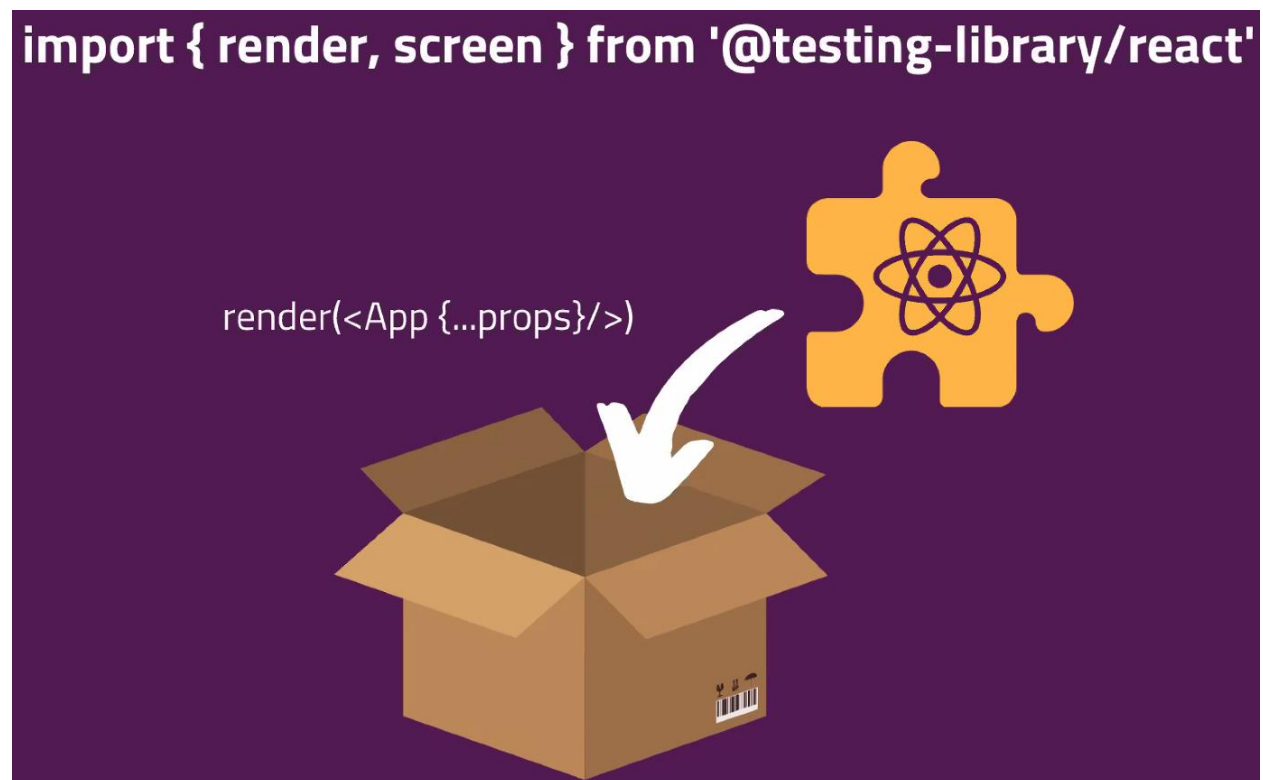
There are always trade-offs, but we will be using behavioural testing.

Remember to use TDD – write the failing test first so you can then write testable code, and the smallest amount of code needed to pass the test (so makes it concise).

An example of basic usage:

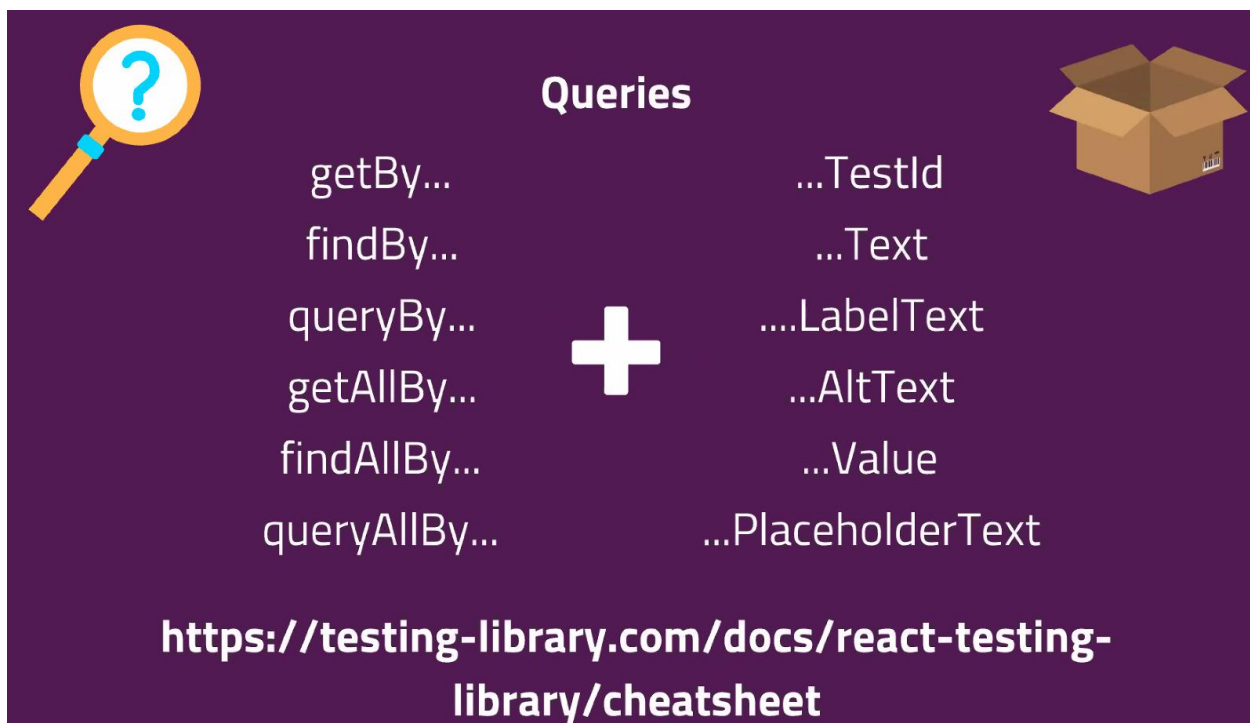
Import render and screen at top of file.

Use the render function and use the JSX code to render your App component, passing in any props using the spread operator (you could just pass props in as a whole object).



Queries are how you select things and use a matcher to select with – using the correct one will give you better feedback on the results if it fails. This is instead of using `querySelector`, as users don't care about selecting by ID/class etc, they're looking for text or label text – such as what a button is displaying as its text (e.g. a submit button should show “submit”). `TestId` is an “exception” case for when there is a specific reason as to why you might not want the `text/labeltext` to match for example (either it doesn't make sense or isn't practical to do so).

<https://testing-library.com/docs/queries/about/> - docs for using queries. Particularly look at “Priority” section as these are great methods for universal testing (including accessibility) in order of priority. Also consider the correct selector – as `get`, `find` and `query` have different behaviours (outlined in docs) and if you want multiple, then need `getAllBy`.



Queries

<code>getBy...</code>		<code>...TestId</code>
<code>findBy...</code>		<code>...Text</code>
<code>queryBy...</code>	+	<code>...LabelText</code>
<code>getAllBy...</code>		<code>...AltText</code>
<code>findAllBy...</code>		<code>...Value</code>
<code>queryAllBy...</code>		<code>...PlaceholderText</code>

<https://testing-library.com/docs/react-testing-library/cheatsheet>

Arshi went on to demo testing a simple `ResetButton` component:

As an aside – this code creates a file and opens it for editing immediately:

```
staff@Schools-MacBook-Pro-20 live-demo % touch src/components/ResetButton/ResetButton.test.js && code $_
```

Plan:

//End goal: Test the `ResetButton` component

Test that the `handleClick` is called when the button is clicked

//Write a test

Import necessary items to do the test (e.g. `{test, expect}` from `Jest`)

Arrange (setting everything up):

Render the ResetButton + pass it the props that it needs
Act (do the thing you're testing):

Click the button

How do we click something in our test?

How do we select the button?

Assert (check the outcome against what you're expecting):

Check that the handleClick prop was called

(Arshi acknowledge his human-readable string wasn't the best but he was running low on time.
We are testing that the handleClick prop gets called when we click the Reset Button)

```
import { test, expect, jest } from '@jest/globals';           for Jest feedback on our tests
import { render, screen } from '@testing-library/react';      To actually render items + check screen
import userEvent from '@testing-library/user-event';          To do the click event
import ResetButton from './index';                             To actually use the button in our tests
```

```
test("ResetButton handleClick", function () {
```

```
    const handleClick = jest.fn();
```

This function allows us to check handleClick has been called. This is a "Mock" function.

```
    render(<ResetButton handleClick={handleClick} />);
```

Pass in above variable

```
    //screen.debug()
```

This line of code is a useful tool we haven't learnt yet. It gives you visibility over what has actually been rendered (like a console.log but prettier). Use as an early step to check things are rendering then can comment it out if not needed)



```
console.log
<body>
  <div>
    <button
      class="reset-button"
    >
      Reset
    </button>
  </div>
</body>
```

```
    const button = screen.getByText('Reset');  Store the button in a variable
    userEvent.click(button);                  Click the button (passing in above variable)
    expect(handleClick).toHaveBeenCalled();    Assert handleClick variable was called using
                                              matcher that checks for a "mock" function
                                              (only)
```

```
  })
```

Arhi's codeblock from Slack:

```

// End goal: Test the ResetButton component
// - Test that the handleClick is called when the button is clicked
import { test, expect, jest } from "@jest/globals";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import ResetButton from "../index";

// Write a test
test("ResetButton handleClick", function () {
  const handleClick = jest.fn();

  // Arrange:
  //   Render the ResetButton + pass it the props that it needs
  render(<ResetButton handleClick={handleClick} />);

  //   screen.debug();
  // Act:
  const button = screen.getByText("Reset");

  userEvent.click(button);
  //   Click the button
  //   How do we click something in our test?
  //   How do we select the button?
  // Assert:
  //   Assert that the handleClick prop was called
  expect(handleClick).toHaveBeenCalled();
});

```

ResetButton code:

```
JS index.js × JS ResetButton.test.js U ●
demoCode > live-demo > src > components > ResetButton > JS index.js > ...
1 import React from 'react';
2 import '../App/app.css';
3
4 function ResetButton({ handleClick }) {
5   return (
6     <button onClick={handleClick} className="reset-button">
7       Reset
8     </button>
9   );
10 }
11
12 export default ResetButton;
13
```

Test code:

```
JS ResetButton.test.js U × JS index.js
demoCode > live-demo > src > components > ResetButton > JS ResetButton.test.js > test
1 // End goal: Test the ResetButton component
2 // - Test that the handleClick is called when the button is clicked
3 import { test, expect, jest } from "@jest/globals";
4 import { render, screen } from "@testing-library/react";
5 import userEvent from "@testing-library/user-event";
6 import ResetButton from "../index";
7
8 // Write a test
9 test("ResetButton handleClick", function () {
10   const handleClick = jest.fn();
11
12   // Arrange:
13   //   Render the ResetButton + pass it the props that it needs
14   render(<ResetButton handleClick={handleClick} />);
```

```
15
16   //   screen.debug();
17   // Act:
18   const button = screen.getByText("Reset");
19
```

```
20     userEvent.click(button);
21     //      Click the button
22     //      How do we click something in our test?
23     //      How do we select the button?
24     //  Assert:
25     //      Assert that the handleClick prop was called
26     expect(handleClick).toHaveBeenCalled();
27   });
28
```

We are using user-event v13.5 as that's what comes bundled. Use this instead of fireEvent, as it seeks to be more realistic/representative of the actual event a user does. (userEvent is more similar to Cypress - fireEvent makes the event happen internally). <https://testing-library.com/docs/ecosystem-user-event/>

Arshi little gems:

Ctrl + space will trigger intellisense for where you are in VS Code