MENU

👍 2 ❤️ 1
🦄 1 👏 1
🍺 1 🏆 1
😍 1 💰 1
🎉 1 🚀 1

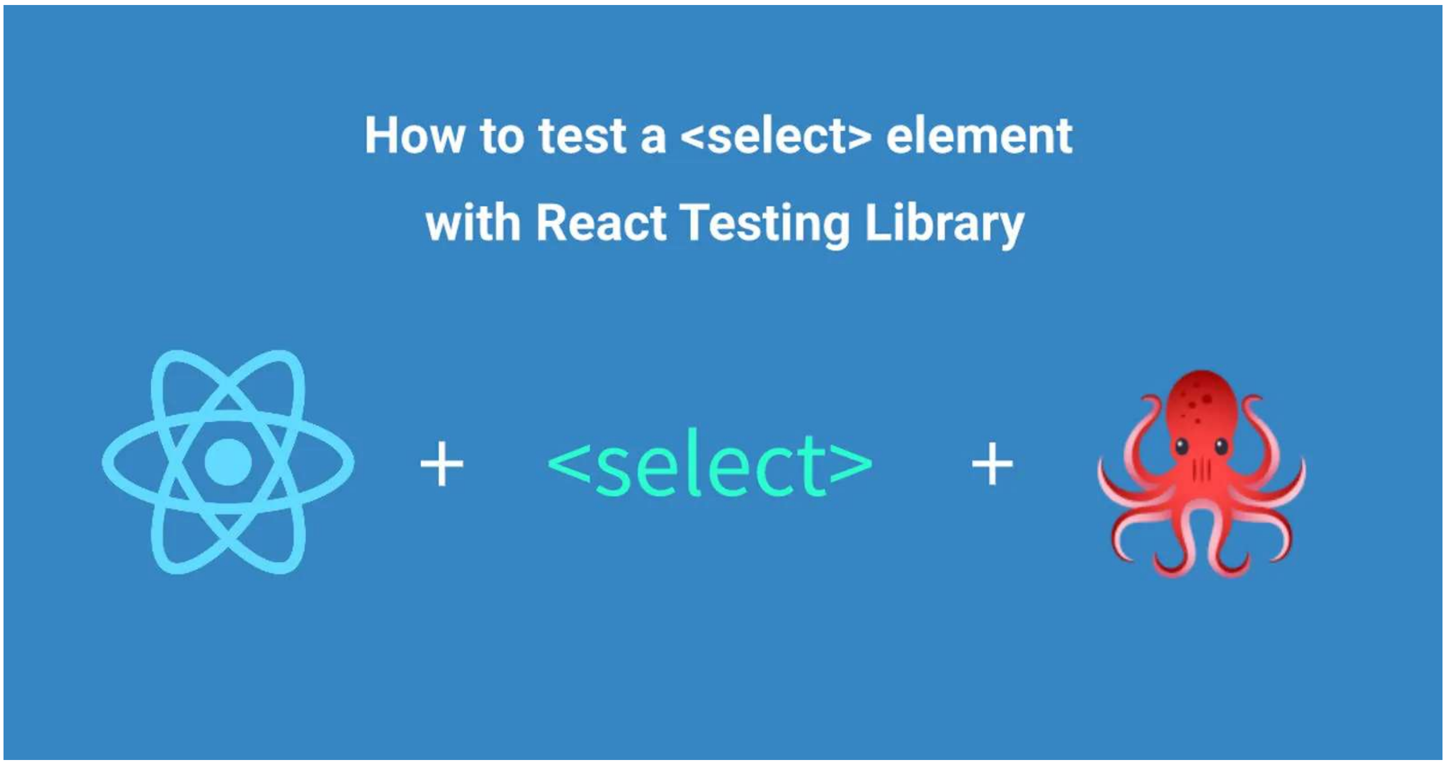
+ Follow

Tech with Cathal 🚀

How to test a select element with React Testing Library

Cathal Mac Donnacha

Oct 6, 2021 • 3 min read



I recently needed to add tests for a `<select>` element I was developing, and I couldn't find a lot of resources on how to do this with **React Testing Library**, so I'll share the approach I went with.

The `<select>` element

First of all, let's create a `<select>` element with some options. Here I have an array with 3 countries:

COPY

```
const countries = [
  { name: "Austria", isoCode: "AT" },
  { name: "United States", isoCode: "US" },
  { name: "Ireland", isoCode: "IE" },
]
```

Here's the `<select>` element itself, it has:

1. A default placeholder `<option>` asking the user to "Select a country".
2. A `.map` method so we can iterate over the `countries` array and add an `<option>` element for each one.

COPY

```
<select>
  <option>Select a country</option>
  {countries.map(country => (
    <option key={country.isoCode} value={country.isoCode}>
      {country.name}
    </option>
  ))}
</select>
```

Tests

Now that we have a basic `<select>` element which displays some countries, let's go ahead and write some tests! Yay...my favourite part 😀

The beauty of React Testing Library is that it makes you focus more on writing tests the way an actual user would interact with your application, so that's the approach I've taken with the tests below. Of course you may have your own unique scenarios, if you do, just think *"How would a real user interact with my select element?"*.

Default selection

COPY

```
it('should correctly set default option', () => {
  render(<App />)
  expect(screen.getByRole('option', { name: 'Select a country' }).selected)
})
```

Correct number of options

COPY

```
it('should display the correct number of options', () => {
  render(<App />)
  expect(screen.getAllByRole('option').length).toBe(4)
})
```

Change selected option

COPY

```
it('should allow user to change country', () => {
  render(<App />)
  userEvent.selectOptions(
    // Find the select element, like a real user would.
```

```
      screen.getByRole('combobox'),
      // Find and select the Ireland option, like a real user would.
      screen.getByRole('option', { name: 'Ireland' }),
    )
    expect(screen.getByRole('option', { name: 'Ireland' }).selected).toBe(tru
  })
```

😍 1 💰 1

🎉 1 🚀 1

Gotchas

Initially when I started to look into writing tests for these scenarios I went with the following approach:

COPY

```
  it('should allow user to change country', () => {
    render(<App />)
    userEvent.selectOptions(
      screen.getByRole('combobox'),
      screen.getByRole('option', { name: 'Ireland' } ),
    )
    expect(screen.getByRole('option', { name: 'Ireland' })).toBeInTheDocument
  })
```

Notice the difference? I was only checking that the "Ireland" `<option>` existed instead of checking if it was actually selected. Yet my test was still passing 🤔

COPY

```
  expect(screen.getByRole('option', { name: 'Ireland' })).toBeInTheDocument()
```

Let's take a look at why this happened. When the component is loaded, the following is rendered:

2 1

COPY

```
<select>
  <option value="">Select a country</option>
  <option value="US">United States</option>
  <option value="IE">Ireland</option>
  <option value="AT">Austria</option>
</select>
```

So from JSDOM's point of view, the "Ireland" `<option>` **always** exists within the document, causing my test to pass!

Whereas the correct approach is to use `.selected`:

COPY

```
expect(screen.getByRole('option', { name: 'Ireland' }).selected).toBe(true)
```

Gotchas like this can be just as dangerous as not writing the test in the first place as it gives you false confidence about your tests. This is why I always recommend intentionally causing your tests to fail, like this:

COPY

```
expect(screen.getByRole('option', { name: 'Austria' }).selected).toBe(true)
```

COPY

```
❌ Test failed: should allow user to change country
Expected: true
Received: false
```

👍 2 ❤️ 1 🦄 1 👏 1 🍺 1 🏆 1 😍 1 💰 1 🎉 1 🚀 1

This way you can be confident that it only passes for the intended scenario 🥳

Full code example

Here's a codesandox which includes the basic examples shown above.

Final thoughts

So there it is, you should now be able to write some basic tests for your `<select>` elements using React Testing Library. Of course, I'm not an expert on this topic, I'm simply sharing what I learned in the hope that I can pass on some knowledge.

If you found this article useful, please give it a like and feel free to leave any feedback in the comments 🙏

Want to see more?

I mainly write about real tech topics I face in my everyday life as a Frontend Developer. If this appeals to you then feel free to follow me on Twitter: twitter.com/cmacdonnacha

Bye for now 👋

#reactjs #javascript #frontend-development #beginners #testing-library

👍 2 ❤️ 1

🦄 1 👏 1

🍺 1 🏆 1

😍 1 💰 1

🎉 1 🚀 1

MORE ARTICLES

Cathal Mac Donnacha

👍 2 ❤️ 1

🦄 1 👏 1

🍺 1 🏆 1

😍 1 💰 1

🎉 1 🚀 1

Working around CORS in create-react-app

One problem we often face as frontend developers is dealing with CORS when making API requests. What...

Cathal Mac Donnacha

Cypress vs Playwright: Which is best for E2E testing?

Cypress was our go-to end-to-end (E2E) testing tool, and we were pretty happy with it, up until rece...

Cathal Mac Donnacha

Why you should make your tests fail

Let's face it, most of us developers don't necessarily love writing tests. We sometimes end up rushi...

👍 2 ❤️ 1

🦄 1 👏 1

🍺 1 🏆 1

😍 1 💰 1

🎉 1 🚀 1

Comments

+ Write a comment