React testing Library

Installation:

```
npm install --save-dev @testing-library/react
```

If you would have asked me a couple of weeks before about the go-to testing library to test ReactJs application, I would have answered **Jest** and **Enzyme** but recently I started working on React Hooks and was trying to test it using **Enzyme** but I found it is still not supporting React Hooks completely. Here is the open issue on GitHub

Why React-testing-library

I started looking for another testing library which is having better support for hooks and I found **React-testing-library** by Kent C. Dodds. This is even recommended by React team in their *official documentation.* So*,* I thought to give it a try.

Benefits of using React-testing-library

While working with **React-testing-library** you need to re-think your test cases. It enforces us to write test cases the way the user will interact with the application.

1. **No need to set up or initialize**. With **Enzyme** you need to set up an adaptor to make it working with React 16 but with **React-testing-library** there is no need to do it.

```
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new Adapter() });
```

2. No Shallow Rendering

It doesn’t have shallow rendering. It will render everything, It means it gives us the same environment as we get on the browser. **Enzyme** also have *mount* API which serves the same purpose. So it is not a real advantage over **Enzyme**

3. No need toforceUpdate

If you are using **Enzyme** with sinon and spying on some arrow function inside the component then you would have to force update the component

```
it('should fire handleClick when button is clicked', () => {
   const wrapper = shallow(<Component {...props} />
   const stubbedHandler = sinon.stub(wrapper.instance(),
      'handleClick');
   this.setProps({prop1: 'string'}); // OR   wrapper.forceUpdate();
   chai.assert.isTrue(stubbedEventHandler.calledOnce);
   stubbedEventHandler.restore()
}
```

With React testing library with don’t have to do all. You just have to deal with **DOM**

4. Avoiding Implementation details

React-testing-library encourages you to avoid implementation detail on your component rather focus on making your tests give you the confidence for which they are intended. This makes your test much more maintainable in the long run. For example, **Enzyme** lets you update state and props of the component which is actually not the way the user is going to use your application

5. Enforce to write better test cases

React-testing-library encourages us to write better components, follow Accessibility and semantic HTML. It offers very few ways to select a dom element by **test-id, placeholder, aria** which it is very less likely to get change while you refactor your code.

It gives you much confidence then CSS and id based selectors.

So when I was writing my test cases I realized my elements are not easily accessible so I went back to HTML changed it so it make it accessible. It is like a feedback mechanism about your component.

The problem

You want to write maintainable tests for your React components. As a part of this goal, you want your tests to avoid including implementation details of your components and rather focus on making your tests give you the confidence for which they are intended. As part of this, you want your testbase to be maintainable in the long run so refactors of your components (changes to implementation but not functionality) don't break your tests and slow you and your team down.

This solution

The React Testing Library is a very light-weight solution for testing React components. It provides light utility functions on top of react-dom and react-dom/test-utils, in a way that encourages better testing practices. Its primary guiding principle is:

The more your tests resemble the way your software is used, the more confidence they can give you.

So rather than dealing with instances of rendered React components, your tests will work with actual DOM nodes. The utilities this library provides facilitate querying the DOM in the same way the user would. Finding form elements by their label text (just like a user would), finding links and buttons from their text (like a user would). It also exposes a recommended way to find elements by a data-testid as an "escape hatch" for elements where the text content and label do not make sense or is not practical.

This library encourages your applications to be more accessible and allows you to get your tests closer to using your components the way a user will, which allows your tests to give you more confidence that your application will work when a real user uses it.

This library is a replacement for Enzyme. While you can follow these guidelines using Enzyme itself, enforcing this is harder because of all the extra utilities that Enzyme provides (utilities which facilitate testing implementation details).

NOTE: This library is built on top of DOM Testing Library which is where most of the logic behind the queries is.

Cheatsheet
A short guide to all the exported functions in DOM Testing Library

Queries

	No Match	**1 Match**	**1+ Match**	**Await?**
getBy	throw	return	throw	No
findBy	throw	return	throw	Yes
queryBy	null	return	throw	No

getAllBy	throw	array	array	No
findAllBy	throw	array	array	Yes
queryAllBy	[]	array	array	No

- **ByLabelText** find by label or aria-label text content
    - getByLabelText
    - queryByLabelText
    - getAllByLabelText
    - queryAllByLabelText
    - findByLabelText
    - findAllByLabelText
- **ByPlaceholderText** find by input placeholder value
    - getByPlaceholderText
    - queryByPlaceholderText
    - getAllByPlaceholderText
    - queryAllByPlaceholderText
    - findByPlaceholderText
    - findAllByPlaceholderText
- **ByText** find by element text content
    - getByText
    - queryByText
    - getAllByText
    - queryAllByText
    - findByText
    - findAllByText
- **ByDisplayValue** find by form element current value
    - getByDisplayValue
    - queryByDisplayValue
    - getAllByDisplayValue
    - queryAllByDisplayValue
    - findByDisplayValue
    - findAllByDisplayValue
- **ByAltText** find by img alt attribute
    - getByAltText
    - queryByAltText
    - getAllByAltText
    - queryAllByAltText

    - findByAltText
    - findAllByAltText
- **ByTitle** find by title attribute or svg title tag
    - getByTitle
    - queryByTitle
    - getAllByTitle
    - queryAllByTitle
    - findByTitle
    - findAllByTitle
- **ByRole** find by aria role
    - getByRole
    - queryByRole
    - getAllByRole
    - queryAllByRole
    - findByRole
    - findAllByRole
- **ByTestId** find by data-testid attribute
    - getByTestId
    - queryByTestId
    - getAllByTestId
    - queryAllByTestId
    - findByTestId
    - findAllByTestId

Async#

See Async API. Remember to await or .then() the result of async functions in your tests!

- **waitFor** (Promise) retry the function within until it stops throwing or times out
- **waitForElementToBeRemoved** (Promise) retry the function until it no longer returns a DOM node

Deprecated since v7.0.0:

- **wait** (Promise) retry the function within until it stops throwing or times
- **waitForElement** (Promise) retry the function until it returns an element or an array of elements. The findBy and findAllBy queries are async and retry until

the query returns successfully, or when the query times out; they wrap waitForElement

- **waitForDomChange** (Promise) retry the function each time the DOM is changed

Events#
See Considerations for fireEvent, Events API

- **fireEvent** trigger DOM event: fireEvent(node, event)
- **fireEvent.*** helpers for default event types
    - **click** fireEvent.click(node)
    - See all supported events

Other#
See Querying Within Elements, Config API

- **within** take a node and return an object with all the queries bound to the node (used to return the queries from React Testing Library's render method): within(node).getByText("hello")
- **configure** change global options: configure({testIdAttribute: 'my-data-test-id'})

Text match option:

<div>Hello World</div>

// Matching a string:

getByText(container, 'Hello World') // full string match

getByText(container, 'llo Worl', {exact: false}) // substring match

getByText(container, 'hello world', {exact: false}) // ignore case

// Matching a regex:

getByText(container, /World/) // substring match

getByText(container, /world/i) // substring match, ignore case

getByText(container, /^hello world$/i) // full string match, ignore case

```
getByText(container, /Hello W?oRlD/i) // advanced regex

// Matching with a custom function:

getByText(container, (content, element) => content.startsWith('Hello'))
```

Given a button that updates the page after some time:

```
test('loads items eventually', async () => {

  // Click button

  fireEvent.click(getByText(node, 'Load'))

  // Wait for page to update with query text

  const items = await findByText(node, /Item #[0-9]: /)

  expect(items).toHaveLength(10)

})
```

List of role elements:

ARIA: alert role

ARIA: application role

ARIA: timer role

ARIA: article role

ARIA: banner role

ARIA: button role

ARIA: cell role

ARIA: checkbox role

ARIA: comment role

ARIA: complementary role

ARIA: contentinfo role

ARIA: dialog role

ARIA: document role

ARIA: feed role

ARIA: figure role

ARIA: form role

ARIA: grid role

ARIA: gridcell role

ARIA: heading role

ARIA: list role

ARIA: listbox role

ARIA: listitem role

ARIA: main role

ARIA: mark role

ARIA: navigation Role

ARIA: region role

ARIA: img role

ARIA: row role

ARIA: rowgroup role

ARIA: search role

ARIA: suggestion role

ARIA: switch role

ARIA: tab role

ARIA: table role

ARIA: tabpanel role

ARIA: textbox role

What you should avoid with Testing Library

Testing Library encourages you to avoid testing implementation details like the internals of a component you're testing (though it's still possible). The Guiding Principles of this library emphasize a focus on tests that closely resemble how your web pages are interacted by the users.

You may want to avoid the following implementation details:

Internal state of a component

Internal methods of a component

Lifecycle methods of a component

Child components

Fire Event:

Most projects have a few use cases for fireEvent, but the majority of the time you should probably use @testing-library/user-event.

fireEvent[eventName]

fireEvent[eventName](node: HTMLElement, eventProperties: Object)

```
fireEvent.change(input, {target: {value: '2020-05-24'}})

fireEvent.keyDown(domNode, {key: 'Enter', code: 'Enter'})

fireEvent.keyDown(domNode, {key: 'A', code: 'KeyA'})

fireEvent.drop(getByLabelText(/drop files here/i), {
  dataTransfer: {
    files: [new File(['(⌐□_□)'], 'chucknorris.png', {type: 'image/png'})],
  },
})

fireEvent.mouseOver(element)

fireEvent.mouseMove(element)

fireEvent.mouseDown(element)

element.focus() (if that element is focusable)

fireEvent.mouseUp(element)

fireEvent.click(element)

- fireEvent.focus(getByText('focus me'));

+ getByText('focus me').focus();
```

- fireEvent.keyDown(getByText('click me'));

+ getByText('click me').focus();

+ fireEvent.keyDown(document.activeElement || document.body);

createEvent[eventName]:

createEvent[eventName](node: HTMLElement, eventProperties: Object)

const myEvent = createEvent.click(node, {button: 2})

fireEvent(node, myEvent)

Alternatives

We will describe a couple of simple adjustments to your tests that will increase your confidence in the interactive behavior of your components. For other interactions you may want to either consider using **user-event** or testing your components in a real environment (e.g. manually, automatic with cypress, etc.).

User-event:

user-event tries to simulate the real events that would happen in the browser as the user interacts with it. For example userEvent.click(checkbox) would change the state of the checkbox.

npm install --save-dev @testing-library/user-event @testing-library/dom

import userEvent from '@testing-library/user-event'

type(element, text, [options])

userEvent.type(input, '{backspace}good')

Writes text inside an <input> or a <textarea>.

keyboard(text, options)

Simulates the keyboard events described by text. This is similar to userEvent.type() but without any clicking or changing the selection range.

You should use userEvent.keyboard if you want to just simulate pressing buttons on the keyboard. You should use userEvent.type if you just want to conveniently insert some text into an input field or textarea.

userEvent.keyboard('{{a[[') // translates to: {, a, [

userEvent.keyboard('{Shift}{f}{o}{o}') // translates to: Shift, f, o, o

userEvent.keyboard('[ShiftLeft][KeyF][KeyO][KeyO]') // translates to: Shift, f, o, o

Keys can be kept pressed by adding a > to the end of the descriptor - and lifted by adding a / to the beginning of the descriptor:

userEvent.keyboard('{shift}{ctrl/}a{/shift}') // translates to: Shift(down), Control(down+up), a, Shift(up)

userEvent.keyboard('{Shift>}A{/Shift}') // translates to: Shift(down), A, Shift(up)

`upload(element, file, [{ clickInit, changeInit }], [options])`

render(

<div>

```
<label htmlFor="file-uploader">Upload file:</label>
<input id="file-uploader" type="file" multiple />
</div>,
)
const input = screen.getByLabelText(/upload file/i)
userEvent.upload(input, files)
expect(input.files).toHaveLength(2)
```

Clear(element)

```
userEvent.clear(screen.getByRole('textbox'))
expect(screen.getByRole('textbox')).toHaveAttribute('value', '')
```

selectOptions(element, values):

```
userEvent.selectOptions(screen.getByRole('listbox'), ['1', '3'])
expect(screen.getByRole('option', {name: 'A'}).selected).toBe(true)
expect(screen.getByRole('option', {name: 'B'}).selected).toBe(false)
expect(screen.getByRole('option', {name: 'C'}).selected).toBe(true)
```

deselectOptions(element, values):

```
userEvent.selectOptions(screen.getByRole('listbox'), '2')
expect(screen.getByText('B').selected).toBe(true)
userEvent.deselectOptions(screen.getByRole('listbox'), '2')
expect(screen.getByText('B').selected).toBe(false)
```

API:

- render
- render Options
  - container
  - baseElement
  - hydrate

    - wrapper
    - queries
- render Result
    - ...queries
    - container
    - baseElement
    - debug
    - rerender
    - unmount
    - asFragment
- cleanup
- act

Cleanup:

```
import {cleanup, render} from '@testing-library/react'

afterEach(cleanup)
```

Unmounts React trees that were mounted with render.

Please note that this is done automatically if the testing framework you're using supports the afterEach global and it is injected to your testing environment (like mocha, Jest, and Jasmine). If not, you will need to do manual cleanups after each test.

Debug:

NOTE: It's recommended to use screen.debug instead.

This method is a shortcut for console.log(prettyDOM(baseElement)).

```
import React from 'react'

import {render} from '@testing-library/react'

const HelloWorld = () => <h1>Hello World</h1>

const {debug} = render(<HelloWorld />)
```

```
debug()
```

Rerender:

It'd probably be better if you test the component that's doing the prop updating to ensure that the props are being updated correctly (see the Guiding Principles section). That said, if you'd prefer to update the props of a rendered component in your test, this function can be used to update props of the rendered component.

```
import {render} from '@testing-library/react'

const {rerender} = render(<NumberDisplay number={1} />)

// re-render the same component with different props

rerender(<NumberDisplay number={2} />)
```

Unmount:

This will cause the rendered component to be unmounted. This is useful for testing what happens when your component is removed from the page (like testing that you don't leave event handlers hanging around causing memory leaks).

This method is a pretty small abstraction over ReactDOM.unmountComponentAtNode

```
import {render} from '@testing-library/react'

const {container, unmount} = render(<Login />)

unmount()

// your component has been unmounted and now: container.innerHTML === ''
```