## ****Introduction and Purpose****

Welcome to the migration guide from Enzyme to React Testing Library. As the React ecosystem evolves, it's essential for projects to stay up-to-date with the latest and recommended testing practices. This migration guide will help you transition from using Enzyme, a popular testing utility for React, to React Testing Library, a testing library that promotes a more user-centric testing approach.

### Why Migrate?

#### The Evolution of React Testing Practices

With the growth of React and the broader JavaScript ecosystem, testing practices have also evolved. React Testing Library has gained popularity for its emphasis on testing user behavior rather than component implementation details. This shift aligns with the philosophy of writing tests that closely resemble how users interact with your application.

#### Enzyme is Deprecated

It's important to note that Enzyme, once widely used for React testing, is considered deprecated as of Enzyme v3.11.0 release. While Enzyme is still functional and might receive bug fixes, it's recommended to explore and adopt modern testing libraries that align with the latest best practices.

### Key Objectives of this Migration Guide

1. **Stay Current with Testing Practices:**Embrace React Testing Library to align with the evolving best practices in the React testing ecosystem.
2. **Improve Test Readability and Maintainability:**Transition from Enzyme's focus on component internals to React Testing Library's emphasis on testing user interactions, leading to more readable and maintainable tests.
3. **Ensure Long-Term Compatibility:**Prepare your project for the future by adopting a testing library that is actively maintained and aligned with the direction of the React community.
4. **Address Deprecated Tooling:**Acknowledge the deprecation of Enzyme and make informed decisions about adopting modern alternatives.

## ****Prerequisites****

**Before migrating your tests from Enzyme to React Testing Library, ensure that your development environment and project meet the following prerequisites.**

### React Version Compatibility

Ensure that your project is using React version 17 or later. React Testing Library is designed to work seamlessly with the latest React versions.

### Installation of React Testing Library

Make sure React Testing Library is installed in your project as a development dependency. Use a package manager like npm to install it.

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

### Installation of ESLint Plugins

Install ESLint plugin to follow best practices and anticipate common mistakes when writing tests.

npm install --save-dev eslint-plugin-jest-dom eslint-plugin-testing-library

### ****Review Existing Enzyme Tests****

Review and analyze your existing Enzyme tests to understand the components and functionality covered. This will help in planning the migration strategy and identifying potential challenges.

## ****Basic Concepts****

### ****Introduction to React Testing Library****

**React Testing Library (RTL) is a testing utility designed to facilitate a more user-centric and behavior-driven approach to testing React components. Unlike Enzyme, which focuses on the internal implementation details of components, RTL encourages developers to test the application the way users interact with it. This shift in perspective results in tests that are more robust, maintainable, and closely aligned with the user experience.**

### ****Philosophy: Testing the Way Users Interact****

**The core philosophy of React Testing Library is centered around the idea of testing the application from the user's perspective. Traditional testing libraries often lead developers to write tests that are tightly coupled with the component's internal structure, making them brittle and prone to breaking when implementation details change.**

React Testing Library takes a different approach. It encourages developers to write tests that simulate user interactions and assert on the resulting changes in the DOM. This philosophy promotes the idea that tests should mirror how a real user interacts with the application, focusing on behaviors rather than implementation specifics.

### Key Principles:

1. **Accessibility-Driven Testing:** RTL emphasizes testing based on accessibility roles, ensuring that components are not only visually correct but also usable by individuals with disabilities.
2. **Queries for Selection:** Instead of relying on complex selector strategies, RTL provides a set of simple queries that mimic how users identify and interact with elements on the page.
3. **Minimal Mocking:** Reduce the need for extensive mocking of internal component logic. Instead, focus on testing the output and behavior that users actually experience.
4. **Avoiding Implementation Details:** Tests should not break when internal component implementations change, promoting a more resilient test suite.
5. **Readable and Maintainable Tests:** Write tests that are easy to understand and maintain, fostering collaboration among team members and making the testing suite a valuable asset.

### Example:

Consider a button click scenario:

**Enzyme Approach:**

it('should increment the counter on button click', () => {

const wrapper = mount(<Counter />);

const button = wrapper.find('button');

button.simulate('click');

expect(wrapper.find('span').text()).toEqual('1');

});

**React Testing Library Approach:**

it('should increment the counter on button click', () => {

const { getByText } = render(<Counter />);

const button = getByText('Increment');

fireEvent.click(button);

expect(screen.getByText('1')).toBeInTheDocument();

});

In the React Testing Library example, we focus on selecting elements based on user interaction, simulating a button click, and asserting the resulting change in the DOM.

## ****Test Structure Changes****

### ****1. Component-Centric vs. Behavior-Driven Testing:****

#### Enzyme (Component-Centric):

Enzyme tests often focus on the internal implementation details of components, including shallow rendering and direct manipulation of states and props.

import React from 'react';

import { shallow } from 'enzyme';

import MyComponent from './MyComponent';

describe('MyComponent', () => {

it('handles click event', () => {

const wrapper = shallow(<MyComponent />);

wrapper.find('.my-button').simulate('click');

expect(wrapper.find('p').text()).toBe('Button clicked!');

});

});

#### React Testing Library (Behavior-Driven):

React Testing Library promotes behavior-driven testing, focusing on user interactions and expected outcomes. Tests interact with the rendered components as users would.

import React from 'react';

import { render, screen, fireEvent } from '@testing-library/react';

import MyComponent from './MyComponent';

describe('MyComponent', () => {

it('handles click event', () => {

render(<MyComponent />);

// Using semantic query to get the button

const button = screen.getByText('Click me');

fireEvent.click(button);

// Using semantic query to get the success message

const successMessage = screen.getByText('Button clicked!');

expect(successMessage).toBeInTheDocument();

});

});

### 2. ****Selectors and Queries:****

#### Enzyme (CSS Class-Based Queries):

Enzyme tests often use CSS class-based queries to select elements within the component.

const wrapper = shallow(<MyComponent />);

const header = wrapper.find('.header');

#### React Testing Library (Semantic Queries):

React Testing Library encourages the use of semantic queries like getByText, getByRole to select elements based on their role or visible text content.

render(<MyComponent />);

const header = screen.getByText('Header');

### 3. ****Component Render Methods:****

#### Enzyme (Shallow Rendering):

Enzyme frequently uses shallow rendering to render only the component without its children.

const wrapper = shallow(<MyComponent />);

#### React Testing Library (Realistic Rendering):

React Testing Library uses render to perform a more realistic rendering, rendering the component and its children as they would appear in a real application.

render(<MyComponent />);

### 4. ****Event Simulation:****

#### Enzyme (Simulate Events):

Enzyme uses the simulate method to simulate events like clicks.

wrapper.find('button').simulate('click');

#### React Testing Library (fireEvent):

React Testing Library recommends using the fireEvent utility to simulate user interactions. But React Testing Library, combined with user-event, provides a more realistic simulation of user interactions, making tests more user-centric.

fireEvent.click(screen.getByRole('button'));

// OR

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

userEvent.type(input, 'Hello');

### 5. ****Assertions:****

#### Enzyme (Component Structure Assertions):

Enzyme assertions often focus on the internal structure of the component.

expect(wrapper.find('.header')).toHaveLength(1);

#### React Testing Library (User-Visible Content Assertions):

React Testing Library encourages assertions based on user-visible content.

expect(screen.getByText('Header')).toBeInTheDocument();

By understanding these differences and adapting your testing approach, you can leverage the strengths of React Testing Library for more robust and user-centric tests in your React applications.

## ****DOM Interactions****

#### 1. ****Click Event:****

##### Enzyme Example:

wrapper.find('button').simulate('click');

##### RTL Example:

fireEvent.click(screen.getByRole('button'));

#### 2. ****Typing:****

##### Enzyme Example:

wrapper.find('input').simulate('change', { target: { value: 'Hello' } });

##### RTL (user-event) Example:

const input = screen.getByRole('textbox');

userEvent.type(input, 'Hello');

#### 3. ****Change Event:****

##### Enzyme Example:

wrapper.find('select').simulate('change', { target: { value: 'Option 2' } });

##### RTL (user-event) Example:

const select = screen.getByRole('combobox');

userEvent.selectOptions(select, 'Option 2');

#### 4. ****Form Submission:****

##### Enzyme Example:

wrapper.find('form').simulate('submit');

##### RTL (user-event) Example:

const form = screen.getByRole('form');

userEvent.submit(form);

## ****Assertions and Matchers****

#### 1. ****Text Content Assertion:****

##### Enzyme Example:

expect(wrapper.find('p').text()).toBe('Hello, World!');

##### RTL Example:

expect(screen.getByText('Hello, World!')).toBeInTheDocument();

#### 2. ****Element Existence Assertion:****

##### Enzyme Example:

expect(wrapper.find('.header')).toHaveLength(1);

##### RTL Example:

expect(screen.getByText('Header')).toBeInTheDocument();

#### 3. ****Element Absence Assertion:****

##### Enzyme Example:

expect(wrapper.find('.non-existent-element')).toHaveLength(0);

##### RTL Example:

expect(screen.queryByText('Non-existent Element')).toBeNull();

#### 4. ****Event Handling Assertion:****

##### Enzyme Example:

wrapper.find('button').simulate('click');

expect(wrapper.find('p').text()).toBe('Button clicked!');

##### RTL Example:

const button = screen.getByRole('button');

fireEvent.click(button);

expect(screen.getByText('Button clicked!')).toBeInTheDocument();

#### 5. ****Component State Assertion:****

##### Enzyme Example:

expect(wrapper.state('isActive')).toBe(true);

##### RTL Example:

// Assumes the state is reflected in the user interface

expect(screen.getByText('Active')).toBeInTheDocument();

#### 6. ****Element Visibility Assertion:****

##### Enzyme Example:

expect(wrapper.find('.hidden-element')).toHaveLength(0);

##### RTL Example:

expect(screen.getByText('Visible Element')).toBeVisible();

## ****Snapshot Testing****

### Snapshot Testing with Enzyme:

In Enzyme, snapshot testing is commonly used to capture a rendered component's output and compare it against a previously saved snapshot. This ensures that the UI does not unintentionally change between test runs.

#### Example:

import { shallow } from 'enzyme';

import MyComponent from './MyComponent';

test('MyComponent snapshot', () => {

const wrapper = shallow(<MyComponent />);

expect(wrapper).toMatchSnapshot();

});

In this Enzyme example, the toMatchSnapshot matcher captures the rendered output of MyComponent and compares it to a stored snapshot. If the snapshots differ, the test will fail, alerting developers to any unintended UI changes.

### Snapshot Testing with React Testing Library (RTL):

React Testing Library also supports snapshot testing, but there are some differences in usage and considerations.

#### Example:

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

import MyComponent from './MyComponent';

test('MyComponent snapshot', () => {

const { asFragment } = render(<MyComponent />);

expect(asFragment()).toMatchSnapshot();

});

In RTL, the asFragment function is used to retrieve a DocumentFragment containing the component's rendered output. This fragment is then compared to the stored snapshot. While the fundamental concept is similar, the implementation details differ between Enzyme and RTL.

### Changes and Considerations for Snapshot Testing in RTL:

1. **Container Element:**
   * Enzyme snapshots often capture the entire component hierarchy, including wrapper elements. In RTL, asFragment captures only the component's content without wrapper elements. Adjust snapshot expectations accordingly.
2. **Semantic Changes:**
   * With RTL's focus on user behavior, UI changes might be more intentional. Regularly review and update snapshots to reflect intentional changes in the UI.
3. toMatchSnapshot **Function:**
   * RTL does not provide a direct toMatchSnapshot function like Enzyme. Instead, use the asFragment function to obtain the document fragment and then compare it to the stored snapshot.
4. **Migration Steps:**
   * When migrating, update existing snapshot tests to use RTL's asFragment function.
   * Review and update snapshots to reflect the desired state of the UI in a more user-centric testing approach.

## ****Testing Async Behavior****

### Enzyme Async Testing Example:

import React from 'react';

import { mount } from 'enzyme';

import MyAsyncComponent from './MyAsyncComponent';

test('fetches data on button click', (done) => {

const wrapper = mount(<MyAsyncComponent />);

wrapper.find('button').simulate('click');

setTimeout(() => {

wrapper.update();

expect(wrapper.find('p').text()).toBe('Data Loaded');

done();

}, 1000);

});

In Enzyme, asynchronous testing often involves using setTimeout or other mechanisms to wait for async actions to complete before making assertions.

### React Testing Library (RTL) Async Testing Example:

import React from 'react';

import { render, screen, waitFor } from '@testing-library/react';

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

import MyAsyncComponent from './MyAsyncComponent';

test('fetches data on button click', async () => {

render(<MyAsyncComponent />);

const button = screen.getByRole('button');

userEvent.click(button);

await waitFor(() => {

expect(screen.getByText('Data Loaded')).toBeInTheDocument();

});

});

### Changes and Considerations for Async Testing in React Testing Library:

#### 1. ****Testing Library Utilities:**** Use render from RTL to render the component and testing utilities like waitFor for handling async behavior.

#### 2. ****Async/Await:**** Leverage async/await syntax for cleaner and more readable asynchronous testing code.

#### 3. ****RTL's**** waitFor****:**** waitFor is a powerful utility in RTL that allows you to wait for specific conditions before making assertions. It's designed for handling asynchronous behavior and is a key tool for async testing.

## ****Best Practices****

1. **Use Semantic Queries: Prioritize semantic queries (**getByText**,** getByRole**, etc.) over class-based or implementation-specific queries. This aligns with RTL's focus on user-centric testing.**
2. **Testing User Interactions:** Emphasize testing user interactions and behaviors rather than internal component structures. This shift leads to more reliable tests that reflect how users interact with your application.
3. **Avoid Direct DOM Manipulation:** Minimize direct DOM manipulation in tests. Instead, interact with the rendered components using user events provided by RTL's userEvent library.
4. **Use Async/Await Syntax:** Adopt async/await syntax for handling asynchronous behavior in tests. Leverage RTL's waitFor utility to ensure that assertions are made after asynchronous actions complete.
5. **Limit Use of** fireEvent**:** While fireEvent is a powerful utility, prefer using user-friendly alternatives like userEvent for simulating user interactions. userEvent closely mimics real user behavior.
6. **Test Accessibility:** Use RTL's queries to test for accessible content. Ensure that components are not only visually correct but also accessible to users with disabilities.
7. **Isolate Test Concerns:** Keep tests focused on specific concerns, avoiding unnecessary dependencies between tests. This improves test maintainability and helps diagnose issues more easily.
8. **Avoid Snapshots for Complex Components:** For complex components, be cautious with snapshot testing, as it may capture a large amount of implementation details. Focus on more granular unit tests and user interaction testing.
9. **Test Edge Cases:** Test various scenarios, including edge cases and error states, to ensure robustness. Cover different user inputs and conditions to catch potential issues.
10. **Use RTL Utils:** Familiarize yourself with the various utilities provided by React Testing Library, such as screen, render, waitFor, and userEvent. Understanding and leveraging these utilities can simplify and enhance your testing.

## ****Resources****

## ****Conclusion****

**Migrating from Enzyme to React Testing Library (RTL) marks a significant shift in your React testing paradigm, bringing with it a host of benefits for test quality and maintainability. Here's a summary of key points and an encouragement to embrace the new testing paradigm:**

1. **User-Centric Testing Philosophy:**
   * RTL encourages a user-centric testing philosophy, focusing on how users interact with your components rather than their internal implementation details. This shift leads to more resilient tests that reflect the real user experience.
2. **Semantic Queries for Improved Readability:**
   * RTL promotes the use of semantic queries (getByText, getByRole) over class-based selectors. This not only aligns tests with user behavior but also enhances test readability and reduces the impact of implementation changes.
3. **Realistic Rendering Approach:**
   * RTL's realistic rendering approach provides a more accurate representation of how components are rendered in a real application, reducing the need for shallow rendering and making tests more reliable.
4. **Powerful Async Testing Utilities:**
   * The waitFor utility in RTL is a powerful tool for handling asynchronous behavior, simplifying the process of waiting for conditions before making assertions. Async testing becomes cleaner and more manageable.
5. **User-Event Library for Realistic Simulations:**
   * The userEvent library in RTL facilitates realistic simulations of user interactions, ensuring that your tests closely resemble how users interact with your application.