React Testing Library and the “not wrapped in act” Errors

David Cai · Follow

4 min read · Apr 30, 2020

I recently upgraded React and React testing library. Very happy about the upgrade. But I start to see test errors like this:

```
When testing, code that causes React state updates should be wrapped
into act(...):

     act(() => {
       /* fire events that update state */
     });
     /* assert on the output */

This ensures that you're testing the behavior the user would see in
the browser. Learn more at https://fb.me/react-wrap-tests-with-act
```

Why? And how do I fix this?

Why do I get this error?

In test, the code to render and update React components need to be included in React's call stack. So the test behaves more similar to the user experience in real browsers.

Here is a simplified example from React's document:

```
// With react-dom/test-utils

it("should render and update a counter", () => {
  // Render a component
  act(() => {
    ReactDOM.render(<Counter />, container);
  });
  ...

  // Fire event to trigger component update
  act(() =>
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
```

```
  ...
});
```

React testing library already integrated `act` with its APIs. So in most cases, we do not need to wrap `render` and `fireEvent` in `act`. For example:

```
// With react-testing-library

it("should render and update a counter", () => {
  // Render a component
  const { getByText } = render(<Counter />;
  ...

  // Fire event to trigger component update
  fireEvent.click(getByText("Save"));
  ...
});
```

However, if your test still complains about "not wrapped in act(...)", you might encounter one of these 4 cases described below.

Case 1: Asynchronous Updates

Test code somehow triggered a testing component to update in an asynchronous way. Here is an example:

```
const MyComponent = () => {
  const [person, setPerson] = React.useState();

  const handleFetch = React.useCallback(async () => {
    const { data } = await fetchData();
    setPerson(data.person); // <- Asynchronous update
  }, []);

  return (
    <button type="button" onClick="handleFetch">
      {person ? person.name : "Fetch"}
    </button>
  );
};
```

The following test will have the “not wrapped in act” error:

```
it("should fetch person name", () => {
  const { getByText } = render(<MyComponent />);
  fireEvent.click(getByText("Fetch"));

  expect(getByText("David")).toBeInTheDocument();
});
```

`fireEvent.click` triggers `fetchData` to be called, which is an asynchronous call. When its response comes back, `setPerson` will be invoked, but at this moment, the update will happen outside of React’s call stack.

Solution

Before assertions, wait for component update to fully complete by using `waitFor`. `waitFor` is an API provided by React testing library to wait for the wrapped assertions to pass within a certain timeout window.

```
it("should fetch person name", async () => {
  const { getByText } = render(<MyComponent />);
  fireEvent.click(getByText("Fetch"));

  await waitFor(() => {
    expect(getByText("David")).toBeInTheDocument();
  });
});
```

Case 2: Jest Fake Timers

When you have `setTimeout` or `setInterval` in your component:

```
const Toast = () => {
  const [isVisible, setIsVisible] = React.useState(true);
  React.useEffect(() => {
    setTimeout(() => { setIsVisible(false);}, 1000);
  }, []);
```

```
  return isVisible ? <div>Toast!</div> : null;
};
```

... and use Jest's fake timers to manipulate time:

```
it("should display Toast in 1 sec", () => {
  jest.useFakeTimers();
  const { queryByText } = render(<MyComponent />);
  jest.advanceTimersByTime(1000);

  expect(queryByText("Toast!")).not.toBeInTheDocument();
});2
```

..., unit test has no idea that advancing timers will cause component updates, and you will get the "not wrapped in act" error.

Solution

Wrap Jest's timer manipulations in an `act` block, so test will know that advancing time will cause component to update:

```
it("should display Toast in 1 sec", () => {
  jest.useFakeTimers();
  const { queryByText } = render(<MyComponent />);
  act(() => {
    jest.advanceTimersByTime(1000);
  });

  expect(queryByText("Toast!")).not.toBeInTheDocument();
});
```

Case 3: Premature Exit

Your test prematurely exits before components finish rendering or updating.

This normally happens in components that have loading state, e.g. components fetching data using GraphQL or REST.

```
const MyComponent = () => {
  const { loading, data } = useQuery(QUERY_ACCOUNTS);

  return loading ? (
    <div>Loading ...</div>
  ) : (
    <div>{data.accounts.length}</div>
  );
};
```

The test checks if “Loading ...” is present.

```
it("should display loading state", () => {
  const { getByText } = render(
    <MockedApolloProvider mocks={accountMock}>
      <MyComponent />
    </MockedApolloProvider>
  );

  expect(getByText("Loading ...")).toBeInTheDocument();
});
```

However, the `it` block exits before the loading state disappears and data comes back. This kind of test will also cause “not wrapped in act” errors.

Solution

Make sure the test exits after all the rendering and updates are done. To do that, we can wait for the loading state to disappear:

```
it("should display loading state", async () => {
  const { getByText, queryByText } = render(
    <MockedApolloProvider mocks={accountMock}>
      <MyComponent />
    </MockedApolloProvider>
  );

  expect(getByText("Loading ...")).toBeInTheDocument();
  await waitFor(() => {
    expect(queryByText("Loading ...")).not.toBeInTheDocument();
  });
});
```

Alternatively, you can use `waitForElementToBeRemoved` which is a wrapper around `waitFor`.

```
await waitForElementToBeRemoved(() => queryByText("Loading ..."));
```

Case 4: Formik Updates

This is actually another variation of Case 1. It goes like this: test simulates events to change values in form inputs, e.g. changing value in a text input. If the form input is managed by Formik, your test will have a chance to run into "not wrapped in act" errors.

```
it("should validate phone numbers", () => {
  ...

  fireEvent.change(getByPlaceholder("Phone"), {
    target: { value: "123456789" }
  });
  fireEvent.click(getByText("Save"));

  expect(getByText(
    "Please enter a valid phone number"
  )).toBeInTheDocument();
});
```

Solution

Similar to Case 1, wait for all updates to complete, then perform assertions:

```
it("should validate phone numbers", async () => {
  ...

  fireEvent.change(getByPlaceholder("Phone"), {
   target: { value: "123456789" }
  });
  fireEvent.click(getByText("Save"));

  await waitFor(() => {
    expect(getByText(
      "Please enter a valid phone number"
    )).toBeInTheDocument();
```

```
  });
});
```

Conclusion

In test, React needs extra hint to understand that certain code will cause component updates. To achieve that, React-dom introduced `act` API to wrap code that renders or updates components. React testing library already wraps some of its APIs in the `act` function. But in some cases, you would still need to use `waitFor`, `waitForElementToBeRemoved`, or `act` to provide such "hint" to test.

React Testing

Written by David Cai

57 Followers

Gamer, hiker, reader, and programmer

More from David Cai