How to test React Context

Oct 22, 2020 (Updated) 5 min read

React Context is a tool for designing flexible Component APIs. How we test it depends on the situation, we are going to explore some of the situations you might find yourself in and the best way to write maintainable tests for each of them.

The best way to test Context is to make our tests unaware of its existence and avoiding mocks. We want to test our components in the same way that developers would use them (behavioral testing) and mimic the way they would run in our applications (integration testing).

Theming example

Let's set up our example which we will then explore how to test. We might choose Context to avoid "prop drilling" where we pass a theme prop into every component. To do this we can create a `ThemeContext`:

JSX

```jsx
import { createContext } from "react";

export const ThemeContext = createContext();
```

To make `ThemeContext` useful we need to wrap components that need access to the theme in a Provider:

JSX

```jsx
import React, { useState } from "react";
import BlogPost from "./BlogPost";
import { ThemeContext } from "./ThemeContext";
```

```
export default function Page() {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider
      value={{
        theme,
        onThemeChange: (newTheme) => setTheme(newTheme),
      }}
    >
      <BlogPost content="This is blog content" />
    </ThemeContext.Provider>
  );
}
```

The source of truth for the active theme is the `theme` property and the `onThemeChange` function allows any component to change the theme.

We can then make use of the `ThemeContext` in the `<BlogPost />` component. It both reads the theme value and updates it through the `onThemeChanged` callback:

JSX

```
import React from "react";
import { useContext } from "react";
import { ThemeContext } from "./ThemeContext";
import { getActiveClasses } from "get-active-classes";
import "./BlogPost.css";

export default function BlogPost({ content }) {
  const { theme, onThemeChange } = useContext(ThemeContext);

  return (
    <article
      className={getActiveClasses({
        "light-theme": theme === "light",
        "dark-theme": theme === "dark",
      })}
    >
      {content}
```

```
        <button
          onClick={() => onThemeChange(theme === "dark" ? "light" : "dark"
        >
          {theme === "dark" ? "Use Light Theme" : "Use Dark Theme"}
        </button>
      </article>
    );
}
```

Behavioral Testing Approaches

> The best way to test Context is to make our tests unaware of its existence

Testing the Provider and Consumer Together

This type of test is available if both the provider and consumer are used within the component that you want to test such as in the case of `<Page />`. This allows us to write our tests without any mention of Context:

JSX

```
import React from "react";
import Page from "./Page";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";

describe("<Page />", () => {
  beforeEach(() => {
    render(<Page />);
  });

  describe("when page is initialized", () => {
    it("then shows the light theme by default", () => {
      // "Use Dark Theme" text is only shown when the light theme is act
      expect(screen.getByText(/Use Dark Theme/i)).toBeTruthy();
    });
```

```
  });

  describe("when the toggle theme button is clicked", () => {
    beforeEach(() => {
      userEvent.click(screen.getByText(/Use Dark Theme/i));
    });

    it("then uses the dark theme", () => {
      // "Use Light Theme" text is only shown when the dark theme is act
      expect(screen.getByText(/Use Light Theme/i)).toBeTruthy();
    });
  });
});
```

If we decided we no longer wanted to use Context and change the implementation our tests would still pass.

Testing a component with children that consume Context

This is a common pattern often used in compound components where the children components can consume Context provided by the *base* component. In this example we have modified our `<Page />` component to accept children in this way:

JSX

```
import React, { useState } from "react";
import { ThemeContext } from "./ThemeContext";

export default function Page({ children }) {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider
      value={{
        theme,
        onThemeChange: (newTheme) => setTheme(newTheme),
      }}
    >
      {children}
```

```
    </ThemeContext.Provider>
  );
}
```

To test that Context is doing its job we can pass in a component that consumes the Context and test the functionality that Context enables for it:

JSX

```jsx
import React from "react";
import Page from "./Page";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import BlogPost from "./BlogPost";

describe("<Page />", () => {
  beforeEach(() => {
    render(
      <Page>
        <BlogPost content="How to do things" />
      </Page>
    );
  });

  describe("when page is initialized", () => {
    it("then shows the light theme by default", () => {
      // "Use Dark Theme" text is only shown when the light theme is act
      expect(screen.getByText(/Use Dark Theme/i)).toBeTruthy();
    });
  });

  describe("when the toggle theme button is clicked", () => {
    beforeEach(() => {
      userEvent.click(screen.getByText(/Use Dark Theme/i));
    });

    it("then uses the dark theme", () => {
      // "Use Light Theme" text is only shown when the dark theme is act
      expect(screen.getByText(/Use Light Theme/i)).toBeTruthy();
    });
```

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

Once again this type of test does not couple us to the implementation detail that is our usage of Context.

Implementation Testing Approaches

Use these with caution as they will make test files hard to maintain and read!

This is an example of *implementation* testing. This is not ideal, but it might be worth doing in some situations to give you confidence in your code.

Testing a Consumer without a Provider

It's going to be easier to maintain our tests if we choose to test the provider and consumer together instead of testing individual components. But if we wanted to test a component individually that relies on consuming Context we need to provide that Context:

JSX

```jsx
import React from "react";
import BlogPost from "./BlogPost";
import { render, screen } from "@testing-library/react";
import { ThemeContext } from "./ThemeContext";
import userEvent from "@testing-library/user-event";

describe("<BlogPost />", () => {
  describe("when theme is dark", () => {
    const theme = {
      theme: "dark",
      onThemeChange: jest.fn(),
    };

    beforeEach(() => {
      render(
```

```
        <ThemeContext.Provider value={theme}>
          <BlogPost />
        </ThemeContext.Provider>
      );
    });

    describe("when clicking toggle", () => {
      beforeEach(() => userEvent.click(screen.getByText(/toggle theme/i)

      it("theme callback is ran", () => {
        expect(theme.onThemeChange).toHaveBeenCalledWith("light");
      });
    });
  });
});
```

Notice here we don't have 100% code coverage of `<BlogPost />`, we are missing tests for the dynamic classes: `.light-theme` and `dark-theme`. Testing styles are best left to tools like Storybook.

Testing a Provider without a Consumer

Don't bother with this trust me. It's possible but it only makes our tests complex and hard to maintain. Instead use the behavioral testing approaches