Conclusion

Kamran Ayub

How to Test React Components in TypeScript

Kamran Ayub

Aug 9, 2019 • 15 Min read • 60,453 Views

Aug 9, 2019 • 15 Min read • 60,453 Views

Web Development React

Introduction

When writing applications, testing is crucial for ensuring code behaves as expected. In this guide, you'll learn how to get started quickly writing tests using TypeScript, React, and Jest in an idiomatic way.

There are several benefits to leveraging TypeScript when testing:

- Enables better refactoring of tests, improving long-term maintenance
- Ensures consistency of component usage and props
- Reduces potential for bugs within tests

Writing unit tests and using TypeScript are not mutually exclusive, when used together they help you build more maintainable codebases.

We'll be using Jest, a popular test framework for JavaScript projects including React.

Using React Testing Library

Jest will take care of running tests and handling assertions but since we are testing React, we will need some testing utilities.

There are two popular testing libraries for React: Enzyme and React Testing Library.

In this guide we will be testing React components using React Testing Library, as it provides a simple and straightforward way to test components that promotes good test practices.

There are several practices that the React Testing Library promotes:

- Avoid testing internal component state
- Testing how a component renders

These two practices help focus tests on behavior and user interaction, treating the internals of a component like a "black box" that shouldn't be exposed.

> *"The more your tests resemble the way your software is used, the more confidence they can give you." -- Kent C. Dodds, creator of RTL.*

Configuring Jest and React Testing Library

Starting with an existing React and TypeScript project, we can add dependencies for Jest and React Testing Library:

```
npm install @types/jest @testing-library/react @testing-library/jest-dom j
```

This installs Jest and React Testing Library with TypeScript support.

Add a new `jest.config.js` file to the root of your project:

Conclusion

170

js

```
module.exports = {
  // The root of your source code, typically /src
  // `<rootDir>` is a token Jest substitutes
  roots: ["<rootDir>/src"],

  // Jest transformations -- this adds support for TypeScript
  // using ts-jest
  transform: {
    "^.+\\.tsx?$": "ts-jest"
  },

  // Runs special logic, such as cleaning up components
  // when using React Testing Library and adds special
  // extended assertions to Jest
  setupFilesAfterEnv: [
    "@testing-library/react/cleanup-after-each",
    "@testing-library/jest-dom/extend-expect"
  ],

  // Test spec file resolution pattern
  // Matches parent folder `__tests__` and filename
  // should contain `test` or `spec`.
  testRegex: "(/__tests__/.*|(\\.|/)(test|spec))\\.tsx?$",

  // Module file extensions for importing
  moduleFileExtensions: ["ts", "tsx", "js", "jsx", "json", "node"]
};
```

This is a typical Jest configuration but with some additional modifications:

170

- **TypeScript support** added via the `ts-jest` package
- **DOM cleanup** when using React Testing Library
- **Extended assertions** when using React Testing Library

Add a new `npm` script to `package.json`:

```
{
        ...
        "scripts": {
            ...
            "test": "jest",
            "test:watch": "jest --watch"
        }
    }
```

Jest supports a powerful "watch" mode that will re-run changed tests in a quick way while you develop.

Tests should be organized into `__tests__` folders under a top-level `src` directory, according to this configuration.

Ensure your `tsconfig.json` has the `esModuleInterop` flag enabled for compatibility with Jest (and Babel):

```json
{
  "compilerOptions": {
    "esModuleInterop": true
  }
}
```

To ensure the additional Jest matchers are available for all test files, create a `src/globals.d.ts` and import the matchers:

```ts
import "@testing-library/jest-dom/extend-expect";
```

Testing a Basic Component

For this guide, we'll test a basic component that has an internal state and uses React hooks to showcase how you'd write a set of tests.

Create `src/LoginForm.tsx` that contains the following:

```tsx
import React from "react";

export interface Props {
  shouldRemember: boolean;
```

Conclusion

170


```
  onUsernameChange: (username: string) => void;
  onPasswordChange: (password: string) => void;
  onRememberChange: (remember: boolean) => void;
  onSubmit: (username: string, password: string) => void;
}

function LoginForm(props: Props) {
  const [username, setUsername] = React.useState("");
  const [password, setPassword] = React.useState("");
  const [remember, setRemember] = React.useState(props.shouldRemember);

  const handleUsernameChange = (e: React.ChangeEvent<HTMLInputElement>) =
    const { value } = e.target;
    setUsername(value);
    props.onUsernameChange(value);
  };

  const handlePasswordChange = (e: React.ChangeEvent<HTMLInputElement>) =
    const { value } = e.target;
    setPassword(value);
    props.onPasswordChange(value);
  };

  const handleRememberChange = (e: React.ChangeEvent<HTMLInputElement>) =
    const { checked } = e.target;
    setRemember(checked);
    props.onRememberChange(checked);
  };

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    props.onSubmit(username, password);
  };

  return (
```

Conclusion

170

170


```
      <form data-testid="login-form" onSubmit={handleSubmit}>
        <label htmlFor="username">Username:</label>
        <input
          data-testid="username"
          type="text"
          name="username"
          value={username}
          onChange={handleUsernameChange}
        />

        <label htmlFor="password">Password:</label>
        <input
          data-testid="password"
          type="password"
          name="password"
          value={password}
          onChange={handlePasswordChange}
        />

        <label>
          <input
            data-testid="remember"
            name="remember"
            type="checkbox"
            checked={remember}
            onChange={handleRememberChange}
          />
          Remember me?
        </label>

        <button type="submit" data-testid="submit">
          Sign in
        </button>
      </form>
    );
```

```
75      }
76
77      export default LoginForm;
```

170

This is a simple login form containing a username, password, and checkbox. It uses the `useState` hook to maintain internal state, as well as a `shouldRemember` prop to set the default state of the checkbox.

Create a new test file, `src/__tests__/LoginForm.test.tsx`:

tsx

```tsx
1   import React from "react";
2   import { render, fireEvent, waitForElement } from "@testing-library/react
3
4   import LoginForm, { Props } from "../LoginForm";
5
6   describe("<LoginForm />", () => {
7     test("should display a blank login form, with remember me checked by de
8       // ???
9     });
10  });
```

This is the skeleton of our test suite. We start by importing the utilities needed from `@testing-library/react`:

- `render` helps render components and returns find helper methods.

- `fireEvent` is for simulating events on DOM elements.
- `waitForElement` is useful when waiting for UI changes to occur.

've defined a test but it has no implementation. To run the test
ite, start the test runner:

```
npm run test:watch
```

This will watch for changes as we implement the tests. Since there's no assertions, the first test passes:

```
> jest --watch
 PASS  src/__tests__/LoginForm.test.tsx
  <LoginForm />
    √ should display a blank login form, with remember me checked by default (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.678s
Ran all test suites related to changed files.
```

Remember, we want to test important behavior, so the first test will ensure that we are rendering the username, password, and "remember me" checkbox for a blank form.

One useful tip is to encapsulate rendering the component you are testing using a render helper function so that you can handle props overriding and make your tests more maintainable:

tsx

```tsx
function renderLoginForm(props: Partial<Props> = {}) {
  const defaultProps: Props = {
    onPasswordChange() {
      return;
    },
    onRememberChange() {
      return;
    },
    onUsernameChange() {
      return;
    },
    onSubmit() {
      return;
    },
    shouldRemember: true
  };
  return render(<LoginForm {...defaultProps} {...props} />);
}

```

Here, we are leveraging TypeScript to ensure our props are consistently applied to the `LoginForm` component. We start by defining some "default" props and then spreading additional props passed into the function as overrides. The override props are typed as `Partial<Props>` since they are optional.

If the `Props` interface changes, TypeScript will throw a compiler error and the test helper will need to be updated, ensuring our tests are kept updated.

React Testing Library offers a quick way to find elements using helpers.

170

will use `findByTestId` to find elements by their `data-testid` attribute value. You can also use `getByTestId` which is the sync version. We've given the `<form>` element a test ID value of `login-form` which we can query.

Testing library provides additional Jest matchers through @testing-library/jest-dom. In our example, we are using semantic form markup using the `<form>` element and input `name` attributes so we can use the `toHaveFormValues` matcher to more easily assert if the form values are what we expect:

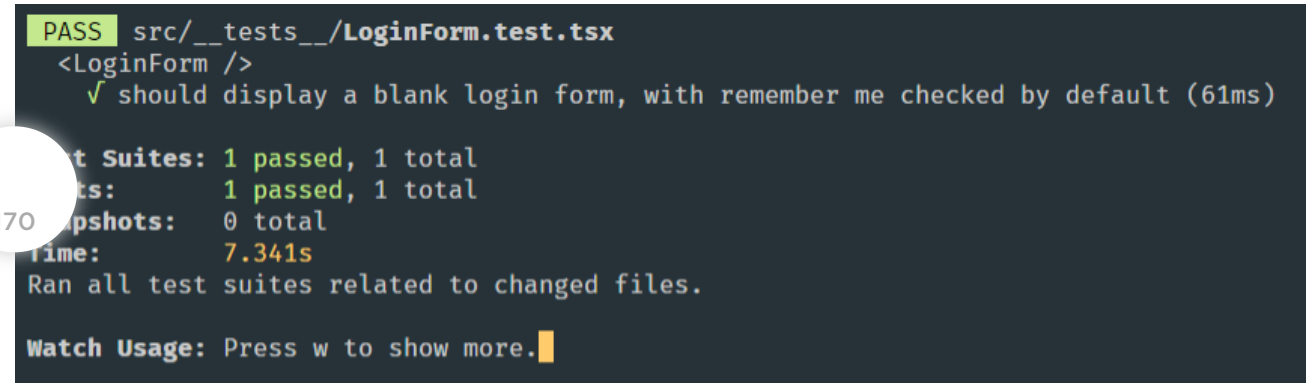
tsx

```tsx
test("should display a blank login form, with remember me checked by defa
  const { findByTestId } = renderLoginForm();

  const loginForm = await findByTestId("login-form");

  expect(loginForm).toHaveFormValues({
    username: "",
    password: "",
    remember: true
  });
});
```

The test still passes in the Jest output:



Testing Event Handling

The next tests should be to ensure that the user can interact with our form in the expected ways:

- Enter a username
- Enter a password
- Check or uncheck "Remember me"
- Submit the form

We'll need to use the `fireEvent` utility to help us modify the form's state. Rather than trying to test the internal state of the form, we've introduced callbacks on the props that we can mock to ensure that we get the value we expect for each interaction. Not only is this useful for testing, it is useful for any consumers of the `LoginForm` component.

Start by entering a username and password, and ensuring we get a change event back with the values we expect:

170

ts

```ts
test("should allow entering a username", async () => {
  const onUsernameChange = jest.fn();
  const { findByTestId } = renderLoginForm({ onUsernameChange });
  const username = await findByTestId("username");

  fireEvent.change(username, { target: { value: "test" } });

  expect(onUsernameChange).toHaveBeenCalledWith("test");
});

test("should allow entering a password", async () => {
  const onPasswordChange = jest.fn();
  const { findByTestId } = renderLoginForm({ onPasswordChange });
  const username = await findByTestId("password");

  fireEvent.change(username, { target: { value: "password" } });

  expect(onPasswordChange).toHaveBeenCalledWith("password");
});
```

Using `fireEvent.change` we can simulate a form change event on the inputs and assert that the right value was passed to the prop callback. We locate the inputs using their respective `data-testid` values of `username` and `password`.

Now, let's make sure the user can check the "remember me" checkbox:

ts

```ts
test("should allow toggling remember me", async () => {
  const onRememberChange = jest.fn();
  const { findByTestId } = renderLoginForm({
    onRememberChange,
    shouldRemember: false
  });
  const remember = await findByTestId("remember");

  fireEvent.click(remember);

  expect(onRememberChange).toHaveBeenCalledWith(true);

  fireEvent.click(remember);

  expect(onRememberChange).toHaveBeenCalledWith(false);
});
```

This time we're using `fireEvent.click` to toggle the checkbox on and off. By passing the initial prop `shouldRemember: false` we know, after the first click, that the value should be `true`.

Finally, let's fill in all the form fields and submit the form and ensure we receive the expected values:

ts

```ts
test("should submit the form with username, password, and remember", asyn
  const onSubmit = jest.fn();
```

```
    const { findByTestId } = renderLoginForm({
      onSubmit,
      shouldRemember: false
    });
    const username = await findByTestId("username");
    const password = await findByTestId("password");
    const remember = await findByTestId("remember");
    const submit = await findByTestId("submit");

    fireEvent.change(username, { target: { value: "test" } });
    fireEvent.change(password, { target: { value: "password" } });
    fireEvent.click(remember);
    fireEvent.click(submit);

    expect(onSubmit).toHaveBeenCalledWith("test", "password", true);
  });
```

The form is fully tested and meets our expectations. Using the React Testing Library encouraged us to write tests *without relying on the internal state* and use the DOM directly, just as our users would.

Testing Custom Hooks

As you can see above, testing React hooks like `useState` doesn't require anything special during tests when used in components. In

general, you **do not** need to test hooks in isolation unless they are complex or you are building a reusable library.

ou do need to test hooks you can leverage react-hooks-testing-library. This makes it easy to test hooks without requiring a component to wrap them.

There is no extra work necessary for writing the tests in TypeScript, you can follow the documentation for guidance on usage.

Conclusion

Testing React components with TypeScript is not too different from testing with JavaScript. In this guide we set up the Jest and React Testing Library with TypeScript support. We also learned the basics of testing a React component's state using `data-testid` attributes and callbacks by firing events.

The source code for this guide is available for reference on GitHub.