BACK

React Snapshot Testing With Jest: An Introduction With Examples

Michiel Mulders Jan 29, 2021 · 5 min read

Testing forms an integral part of software development. Both frontend and backend developers write tests. You can't escape them. Testing is an excellent way to improve trust in your code but also to guarantee code quality.

Many companies prefer to adopt agile development. Here, testing is part of the iterative development methodology. It allows software teams to catch bugs early on in the software development lifecycle. Catching those bugs early on saves your organization a lot of money. The cost of fixing a bug increases the longer it stays undetected.

This article takes a look at snapshot testing specifically. It's a technique for testing React components. Snapshot testing ensures that your UI doesn't change unexpectedly. Further, we'll be using the Jest CLI tool to maintain and update our snapshot tests.

So, why is snapshot testing such a powerful concept?

Why is snapshot testing so powerful for React component testing?

First of all, it's essential to understand how snapshot testing works. It's a type of comparison testing that compares the stored "good output" with the current output of a component. Therefore, snapshot testing is different from unit and functional testing. These types of testing focus on making assertions about the expected behavior or output. Instead, snapshot testing is only concerned with detecting unexpected UI changes. It won't tell you anything whether the component's behavior is false or correct. Snapshot testing only tells you that the output is different.

So, why would you use snapshot testing?

Snapshot testing has been created due to the need for an easier way to write tests for React components. Many React developers reported that they spend more time writing tests than the actual component. Therefore, snapshot testing allows React developers to quickly generate tests using its simple syntax.

You can create a snapshot test with just two lines of code. First, you have to pass a component and its specific properties you want to test. By calling the `toJSON()`, function you can generate an output for your component which you can later compare to detect unexpected UI changes. Next, we can pass the output to the `expect()` assertion and call the `toMatchSnapshot()` function to compare the current snapshot with the newly generated output.

Here's an example.

```
const elem = renderer.create(<MyComponent foo="bar">).toJSON();
expect(elem).toMatchSnapshot();
```

Note that snapshot testing doesn't support Test-Driven Development (TDD) as we need a finished component before we can "snapshot" it.

Now, let's prepare a small project to experiment with snapshot testing.

Project setup: Create a React Component

Let's create a new React project using `create-react-app`.

```
$ npx create-react-app@3.4.1 react-snapshot-testing
```

Next, let's change into the newly created folder.

```
$ cd react-snapshot-testing
```

Now, let's start the app to check if the setup works.

```
$ npm start
```

This command will start the React app and open the webpage in your browser at `http://localhost:3000/`. Next, let's create a React component. In this example, let's create a simple `Books` component that renders a list of books. To make this an exciting testing example, let's render a different UI depending on the length of the passed `books` array.

In your terminal, make a `components` directory under the `src` directory.

```
$ mkdir src/components
```

Further, let's create a `Books.js` file inside the `components` directory.

```
$ touch src/components/Books.js
```

Lastly, add the following code to `Books.js`:

```
import React from 'react';
import PropTypes from 'prop-types';

/**
 * Render a list of books
 *
 * @param {Object} props - List of books
 */
function Books(props) {
  const { books = [] } = props;

  // A single book in the list, render book in paragraph element
  if (books.length === 1) {
    return <p>{books[0]}</p>;
  }

  // Multiple books on the list, render a list.
  if (books.length > 1) {
    return (
      <ol>
        {books.map(book => <li key={book}>{book}</li>)}
      </ol>
    );
  }

  // No books on the list, render an empty message.
  return <span>We didn't find any books.</span>;
}

Books.propTypes = {
    books: PropTypes.array,
};

Books.defaultProps = {
    books: []
};

export default Books;
```

There are three possible outcomes for our `Books` component:

1. Render a message when there are no books in a `<span>` tag
2. Render a single book in a `<p>` tag
3. Render multiple books in a `<ol>` tag

Finally, we have to update the `src/App.js` code to render the `Books` component. Make sure to replace the contents of the `App.js` file with the code below:

```
import React, { Component } from 'react';
import Books from './components/Books';

class App extends Component {
  render() {
    const books = [
      'Harry Potter',
      'The Lord of the Rings',
```

```
      'The City of Dreaming Books'
    ];
    return (
      <div className="App">
        <Books books={books} />
      </div>
    );
  }
}

export default App;
```

You can verify if everything works correctly by visiting `localhost:3000` in your browser. Make sure the `npm start` command is still running in your terminal. If yes, your browser renders the following output.

```
1. Harry Potter
2. The Lord of the Rings
3. The City of Dreaming Books
```

All good? Let's prepare the testing setup.

Prepare the testing setup

Firstly, delete the `src/App.test.js` file as we don't need it for this tutorial.

```
$ rm src/App.test.js
```

Next, let's create a test file for the `Books` component.

```
$ touch src/components/Books.test.js
```

Next, install the `react-test-renderer` library that enables you to render React components to pure JavaScript objects without depending on the DOM or a native mobile environment.

```
$ npm install react-test-renderer@16.13.1
```

For instance, a `<Link>` component might look like this:

```
{
    type: 'a',
    props: { href: 'https://www.facebook.com/' },
    children: [ 'Facebook' ]
}
```

Done? Let's get testing!

How to write a snapshot test?

Remember your `Books` component that accepts a `books` array via the props?

```
<Books books={books} />
```

Let's create a snapshot for the three possible logical flows for our component:

1. Render a message when there are no books in a `<span>` tag
2. Render a single book in a `<p>` tag
3. Render multiple books in a `<ol>` tag

First, let's snapshot the `Books` component when we pass no books. We use the `react-test-renderer` to snapshot the component's output. Add the code below to your `src/components/Books.test.js` file.

```
import React from 'react';
import renderer from 'react-test-renderer';

import Books from './Books';

it('should render an empty message when no books', () => {
  const elem = renderer.create(<Books />).toJSON();
  expect(elem).toMatchSnapshot();
});
```

Next, try to write test cases for scenarios `2` and `3`.

Ready?

Here's the solution.

```
import React from 'react';
import renderer from 'react-test-renderer';

import Books from './Books';

it('should render an empty message when no books', () => {
  const elem = renderer.create(<Books />).toJSON();
  expect(elem).toMatchSnapshot();
});

it('should render a single book', () => {
    const books = ['Harry Potter']
    const elem = renderer.create(<Books books={books} />).toJSON();
    expect(elem).toMatchSnapshot();
});

it('should render multiple books', () => {
    const books = ['Harry Potter', 'The Lord of the Rings']
    const elem = renderer.create(<Books books={books} />).toJSON();
    expect(elem).toMatchSnapshot();
});
```

Alright, let's run our tests via the terminal.

```
$  npm test
```

If everything works well, all three tests should pass.

Note the `src/components/__snapshots__` folder that the `npm test` command has created. This folder contains the snapshot outputs. Let's take a look:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`should render a single book 1`] = `
<p>
  Harry Potter
</p>
`;

exports[`should render an empty message when no books 1`] = `
<span>
  We didn't find any books.
</span>
`;

exports[`should render multiple books 1`] = `
<ol>
  <li>
    Harry Potter
  </li>
  <li>
    The Lord of the Rings
  </li>
</ol>
`;
```

It contains the rendered output for each of your test cases. But what happens when we update the UI? Let's find out!

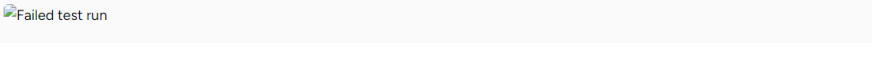
How to update snapshot tests?

So, you've rendered snapshot outputs to detect UI changes. Yet, you update the `Books` component, and now all tests fail. This section explains how you can update your snapshot tests.

First, change the `<span>` tag to a `<b>` tag when we pass an empty `books` array in `src/components/Books.js`.

```
// No books on the list, render an empty message.
return <b>We didn't find any books.</b>;
```

If you run `npm tests` in your terminal, the test case `should render an empty message when no books` fails.



Now, Jest comes into play. Actually, `npm test` uses the Jest CLI to run your tests. You will likely see the interactive mode. Press `u` to update the failing tests.

However, you can also install the Jest CLI globally with `npm install jest -g`. This allows you to use the `jest --updateSnapshot` command from the terminal to update all snapshots. Personally, I prefer using Jest's interactive mode.

Further, take a look at the `src/components/__snapshots__/` folder. The `Books.test.js.snap` file contains the updated snapshot.

```
exports[`should render an empty message when no books 1`] = `
<b>
  We didn't find any books.
</b>
`;
```

Also, you should see passing tests again. That's simple, right?

Conclusion

This tutorial has shown you how to write snapshot tests and update them when your UI changes.

Jest snapshot testing is a great tool for React developers to detect unexpected UI changes. They are easy to create and maintain. Yet, make sure to write test cases for all possible flows in your React component. Remember that you can't replace unit or functional tests with snapshot tests. Snapshot tests act as a simple tool to improve your code's quality. In the end, you need unit and functional tests to verify your application's behavior.

If you want to learn more about Jest snapshot testing, take a look at the best practices documentation by Jest.

Frontend Application Monitoring

A broken experience is not always due to an error or a crash, but may be the consequence of a bug or a slowdown that went unnoticed. Did my last release introduce that? Is it coming from the backend or the frontend? OpenReplay helps answer those questions and figure out what part of your code requires fixing or optimization. Because inconsistent performance simply drives customers away, causing retention and revenues to drop.

As we embrace agility, we push code more frequently than ever, and despite our best testing efforts, our code may end up breaking for a variety of reasons. Besides, frontend is different. It runs on different browsers, relies on complex JS frameworks, involves multiple CDN layers, gets affected by 3rd-party APIs, faulty internet connections, not-so-powerful devices and slow backends. In fact, frontend requires better visibility, and OpenReplay provides just that.

Your frontend is your business. It's what people touch and feel. Let's make it fast, reliable and delightful!

Start monitoring your web app for free.

More articles from OpenReplay Blog

May 10, 2023, 3 min read
Generics in TypeScript

Jul 28, 2022, 3 min read
Generating barcodes with Vue