Snapshot Tests with Jest

In the next couple of lessons, we'll cover testing, an important practice to keep source code robust. In this lesson, we'll brush upon the basics of software testing and then learn the basics of testing snapshot testing React components with Jest - Facebook's JavaScript testing library.

Software Testing

Testing source code is essential to programming, and should be seen as mandatory. We want to keep the quality of your code high and make sure everything works before using it in production. We will use the testing pyramid as our guide.

The testing pyramid includes end-to-end tests, integration tests, and unit tests. A unit test is for an isolated and small block of code, such a single function. However, sometimes units work well in isolation but not in combination with other units, so they need to be tested as a group. That's where integration tests can help us figure out if units work well together. An end-to-end test is a simulation of a real-life scenario, such as the automated setup of a browser simulating the login flow in a web application. Where unit tests are fast and easy to write and maintain, end-to-end tests are at the opposite of the spectrum.

We want to have many unit tests covering the isolated functions. After that, we can use several integration tests to make sure the most important functions work in combination as expected. Finally, we may need a few end-to-end tests to simulate critical scenarios.

Testing React

The foundation for testing in React are component tests, generalized partly as unit tests and partly as snapshot tests. Unit tests for our components will be covered in the next chapter with a library called Enzyme. In this section, we focus on snapshot tests, using Jest.

Introduction to Jest

Jest is a JavaScript testing framework used by Facebook. It is used for component tests by the React community. Fortunately, *create-react-app* already comes with Jest, so you don't need to set it up.

Jest Setup

Before you can test your first components, you have to export them from your *App.js* file. Afterward, you can test them in a different file.

```
...

class App extends Component {
  ...
}

...

export default App;

export {
  Button,
  Search,
  Table,
};
```

In your *App.test.js* file, you will find the first test that came with *create-react-app*. It verifies the App component will render without errors.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

The "it"-block describes one test case. It comes with a test description, and when you test it, it can either succeed or fail. Furthermore, you could wrap it into a "describe"-block that defines your test suite. A test suite could include a bunch of the "it"-blocks for one specific component. You will see "describe"-blocks later. Both blocks are used to separate and organize your test cases.

Note that the `it` function is acknowledged in the JavaScript community as the

function where you run a single test. However, in Jest it is often found as an alias `test` function.

You can run the test cases using the interactive *create-react-app* test script on the command line. You will get the output for all test cases on your command line interface.

```
npm test
```

Jest enables you to write snapshot tests. These tests make a snapshot of your rendered component and runs it against future snapshots. When a future snapshot changes, you will get notified in the test. You can either accept the snapshot change, because you changed the component implementation on purpose, or deny the change and investigate for the error. It complements unit tests very well, because you only test the differences of the rendered output. It doesn't add big maintenance costs since you can accept snapshots for intentional changes.

Jest stores snapshots in a folder so it can validate the diff against a future snapshot. This also lets use share snapshots across teams when having version control such as git in place.

Before writing your first snapshot test with Jest, you have to install its utility library:

```
npm install --save-dev react-test-renderer
```

Now you can extend the App component test with your first snapshot test. First, import the new functionality from the node package and wrap your previous "it"-block for the App component into a descriptive "describe"-block. In this case, the test suite is only for the App component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
```

```
  });

});
```

Testing with Jest

Now you can implement your first snapshot test by using a “test”-block:

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <App />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Run your tests again and observe how they succeed or fail. Once you change the output of the render block in your App component, the snapshot test should fail. Then you can decide to update the snapshot or investigate in your App component.

The `renderer.create()` function creates a snapshot of your App component. It renders it virtually, and then stores the DOM into a snapshot. Afterward, the snapshot is expected to match the previous version from the last test. This is how we make sure the DOM stays the same and doesn’t change anything by accident.

Testing the Search component

Let’s add more tests for our independent components. First, the Search component:

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

```
import renderer from 'react-test-renderer';
import App, { Search } from './App';

...

describe('Search', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

The Search component has two tests similar to the App component. The first test simply renders the Search component to the DOM and verifies that there is no error during the rendering process. If there would be an error, the test would break even though there isn't any assertion (e.g. expect, match, equal) in the test block. The second snapshot test is used to store a snapshot of the rendered component and to run it against a previous snapshot. It fails when the snapshot has changed.

Second, you can test the Button component whereas the same test rules as in the Search component apply.

```
...
import App, { Search, Button } from './App';

...

describe('Button', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
```

```
});
```

Finally, the Table component that you can pass a bunch of initial props to render it with a sample list.

```
...
import App, { Search, Button, Table } from './App';
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Snapshot tests usually stay pretty basic. You only want to make sure the component doesn't change its output. Once it does, you have to decide if you accept the changes, otherwise you have to fix the component.

Exercises

- See how a snapshot test fails once you change your component's return value in the `render()` method
  - Accept or deny the snapshot change(s)
- Keep your snapshots tests up to date when the implementation of components change in next chapters

Further Reading:

- Read about Jest in React

Quick quiz on Jest!

Q

The `it` block can contain multiple test cases.

COMPLETED 0%

1 of 1