Master Frontend Testing: A Complete Guide Using Jest

Follow For More

Web Development | Design | Coding

Swipe

Introduction

Writing tests is one of the hardest things that a developer has to do. Writing frontend tests is even harder, but it is necessary, it allows us to detect bugs early on before they reach our users, and to build a stable, reliable codebase.

If you're willingly avoiding frontend testing, it's time to stop, it will not help you in the long run and it brings no benefits to the applications you're working on.

In this post, we will go through a testing journey and get familiar with testing types, concepts and even write a few test cases for a React application using Jest & Enzyme!

Let's Get Started! ⟶

Next

1 Types of Frontend Testing

Frontend testing can be easily divided into three major categories: E2E (End To End) Testing, Integration Testing and Unit Testing.

E2E Testing

Testing that the app performs as intended from start to finish by using real world flows. This includes testing component communication and API calls.

Integration Testing

Usually means testing the communication between the UI and APIs it communicates with to ensure integration.

Unit Testing

The most common and easiest to implement, used for testing isolated parts of the code as units. A unit can be a method, component, action, etc.

In our examples, we will be writing unit tests!

Next

2

Unit Testing Basics

To write unit tests, we use testing frameworks such as Jest, which is a great & popular choice! With Jest, we can set up a testing environment and then create test suites with multiple test cases.

When writing unit tests, it's important to know what these topics are:

Test Suite

A group of unit tests that are related, for example, all the tests for a specific component or method.

Unit Test

The actual test that we write for our unit, complete with a relevant description and the test code.

Test Assertion

Assertions are expectations that we have about our code, for example, we could expect a value to be truthy, if assertions are met, the test passes!

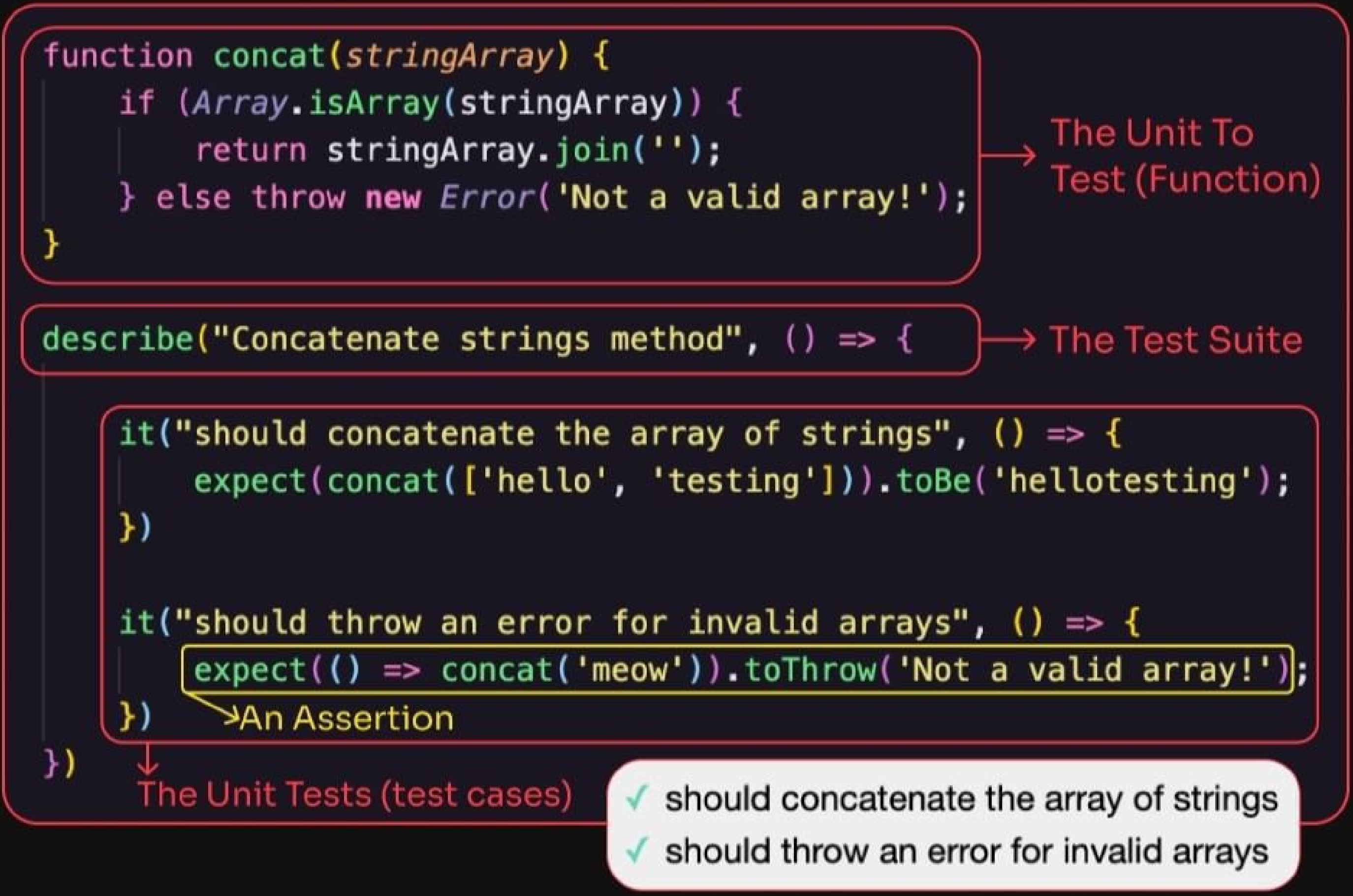
Now let's do some practice!

Next

3

Writing Unit Tests - Example

Let's say we need to test that a function used for the concatentation of an array of strings works correctly!



```
function concat(stringArray) {
    if (Array.isArray(stringArray)) {
        return stringArray.join('');
    } else throw new Error('Not a valid array!');
}

describe("Concatenate strings method", () => {

    it("should concatenate the array of strings", () => {
        expect(concat(['hello', 'testing'])).toBe('hellotesting');
    })

    it("should throw an error for invalid arrays", () => {
        expect(() => concat('meow')).toThrow('Not a valid array!');
    })
})
```

✓ should concatenate the array of strings
✓ should throw an error for invalid arrays

As you can see, our tests make sure that the code we wrote functions as intended! We made sure to cover two important cases, and can safely say the function is implemented in a proper way!

Next

4 Writing Unit Tests - React Example

Let’s take it a step further, now we will write some unit tests for a React counter component that we created!

```
import Enzyme, { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import Counter from './Counter';
import App from './App';

Enzyme.configure({ adapter: new Adapter() });

describe('Counter component', () => {
    it('renders the component successfully', () => {
        const app = shallow(<App />);
        expect(app.find(Counter)).toBeTruthy();
    });

    it('should increment the counter', () => {
        const wrapper = shallow(<Counter />);
        const incrementBtn = wrapper.find('button');
        const count = wrapper.find('p').text();

        expect(count).toEqual('0');

        incrementBtn.simulate('click')
        expect(count).toEqual('1');
    })
})
```

Counter.test.js

```
const Counter = () => {
    const [counter, setCounter] = useState(0);

    const incrementCounter = () => {
        setCounter((count) => count + 1);
    };

    return (
        <>
            <button onClick={incrementCounter}>
                +
            </button>
            <p>{counter}</p>
        </>
    );
}
```

Counter.jsx

We use Enzyme to simulate rendering & certain events in our component! With this technique we can test multiple behaviours and cover all the relevant cases!

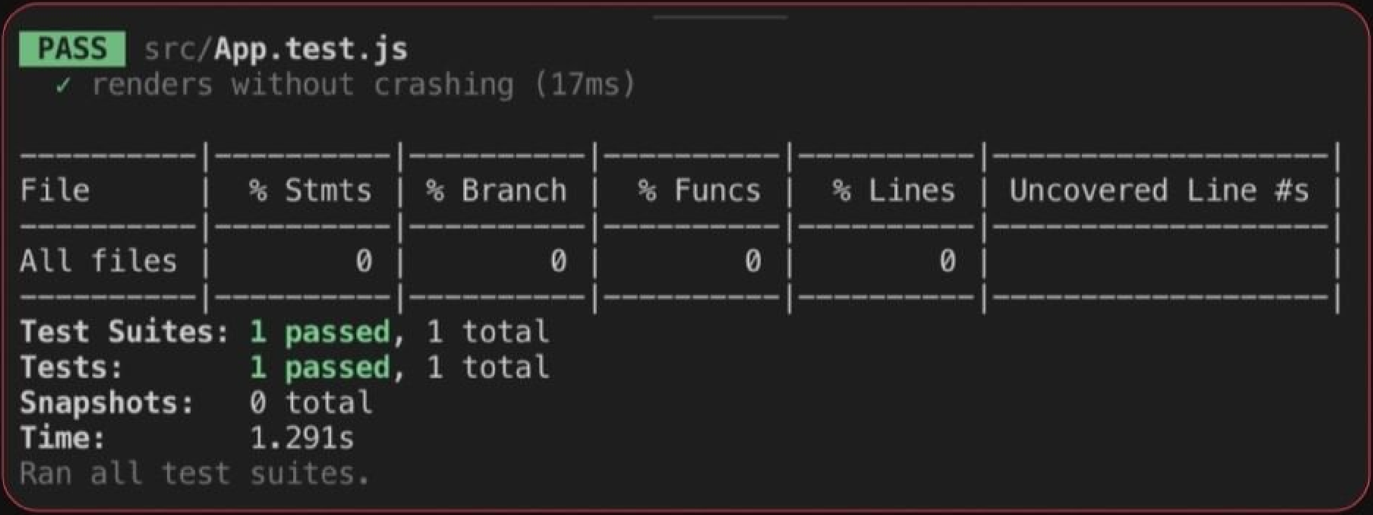
Next

5 Code Coverage Metrics

Another important testing concept to understand is code coverage. Code coverage is a testing metric to help assess the test performance and quality of the software, however, it can be a bad metric if you're only writing tests to meet the coverage and not to write relevant test cases for a given unit.

Code coverage is split between 4 main metrics, which are: Function, Statement, Branch & Line coverage.

Code coverage is measured in percentages from 0 to 100, and usually looks similar to this for a unit:



```
PASS  src/App.test.js
  ✓ renders without crashing (17ms)

----------|----------|----------|----------|----------|-------------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------|----------|----------|----------|----------|-------------------|
All files |        0 |        0 |        0 |        0 |                   |
----------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.291s
Ran all test suites.
```

Next

AMER SHBOUL

Was this useful?

Follow me for more posts like this

Let me know with a comment

Share it with others as well

Let me know you liked it!

Save for later!