Code coverage with Jest

Internal reference: topics/06-2.md

Introduction

Popular JavaScript frameworks can use Facebook's Jest to perform unit tests. Jest has the Coverage Report feature that allows us to check if our code covers all lines of the files we choose by generating an HTML file that we can open.

Code coverage criteria in Jest

1. `Statements` represent instructions that have been executed at least once during the unit tests. For example, we can have a line that contains two statements:

```
var age= 18; console.log(age)
```

This contains a variable declaration statement and a statement that executes the log function that belongs to the console object.

1. `Branches` represent if statements which conditions have been fulfilled at least once during the unit tests.
2. `Functions` represent functions that have been called at least once during the unit tests.
3. `Lines` represent code lines that have executed at least once during the unit tests.

Jest code coverage report in your terminal

Seeing your code coverage can be as simple as adding the `–coverage` flag when running your Jest unit tests:

```
jest --coverage
```

After you run the coverage command you'll get a summary report that looks like this:

```
$ jest --coverage
 PASS  tests/trafficlight.test.js
  test suite traffic light
    √ case red light (2 ms)

----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |   44.44 |       25 |     100 |   44.44 |
 trafficlight.js |   44.44 |       25 |     100 |   44.44 | 28,33-37
----------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.708 s, estimated 1 s
Ran all test suites.
```

By adding –`collectCoverageFrom`, Jest will calculate code coverage for all the files that you specify. Even ones without any tests:

```
jest --coverage --collectCoverageFrom='src/**/*.js'
```

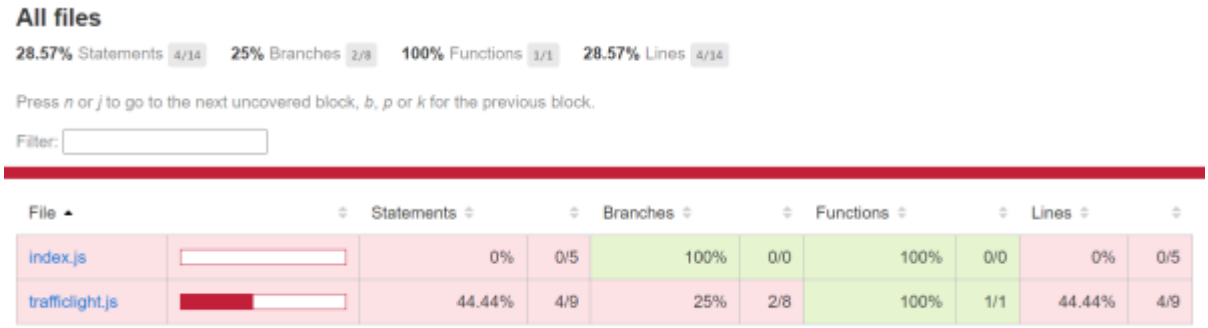
```
$ jest --coverage --collectCoverageFrom='src/**/*.js'
 PASS  tests/trafficlight.test.js
  test suite traffic light
    √ case red light (3 ms)

----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |   28.57 |       25 |     100 |   28.57 |
 index.js       |       0 |      100 |     100 |       0 | 1-6
 trafficlight.js |   44.44 |       25 |     100 |   44.44 | 28,33-37
----------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.555 s
Ran all test suites.
```

Jest code coverage report in your browser

When looking at the summary table, it can be very hard to determine where you are missing coverage! A much easier way is to generate an HTML report in the folder you specified with –`coverageDirectory`:

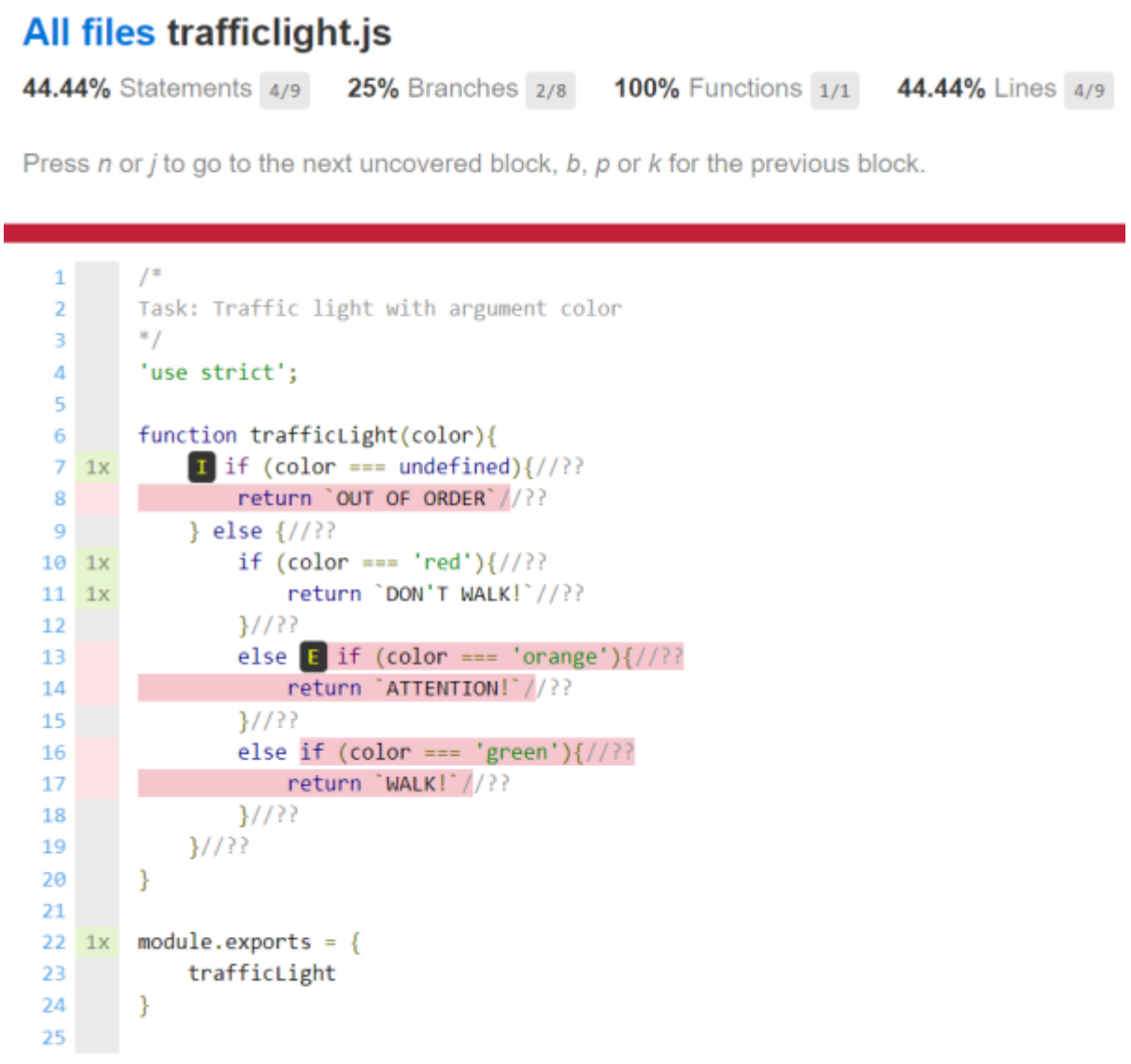
```
jest --coverage --coverageDirectory='coverage' --
collectCoverageFrom='src/**/*.js'
```



If you open up the index.html file in your browser, you will see lines highlighted in red. These are the lines that are not currently covered by your unit tests.

Symbols and Numbers

If you open up the index.html file in your browser, you will see lines highlighted in red. These are the lines that are not currently covered by your unit tests.



By analyzing the lines of code we can see 1x on the left hand side. It means that we executed that part of the code one time during our unit tests. This happens because I only tested one of the functional requirements on my unit tests.

The `I` symbol at line 7 means that this branch has not been entered during unit tests (if path not taken). Similar is the `E` symbol at line 13 means that alternative branch has not been entered during unit tests (else path not taken).

If there is a part of line that is highlighted in yellow, it means that the possible branch is not covered.

The pitfalls of aiming for 100% coverage

As you increase your code coverage, sometimes it will be too hard to cover certain lines of code with unit tests. Spending your time trying to find a workaround to cover that line of code is never worth it. There are much better things you could be spending your time on than striving for 100% coverage!

Even if you do cover a line of code, there's no guarantee that it will be perfect and bug-free, either. If

we take a look at a double function that doubles the number you pass in:

```
const double = (number) => 2;
```

You could test that double(1) = 2 and that test would pass. You would have 100% code coverage as well. But your function would fail with all other numbers.

Code coverage is useful, but it's important not to use it as the only metric to measure your unit tests. Make sure to keep in mind all the possible edge cases and different scenarios, as code coverage won't pick these up.
