Testing in JavaScript

Part 2 of 5 in our *Testing like a Pro in JavaScript* series.
Written by Alex Jover Morales

In the second part of this 5 part series about testing in JavaScript, we're going to look at a few tools we can use and then write our very first unit tests.

Before going hands-on with some JavaScript testing and its workflow, let's have a comparison of the tools we can use.

Tools

For **unit tests** we have several options. The most popular ones:

- **Jest**: it has become the de facto testing tool in the JavaScript community. It's a modern tool that includes all what's needed for testing: great mocking capabilities, expect library, snapshot testing, etc. It uses a fast emulated browser called JSDom, executes in parallel, integrates well with other libraries and it's very easy to configure starting from zero-config.
- **AVA**: along with Jest, AVA is another modern tool for testing. It's not as fully featured as Jest and requires other tools in order to get more capabilities. It's perfect though for libraries or small apps, but not limited to them.
- **Karma + Mocha + Chai**: These are the tools conforming the traditional testing stack. It's more mature than the other two, but it has some disadvantages in comparison: more dependencies, slower tests, less capabilities. Karma can provide a real browser environment, although that's not needed in unit testing and it makes it run quite slower. The configuration can be painful as well.

As per **E2E tests**, some of the most populars high level testing frameworks are:

- Cypress: one of the newest tools that aims to provide an easy and concise API for testing. It uses its own engine.
- Nightwatch: it's one of the most used and popular in the JavaScript community. It runs on WebDriver or Selenium under the hood.

I'm choosing Jest given its simplicity, popularity and power, and I'll use it for these tutorial.

Setup Jest

Start by creating an empty project (via `npm init` for instance) and install Jest:

```
npm install -D jest
```

And create a `test` script in the *package.json*:

```
{
  "name": "jest-testing",
  "version": "0.1.0",
  "scripts": {
    "test": "jest"
  }
}
```

Using ECMA Script Modules

Just with what we've done so far, jest will already work running `npm test`. By default, Jest understands CommonJS modules, the module format used by NodeJS with the `require` syntax. If you want to use ECMA Script modules with the `import` and `export` syntax, you'll need to do some extra steps.

Let's use the latest babel-preset-env. Install it by running:

```
npm install -D babel-preset-env
```

Then create a *.babelrc* file with the following content:

```
{
  "presets": ["env"]
}
```

Creating the First Test

In this example, let's create a `Calculator` as the test object.

```
// calculator.js
export default class Calculator {
  add(a, b) {
    return a + b;
  }
  subtract(a, b) {
    return a - b;
  }
}
```

In Jest, a test starts with a `describe` function referring to the test object. It contains several `it` statements for each test case.

Let's start by testing that the Calculator is in fact an instanceable class:

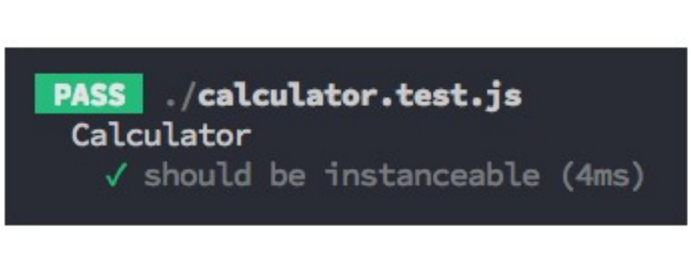
```
// calculator.test.js
import Calculator from "./calculator";

describe("Calculator", () => {
  it("should be instanceable", () => {
    expect(new Calculator()).toBeInstanceOf(Calculator);
  });
});
```

As you can see, *Calculator* is the first test subject, and it contains an `it` function describing the test case *should be instanceable* by using the `toBeInstanceOf(…)` matcher from `expect`. You can see the whole list of matchers in Jest's expect documentation, but we'll be using them along the way.

Jest automatically detects test files that have the `.test` or `.spec` suffix, as well as the `.js` files within a `__tests__` folder. That's why I've named the calculator test `calculator.test.js`.

Then, just run `npm test` and the test should pass \o/.



```
PASS ./calculator.test.js
  Calculator
    ✓ should be instanceable (4ms)
```

Testing the Calculator

The calculator has several functions, each of them being a test item. A test item can have several test cases as well.

We can nest several `describe` in order to structure them, so we'll have:

```
// calculator.test.js
import Calculator from "./calculator";

describe("Calculator", () => {
  it("should be instanceable", () => {
    expect(new Calculator()).toBeInstanceOf(Calculator);
  });
  describe("add", () => {
    //...
  });
  describe("subtract", () => {
    //...
  });
});
```

Let's add a test for the cases where they should just work:

```
// calculator.test.js
// ...
describe("add", () => {
  it("should sum up 2 numbers", () => {
    const calculator = new Calculator();
    expect(calculator.add(3, 2)).toBe(5);
  });
});
describe("subtract", () => {
  it("should subtract 2 numbers", () => {
    const calculator = new Calculator();
    expect(calculator.subtract(3, 2)).toBe(1);
  });
});
```

Test Driven Development (TDD)

Test Driven Development is a programming methodology based on writing the tests before implementing the code. That makes you first think on the requirements, then how it should behave and how the final API of a module should look like, and finally the implementation details.

It's been adopted and fits perfectly in agile workflow methodologies because of the small and continuous development cycles that come naturally with TDD.

Jest has a **watch mode** that comes in handy for it. It doesn't only watch for files, but also provides you with some options to run only files changed or filter by file or test names. Just run the test script with the watch option:

```
npm test -- --watch
```

Note: Jest uses Git under the hood to determine which files have changed. If you're not using Git, use the watchAll option instead.

Then, we can use TDD to define more use cases for the `add` method:

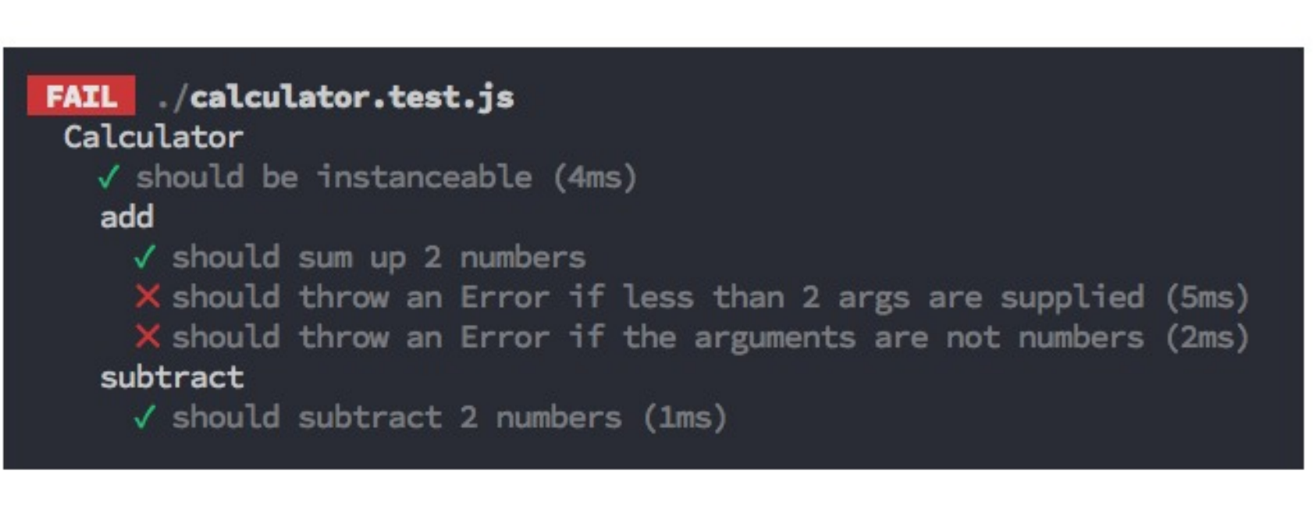
```
// calculator.test.js
// ...
describe("add", () => {
  // ...
  it("should throw an Error if less than 2 args are supplied", () => {
    const calculator = new Calculator();
    expect(() => calculator.add(3)).toThrow();
  });
  it("should throw an Error if the arguments are not numbers", () => {
    const calculator = new Calculator();
    expect(() => calculator.add(3, "2")).toThrow();
  });
});
```

As you can notice, when using error throwing matchers, like `toThrow`, we need to wrap the expected object in a function, since the expect wraps it internally in a `try/catch`.

You can also specify a specific error by passing a string as an argument or an Error instance to the `toThrow` function:

```
// calculator.test.js
// ...
describe("add", () => {
  // ...
  it("should throw an Error if less than 2 args are supplied", () => {
    const calculator = new Calculator();
    expect(() => calculator.add(3)).toThrow("2 arguments are required");
  });
  it("should throw an Error if the arguments are not numbers", () => {
    const calculator = new Calculator();
    expect(() => calculator.add(3, "2"))
      .toThrow(Error("The arguments must be numbers"));
  });
});
```

When you save the file, you'll see that the test are failing. That's great, now you have a development cycle where you write tests, it goes red, then write code until it goes green.



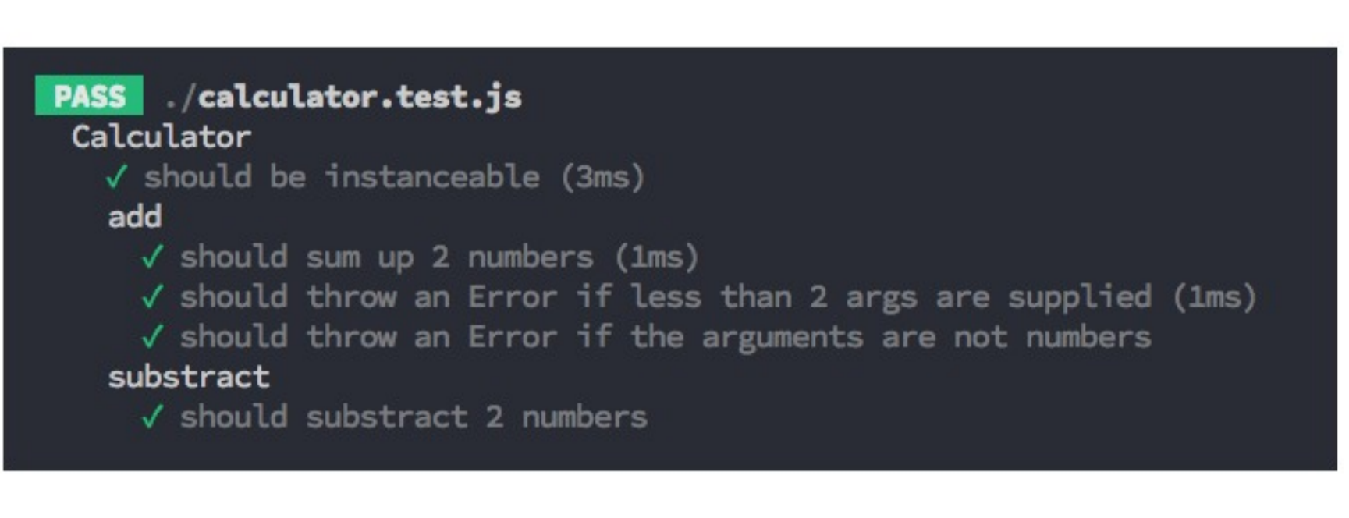
```
FAIL ./calculator.test.js
  Calculator
    ✓ should be instanceable (4ms)
    add
      ✓ should sum up 2 numbers
      ✕ should throw an Error if less than 2 args are supplied (5ms)
      ✕ should throw an Error if the arguments are not numbers (2ms)
    subtract
      ✓ should subtract 2 numbers (1ms)
```

That's exactly the principle of the red-green refactor, a technique where you safely refactor a piece of code. It will probably go red, but the refactor finishes when it goes green. That allows to move code around safely knowing that the application is still working as expected, without any regression.

Let's make our test go green and implement the type checks on the Calculator:

```
// calculator.js
export default class Calculator {
  _checkArgs(a, b) {
    if (a === undefined || b === undefined) {
      throw new Error("2 arguments are required");
    }
    if (typeof a !== "number" || typeof b !== "number") {
      throw new Error("The arguments must be numbers");
    }
  }
  add(a, b) {
    this._checkArgs(a, b);
    return a + b;
  }
  subtract(a, b) {
    this._checkArgs(a, b);
    return a - b;
  }
}
```

They should be green again \o/.



```
PASS ./calculator.test.js
  Calculator
    ✓ should be instanceable (3ms)
    add
      ✓ should sum up 2 numbers (1ms)
      ✓ should throw an Error if less than 2 args are supplied (1ms)
      ✓ should throw an Error if the arguments are not numbers
    substract
      ✓ should substract 2 numbers
```

Coverage, and What to Test

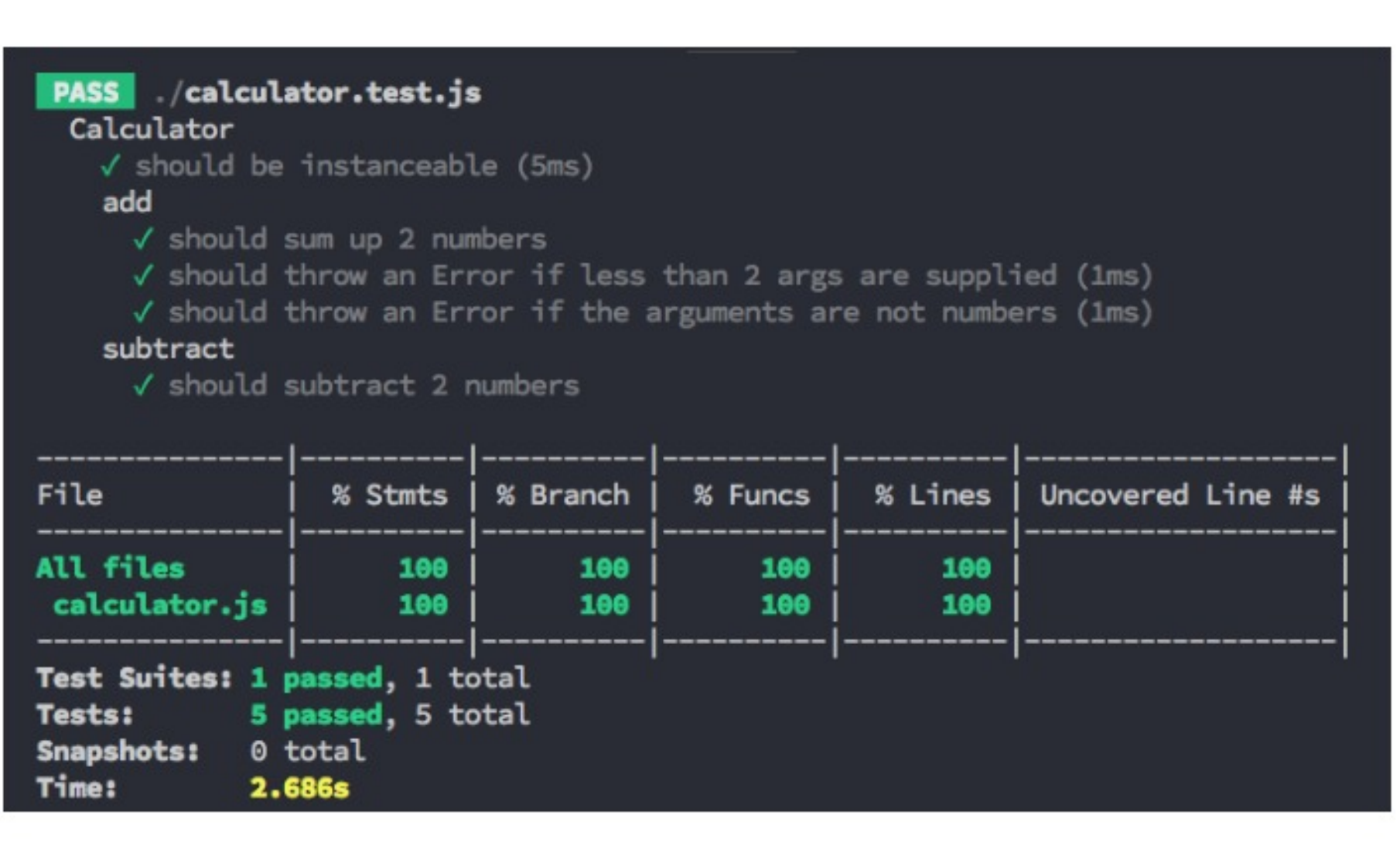
Have you noticed I've added a `_checkArgs` method to delegate the check logic? Probably you're wondering… should we test that method?

Here's the answer is clear: no. That method is a helper method used by `add` and `subtract`, and it's implicitly tested. Think about it, with the tests we've added to the `add` method, the test cases for `_checkArgs` are tested as well.

The **coverage** is a metric we can use to know how much of our code is tested. Jest comes with a built-in coverage tool that we can use by running it with the coverage argument:

```
npm run test -- --coverage
```

That should give us a 100% coverage right now:



```
PASS ./calculator.test.js
  Calculator
    ✓ should be instanceable (5ms)
    add
      ✓ should sum up 2 numbers
      ✓ should throw an Error if less than 2 args are supplied (1ms)
      ✓ should throw an Error if the arguments are not numbers (1ms)
    subtract
      ✓ should subtract 2 numbers

---------------|----------|----------|----------|----------|-------------------|
File           |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
---------------|----------|----------|----------|----------|-------------------|
All files      |      100 |      100 |      100 |      100 |                   |
 calculator.js |      100 |      100 |      100 |      100 |                   |
---------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        2.686s
```

I think statements, functions and lines are clear, but what the heck are branches? They're the possible executions path of your code. For example, if your code has an `if` statement, it has two (or more) paths: the one that enters the `if` and the one that doesn't.

Keep in mind that coverage is not the ultimate metric for testing, in fact it has some physiological effect. If your target is a developer, for example when you create a library, they will like it with a high coverage.

Having a 100% of coverage doesn't mean the code is unbreakable. In fact, it could be poorly or badly tested and still give a 100% coverage.

Keep in mind that, while testing saves time on the long term, it also takes time to write them. And the truth is, not all tests provide the same value. Some tests are harder to write than others, and there are modules on the codebase that are more important than others. For example, a payment module should be quite important to test, while some logging tools might not.

Think of the time you have to develop something in your sprint, the importance of it, the time you've been provided with and make your decisions. Lots of companies set a rule of a minimum coverage on the test suite, usually around 70% to 80%, which makes sense.

Remember: not everything needs to be tested, be pragmatic.

This tutorial is part of our *Testing like a Pro in JavaScript* series
Next: Test Doubles (3 of 5)
Previous: What's testing and why should we do it? (1 of 5)

Article written by Alex Jover Morales

Passionate web developer. Author of Test Vue.js components with Jest on Leanpub. I co-organize Alicante Frontend. Interested in web performance, PWA, the human side of code and wellness. Cat lover, sports practitioner and good friend of his friends. His bike goes with him.

2 Comments Vue School 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

MM · 3 months ago
You need to change your package.json file for any test to run and for coverage to work. I would change the article to include that. For example, coverage doesn't run for me unless I add the --coverage to my package.json. My test files don't run unless I install other dependencies such as mocha and edit my babel.rc file and package.json file etc.
Reply · Share ›

the_aceix · 4 months ago
Good series! Moving to part 3..
Reply · Share ›

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS