

Testing JavaScript with Jest

Jest is a library for testing JavaScript code. It's an open source project maintained by Facebook, and it's especially well suited for React code testing, although not limited to that: it can test any JavaScript code. Jest is very fast and easy to use.

Jest is a library for testing JavaScript code.

It's an open source project maintained by Facebook, and it's especially well suited for React code testing, although not limited to that: it can test any JavaScript code. Its strengths are:

- it's fast
- it can perform snapshot testing
- it's opinionated, and provides everything out of the box without requiring you to make choices

Jest is a tool very similar to Mocha, although they have differences:

- Mocha is less opinionated, while Jest has a certain set of conventions
- Mocha requires more configuration, while Jest works usually out of the box, thanks to being opinionated
- Mocha is older and more established, with more tooling integrations

In my opinion the biggest feature of Jest is it's an out of the box solution that works without having to interact with other testing libraries to perform its job.

<https://flaviocopes.com/jest/>

Installation

```
npm install --save-dev jest
```

notice how we instruct both to put Jest in the devDependencies part of the package.json file, so that it will only be installed in the development environment and not in production.

Add this line to the scripts part of your package.json file:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

so that tests can be run using yarn test or npm run test.

```
npm run test
```

Let's create the first test. Open a math.js file and type a couple functions that we'll later test:

```
const sum = (a, b) => a + b
const mul = (a, b) => a * b
const sub = (a, b) => a - b
const div = (a, b) => a / b

export default { sum, mul, sub, div }
```

Now create a math.test.js file, in the same folder, and there we'll use Jest to test the functions defined in math.js:

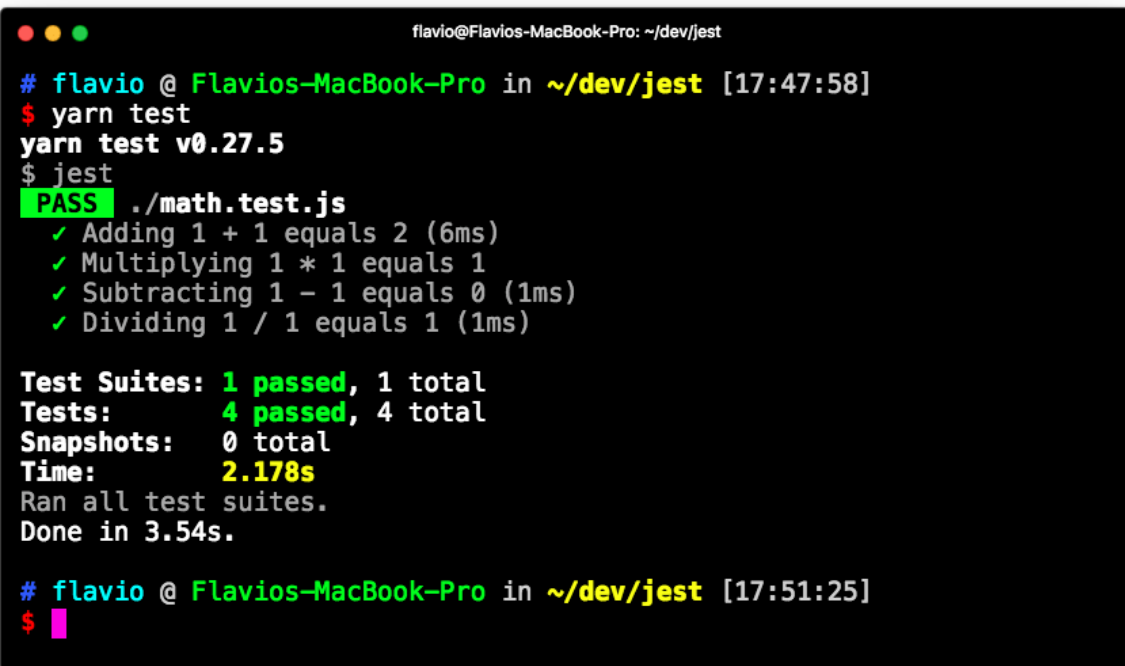
```
const { sum, mul, sub, div } = require("./math")

test("Adding 1 + 1 equals 2", () => {
  expect(sum(1, 1)).toBe(2)
})
test("Multiplying 1 * 1 equals 1", () => {
  expect(mul(1, 1)).toBe(1)
})
test("Subtracting 1 - 1 equals 0", () => {
  expect(sub(1, 1)).toBe(0)
})
```

<https://flaviocopes.com/jest/>

```
})  
test("Dividing 1 / 1 equals 1", () => {  
  expect(div(1, 1)).toBe(1)  
})
```

Running yarn test results in Jest being run on all the test files it finds, and returning us the end result:



```
flavio@Flavios-MacBook-Pro: ~/dev/jest  
# flavio @ Flavios-MacBook-Pro in ~/dev/jest [17:47:58]  
$ yarn test  
yarn test v0.27.5  
$ jest  
PASS ./math.test.js  
  ✓ Adding 1 + 1 equals 2 (6ms)  
  ✓ Multiplying 1 * 1 equals 1  
  ✓ Subtracting 1 - 1 equals 0 (1ms)  
  ✓ Dividing 1 / 1 equals 1 (1ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      4 passed, 4 total  
Snapshots:  0 total  
Time:       2.178s  
Ran all test suites.  
Done in 3.54s.  
  
# flavio @ Flavios-MacBook-Pro in ~/dev/jest [17:51:25]  
$
```

Matchers

In the previous article I used `toBe()` as the only matcher:

```
test("Adding 1 + 1 equals 2", () => {  
  expect(sum(1, 1)).toBe(2)  
})
```

A matcher is a method that lets you test values.

Most commonly used matchers, comparing the value of the result of `expect()` with the value passed in as argument, are:

- `toBe` compares strict equality, using `===`
- `toEqual` compares the values of two variables. If it's an object or array, checks equality of all the properties or elements
- `toBeNull` is true when passing a null value
- `toBeDefined` is true when passing a defined value (opposite as above)
- `toBeUndefined` is true when passing an undefined value
- `toBeCloseTo` is used to compare floating values, avoid rounding errors
- `toBeTruthy` true if the value is considered true (like an if does)
- `toBeFalsy` true if the value is considered false (like an if does)
- `toBeGreaterThan` true if the result of `expect()` is higher than the argument
- `toBeGreaterThanOrEqual` true if the result of `expect()` is equal to the argument, or higher than the argument
- `toBeLessThan` true if the result of `expect()` is lower than the argument
- `toBeLessThanOrEqual` true if the result of `expect()` is equal to the argument, or lower than the argument
- `toMatch` is used to compare strings with regular expression pattern matching
- `toContain` is used in arrays, true if the expected array contains the argument in its elements set
- `toHaveLength(number)`: checks the length of an array
- `toHaveProperty(key, value)`: checks if an object has a property, and optionally checks its value
- `toThrow` checks if a function you pass throws an exception (in general) or a specific exception
- `toBeInstanceOf()`: checks if an object is an instance of a class

<https://flaviocopes.com/jest/>

<https://flaviocopes.com/jest/>

All those matchers can be negated using `.not.` inside the statement, for example:

```
test("Adding 1 + 1 does not equal 3", () => {  
  expect(sum(1, 1)).not.toBe(3)  
})
```

For use with promises, you can use `.resolves` and `.rejects`:

```
expect(Promise.resolve('lemon')).resolves.toBe('lemon')  
expect(Promise.reject(new Error('octopus'))).rejects.toThrow('octopus')
```

Setup

Before running your tests you will want to perform some initialization.

To do something once before all the tests run, use the `beforeAll()` function:

```
beforeAll(() => {  
  //do something  
})
```

To perform something before each test runs, use `beforeEach()`:

```
beforeEach(() => {  
  //do something  
})
```

Teardown

Just as you could do with the setup, you can perform something after each test runs:

```
afterEach(() => {  
  //do something  
})
```

and after all tests end:

```
afterAll(() => {  
  //do something  
})
```

Group tests using describe()

You can create groups of tests, in a single file, that isolate the setup and teardown functions:

```
describe('first set', () => {  
  beforeEach(() => {  
    //do something  
  })  
  afterAll(() => {  
    //do something  
  })  
  test(/*...*/)  
  test(/*...*/)  
})
```

```
describe('second set', () => {  
  beforeEach(() => {  
    //do something  
  })  
  beforeAll(() => {  
    //do something  
  })  
  test(/*...*/)  
  test(/*...*/)  
})
```


Promises

With functions that return promises, we simply return a promise from the test:

```
//uppercase.js
const uppercase = (str) => {
  return new Promise((resolve, reject) => {
    if (!str) {
      reject('Empty string')
      return
    }
    resolve(str.toUpperCase())
  })
}
module.exports = uppercase

//uppercase.test.js
const uppercase = require('./uppercase')
test(`uppercase 'test' to equal 'TEST`, () => {
  return uppercase('test').then(str => {
    expect(str).toBe('TEST')
  })
})
```


<https://flaviocopes.com/jest/>



The image shows a code editor with two tabs. The top tab, 'uppercase.js', contains a function 'uppercase' that takes a string 'str' and returns a Promise. The Promise resolves to 'str.toUpperCase()' if 'str' is non-empty, and rejects with the message 'Empty string' if 'str' is empty. The bottom tab, 'uppercase.test.js', contains a Jest test. It imports the 'uppercase' function and uses 'otest' (likely a typo for 'it') to test that 'uppercase('test')' resolves to 'TEST'. The test uses 'expect(str).toBe('TEST')' to verify the result.

```
uppercase.js simple-jest-test-example x
1  const uppercase = (str) => {
2    return new Promise((resolve, reject) => {
3      if (!str) {
4        reject('Empty string')
5        return
6      }
7      resolve(str.toUpperCase())
8    })
9  }
10
11  module.exports = uppercase
12

uppercase.test.js simple-jest-test-example x
1  const uppercase = require('./uppercase')
2
3  otest(`uppercase 'test' to equal 'TEST'`, () => {
4    return uppercase('test').then(str => {
5      expect(str).toBe('TEST')
6    })
7  })

x 0 a 0 Ln 1, Col 1 Spaces: 2 UTF-8 LF Javascript (Babel) ☺ 🔔
```

<https://flaviocopes.com/jest/>

Promises that are rejected can be tested using `.catch()`:


```
//uppercase.js
const uppercase = (str) => {
  return new Promise((resolve, reject) => {
    if (!str) {
      reject('Empty string')
      return
    }
    resolve(str.toUpperCase())
  })
}

module.exports = uppercase

//uppercase.test.js
const uppercase = require('./uppercase')

test(`uppercase 'test' to equal 'TEST`, () => {
  return uppercase('').catch(e => {
    expect(e).toMatch('Empty string')
  })
})
```

<https://flaviocopes.com/jest/>



```
uppercase.js — simple-jest-test-example
1  const uppercase = (str) => {
2    return new Promise((resolve, reject) => {
3      if (!str) {
4        reject('Empty string')
5        return
6      }
7      resolve(str.toUpperCase())
8    })
9  }
10
11  module.exports = uppercase
12

uppercase.test.js simple-jest-test-example
1  const uppercase = require('./uppercase')
2
3  test(`uppercase 'test' to equal 'TEST'`, () => {
4    return uppercase('').catch(e => {
5      expect(e).toMatch('Empty string')
6    })
7  })

0 0  Ln 10, Col 1  Spaces: 2  UTF-8  LF  Javascript (Babel)
```

Mocking

In testing, mocking allows you to test functionality that depends on:

- Database
- Network requests
- access to Files
- any External system

so that:

1. your tests run faster, giving a quick turnaround time during development
2. your tests are independent of network conditions, the state of the database
3. your tests do not pollute any data storage because they do not touch the database
4. any change done in a test does not change the state for subsequent tests, and re-running the test suite should start from a known and reproducible starting point
5. you don't have to worry about rate limiting on API calls and network requests

Mocking is useful when you want to avoid side effects (e.g. writing to a database) or you want to skip slow portions of code (like network access), and also avoids implications with running your tests multiple times (e.g. imagine a function that sends an email or calls a rate-limited API).

Even more important, if you are writing a Unit Test, you should test the functionality of a function in isolation, not with all its baggage of things it touches.

Using mocks, you can inspect if a module function has been called and which parameters were used, with:

- `expect().toHaveBeenCalled()`: check if a spied function has been called
- `expect().toHaveBeenCalledTimes()`: count how many times a spied function has been called
- `expect().toHaveBeenCalledWith()`: check if the function has been called with a specific set of parameters
- `expect().toHaveBeenLastCalledWith()`: check the parameters of the last time the function has been invoked

<https://flaviocopes.com/jest/>

Spy packages without affecting the functions code

When you import a package, you can tell Jest to “spy” on the execution of a particular function, using `spyOn()`, without affecting how that method works.

Example:

```
const mathjs = require('mathjs')

test(`The mathjs log function`, () => {
  const spy = jest.spyOn(mathjs, 'log')
  const result = mathjs.log(10000, 10)

  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

Mock an entire package

Jest provides a convenient way to mock an entire package. Create a `__mocks__` folder in the project root, and in this folder create one JavaScript file for each of your packages.

Say you import `mathjs`. Create a `__mocks__/mathjs.js` file in your project root, and add this content:

```
module.exports = {
  log: jest.fn(() => 'test')
}
```

This will mock the `log()` function of the package. Add as many functions as you want to mock:

```
const mathjs = require('mathjs')

test(`The mathjs log function`, () => {
  const result = mathjs.log(10000, 10)
  expect(result).toBe('test')
  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

Mock a single function

More simply, you can mock a single function using `jest.fn()`:

<https://flaviocopes.com/jest/>

```
const mathjs = require('mathjs')

mathjs.log = jest.fn(() => 'test')
test(`The mathjs log function`, () => {
  const result = mathjs.log(10000, 10)
  expect(result).toBe('test')
  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

You can also use `jest.fn().mockReturnValue('test')` to create a simple mock that does nothing except returning a value.

Pre-built mocks

You can find pre-made mocks for popular libraries. For example this package <https://github.com/jefflau/jest-fetch-mock> allows you to mock `fetch()` calls, and provide sample return values without interacting with the actual server in your tests.