



Unit Testing

or how to write ~~good~~ not that bad unit tests

Frontend Junior Program - 2022

CONFIDENTIAL | © 2022 EPAM Systems, Inc.

Your application is a special snowflake



Expert

Excuses for Not Writing Unit Tests

ORLY?

@ThePracticalDev

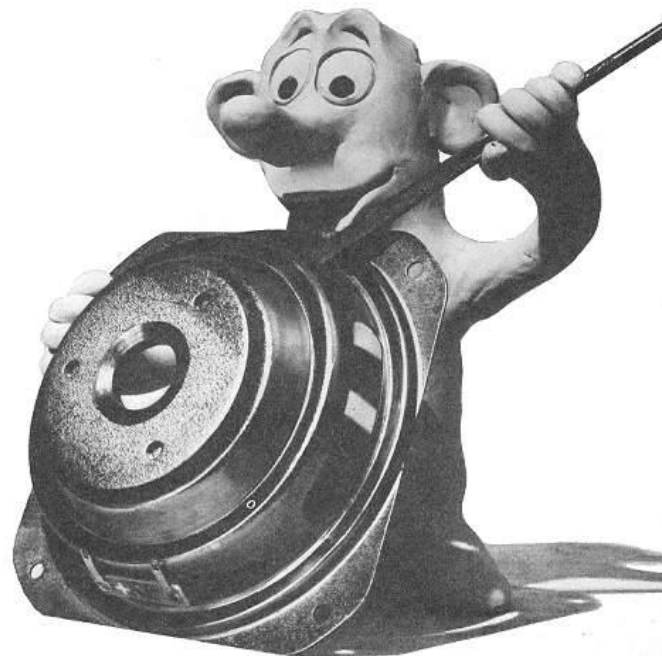
How to create a bad ~~unit test~~ speaker?

Recently (1985), a truly excellent article was published in the HIFI Magazine about [how to make bad loudspeakers \(part I, part II](#) – Hungarian, only). You know, back then it was very hard to get [loudspeakers](#) in Hungary, and basically, it is a simple device: a box, some speaker units and a little bit of electronics, [nothing special really](#) – so, many hobbyist had a go.

Still, creating good loudspeakers is an extremely complex task. Unfortunately, the same is true for unit tests.

[Writing good unit tests is very challenging](#). So, the best we can achieve here is to show you: how the avoid the catastrophe.

Even that will be hard, still, it is worth to try, we promise.



it is really not easy to write bad unit tests – one need years of experience, professional excellence and dedicated, hard work to achieve that




There is a strong **belief**, that writing unit tests (especially in some ways, such as TDD), could be the ultimate weapon against every problem in software development: there will be **less bugs**, makes the **refactoring a bliss** and **leads to good architecture**.

But what is a unit test?

A **unit test** is simply a **code**, that **evaluates** the **result of** another specific **code**

System Under Test,	→	> <i>/* SUT */</i>
this is our unit now	→	<i>function add(a, b) { return a + b }</i>
		<i>/* Unit tests */</i>
test suite {	test case →	<i>console.log(add(1, 2) === 3);</i>
	test case →	<i>console.log(add(-1, 2) === 0);</i>
	test results {	<i>true</i>
		<i>false</i>

 a bug in the unit test

Unit Test



uni means “one”

A unit test is concerned about one part only.

A unit can be anything: an application, a component, an object, but usually
unit is a function or a method

a unit test validates the output against values

It compares the result of the unit to an expected value. A result can be the *return value* or a *state space* (i.e., changes of the state: global variables, object properties, database records) or both at the same time.

A unit test must be...

reproducible

Through time, locations and other external conditions.

consistent

Across host environments. While the FE code runs in a browser, the unit tests are invoked from node.js (using a headless browser, or JSDOM).

Test results should not vary depending on the environment.

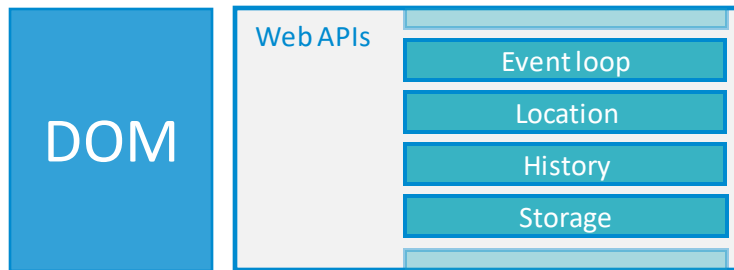
fast

Even a simple application could utilize thousands of test cases. As all tests should run before a commit, or in CI/CD, the running time is critical.

Unit testing in the Front-end world is a *bit more* complex task

A FE code is not just core JavaScript code

1. We use DOM, we use browser API calls, therefore the **host environment must be presented** or simulated somehow. To do this, test runners invoke real browsers or use JSDOM to provide a lookalike host.
2. Also, many times the **result** of a component **is a change in a DOM**. While there are different techniques to address this (visual diffs and snapshots), the fundamental complexity will be there.



FE \neq core JavaScript, even a simple FE unit test requires a vast background

In practice, for running front-end unit tests we use a testing framework



Jest (from Facebook) for React applications



JSDOM emulates the browser for Jest

typically used with React



Jasmine is a battle-hardened testing framework



Karma (from Google) provides the host (via browsers) for unit tests

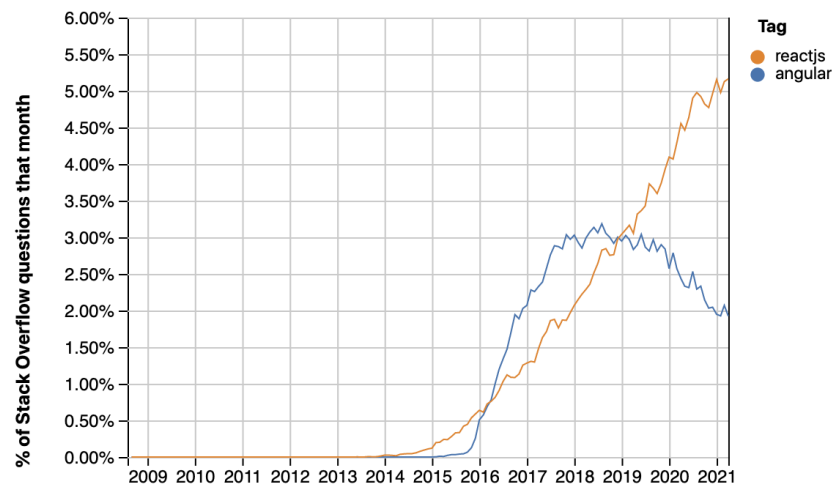
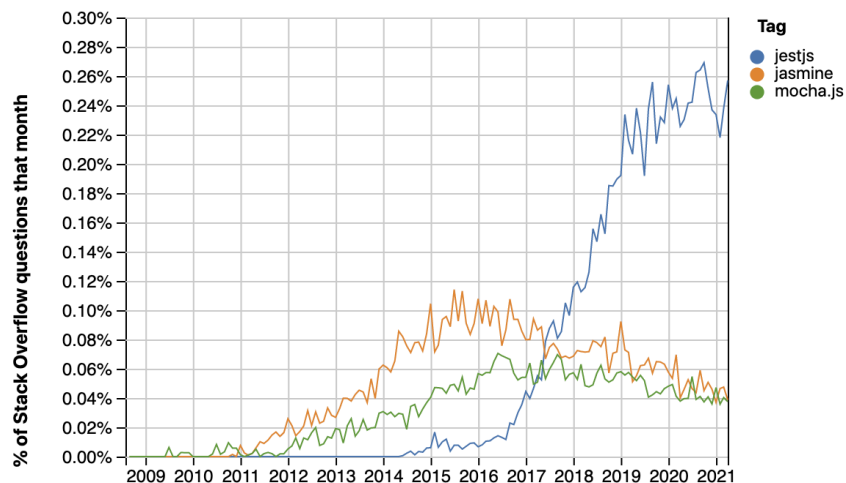
a common Angular setup

Mocha could be a flexible option

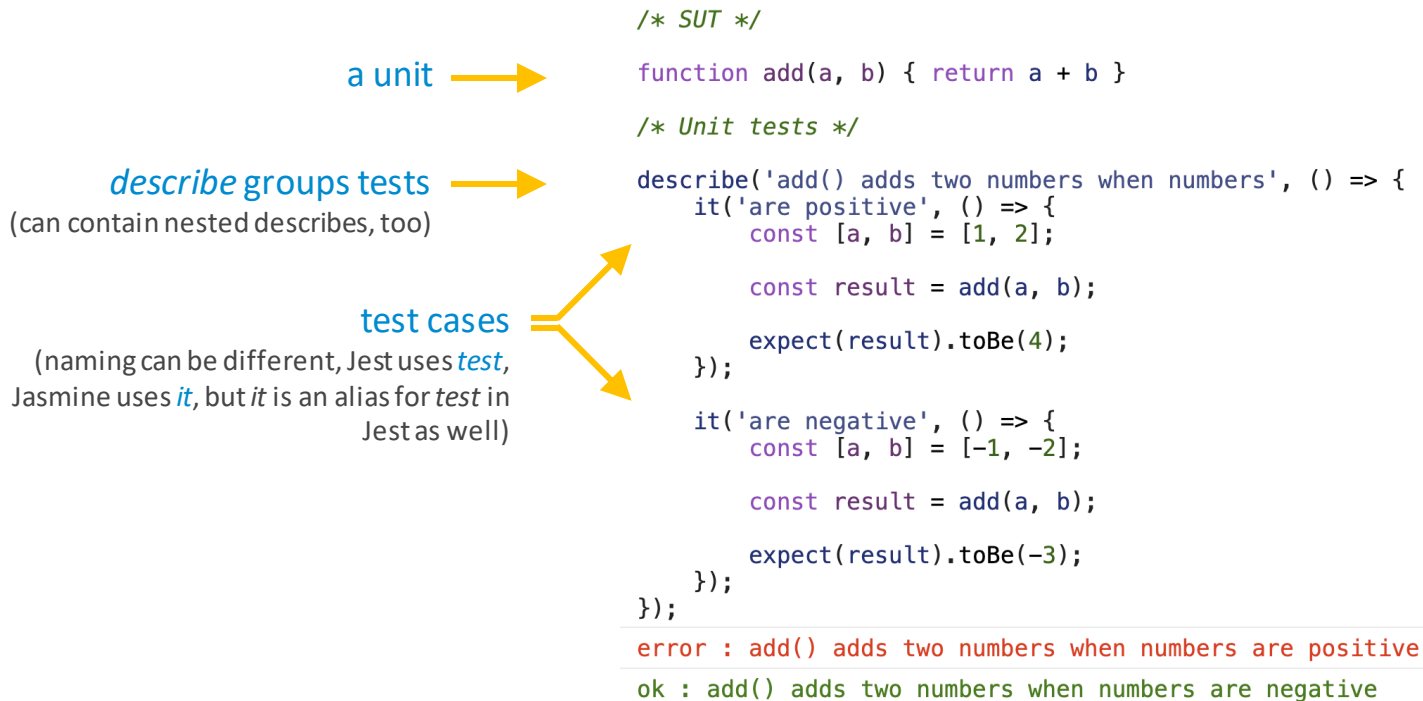


While **testing frameworks** does have different configuration, from unit testing perspective they **are very similar**

If you master Jasmine, you can use Jest out of the box.



Basics



Demystifying a Testing Framework

At first sight, a **testing framework** does magic

For your convenience, we developed a super-simple testing framework (*of course*), for demonstration purposes. We will analyze it, then we'll have a deep dive in Jest.

that's all
it works, really!
moreover, it is compatible
with Jest (`describe`, `it`)

```
/* Testing Framework */  
  
function describe(name, fn) {  
  __suite = { name, tests: [] }  
  
  // registering tests  
  fn();  
  
  // running tests  
  __suite.tests.forEach(test => {  
    __testName = `: ${__suite.name} ${test.name}`;  
    test.fn();  
  });  
}  
  
function it(name, fn) {  
  __suite.tests.push({ name, fn });  
}  
  
/* Assertion library */  
  
function expect(value) {  
  return {  
    toBe: function(expected) {  
      console.log(...(value === expected  
        ? [`%cok ${__testName}`, 'color: green']  
        : [`%cerrror ${__testName}`, 'color: red']  
      ));  
    }  
  }  
}
```

this argument setup
should not surprise you at this
point. We use a **ternary operator**
and **spread syntax** here.

Admittedly, Jest and Jasmine could be a bit more complex

But the under the hood they are similar to this: the test cases provides only a setup, running is a different story.

it does not run a test, only registers it
as a consequence, consecutive tests (in test
code), can run parallel, and the
calling order is not fixed!



an assertion simply compares values
and somehow logs the results



```
/* Testing Framework */

function describe(name, fn) {
  __suite = { name, tests: [] }

  // registering tests
  fn();

  // running tests
  __suite.tests.forEach(test => {
    __testName = `: ${__suite.name} ${test.name}`;
    test.fn();
  });
}

function it(name, fn) {
  __suite.tests.push({ name, fn });
}

/* Assertion library */

function expect(value) {
  return {
    toBe: function(expected) {
      console.log(...(value === expected
        ? [`%cok ${__testName}`, 'color: green']
        : [`%cerrror ${__testName}`, 'color: red']
      ));
    }
  }
}
```

Working with Jest

> mkdir indiana_jones
> cd indiana_jones
> npm init -y
> npm i jest
> mkdir src

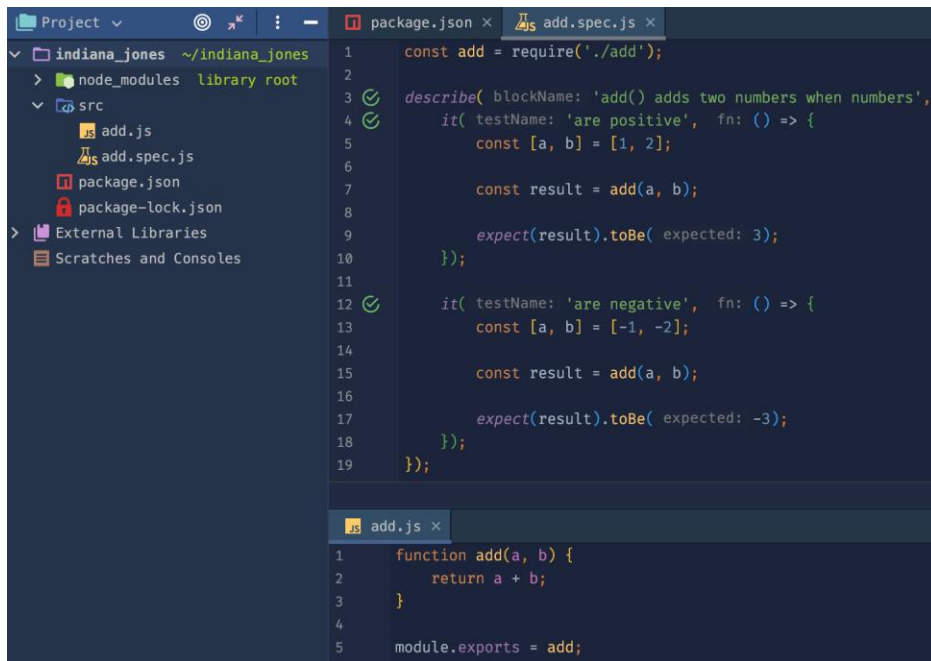
add "test": "jest" to the
scripts part of package.json

create add.js and add.spec.js
in the src folder

add content to the JS files

run the test suite:
> npm run test

the files we prepared for our
testing framework are already
Jest compatible :)



The screenshot shows a VS Code editor with two files open: package.json and add.spec.js. The left sidebar shows the project structure with a folder named 'indiana_jones' containing 'node_modules', 'src', 'package.json', and 'package-lock.json'. The 'src' folder contains 'add.js' and 'add.spec.js'. The main editor shows the content of 'add.spec.js' and 'add.js'.

```
package.json
1  "test": "jest",
2
3  "scripts": {
4    "test": "jest"
5  }
6
7  "dependencies": {
8    "jest": "^27.0.6"
9  }
10
11 "devDependencies": {
12   "jest": "^27.0.6"
13 }
14
15 "name": "indiana_jones",
16 "version": "1.0.0",
17 "description": "",
18 "main": "index.js",
19 "author": "EPAM",
20 "license": "ISC"
```

```
add.spec.js
1  const add = require('./add');
2
3  describe('add() adds two numbers when numbers', () => {
4    it('are positive', () => {
5      const [a, b] = [1, 2];
6
7      const result = add(a, b);
8
9      expect(result).toBe(3);
10    });
11
12    it('are negative', () => {
13      const [a, b] = [-1, -2];
14
15      const result = add(a, b);
16
17      expect(result).toBe(-3);
18    });
19  });
```

```
add.js
1  function add(a, b) {
2    return a + b;
3  }
4
5  module.exports = add;
```

we are in node.js, the **export/import**
from JS Modules are not presented

Add JS Modules support

Jest recognizes babeljs, and can use it to transpile the JS files before running the tests

We just need to [install babel](#) and add a minimal [configuration](#).

install babel

> `npm i babel-jest @babel/core @babel/preset-env`

create `babel.config.js`

add content

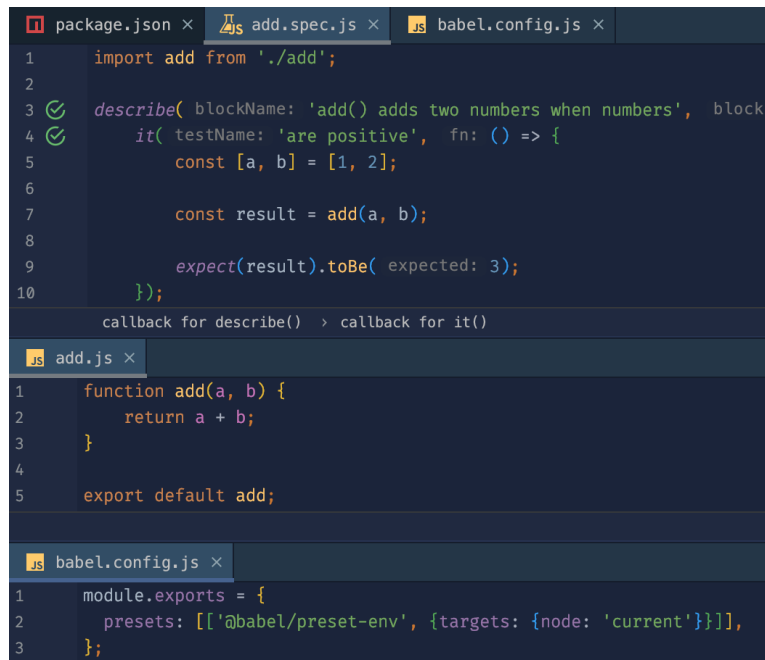
change the `require` and `module.exports` in JS files

run the test suite:

> `npm run test`

or

> `npm test`



```
package.json x  add.spec.js x  babel.config.js x
1  import add from './add';
2
3  ✓ describe( blockName: 'add() adds two numbers when numbers', block
4  ✓ it( testName: 'are positive', fn: () => {
5      const [a, b] = [1, 2];
6
7      const result = add(a, b);
8
9      expect(result).toBe( expected: 3);
10  });
    callback for describe() > callback for it()
JS  add.js x
1  function add(a, b) {
2      return a + b;
3  }
4
5  export default add;
JS  babel.config.js x
1  module.exports = {
2      presets: [['@babel/preset-env', {targets: {node: 'current'}}]],
3  };
```

babel.config.js is still processed by node.js directly, therefore it requires the [require](#)

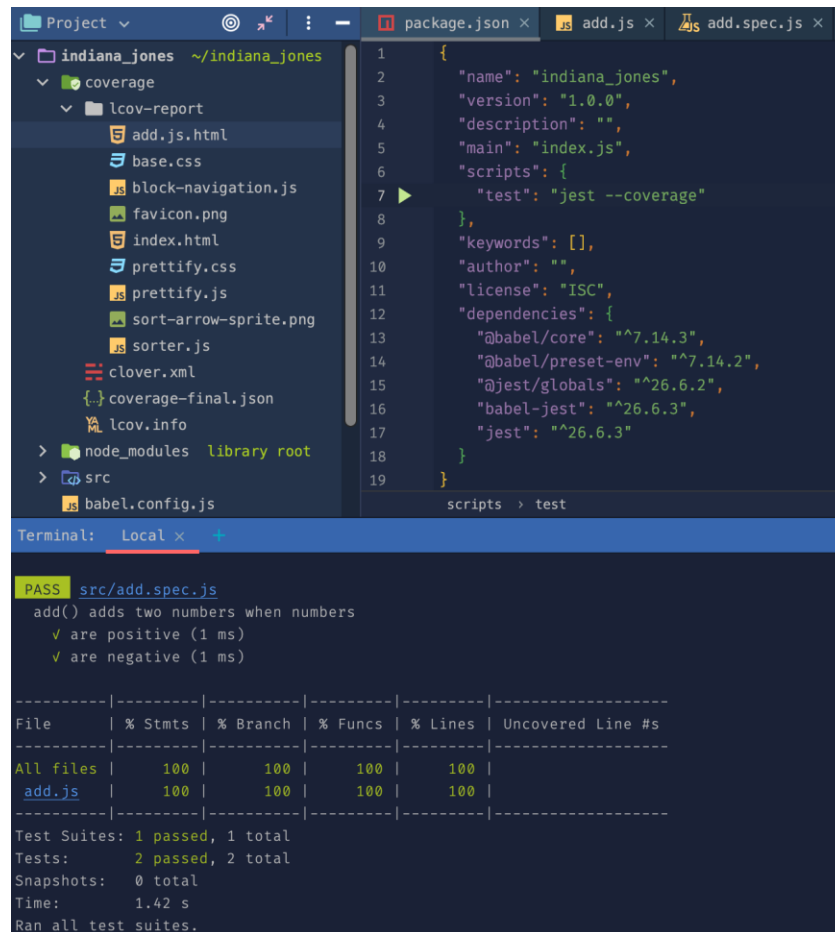
much better, we have [export and import](#) now

Coverage report

By adding `--coverage` as a CLI parameter to the `jest` command, Jest will create a coverage report for us

generated coverage report
as a web page

coverage report in the terminal
this looks useful now, however, after
having several hundreds of source files, it
could be a bit of lot



The screenshot shows a code editor with a project structure on the left and a `package.json` file on the right. The project structure includes a `coverage` directory with an `lcov-report` subdirectory, which contains an `add.js.html` file. The `package.json` file has a `scripts` section with a `test` script that runs `jest --coverage`. Below the editor, a terminal window shows the output of the `jest` command, including a `PASS` message and a coverage report table.

```
package.json
{
  "name": "indiana_jones",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest --coverage"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@babel/core": "^7.14.3",
    "@babel/preset-env": "^7.14.2",
    "@jest/globals": "^26.6.2",
    "babel-jest": "^26.6.3",
    "jest": "^26.6.3"
  }
}
```

```
Terminal: Local x
PASS src/add.spec.js
  add() adds two numbers when numbers
    ✓ are positive (1 ms)
    ✓ are negative (1 ms)

-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files | 100     | 100      | 100      | 100      |
add.js   | 100     | 100      | 100      | 100      |
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        1.42 s
Ran all test suites.
```

But what is **coverage**?

With coverage, we can have a detailed view on which **statements** / **branches** / **functions** and **lines** had been covered by our tests.

It is very useful to ~~comply~~ **project-specific coverage requirements** and to understand what was tested specifically.

Let's see a bit more complex example!

Jest uses [istanbul](#) for generating the coverage report under the hood



← → ↺ ⓘ localhost:63342/indiana_jones/coverage/lcov-report/add.js.html

All files add.js

100% Statements 1/1 100% Branches 0/0 100% Functions 1/1 100% Lines 1/1

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 function add(a, b) {  
2   2x   return a + b;  
3 }  
4  
5 export default add;  
6
```


All files math.js

50% Statements 3/6

50% Branches 2/4

50% Functions 2/4

50% Lines 3/6

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

this function is covered
while we did not run it directly, it was
called by the *add* function

this function is not covered
we added only numbers

this branch is not covered
because this line has never run

this function is not covered
we just tested the *add* function

```
1  function addNumbers(a, b) {  
2    return a + b;  
3  }  
4  
5  function addStrings(a, b) {  
6    return a + b;  
7  }  
8  
9  function add(a, b) {  
10   if (typeof a === 'string' && typeof b === 'string') {  
11     return addStrings(a, b);  
12   } else {  
13     return addNumbers(a, b);  
14   }  
15 }  
16  
17 function subtract(a, b) {  
18   return a + b;  
19 }  
20  
21 export { add, subtract };  
22
```

this condition was never evaluated
because *typeof a* was always 'number'

```
import { add } from './math';

describe( blockName: 'add() adds two', blockFn: () => {
  describe( blockName: 'numbers when numbers', blockFn: () => {
    it( testName: 'are positive', fn: () => {
      const [a, b] = [1, 2];

      const result = add(a, b);

      expect(result).toBe( expected: 3);
    });

    it( testName: 'are negative', fn: () => {
      const [a, b] = [-1, -2];

      const result = add(a, b);

      expect(result).toBe( expected: -3);
    });
  });

  it( testName: 'strings', fn: () => {
    const [a, b] = ['Indiana', 'Jones'];

    const result = add(a, b);

    expect(result).toBe( expected: 'Indiana Jones');
  });
});
```

All files math.js

83.33% Statements 5/6 100% Branches 4/4 75% Functions 3/4 83.33% Lines 5/6

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 function addNumbers(a, b) {
2   2x   return a + b;
3 }
4
5 function addStrings(a, b) {
6   1x   return `${a} ${b}`;
7 }
8
9 function add(a, b) {
10  3x   if (typeof a === 'string' && typeof b === 'string') {
11  1x     return addStrings(a, b);
12     } else {
13  2x     return addNumbers(a, b);
14     }
15   }
16
17 function subtract(a, b) {
18   return a + b;
19 }
20
21 export { add, subtract };
22
```

by adding a new test case, we can have a bit more coverage



please do not touch private things, not even with unit tests!

The fact, that the **private functions are already covered** does have significant importance. It means that we don't need to export private functions to test. These functions can be tested indirectly.

let's see what happens, if we do want to test them as well!

Always test the public API of your module

Private functions and methods are the internals of the module, their inner knowledge should not be exposed. The implementation details could change with the same API – if the tests depend on the internals, they are brittle: any refactoring* could break the tests.

****refactoring** means no changes in the business functions. If there is a change, then it is a **rework**, which must be covered by a **change request** or a **user story**.*

How to write ~~good~~ not that bad unit tests – rule 1.

Still, let's see what happens, if we do want to test them!

We exported `addNumbers` and added a test for it as well. What happened? Right now, everything looks fine!

we run this 3 times:
2 times in the `add` test case, and
once in the `addNumbers` test case

```
1 function addNumbers(a, b) {  
2   3x   return a + b;  
3 }  
4  
5 function addStrings(a, b) {  
6   1x   return `${a} ${b}`;  
7 }  
8  
9 function add(a, b) {  
10  3x   if (typeof a === 'string' && typeof b === 'string') {  
11  1x     return addStrings(a, b);  
12     } else {  
13  2x     return addNumbers(a, b);  
14     }  
15 }
```

we called `addNumbers`
here only 2 times



All tests are green, we can push the code and let's call it a day!

however, let's introduce a `small bug`!

Guess what? We have 3 failed tests now!

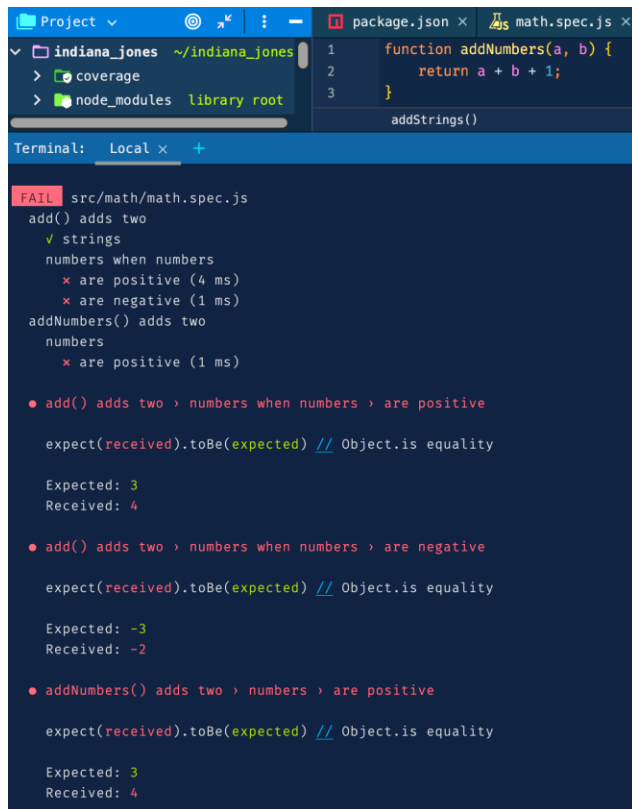
1 bug => 3 failed tests. The root cause of this is that we test the same code in different places (*redundant coverage*).

*In engineering, this is called an **overdetermined system**, and it always can be considered as an issue, because these systems are never stable.*

Think about a chair: a chair with 3 legs is always stable, because 3 points determine a plane. The fourth can be out of the plane. In real world is always out of the plane. A chair with 4 legs is always instable (some flexibility in the legs will save us, thankfully)

So, how can we avoid that?

1. if this was a private function then **don't test it directly!**
2. if this is a library function, then we need to prevent by calling the *addNumbers*. But how?



```
Project v [icons] [menu] [search] [run] [debug] [terminal] [package.json] [math.spec.js] x
indiana_jones ~/indiana_jones
├─ coverage
└─ node_modules library root

Terminal: Local x +

FAIL src/math/math.spec.js
add() adds two
  ✓ strings
  numbers when numbers
    ✗ are positive (4 ms)
    ✗ are negative (1 ms)
addNumbers() adds two
  numbers
    ✗ are positive (1 ms)

  • add() adds two > numbers when numbers > are positive

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 4

  • add() adds two > numbers when numbers > are negative

    expect(received).toBe(expected) // Object.is equality

    Expected: -3
    Received: -2

  • addNumbers() adds two > numbers > are positive

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 4
```



Indy is just mocking an Arab here, but a mock can do wonders!

How can we prevent from calling the `addNumbers`, while having a return value?

We can use a [test double](#), instead. A test double is a [replacement of the original code](#), [that we can have control over](#), and we can analyze its activities. There are different types of test doubles ([stub](#), [mock](#), [spy](#), [fake](#), [dummy](#)), yet at this point we can simply call them [mocks](#) and the process itself is [mocking](#).

let's mock the `addNumbers` then!

But before mocking out the *addNumbers* function, we have to extract that. The reason behind that we should...

Never mock parts of the SUT*

If you feel that there is a temptation for that, then it could be a clear sign of that part is an independent entity, therefore it should be extracted.

**In other words, if you write unit tests for a module, then its functions should not be mocked in that test suite.*

How to write ~~good~~ not that bad unit tests – rule II.

with `xdescribe`, we can temporarily disable the test

```
math.spec.js x
1 import { add } from './math';
2 import * as mathPrimitives from './math.primitives';
3
4 >> describe( blockName: 'add() adds two', blockFn: () => {
5 >>   describe( blockName: 'numbers when numbers', blockFn: () => {
6 >>     it( testName: 'are positive', fn: () => {
7 >>       const [a, b] = [1, 2];
8 >>       mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);
9 >>
10 >>       const result = add(a, b);
11 >>
12 >>       expect(result).toBe( expected: 3);
13 >>     });
14 >>   });
15 >>
16 >>   it( testName: 'strings', fn: () => {
17 >>     const [a, b] = ['Indiana', 'Jones'];
18 >>
19 >>     const result = add(a, b);
20 >>
21 >>     expect(result).toBe( expected: 'Indiana Jones');
22 >>   });
23 >> }
```

```
math.primitives.spec.js
1 import { addNumbers } from './math.primitives';
2
3 >> xdescribe( blockName: 'addNumbers() adds two', blockFn: () => {
4 >>   describe( blockName: 'numbers when they', blockFn: () => {
5 >>     it( testName: 'are positive', fn: () => {
6 >>       const [a, b] = [1, 2];
7 >>
8 >>       const result = addNumbers(a, b);
9 >>
10 >>       expect(result).toBe( expected: 3);
11 >>     });
12 >>   });
13 >> };
```

we have **extracted** the primitive functions

now we have to **import** the extracted functions

```
math.js x
1 import { addNumbers, addStrings } from './math.primitives';
2
3 function add(a, b) {
4   if (typeof a === 'string' && typeof b === 'string') {
5     return addStrings(a, b);
6   } else {
7     return addNumbers(a, b);
8   }
9 }
10
11 function subtract(a, b) {
12   return a + b;
13 }
14
15 export { add, subtract };
```

WebStorm is smart, it is **not used**

```
math.primitives.js x
1 function addNumbers(a, b) {
2   return a + b + 1;
3 }
4
5 function addStrings(a, b) {
6   return `${a} ${b}`;
7 }
8
9 export { addNumbers, addStrings };
```

we still have the bug here

here you are the big picture, but let's see [mock.spec.js](#) for the details!

If we run the tests now, we'll have only one fail, in the *math.primitives.spec.js*. Nice!

we've changed the import, *mathPrimitives* acts as a namespace now

we simply *override* the *addNumbers* with *jest.fn()* which returns a *mock function*

also, we can define the return value with *mockReturnValue()*, so from now on the *add* function will use our mock function, instead of the original

```
math.spec.js x
1  import { add } from "../math";
2  import * as mathPrimitives from "../math.primitives";
3
4  describe( blockName: 'add() adds two', blockFn: () => {
5    describe( blockName: 'numbers when numbers', blockFn: () => {
6      it( testName: 'are positive', fn: () => {
7        const [a, b] = [1, 2];
8        mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);
9
10       const result = add(a, b);
11
12       expect(result).toBe( expected: 3);
13     });
14   });
15
16   it( testName: 'strings', fn: () => {
17     const [a, b] = ['Indiana', 'Jones'];
18
19     const result = add(a, b);
20
21     expect(result).toBe( expected: 'Indiana Jones');
22   });
23 });
```

If we mock out the dependencies of a unit, then we test only what is the real job (a single responsibility, optimally) of that unit – and nothing else.

If we don't mock, then we test our unit, plus we test the dependency, plus the dependencies of that dependency (because we do not have a control on that), plus we really don't know what we test – it could be anything: maybe the whole application, maybe the whole internet...

A test case should only test the unit's responsibility

If you want to test a banana, please don't test a gorilla holding the banana and the entire jungle with it at the same time.

How to write ~~good~~ not that bad unit tests – rule III.

You may have guessed now: we have just run into a **serious issue with** our new **mock**



The problem is: now **we are aware of the implementation** of our unit. We've just exposed the **internals** of our unit into the test.

Now we do know that *add* calls *addNumbers*.

We are aware of the internals. We can see inside. It is a...



White box

What is going on inside the *add*? - we don't know anything about the internals. Does that call another function? What is the implementation?



We don't know. We cannot see inside. It is a...

Black box

```
math.spec.js x
1 import { add } from "../math";
2 import * as mathPrimitives from "../math.primitives";
3
4 describe( blockName: 'add() adds two', blockFn: () => {
5   describe( blockName: 'numbers when numbers', blockFn: () => {
6     it( testName: 'are positive', fn: () => {
7       const [a, b] = [1, 2];
8       mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);
9
10      const result = add(a, b);
11
12      expect(result).toBe( expected: 3);
13    });
14  });
15
16   it( testName: 'strings', fn: () => {
17     const [a, b] = ['Indiana', 'Jones'];
18
19     const result = add(a, b);
20
21     expect(result).toBe( expected: 'Indiana Jones');
22   });
23 });
```

Don't expose the internals of the unit into the test

If you do so, the test will be tightly coupled with the implementation of the unit - any changes in the unit could break the test. The refactoring will be very hard now.

One of the major benefits of the unit testing is that we can [change the implementation details without breaking the tests](#) – we can [refactor](#) the code. Theoretically, if the unit tests covers the business requirements, then we can do anything in the unit, while the tests are a green, we are safe.

Exposing the implementation details into the test clearly defies that goal.

How to write ~~good~~ not that bad unit tests – rule IV.

White box

Exposes the internals
of the unit into the test.

—

refactoring could break the tests, even when
there is no new bugs;

+

a bug in any module could be separated and
can be detected identified easily;

structural test: branches, edge cases can be
identified and covered;

mocking many times is important and
unavoidable (API calls);

Black box

Hides the internals
of the unit from the test.

—

mocking is impossible;

a bug in a module could lead to failed tests in
several modules;

+

refactoring won't break the tests;

tests the external behavior: unit's public API;



The [balance is very important](#). As mocking is expensive (requires a lot of effort) it should be used when it makes sense or necessary. On the other hand, mocking does not make the refactoring impossible – it just requires a bit more attention and understanding. Also, the general testing approach and conventions do have impact on what, and when needed to mock.

Let's add a new test – now without mocking

But wait a minute, our new test fails!

What is going on?

It seems that the mocked return value is leaking into the new test. **The mock still in place!**

this should be ok:

$-1 + -2 === -3$



still, it fails...



this “3” is very suspicious: maybe it is the defined **mockReturnValue**?



```
4 >> describe( blockName: 'add() adds two', blockFn: () => {
5 >>   describe( blockName: 'numbers when numbers are', blockFn: () => {
6 >>     it( testName: 'positive', fn: () => {
7       const [a, b] = [1, 2];
8       mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);
9
10      const result = add(a, b);
11
12      expect(result).toBe( expected: 3);
13    });
14
15 >>     it( testName: 'negative', fn: () => {
16       const [a, b] = [-1, -2];
17
18       const result = add(a, b);
19
20       expect(result).toBe( expected: -3);
21     });
22   });
23 }
```

callback for describe()

Terminal: Local x +

FAIL src/math/math.spec.js

• add() adds two > numbers when numbers are > negative

expect(received).toBe(expected) // Object.is equality

Expected: -3

Received: 3

Mocks should be restored

When creating mocks with `jest.fn()` we have to restore it manually.

saving the original function →

restoring manually →

```
it( name: 'positive', fn: () => {  
  const [a, b] = [1, 2];  
  const addNumbersOriginal = mathPrimitives.addNumbers;  
  mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);  
  
  const result = add(a, b);  
  
  mathPrimitives.addNumbers = addNumbersOriginal;  
  expect(result).toBe( expected: 3);  
});
```

however...

Can you spot the difference?

One of these tests is fine, the other does have a catastrophic consequence!

```
it( name: 'positive', fn: () => {  
  const [a, b] = [1, 2];  
  const addNumbersOriginal = mathPrimitives.addNumbers;  
  mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);  
  
  const result = add(a, b);  
  
  mathPrimitives.addNumbers = addNumbersOriginal;  
  expect(result).toBe( expected: 3);  
});
```

```
it( testName: 'positive', fn: () => {  
  const [a, b] = [1, 2];  
  const addNumbersOriginal = mathPrimitives.addNumbers;  
  mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);  
  
  const result = add(a, b);  
  
  expect(result).toBe( expected: 3);  
  mathPrimitives.addNumbers = addNumbersOriginal;  
});
```

A failed *expect* call will stop the code processing

In this case, if the test fails, the mock won't be restored.



```
it( testName: 'positive', fn: () => {  
  const [a, b] = [1, 2];  
  const addNumbersOriginal = mathPrimitives.addNumbers;  
  mathPrimitives.addNumbers = jest.fn().mockReturnValue( value: 3);  
  
  const result = add(a, b);  
  
  expect(result).toBe( expected: 3);  
  mathPrimitives.addNumbers = addNumbersOriginal;  
});
```

— if the *expect* fails, this code **won't run**


let's see if we can do it in a more convenient and safe way!

A `spy` can act as mock

The main goal of a spy to have an assertion about a function (a method) call: whether [it has been called](#), [how many times](#), with [what parameters](#). The name “spy” is very descriptive here, we can [spy on a function](#)!

However, spies also useful for mocking functions:

`spyOn` returns a [mock function](#), similarly to `jest.fn()`



```
it( name: 'Infinity', fn: () => {  
  const [a, b] = [Infinity, Infinity];  
  const addNumbersSpy = jest.spyOn(mathPrimitives, method: 'addNumbers').mockReturnValue(Infinity);  
  
  const result = add(a, b);  
  
  addNumbersSpy.mockRestore();  
  expect(result).toBe(Infinity);  
});
```

restoring mocks in every test, however, is very cumbersome and error prone...

[beforeEach\(\)](#) and [afterEach\(\)](#)

can be useful when we have tasks (preparation or cleanup),
that we need to run after every tests.

```
beforeEach( fn: () => {  
    jest.restoreAllMocks();  
});  
  
it( name: 'Infinity', fn: () => {  
    const [a, b] = [Infinity, Infinity];  
    jest.spyOn(mathPrimitives, method: 'addNumbers').mockReturnValue(Infinity);  
  
    const result = add(a, b);  
  
    expect(result).toBe(Infinity);  
});
```

Q&A