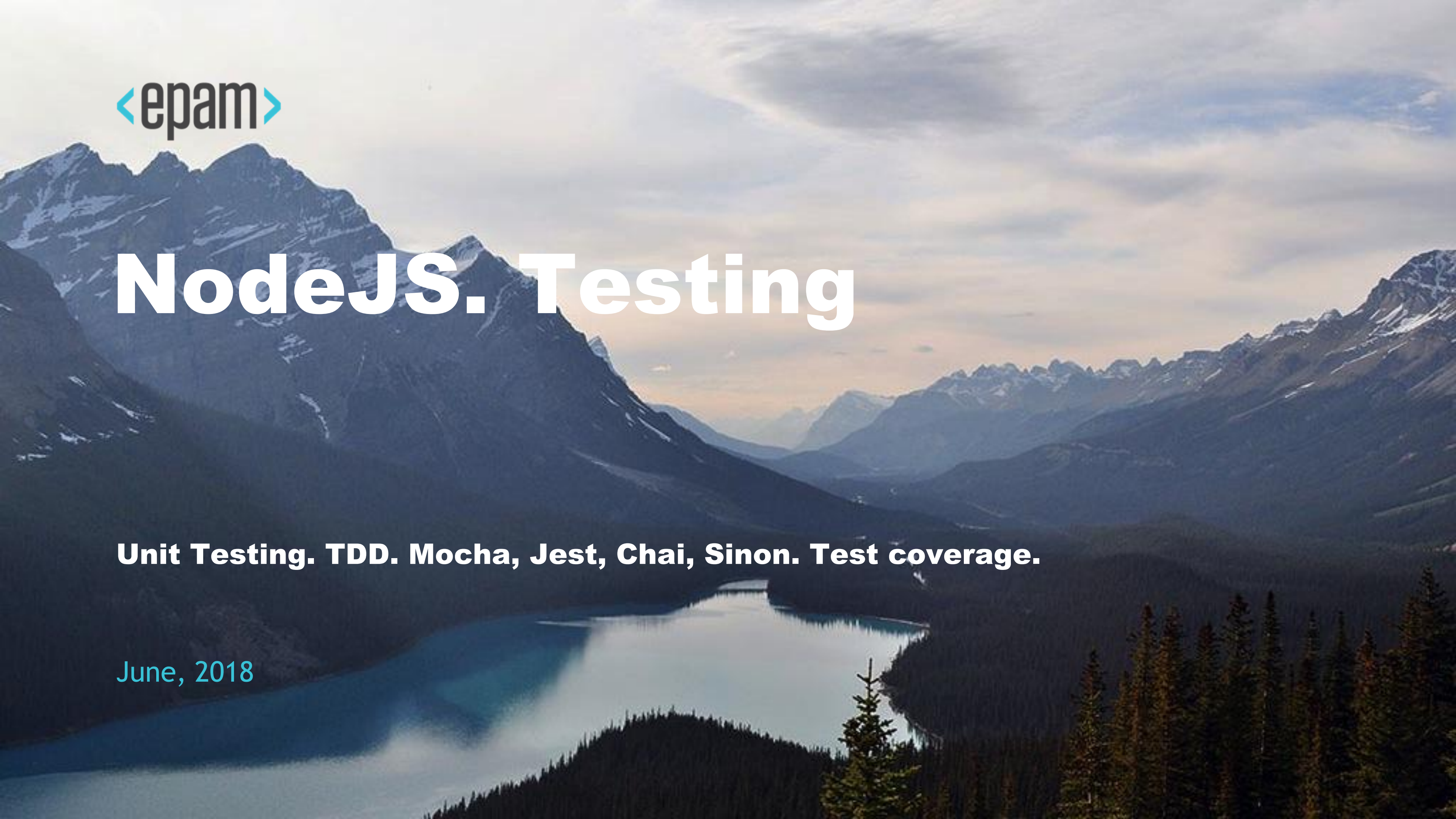


NodeJS. Testing

Unit Testing. TDD. Mocha, Jest, Chai, Sinon. Test coverage.

June, 2018

AGENDA

Unit Testing

Unit Testing - software testing method to test small, **independent** and **isolated** unit of code.

In ideal situation, a unit of code is smallest part of code which has one or several inputs and usually one output, but most of the time you need to deal with side effects while running your unit, and I'll need to use mocks and stubs for that.

The **stub** implements just enough behavior to allow the object under test to execute the test.

The **mock** is like a stub but the test will also verify that the object under test calls the mock as expected.

We will talk about it later….

Unit Testing

We write unit tests to see if a given unit works. And keep in mind, **you should write the test for the exposed methods**, not for internal part of code.

Each unit test has the following workflow:

- Setup test inputs and outputs
- Call the tested method
- Asserting results

In each test you can use as many asserting as you want, but you should test one **concern only**.

Unit Testing

We have such code:

```
function findGCD(a, b) {
  let x = Math.abs(a);
  let y = Math.abs(b);

  while (x && y) {
    if (x > y) {
      x = x % y;
    } else {
      y = y % x;
    }
  }

  return x + y;
}
```

And we’d like to test it.

Unit Testing

First of all, let's write down test cases. We have 2 inputs and 1 output. So, we can create a table.

#	a	b	result
1	30	15	15
2	7	49	7
3	72	16	8
4	13	5	1

Unit Testing

Simplest way to test our code, just to run it some times with different input.

```
console.log(findGCD(30, 15));
console.log(findGCD(7, 49));
console.log(findGCD(72, 16));
console.log(findGCD(13, 5));
```

By if we have a lot of tests, or we need to run some commands per test. It’ll become mess…

Let’s write a function, which will take our test cases and run tests for each of them.

Unit Testing

Testing function:

```
function testIt (cases, func) {
  for (let item of cases) {
    console.log('Testing', item.input, 'should return', item.output);
    const res = func.apply(null, item.input);

    if (res === item.output) {
      console.log('Passed');
    } else {
      console.log('Failed,', 'return', res);
    }

    console.log('---------');
  }
}
```

Test cases:

```
const CASES = [
  { input: [30, 15], output: 15 },
  { input: [7, 49], output: 7 },
  { input: [72, 16], output: 8 },
  { input: [13, 5], output: 1 },
];

testIt(CASES, findGCD);
```

Unit Testing

And now, if we need to add any test case, we just need to add new line to test cases array.

```
$ node index.js
Testing [ 30, 15 ] should return 15
Passed
---------
Testing [ 7, 49 ] should return 7
Passed
---------
Testing [ 72, 16 ] should return 8
Passed
---------
Testing [ 13, 5 ] should return 1
Passed
---------
```

Mocha framework

We can write our testing function every time, but better use exist testing framework with a lot of additional features.

Mocha is the one of them. It has such features:

- can be user on both NodeJS and Browser
- flexible and accurate reporting
- ability to test asynchronous methods
- and more …

Mocha framework

To add mocha to our project, we just need to install it as npm package. To run it, we need to add command “mocha” to npm scripts. Let’s prepare testing code:

```
describe('findGCD', () => {
  it('should return 15 for [30, 15]', () => {
    assert.equal(findGCD(30, 15), 15);
  });

  it('should return 7 for [7, 49]', () => {
    assert.equal(findGCD(7, 49), 7);
  });

  it('should return 8 for [72, 16]', () => {
    assert.equal(findGCD(72, 16), 8);
  });

  it('should return 1 for [13, 5]', () => {
    assert.equal(findGCD(13, 5), 1);
  });
});
```

Mocha framework

And as result, we can test our function again. As you can see, it’s more structured and beautiful as for me ☺

```
$ ./node_modules/.bin/mocha


  findGCD
    ✓ should return 15 for [30, 15]
    ✓ should return 7 for [7, 49]
    ✓ should return 8 for [72, 16]
    ✓ should return 1 for [13, 5]


  4 passing (6ms)
```

Mocha framework

P.S. You no need to write each test by yourself again, you can write it in the following way for identical tests with different data.

```
describe('findGCD', () => {
  [
    { input: [30, 15], output: 15 },
    { input: [7, 49], output: 7 },
    { input: [72, 16], output: 8 },
    { input: [13, 5], output: 1 },
  ].forEach((item) => {
    it('should return ' + item.output + ' for ' + item.input, () => {
      assert.equal(findGCD(30, 15), 15);
    });
  })
});
```

Mocha framework

If we have any failed test, mocha will show us all details about it.

```
$ ./node_modules/.bin/mocha ./test-loop.js


  findGCD
    ✓ should return 15 for 30,15
    1) should return 9 for 7,49
    ✓ should return 8 for 72,16
    ✓ should return 1 for 13,5


  3 passing (25ms)
  1 failing

  1) findGCD
       should return 9 for 7,49:

      AssertionError [ERR_ASSERTION]: 7 == 9
      + expected - actual

      -7
      +9

      at Context.it (test-loop.js:12:14)
```

Mocha framework

More complex errors…

```
$ ./node_modules/.bin/mocha


  ShopingCart
    constructor
      ✓ should call `get` method of DB once
      ✓ should call `get` method of DB twice
    canculate
      ✓ should calculate price of products
    canculate
      ✓ should call `put` method of DB once
      1) should call `put` method of DB once


  4 passing (15ms)
  1 failing

  1) ShopingCart
       canculate
         should call `put` method of DB once:

      AssertionError: { id: 3, price: 30, amount: 99 } deepEqual { id: 3, price: 30, amount: 87 }
      + expected - actual

       {
      -  "amount": 99
      +  "amount": 87
         "id": 3
         "price": 30
       }

      at Context.it (test.js:68:14)
```

Mocha framework

Mocha also has such helping tools and feature.

- Hooks (ability to run code before and after each test or before and after whole process)
- Timeouts
- Asynchronous and Synchronous tests
- Pending, exclusive inclusive tests

Chai and Sinon

Previous tasks were very simple and “unit” was using just inputs to produce output. But most of time, we use not pure functions, which get or write info somewhere else, or perform any side actions. In this case, we need to user **stubs** and **mocks**.

The next problem, we was comparing output of unit with expected result just by typing “===” or using build-in library assert, but in other cases. We may need to compare object, arrays, or just some property of result. For example we need to check, if unit return string with no care about content of this string, or we need to check, if unit return array that consist just of 3 elements.

For this purpose there are a lot of **assertion libraries**.

Chai and Sinon

Chai is an assertion library, which have some assertion styles for different development testing processes, and it an be used with any testing framework and even without it.

There are three assertion styles:

Assert – more classic notation similar to NodeJS Build-in package

```
assert.typeOf(foo, 'string')
assert.typeOf(foo, 'string', 'foo is a string')
```

Expect – notation which allows you chain together natural language assertions

```
expect(foo).to.be.a('string')
expect(foo).to.equal('bar')
```

Should – also chainable notation, but in this case each object are extended with method ‘should’

```
foo.should.be.a('string')
foo.should.equal('bar')
```

Chai and Sinon

Sinon is a standalone library with spies, stubs and mocks. It can be used with any testing framework.

Spies - a test spy is a function that records arguments, return value, the value of this and exception thrown (if any) for all its calls.

Stubs - functions (spies) with pre-programmed behavior, for example when you need function to return something.

Mocks - are fake methods (like spies) with pre-programmed behavior (like stubs) as well as pre-programmed expectations, for example you need function to return something, but only if arguments are correct.

Chai and Sinon

Let’s look into spies.

Creating:

```
const spy = sinon.spy(); // Creates as anonymous spies
const spy = sinon.spy(myFunc); // Spies on the provided function
const spy = sinon.spy(object, 'method'); // Spies method of object
```

Usage:

```
it("should call method once with each argument", () => {
  const object = { method: function () {} };
  const spy = sinon.spy(object, 'method');

  object.method(42);
  object.method(1)

  assert(spy.withArgs(42).calledOnce);
  assert(spy.withArgs(1).calledOnce);
})
```

Chai and Sinon

Let’s look into stubs.

Creating:

```
const stub = sinon.stub(); // Jest stub
const stub = sinon.stub(object, 'method'); // Stub method of object
const stub = sinon.stub(obj); // Stub all methods of object
```

Usage:

```
it("test should stub method", () => {
  const callback = sinon.stub();
  callback.withArgs(42).returns(1);
  callback.withArgs(1).throws('TypeError');

  callback(); // No value, no exception
  callback(42); // Return 1
  callback(1); // Throws TypeError
})
```

```
it("test should stub method", () => {
  const callback = sinon.stub();
  callback.onCall(0).returns(1);
  callback.onCall(1).returns(1);
  callback.returns(2);

  callback(); // Return 1
  callback(); // Return 2
  callback(); // All following calls return 3
})
```

Chai and Sinon

Let’s look into mocks.

```
it("test should call all subscribers when exception", () => {
  const myAPI = { method: function () {} };

  const spy = sinon.apy();
  const mock = sinon.mock(myAPI);
  mock.expects('method').once().throws();

  PubSub.subscribe('message', myAPI.method);
  PubSub.subscribe('message', spy);
  PubSub.publichSync('message', undefined);

  mock.verify();
  assert(spy.calledOnce);
})
```

Chai and Sinon

Now let’s practice. We have database:

```
const Database = {
  // Get stored object by ID
  get(id) {},

  // Update stored object by ID with new value
  put(id, value) {},
}
```

In real life it can contain async calls to API or database driver, but we no need it. Also we have a class, which named ShoopingCart with following API. And we need to write test for this class.

```
class ShopingCart {
  // Init cart and save DB instance
  constructor(db, ids) {}

  // Calculate price of all products
  calculate() {}

  // Buy products (side effect - decrement available amount of products in DB)
  buy() {}
}
```

Chai and Sinon

First of all, we need to stub Database methods

```
const PRODUCTS = {
  1: { id: 1, price: 10, amount: 100 },
  2: { id: 2, price: 20, amount: 100 },
  3: { id: 3, price: 30, amount: 100 },
};

const stubGet = sinon.stub(Database, 'get');
stubGet.withArgs(1).returns(PRODUCTS[1]);
stubGet.withArgs(2).returns(PRODUCTS[2]);
stubGet.withArgs(3).returns(PRODUCTS[3]);
const stubPut = sinon.stub(Database, 'put');
```

Also we need to reset history of this stubs each test, so lets use mocha hook for it:

```
afterEach(() => {
  stubGet.resetHistory();
  stubPut.resetHistory();
});
```

Chai and Sinon

Now let's write tests for constructor. ‘get’ method of Database should be called for each product in shopping cart.

```
describe('constructor', () => {
  it('should call `get` method of DB once', () => {
    const cart = new ShopingCart(Database, [1]);

    assert.equal(stubGet.callCount, 1);
    assert.equal(stubGet.getCall(0).args[0], 1);
  });

  it('should call `get` method of DB twice', () => {
    const cart = new ShopingCart(Database, [1, 2]);

    assert.equal(stubGet.callCount, 2);
    assert.equal(stubGet.getCall(0).args[0], 1);
    assert.equal(stubGet.getCall(1).args[0], 2);
  });
});
```

Chai and Sinon

As you see I’ve stubbed ‘get’ method and it should return real products. Let test if by calling ‘calculate’.

```
describe('canculate', () => {
  it('should calculate price of products', () => {
    const cart = new ShopingCart(Database, [1, 2, 3]);

    assert.equal(cart.calculate(), 60);
  });
});
```

Chai and Sinon

And at the end, we need to write tests for ’buy’ method of shopping cart, as a side effect, it should call ‘put’ method of Database for each product and provide correct product data.

```
describe('canculate', () => {
  it('should call `put` method of DB once', () => {
    const cart = new ShopingCart(Database, [1]);
    cart.buy()

    assert.equal(stubPut.callCount, 1);
    assert.equal(stubPut.getCall(0).args[0], 1);
    assert.deepEqual(stubPut.getCall(0).args[1], { id: 1, price: 10, amount: 99 });
  });

  it('should call `put` method of DB once', () => {
    const cart = new ShopingCart(Database, [1, 3]);
    cart.buy()

    assert.equal(stubPut.callCount, 2);
    assert.equal(stubPut.getCall(0).args[0], 1);
    assert.deepEqual(stubPut.getCall(0).args[1], { id: 1, price: 10, amount: 99 });
    assert.equal(stubPut.getCall(1).args[0], 3);
    assert.deepEqual(stubPut.getCall(1).args[1], { id: 3, price: 30, amount: 99 });
  });
});
```

Chai and Sinon

So, tests are ready and let’s run them…

```
$ ./node_modules/.bin/mocha


  ShopingCart
    constructor
      ✓ should call `get` method of DB once
      ✓ should call `get` method of DB twice
    canculate
      ✓ should calculate price of products
    canculate
      ✓ should call `put` method of DB once
      ✓ should call `put` method of DB once


  5 passing (13ms)
```

TDD and BDD techniques

Always you have a choice, write tests before the code or after. In common case it’s better to write them before. If you do it, you can be sure you have big test coverage and almost all breaking changes will be detected at testing stage and not at production stage ☺

But even if you want to write tests before code, you have two ways…

TDD - Test Driven Development. First write failing tests for new piece of functionality, then write code that satisfies tests, after that refactor your code and start from begin.

BDD – Behaviour Driven Development. First focus on **what** your application should do, write tests for this cases and only after then figure out **how** it will do.

TDD and BDD techniques

In common way TDD likes like step by step implementation of functionality. For example, if you want create a module with method, which will return multiplication of two numbers, you’ll create following tests one by one:

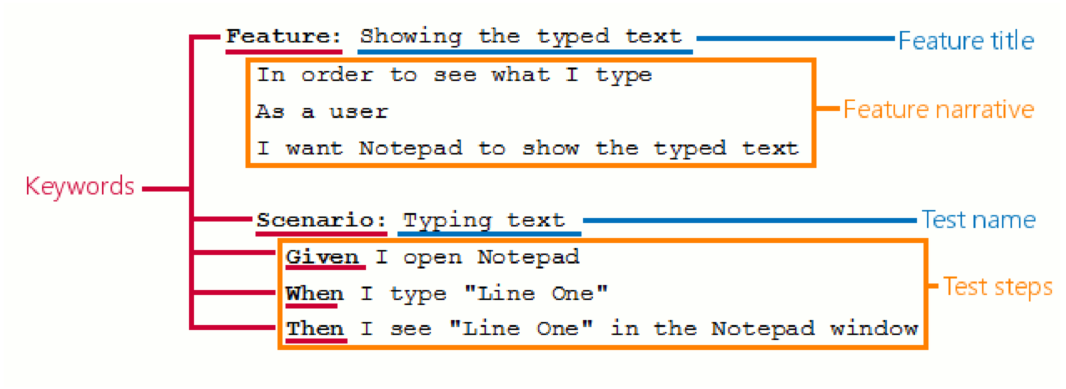
1. Test if file return module
2. Test if returned module has method
3. Test if exist method return a number
4. Test if exist method return correct result

In a result of that, you’ll be notified if someone move this module, rename method or change type of return value. So, your code is under control and it’s hard to break something.

TDD and BDD techniques

BDD is something a little different. We also are writing tests before code. But…

At the beginning we need to point out common functions of our new module. After that we need to write test for each function. And only after that we should start implementing code of our new module.



```
Feature: Showing the typed text
  In order to see what I type
  As a user
  I want Notepad to show the typed text

  Scenario: Typing text
    Given I open Notepad
    When I type "Line One"
    Then I see "Line One" in the Notepad window
```

TDD and BDD techniques

For example I want to create a queue with limit of elements. I'll point out next functions.

1. Adding elements
   - When it is not full, it should add element to the end and increment it's size
   - When it is full, it should pop the first element and add new element to the end
2. Removing elements
   - When it is not empty, it should remove first element and decrement it's size
   - When it is empty, it should throw an error

After that I'll write 4 tests, one for each case. And then I'll start writing code to pass all test. So I do not care about internal functions at the beginning, I only want to solve my problem.

Istanbul. Setup and usage

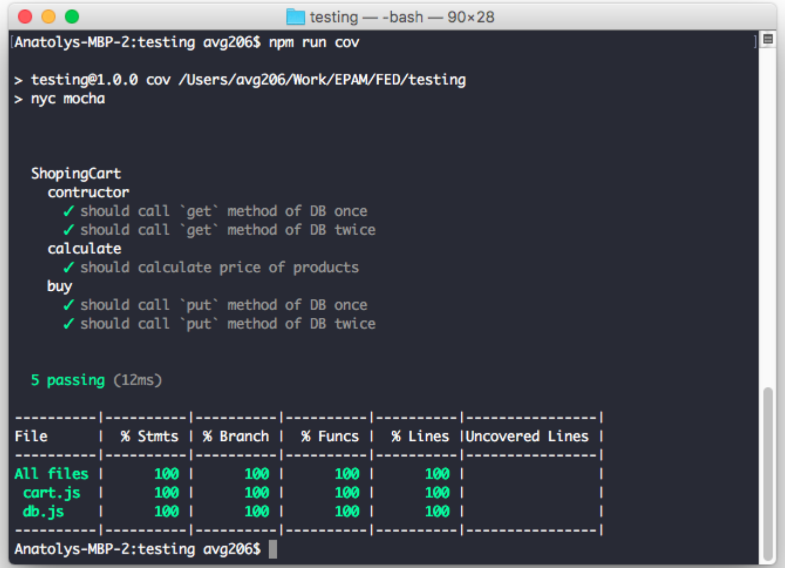
After writing test and working with a big project, you may want to see any metrics. One of them is test coverage.

Istanbul can calculate how much lines of code, function and statements are covered with tests. Also it can show rich report with details about each source file.

If we use mocha, it’s very easy to use it. Just install ‘nyc’ npm package. Add ‘nyc mocha’ to your npm scripts and run it.

Istanbul. Setup and usage

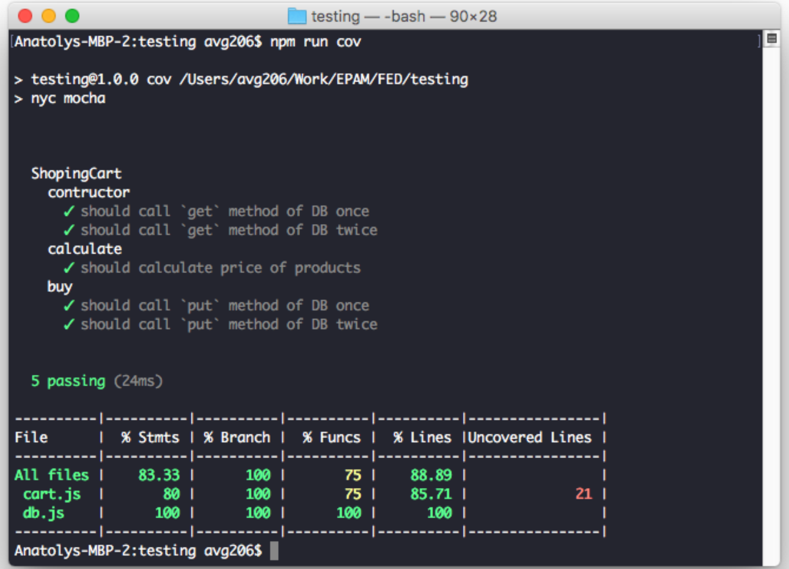
For example if we test coverage of project with Database and ShoppingCart, we will see such results:



```
Anatolys-MBP-2:testing avg206$ npm run cov

> testing@1.0.0 cov /Users/avg206/Work/EPAM/FED/testing
> nyc mocha



  ShopingCart
    contructor
      ✓ should call `get` method of DB once
      ✓ should call `get` method of DB twice
    calculate
      ✓ should calculate price of products
    buy
      ✓ should call `put` method of DB once
      ✓ should call `put` method of DB twice


  5 passing (12ms)

----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files |      100 |      100 |      100 |      100 |                |
 cart.js  |      100 |      100 |      100 |      100 |                |
 db.js    |      100 |      100 |      100 |      100 |                |
----------|----------|----------|----------|----------|----------------|
Anatolys-MBP-2:testing avg206$
```

Istanbul. Setup and usage

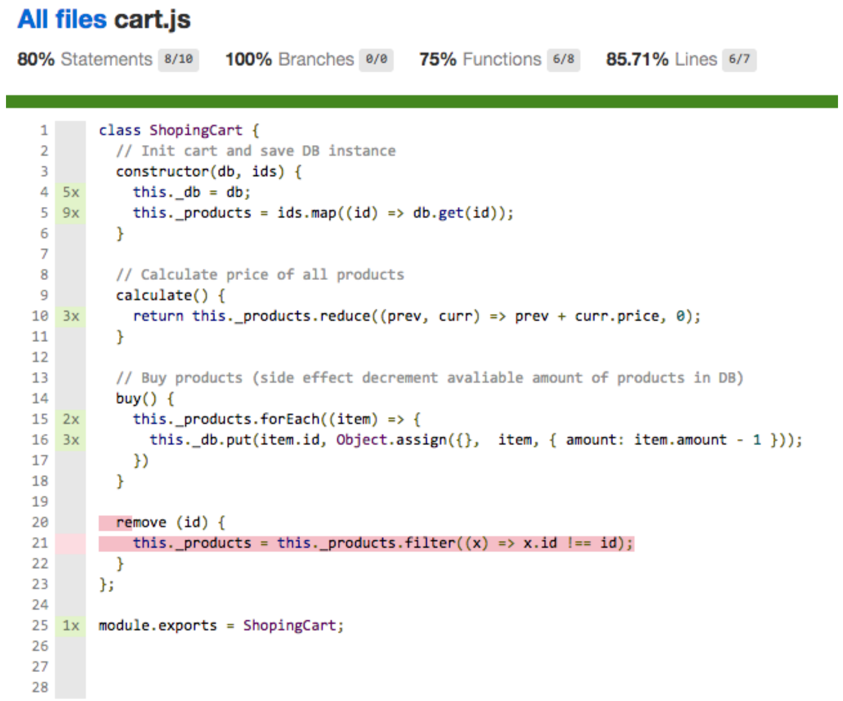
Next I’ve added new method ‘remove’ to ShoppingCart and coverage looks like:



```
Anatolys-MBP-2:testing avg206$ npm run cov

> testing@1.0.0 cov /Users/avg206/Work/EPAM/FED/testing
> nyc mocha



  ShopingCart
    contructor
      ✓ should call `get` method of DB once
      ✓ should call `get` method of DB twice
    calculate
      ✓ should calculate price of products
    buy
      ✓ should call `put` method of DB once
      ✓ should call `put` method of DB twice


  5 passing (24ms)

----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files |    83.33 |      100 |       75 |    88.89 |                |
 cart.js  |       80 |      100 |       75 |    85.71 |             21 |
 db.js    |      100 |      100 |      100 |      100 |                |
----------|----------|----------|----------|----------|----------------|
Anatolys-MBP-2:testing avg206$ 
```

Istanbul. Setup and usage

Detailed report for each file looks like:



All files cart.js

80% Statements 8/10   100% Branches 0/0   75% Functions 6/8   85.71% Lines 6/7

```
    class ShopingCart {
      // Init cart and save DB instance
      constructor(db, ids) {
5x      this._db = db;
9x      this._products = ids.map((id) => db.get(id));
      }

      // Calculate price of all products
      calculate() {
3x      return this._products.reduce((prev, curr) => prev + curr.price, 0);
      }

      // Buy products (side effect decrement avaliable amount of products in DB)
      buy() {
2x      this._products.forEach((item) => {
3x        this._db.put(item.id, Object.assign({},  item, { amount: item.amount - 1 }));
        })
      }

      remove (id) {
        this._products = this._products.filter((x) => x.id !== id);
      }
    };

1x  module.exports = ShopingCart;
```

Jest

And what if we want to merge all these libraries?

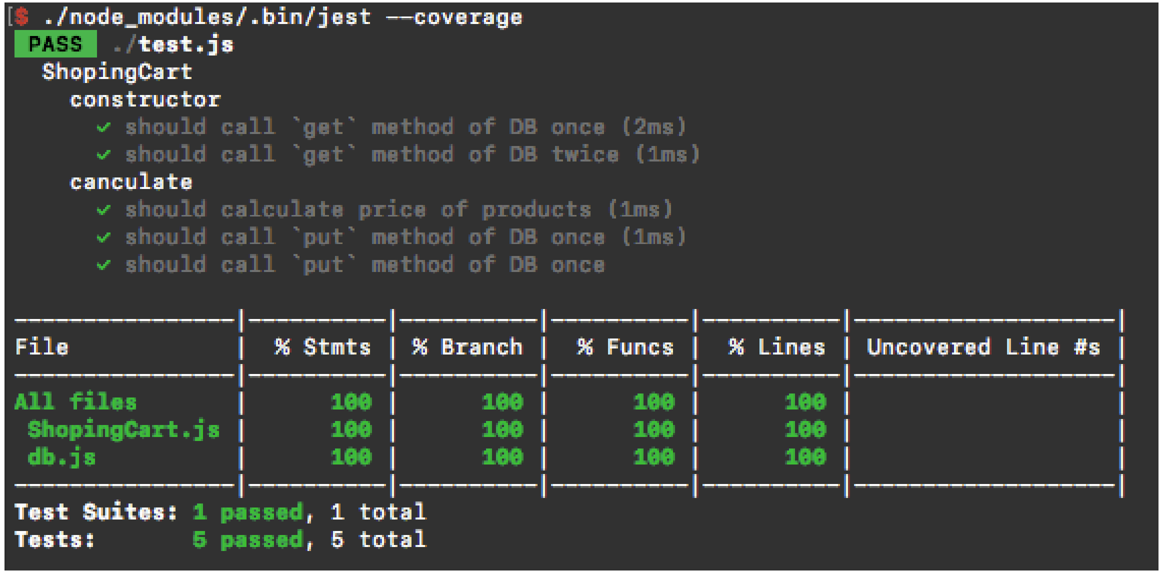
Jest

Jest - Zero configuration testing platform. One of Jest's philosophies is to provide an integrated "zero-configuration" experience.

Interesting features:

- Fast and sandboxed
- Built-in code coverage reports
- Zero configuration
- Powerful mocking library
- …

Jest



```
[$ ./node_modules/.bin/jest --coverage
 PASS  ./test.js
  ShopingCart
    constructor
      ✓ should call `get` method of DB once (2ms)
      ✓ should call `get` method of DB twice (1ms)
    canculate
      ✓ should calculate price of products (1ms)
      ✓ should call `put` method of DB once (1ms)
      ✓ should call `put` method of DB once

----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |      100 |      100 |      100 |      100 |                   |
 ShopingCart.js |      100 |      100 |      100 |      100 |                   |
 db.js          |      100 |      100 |      100 |      100 |                   |
----------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
```

Links

https://mochajs.org/

http://sinonjs.org/

http://chaijs.com/

https://istanbul.js.org/

https://blog.risingstack.com/node-hero-node-js-unit-testing-tutorial/

https://www.sitepoint.com/unit-test-javascript-mocha-chai/

https://www.sitepoint.com/sinon-tutorial-javascript-testing-mocks-spies-stubs/

https://blog.risingstack.com/getting-node-js-testing-and-tdd-right-node-js-at-scale/

QUESTIONS???

THANKS!

UNIT TESTING. TDD. MOCHA, CHAI, SINON. TEST COVERAGE.
BY
ANATOLI HUSEU