



NodeJS Testing

Node.js Global Mentoring Program 2018Q4

MARCH, 2019

AGENDA

- Introduction & Prerequisites;
- What is Unit Testing;
- Mocha usage;
- Chai and Sinon usage;
- TDD and BDD techniques;
- Istanbul. Setup and usage;
- Links;
- Q&A.

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 basic 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're able to fully control what's inside mock.

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 you as many asserting as you want, but you should test one concern only.

Unit Testing

We have such code:

```
1
2
3  function findGCD(a, b) {
4      let x = Math.abs(a);
5      let y = Math.abs(b);
6
7      while (x && y) {
8          if (x > y) {
9              x = x % y;
10         } else {
11             y = y % x;
12         }
13     }
14
15     return x + y;
16 }
17
18
```

And we'd like to test it.

Unit Testing

First of all, lets 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.

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

By if we have a lot of tests, or we need to to some command 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:

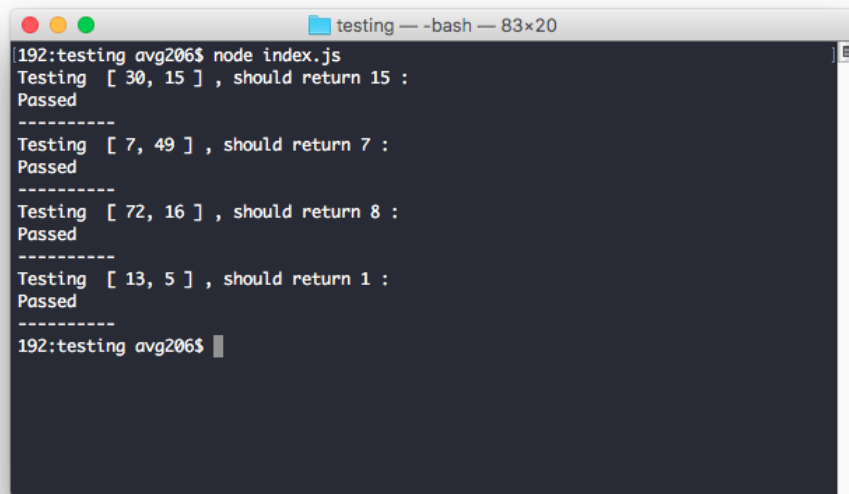
```
3  function testIt(cases, func) {
4      for (let item of cases) {
5          console.log('Testing ', item.input, ', should return', item.output, ':');
6          const res = func.apply(null, item.input);
7
8          if (res === item.output) {
9              console.log('Passed');
10             } else {
11                 console.log('Failed,', 'return', res);
12             }
13
14             console.log('-----');
15         }
16     }
```

Test cases:

```
33  const CASES = [
34      { input: [30, 15], output: 15 },
35      { input: [7, 49], output: 7},
36      { input: [72, 16], output: 8 },
37      { input: [13, 5], output: 1 },
38  ]
39
40
41  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.

A terminal window titled 'testing — -bash — 83x20' showing the execution of a Node.js script. The prompt is '192:testing avg206\$'. The command 'node index.js' has been executed, resulting in four test cases being run and all passing. Each test case is displayed on a new line, separated by a dashed line. The test cases are: '[30, 15], should return 15 : Passed', '[7, 49], should return 7 : Passed', '[72, 16], should return 8 : Passed', and '[13, 5], should return 1 : Passed'. The terminal prompt '192:testing avg206\$' is shown again at the bottom, ready for the next command.

```
192:testing avg206$ 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
-----
192:testing avg206$
```

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 used on both NodeJS and Browser
- flexible and accurate reporting
- ability to test asynchronous methods
- and more ...

Mocha framework

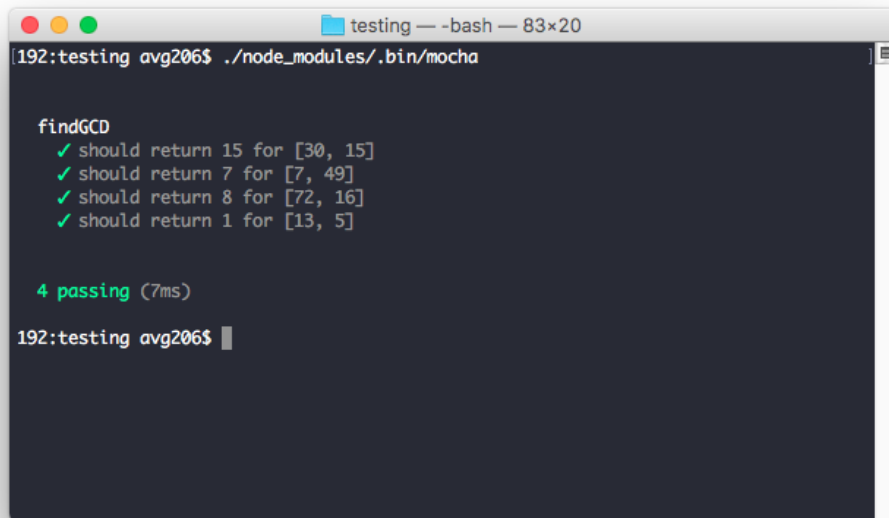
To add mocha to our project, we just need to install it as npm package. To run it, we need to add command “mocha” to npm scripts.

But, let's prepare testing code:

```
1  const assert = require('assert');
2  const findGCD = require('../index.js');
3
4  describe('findGCD', function() {
5    it('should return 15 for [30, 15]', function() {
6      assert.equal(15, findGCD(30, 15));
7    });
8    it('should return 7 for [7, 49]', function() {
9      assert.equal(7, findGCD(7, 49));
10   });
11   it('should return 8 for [72, 16]', function() {
12     assert.equal(8, findGCD(72, 16));
13   });
14   it('should return 1 for [13, 5]', function() {
15     assert.equal(1, findGCD(13, 5));
16   });
17 });
```

Mocha framework

- And as result, we can test our function again. As you can see, it's more structured and beautiful 😊



```
testing — -bash — 83x20
192:testing avg206$ ./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 (7ms)
192:testing avg206$
```

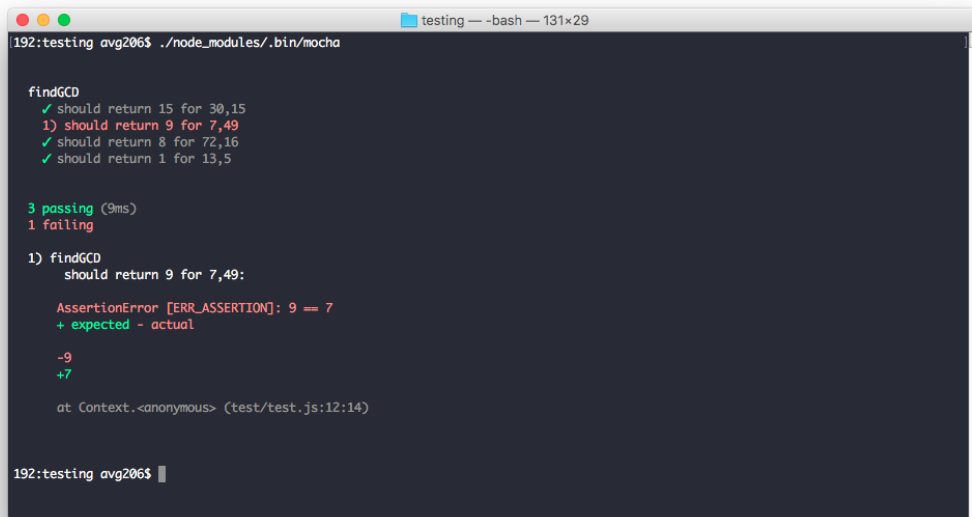
Mocha framework

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

```
1  const assert = require('assert');
2  const findGCD = require('../index.js');
3
4  describe('findGCD', function() {
5    [
6      { input: [30, 15], output: 15 },
7      { input: [7, 49], output: 7},
8      { input: [72, 16], output: 8 },
9      { input: [13, 5], output: 1 },
10   ].forEach(function (item) {
11     it('should return ' + item.output + ' for ' + item.input, function() {
12       assert.equal(item.output, findGCD.apply(null, item.input));
13     });
14   })
15 });
```

Mocha framework

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



```
192:testing avg206$ ./node_modules/.bin/mocha

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 (9ms)
1 failing

1) findGCD
   should return 9 for 7,49:

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

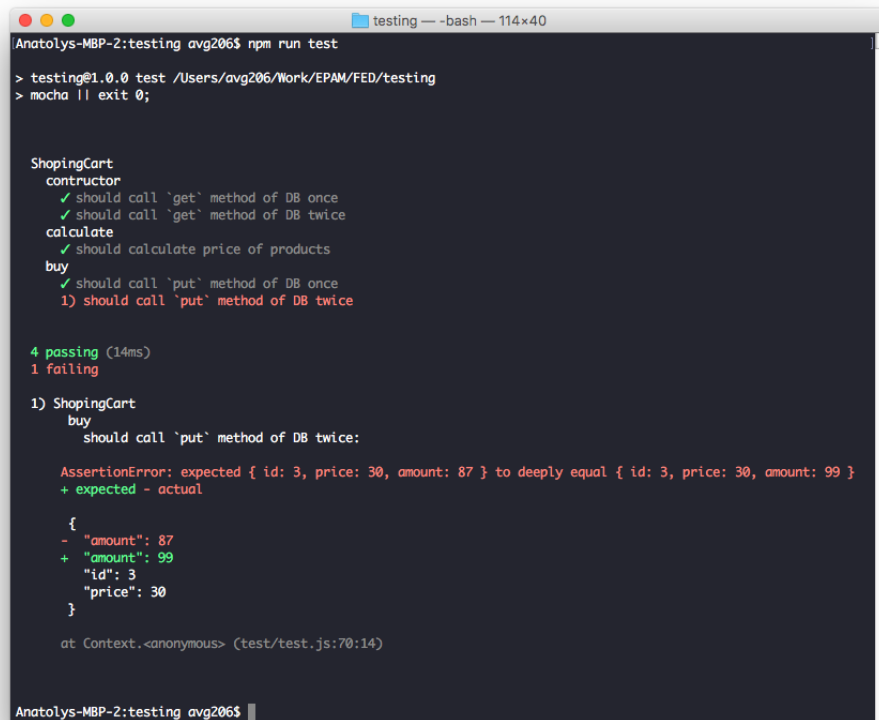
     -9
     +7

   at Context.<anonymous> (test/test.js:12:14)

192:testing avg206$
```

Mocha framework

More complex errors...



```
Anatolys-MBP-2:testing avg206$ npm run test
> testing@1.0.0 test /Users/avg206/Work/EPAM/FED/testing
> mocha || exit 0;

ShoppingCart
  constructor
    ✓ 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
    1) should call `put` method of DB twice

4 passing (14ms)
1 failing

1) ShoppingCart
   buy
     should call `put` method of DB twice:
     AssertionError: expected { id: 3, price: 30, amount: 87 } to deeply equal { id: 3, price: 30, amount: 99 }
+ expected - actual

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

at Context.<anonymous> (test/test.js:70:14)

Anatolys-MBP-2:testing avg206$
```

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 was very simple and “unit” was using just inputs to produce output. But most of time, do 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 can be used with any testing framework and even without it.

There are three assertion styles:

Assert - more classis notation similar to NodeJs Build-in package

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

Expect - notation which allows you chain together natural language assertions

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

Should - also chainable notation, but in this case each object are extended with method 'should'

```
1 foo.should.be.a('string');  
2 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:

```
1  const spy = sinon.spy(); // Creates an anonymous spies
2  const spy = sinon.spy(myFunc); // Spies on the provided function
3  const spy = sinon.spy(object, "method"); // Spies method of object
```

Usage:

```
1  "should call method once with each argument": function () {
2    var object = { method: function () {} };
3    var spy = sinon.spy(object, "method");
4
5    object.method(42);
6    object.method(1);
7
8    assert(spy.withArgs(42).calledOnce);
9    assert(spy.withArgs(1).calledOnce);
10 }
```

Chai and Sinon

Let's look into stubs.

Creating:

```
1 var stub = sinon.stub(); // Just stub
2 var stub = sinon.stub(object, "method"); // Stub method of object
3 var stub = sinon.stub(obj); // Stub all methods of object
```

Usage:

```
1 "test should stub method differently based on arguments": function () {
2   var callback = sinon.stub();
3   callback.withArgs(42).returns(1);
4   callback.withArgs(1).throws("TypeError");
5
6   callback(); // No return value, no exception
7   callback(42); // Returns 1
8   callback(1); // Throws TypeError
9 }
```

```
1 "test should stub method differently on consecutive calls": function () {
2   var callback = sinon.stub();
3   callback.onCall(0).returns(1);
4   callback.onCall(1).returns(2);
5   callback.returns(3);
6
7   callback(); // Returns 1
8   callback(); // Returns 2
9   callback(); // All following calls return 3
10 }
```

Chai and Sinon

Let's look into mocks.

```
1  "test should call all subscribers when exceptions": function () {  
2      var myAPI = { method: function () {} };  
3  
4      var spy = sinon.spy();  
5      var mock = sinon.mock(myAPI);  
6      mock.expects("method").once().throws();  
7  
8      PubSub.subscribe("message", myAPI.method);  
9      PubSub.subscribe("message", spy);  
10     PubSub.publishSync("message", undefined);  
11  
12     mock.verify();  
13     assert(spy.calledOnce);  
14 }
```

Chai and Sinon

Now let's practice. We have database:

```
1  const Database = {  
2    // Get Stored object by ID  
3    get(id) {},  
4  
5    // Update stored object by ID with new value  
6    put(id, value) {}  
7  }  
8  
9  module.exports = Database;
```

In real life it can contain async calls to API or database driver, but we do not need it. Also we have a class, which called ShopingCart with following API. And we need to write test for this class.

```
1  class ShopingCart {  
2    // Init cart and save DB instance  
3    constructor(db, ids) {}  
4  
5    // Calculate price of all products  
6    calculate() {}  
7  
8    // Buy products (side effect decrement available amount of products in DB)  
9    buy() {}  
10 };  
11  
12 module.exports = ShopingCart;
```

Chai and Sinon

First of all, we need to stub Database methods

```
9  const PRODUCTS = {
10    1: { id: 1, price: 10, amount: 100 },
11    2: { id: 2, price: 20, amount: 100 },
12    3: { id: 3, price: 30, amount: 100 },
13  };
14
15  const stubGet = sinon.stub(Database, 'get');
16  stubGet.withArgs(1).returns(PRODUCTS[1]);
17  stubGet.withArgs(2).returns(PRODUCTS[2]);
18  stubGet.withArgs(3).returns(PRODUCTS[3]);
19  const stubPut = sinon.stub(Database, 'put');
```

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

```
22  afterEach(function() {
23    stubGet.resetHistory();
24    stubPut.resetHistory();
25  });
```


Chai and Sinon

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

```
27 describe('constructor', function() {
28   it ('should call `get` method of DB once', function() {
29     const cart = new ShoppingCart(Database, [1]);
30
31     assert.equal(stubGet.callCount, 1);
32     assert.equal(stubGet.getCall(0).args[0], 1)
33   })
34
35   it ('should call `get` method of DB twice', function() {
36     const cart = new ShoppingCart(Database, [1, 2]);
37
38     assert.equal(stubGet.callCount, 2);
39     assert.equal(stubGet.getCall(0).args[0], 1);
40     assert.equal(stubGet.getCall(1).args[0], 2);
41   })
42 })
```

Chai and Sinon

You've stubbed 'get' method and it should return real products. Let test if that is true by calling 'calculate'.

```
44 describe('calculate', function() {  
45   it ('should calculate price of products', function() {  
46     const cart = new ShoppingCart(Database, [1, 2, 3]);  
47  
48     assert.equal(cart.calculate(), 60);  
49   })  
50 })
```

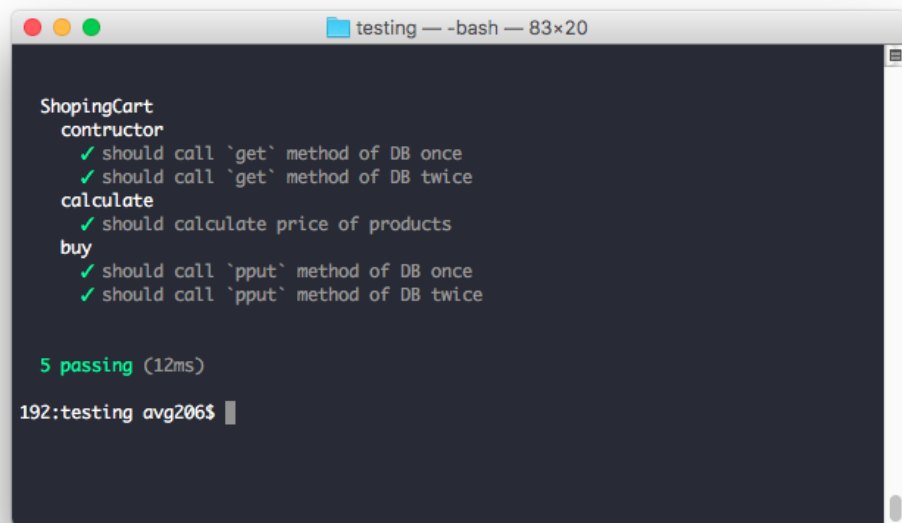
Chai and Sinon

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.

```
52 describe('buy', function() {
53   it ('should call `pput` method of DB once', function() {
54     const cart = new ShoppingCart(Database, [1]);
55     cart.buy();
56
57     assert.equal(stubPut.callCount, 1);
58     assert.equal(stubPut.getCall(0).args[0], 1);
59     assert.deepEqual(stubPut.getCall(0).args[1], { id: 1, price: 10, amount: 99 });
60   })
61
62   it ('should call `pput` method of DB twice', function() {
63     const cart = new ShoppingCart(Database, [1, 3]);
64     cart.buy();
65
66     assert.equal(stubPut.callCount, 2);
67     assert.equal(stubPut.getCall(0).args[0], 1);
68     assert.deepEqual(stubPut.getCall(0).args[1], { id: 1, price: 10, amount: 99 });
69     assert.equal(stubPut.getCall(1).args[0], 3);
70     assert.deepEqual(stubPut.getCall(1).args[1], { id: 3, price: 30, amount: 99 });
71   })
72 })
```

Chai and Sinon

So, tests are ready and let's run them...



```
testing — -bash — 83x20

ShoppingCart
  constructor
    ✓ should call `get` method of DB once
    ✓ should call `get` method of DB twice
  calculate
    ✓ should calculate price of products
  buy
    ✓ should call `pput` method of DB once
    ✓ should call `pput` method of DB twice

5 passing (12ms)

192:testing avg206$
```

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 event 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:

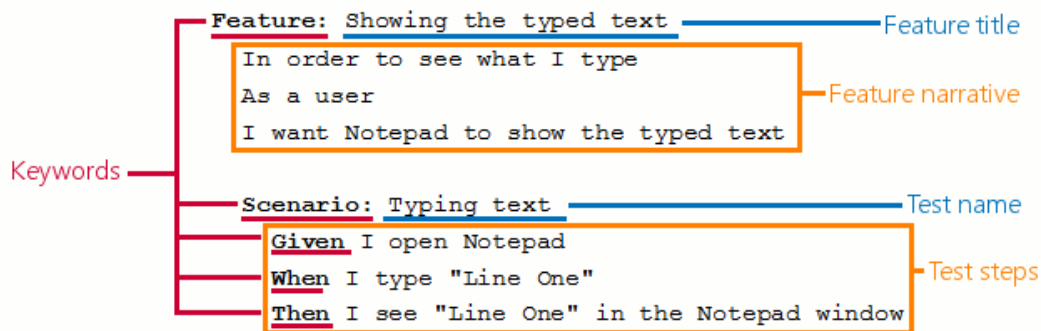
1. Test if file return module
2. Test if returned module has method
3. Test if exist method return a number
4. Test if exist method return correct result

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.



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.

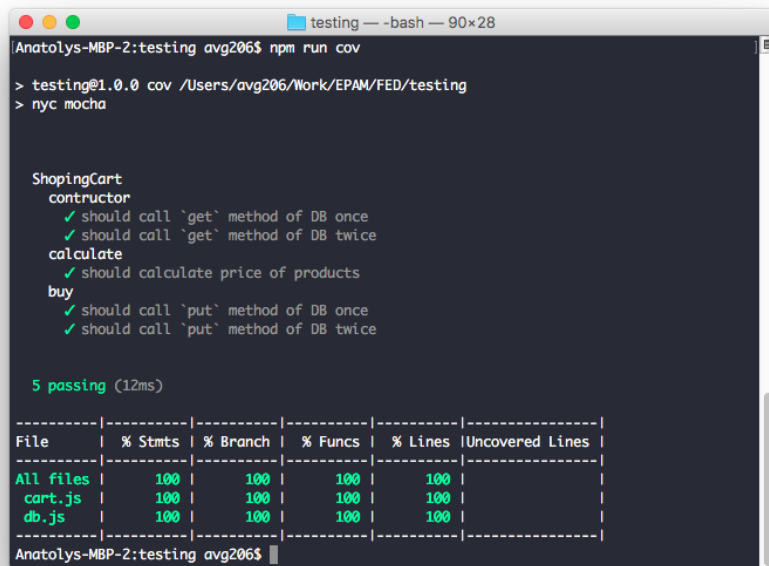
Istanbul can calculate how much lines of code, function and statements are covered with tests. Also it can show rich report with details about each source file.

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

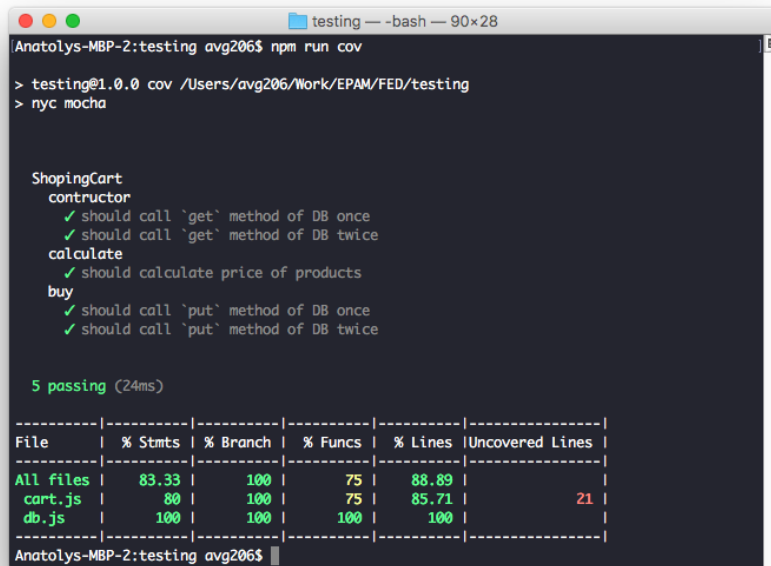
ShoppingCart
  constructor
    ✓ should call `get` method of DB once
    ✓ should call `get` method of DB twice
  calculate
    ✓ should calculate price of products
  buy
    ✓ should call `put` method of DB once
    ✓ should call `put` method of DB twice

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

ShoppingCart
  constructor
    ✓ 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

```
1  class ShoppingCart {
2    // Init cart and save DB instance
3    constructor(db, ids) {
4      5x    this._db = db;
5      9x    this._products = ids.map((id) => db.get(id));
6    }
7
8    // Calculate price of all products
9    calculate() {
10     3x    return this._products.reduce((prev, curr) => prev + curr.price, 0);
11   }
12
13   // Buy products (side effect decrement available amount of products in DB)
14   buy() {
15     2x    this._products.forEach((item) => {
16       3x      this._db.put(item.id, Object.assign({}, item, { amount: item.amount - 1 }));
17     })
18   }
19
20   remove(id) {
21     this._products = this._products.filter((x) => x.id !== id);
22   }
23 };
24
25 1x module.exports = ShoppingCart;
26
27
28
```

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/>

<https://docs.pact.io/>

A stylized world map in a light blue color, centered on the Atlantic Ocean. The map shows the outlines of continents and countries. Overlaid on the map is the text 'Q&A' in a large, white, serif font, positioned in the center of the image.

Q&A

A light blue world map is centered in the background of the slide, showing the outlines of continents and countries. The map is semi-transparent, allowing the white text to stand out.

THANKS!

**NGMP 2018Q4: NODEJS TESTING
BY
PAVEL AURAMENKA**