Published in codebursı

NC Patro Follow

Mar 7, 2018 · 7 min read · Listen

JavaScript — Unit Testing using Mocha and Chai

This article will cover testing of basic function, testing of async callback functions and testing of promises with Mocha and Chai.

Code Repo: (https://github.com/npatro/javascript-unit-testing-with-mocha)

Find Bug with help of Light and Testing

The smallest parts of an hose units to check whether it is fit for use or not is called unit testing.

If we are going to create a test for any function, then we need to make sure that the function by itself, separate from everything around, should do what it is intended to do, not more, not less and mock rest of things which are not under test. And that's basic principle of unit test.

Integration Testing

In the context of Unit Testing, testing the interactions between two units called Integration Testing. Scenarios like function under test calling another function with some context. We should still mock the outside resources but need to test those integration links.

Prerequisite:

Your Machine should have node and npm installed.

Install the node.js LTS version from Node website. npm gets installed along with node automatically.

Run below in command line to check the successful installation of node and npm.

```
npm -v      // will return installed npm version
node -v     // will return installed node version
```

Mocha

- Mocha is a JavaScript Test Framework.
- Runs on Node.js and Browser
- Installation: (Run the below commands in terminal or cmd)

```
npm install --global mocha
```


`--global` helps to insta el which helps to run mocha test through command line.

`--save-dev` helps to add the mocha as dependency in `package.json` file for that particular project.

Mocha Basic Spec

```
var assert = require('assert');

describe('Basic Mocha String Test', function () {
 it('should return number of charachters in a string', function () {
        assert.equal("Hello".length, 4);
    });

 it('should return first charachter of the string', function () {
        assert.equal("Hello".charAt(0), 'H');
    });
});
```

In the above test snippet,

- ***assert*** helps to determine the status of the test, it determines failure of the test.
- ***describe*** is a function which holds the collection of tests. It takes two parameters, first one is the meaningful name to functionality under test and second one is the function which contains one or multiple tests. We can have nested *describe* as well.
- ***it*** is a function again which is actually a test itself and takes two parameters, first parameter is name to the test and second parameter is function which holds the body of the test.

Steps to Run the test:

- Download or clone the Github Repo, navigate to the repo in command line
- Run `npm install` to install all dependencies from package.json


```
"scripts": {
    "test": "mocha"
  }
```

Below is the output which shows up after running the test.

```
> mocha


  Basic Mocha String Test
    1) should return number of charachters in a string
    ✓ should return first charachter of the string


  1 passing (10ms)
  1 failing

  1) Basic Mocha String Test
       should return number of charachters in a string:

      AssertionError: 5 == 4
      + expected - actual

      -5
      +4

      at Context.<anonymous> (test/test.js:4:16)


npm ERR! Test failed.  See above for more details.
```

Mocha Test Output

Test Assertion

- ***Assertion*** is an expression which helps system (Mocha in this case) to know code under test failed.
- ***Assert***'s job is to just throw an error when things are not correct or right.
- ***Assert*** tests the expression and it does nothing when expression passes but throws exception in case of failure to tell the test framework about it.
- We can just throw an exception to fail the test as well.

Testing Actual Code with Mocha

Every function does a sp ction needs to be called from test or spec file with required *inputs*. Then we will put *assert* to validate the *output* or task of the function.

```
/* Code */
function LoginController() {

function isValidUserId(userList, user) {
   return userList.indexOf(user) >= 0;
}

return {
    isValidUserId
  }
}
module.exports = LoginController();

/* Test */
it('should return true if valid user id', function(){
      var isValid =
loginController.isValidUserId(['abc123','xyz321'], 'abc123')
      assert.equal(isValid, true);
});
```

Code and Test available at this Github Repo

Testing Asynchronous Function (callback)

While testing callback function, the only major difference is, we need to tell Mocha that the test is complete because of async nature of function under test. In the below example, Mocha waits for the ***done()*** function to be get called to complete the test.

```
/* Code */
function isValidUserIdAsync(userList, user, callback) {
    setTimeout(function(){
      callback(userList.indexOf(user) >= 0)
    }, 1);
}
Note: setTimeout has been used to simulate the async behavior.

/* Test */
```


```
  });
});
```

Code and Test available at this Github Repo

Hooks like beforeEach and afterEach

- Few steps or code we might want to execute before or after each test to either setup preconditions before test or cleanup after test. so those code can be put inside ***beforeEach()*** and ***afterEach()*** methods.
- It is always good practice to have named function or description to hooks, which helps to identify errors quickly in tests.

```
beforeEach('Setting up the userList', function(){
  console.log('beforeEach');
  loginController.loadUserList(['abc123','xyz321']);
});

describe('LoginController', function () {
...
}
```

Code and Test available at this Github Repo

Chai

- Chai is BDD/TDD assertion library.
- Can be paired with any javascript testing framework.
- Assertion with Chai provides natural language assertions, expressive and readable style.
- Installation: (Run the below commands in terminal or cmd)

```
npm install --save-dev chai
```


- The ***expect*** interface
- The ***should*** interface extends each object with a should property for assertion.
- ***should*** property gets added to the *Object.Prototype*, so that all object can access it through prototype chain.

> *You can go through article* JavaScript — Prototype *to understand more on prototype chain.*

Below is the usage of expect and should instead of Mocha assert

```
var assert = require('assert');
var expect = require('chai').expect;
var should = require('chai').should();

it('should return true if valid user id', function(){
      var isValid = loginController.isValidUserId('abc123')
      //assert.equal(isValid, true);
      expect(isValid).to.be.true;
});

it('should return false if invalid user id', function(){
      var isValid = loginController.isValidUserId('abc1234')
      //assert.equal(isValid, false);
      isValid.should.equal(false);
});
```

Code and Test available at this Github Repo

Testing Promises

- ***Promise*** is asynchronous in nature. Let's understand what *promise* is in simple form, *promise* is like you ask for something and instead of getting it immediately, you get something else(*promise*) that says you will get actual thing once its ready.
- We depend on one more Chai library `chai-as-promised` to test promises
- Installation:

- In *callback function* t out test completion by calling *done()* method but in case of *promise*, we just need to return the *promise* and Mocha will watch the *promise* by itself for test completion.

```
/* Code */
  function isAuthorizedPromise(user){
    return new Promise(function(resolve){
        setTimeout(function(){resolve(userList.indexOf(user) >= 0)},
10);
    });
  }
Note: setTimeout has been used to simulate the async behavior.

/* Test */
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised).should();

describe('isAuthorizedPromise', function(){

  it('should return true if valid user id', function(){
      return
loginController.isAuthorizedPromise('abc123').should.eventually.be.t
rue;
    });

});
```

Code and Test available at this Github Repo

Assertion with Chai is expressive

- Assertion with Chai provides expressive language & readable style like below test.
- In below sample test, we put assertion like `car.should.have.property(propertyName)` which explains itself about the objective of test.
- `should` is available for the object `car` because of *prototype* chain but in case of null object, we can use the `should` instance directly.


```
    car.should.hav                                    ;
});

it('Checking for null', function(){
    var car = null;
    //car.should.not.exist; (Cannot read property 'should' of null)
    should.not.exist(car);
});
```

Code and Test available at this Github Repo

Managing test-suite in Mocha

Skip the test-case or test-suite:

- Never comment out the test-case or test-suite in test/spec files, always skip the test. Commenting out the test is equivalent of deleting the test, It is hard to get noticed about commented tests but skip tests shows up on result file so we can act on those later.
- ***skip() method*** helps to skip the particular test or group of tests, means `describe.skip()` and `it.skip()` both allowed.
- Skipped tests shows as pending in test result summary.

```
describe('LoginController', function () {

  describe('isValidUserId', function(){

    it.skip('should return true if valid user id', function(){
      ...
    });

    it('should return false if invalid user id', function(){
      ...
    });

  });

  describe.skip('isValidUserIdAsync', function(){
```


```
});
```

```
> mocha './test/**/*.spec.js'


  LoginController
    isValidUserId
      - should return true if valid user id
      ✓ should return false if invalid user id
    isValidUserIdAsync
      - should return true if valid user id
    isAuthorizedPromise
      ✓ should return true if valid user id


  2 passing (24ms)
  2 pending
```

Test summary with skipped tests

Run specific test-case or test-suite:

- There might be situation when we want to run specific test-case or test-suite to check the functionality without worrying about all test cases.
- **only()** method helps to run specific test or test-suite. we can have multiple *only()* in entire test-suite.

```
describe('LoginController', function () {

  describe('isValidUserId', function(){

    it.only('should return true if valid user id', function(){
      ...
    });

    it('should return false if invalid user id', function(){
      ...
    });

  });
```

```
    });

});
```

Pending tests

- We can add pending tests in test-suites with having **it()** method without second argument.

```
describe('isValidUserId', function(){
  it('should return false if user id blank');
});
```

Summary

Some writes tests after writing code, some before writing code and some in parallel with code. It is debatable which approach is better but at the end all agree to the point that unit testing is critical part of development. So, we should be aware of all tools and techniques of unit testing.

If you like this post and it was helpful, please click the clap 👏 button multiple times to show the support, thank you.

codeburst.io