JEST for JavaScript unit testing

Bhupendra Singh Parihar
Senior Software Developer
Ness Technologies
Bangalore, India
bhupendrasparihar@gmail.com

Abstract— There are many framework or library for JavaScript unit testing like jasmine, mocha, yolpo, ava, qunit etc. Out of all these Jasmine is the quite famous one. Jasmine is the one of the framework which is being widely accepted by developer community. Though, Facebook added one more to this list called "JEST". This paper explains the need for another framework and the comparison between JEST and JASMINE. It also answers few of the questions like "Why we need to write unit test for your code, what are the main features of testable code?"

I. Introduction

You probably know that testing is good, but the first hurdle to overcome when trying to write unit tests for client-side code is the lack of any actual units; JavaScript code is written for each page of a website or each module of an application and is closely intermixed with back-end logic and related HTML. In the worst case, the code is completely mixed with HTML, as inline events handlers. What is a unit anyway? In the best case, it is a pure function that you can deal with in some way — a function that always gives you the same result for a given input.

This makes unit testing pretty easy, but most of the time you need to deal with side effects, which here means DOM manipulations. It's still useful to figure out which units we can structure our code into and to build unit tests accordingly. Testing JavaScript code is not just a matter of using some test runner and writing a few tests; it usually requires some heavy structural changes when applied to code that has been tested only manually before.

II. Why do you need a unit test?

- Unit test reduces the fear of new changes. As new changes might break the existing functionality, writing unit test for existing code give you confidence.
- Unit test helps you to stop writing code and think about the feature you are developing.
- Unit test also provide a good documentation if you have written well.
- Unit testing helps you write more modular and readable code. Classes and functions are main unit which you want to test. But when you are thinking in unit testing perspective, you start writing smallest unit that can be tested. You can name these units like models, services and utility function. Whereas when you don't think like this, you end up writing bigger chunks of code, which do many things with many responsibilities. Like doing the ajax calls, manipulating the DOM, binding the events etc. Look at the code below.

```
var baseUrl = 'https://api.myjson.com/bins/';

function showMessage(id) {
  $.ajax({
    url: baseUrl + id,
    type: 'GET',
    dataType: 'json',
    success: function(data) {
      $("#messagebox").html(data.name);
    }
  });
}

var id = '19qtx';
showMessage(id);
```

Above function seems to be doing lot more stuff than just AJAX call. We can divide this function into sub units.

```
//config - Configuration part of the program
// Mostly user input for the program
var config = (function() {
  var baseUrl = 'https://api.myjson.com/bins/';

  return {
    baseUrl: baseUrl
  };
})();
```

```
//dataservice(responsible for just AJAX call)
var dataservice = (function($, config) {
  var callApi = function(url, type, callback) {
      $.ajax({
        url: url,
        type: type,
        dataType: 'json',
        success: function(data) {
          callback(data);
        }
      });
    },
    getMessage = function(id, callback) {
      url = config.baseUrl + id;
      callApi(url, 'GET', callback);
    };

  return {
    getMessage: getMessage
  };
})($, config);
```

```
//messenger - Responsible for updating the DOM
var messenger = (function($, dataservice) {
  var showMessage = function(id) {
    dataservice.getMessage(id, function(message) {
      $("#messagebox").html(message.name);
    });
  };

  return {
    showMessage: showMessage
  };
})($, dataservice);
```

```
//main - invoking the whole function
(function(messenger) {
  var id = '19qtx';
  messenger.showMessage(id);
})(messenger);
```

These four units can be tested individually. One thing we need to make sure for running these units is that, they all needs to be in the same order as mentioned above. This means they are dependent on other units. We can use *require.js* to pass the dependency of these units. Requirejs uses the AMD api for defining modules. Other common ways of defining modules in javascript is CommonJS and ES Harmony pattern. AMD way of defining a module.

```
define(
   module_id /*optional*/ ,
   [dependencies] /*optional*/ ,
   definition : function /*function for instantiating
the               module or object*/
);
```

Let’s define our all modules using *Require.js*

```
define('config', [],
  function() {
   var baseUrl = 'https://api.myjson.com/bins/';

    return {
      baseUrl: baseUrl
    };
  }
);
```

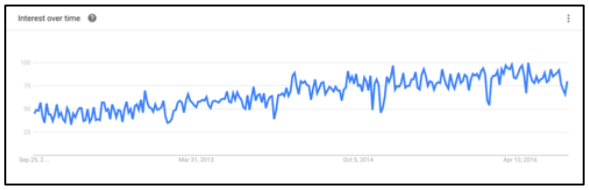
```
define('dataservice', ['jquery', 'config'],
  function($, config) {
    var
      callApi = function(url, type, callback) {
        $.ajax({
          url: url,
          type: type,
          dataType: 'json',
          success: function(data) {
            callback(data);
          }
        });
      },

      getMessage = function(id, callback) {
        url = config.baseUrl + id;
        callApi(url, 'GET', callback);
      };

    return {
      getMessage: getMessage
    };
  }
);
```

```
define('messenger', ['jquery', 'dataservice'],
  function($, dataservice) {
    var showMessage = function(id) {
      dataservice.getMessage(id, function(message) {
        $("#messagebox").html(message.name);
      });
    };

    return {
      showMessage: showMessage
    };
  }
);
```

```
(function() {
  requirejs.config({
    paths: {
      'jquery': '../scripts/jquery-1.11.2.min'
    }
  });

  require(
    ['messenger'],
    function(messenger) {
      var id = '19qtx';
      messenger.showMessage(id);
    }
  );
})();
```

Now all our modules have clear responsibility and *Require.js* handles any kind of dependencies which we have defined in module definition.

Google Trends graph of JasmineJS. It clearly shows how the interest of writing testable code has been increased during last five years.



Now we understood why it is important to write unit tests; question comes, how to write it. There are many libraries and framework for unit testing in JavaScript. Eg. AVA ,Suitest ,DOH ,LBRTW UT ,Enhance JS ,RhUnit ,Crosscheck ,J3Unit ,JSNUnit ,YUI Test ,JSSpec ,UnitTesting ,JSpec ,screw-unit ,Test.Simple ,Test.More ,TestCase ,TestIt ,jsUnitTest ,JSTest ,JSTest.NET ,jsUnity ,RhinoUnit ,JasUnit ,FireUnit ,Js-test-driver ,Js-test-runner ,Sinon.js ,Vows ,Nodeunit ,Tyrtle ,wru ,Buster.JS, Jasmine etc.

III. What is Unit Testing

Essentially, a unit test is a method that instantiates a small portion of our application and verifies its behavior independently from other parts. A typical unit test contains 3 phases: First, it initializes a small piece of an application it wants to test (also known as the system under test, or SUT), then it applies some stimulus to the system under test (usually by calling a method on it), and finally, it observes the resulting behavior. If the observed behavior is consistent with the expectations, the unit test passes, otherwise, it fails, indicating that there is a problem somewhere in the system under test. These three unit test phases are also known as **Arrange**, **Act** and **Assert**, or simply **AAA**.

A unit test can verify different behavioral aspects of the system under test, but most likely it will fall into one of the following two categories: state-based or interaction-based. Verifying that the system under test produces correct results, or that its resulting state is correct, is called **state-based** unit testing, while verifying that it properly invokes certain methods is called **interaction-based** unit testing.

For this paper we will only focus on Jasmine and JEST. Let’s understand some terminology related to Javascript Unit Testing.

Assertion: A narrative communication which by virtue of its meaning is true or false. In Unit Testing terms, assertions are functions that verify values are what you think they are.

```
function assert(value, description){
  if(!value){
    throw new Error(description + 'is false!');
  }
}

assert(5 + 6 === 12,'5 + 6 is 11');
```

Synchronous: A program that runs all the way through without stopping.

Asynchronous: Javascript has single thread of execution, so you cannot run anything in parallel. But you can run a piece of code later, by scheduling it using callback.

- NodeJs uses asynchronous IO(libuv)
- Browser uses events
- Developer can use setTimeout/setInterval

In Async programming, test needs to wait until ‘later’

```
var delay = function(func, delayInSeconds) {
  setTimeout(func, delayInSeconds * 1000);
};

function testDelay() {
  var now = new Date().getTime() / 1000;
  delay(function() {
      assert(new Date().getTime() / 1000 - now === 5);
  },5);
}
```

Stubs: A function that returns known dummy data.

Spies: A function that returns known dummy data and extra information like, how many times it is called and what are the parameters when it was called.

Mock Objects: When we test our code in isolation without calling its dependencies, we need to mock the dependencies.

Fixtures: Fake Data(JSON object, DOM nodes)

By analyzing the above code, let’s write a small test case of messenger module

IV. Walk with JASMINE

We mentioned a unit is a piece of code, which is handling a single responsibility. Now broaden our definition of *“UNIT”*. Let’s assume a UNIT is a small piece of code(HTML+JS+CSS) which is a small app component in its own. By using *JASMINE,* you cannot test a HTML, or a CSS or a behavior of a component. Here comes the power of JEST.

```
module.exports = sum

function sum(a, b){
     return a + b;
}
```

```
const sum = require('./sum')

test('add 1 + 2 to equal 3', () => {
  expect(sum(1,2)).toBe(3)
});
```

Here I won’t see much difference for using JEST instead of JASMINE.

```
describe('Testing messenger for ajax call', function()
{
  var injector;

  it('messenger invokes showMessage and then getMessage
of dataservice', function() {

    spyOn(dataservice,
'getMessage').and.callThrough();
    spyOn($, 'ajax');

    messenger.showMessage('123');


expect(dataservice.getMessage).toHaveBeenCalled();

expect(dataservice.getMessage.calls.mostRecent().args
).toEqual([jasmine.any(String),
jasmine.any(Function)]);

    expect($.ajax).toHaveBeenCalled();

  });
```

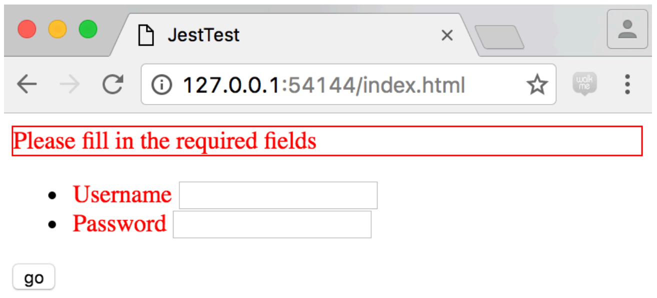
```
});
```

We are able to test the invocation of dataservice.getMessage() but not dataservice.callApi() because it is not exposed and is kind of private method of dataservice module. JASMINE looks good when you want to do state-based unit testing.

V. Testing with JEST

Consider a simple page with some form validation.

```
<style>
  .hidden {display: none}
  .error {color: red}
  .error-box {border: 1px solid red}
</style>
<p class="error error-box hidden" id="err">
 Please fill in the required fields
</p>
<form  onsubmit="return  validateSubmit(this)"  method="post"
action="/cgi-bin/perlbaby.pl">
    <ul>
      <li>
         <label id="username-label" for="username">
          Username
         </label>
        <input id="username">
      </li>
      <li>
<label id="password-label" for="password">Password</label>
          <input id="password"></li>
    </ul>
    <button type="submit" id="button">go</button>
  </form>
  <script>
    /* A simple function to validata form on form submit,
    if any error , show the error message */
    function validateSubmit(f) {
      var validates = true;
      ['username', 'password'].forEach(function(field) {
        if (!document.getElementById(field).value) {
          validates = false;
          document.getElementById(field + '-label').className =
'error';
        } else {
          document.getElementById(field + '-label').className =
'';
        }
      });
      document.getElementById('err').className = validates
        ? 'hidden'
        : 'error error-box';

      if (validates) {
        // fancy stuff goes here
      }

      return false;
    }
  </script>
</body>
</html>
```



Please fill in the required fields

- Username
- Password

Now add a file *package.json* to your app directory and add the following code.

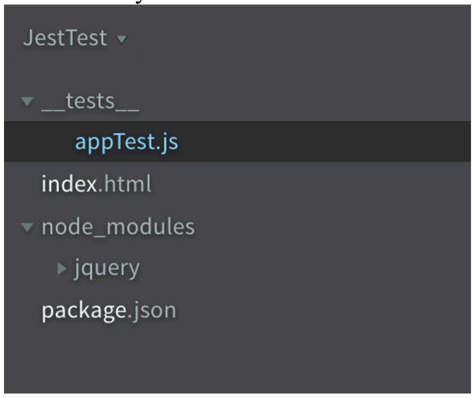
```
{
  "scripts": {
    "test": "jest"
  }
}
```

We are going to use jest for testing our single page app. To use jest install jest using npm

```
$ npm install -g jest
```

-g stands for globally installing the node package

Our directory structure should look like this.



As we are going to use jquery node module for our unit testing, install jquery as well

```
$ npm install jquery -save-dev
```

Source code for appTest.js

```
jest
  .dontMock('fs')
  .dontMock('jquery');
```

JEST mocks each & everything, which is so good for unit testing, because we want to run our unit in isolation. When you don't want to mock something in jest, it is just easy as calling **`dontMock()`** method

```
var $ = require('jquery');
var html =
require('fs').readFileSync('./app.html').toString();

describe('validateSubmits', function() {

  it('shows/hides error banner', function() {
    document.documentElement.innerHTML = html;

    // initial state
    expect($('#err').hasClass('hidden')).toBeTruthy();

    // submit blank form, get an error
    $('form').submit();
    expect($('#err').hasClass('hidden')).toBeFalsy();

    // fill out completely, error gone
    $('#username').val('Bob');
    $('#password').val('123456');
    $('form').submit();
    expect($('#err').hasClass('hidden')).toBeTruthy();
  });

  it('adds/removes error classes to labels', function() {
    document.documentElement.innerHTML = html;

    // initially - no errors
    expect($('#username-
label').hasClass('error')).toBe(false);
    expect($('#password-
label').hasClass('error')).toBe(false);

    // errors
    $('form').submit();
    expect($('#username-label').hasClass('error')).toBe(true);
    expect($('#password-label').hasClass('error')).toBe(true);

    // fill out username, missing password still causes an
error
    $('#username').val('Bob');
    $('form').submit();
    expect($('#username-
label').hasClass('error')).toBe(false);
    expect($('#password-label').hasClass('error')).toBe(true);

    // add the password already
    $('#password').val('123456');
    $('form').submit();
    expect($('#username-
label').hasClass('error')).toBe(false);
    expect($('#password-
label').hasClass('error')).toBe(false);
  });

});
```

To run your test file, just run `$ npm test`

VI. Conclusion

JASMINE may be good in state-based unit testing, but interaction-based unit testing is equally important. I found JEST comfortable and power full in case of interaction-based unit testing. Usually widget or component (react component) which are unit in itself can be well tested with JEST, and probably testing the React Component might be the inspiration for Facebook, to come up with JEST.

VII. References

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#JavaScript
https://en.wikipedia.org/wiki/Jasmine_(JavaScript_testing_framework)
https://github.com/facebook/jest
http://stackoverflow.com/questions/4662641/how-do-i-verify-jquery-ajax-events-with-jasmine
http://www.phpied.com/jest-jquery-testing-vanilla-app/
https://facebook.github.io/jest/docs/api.html
https://coderwall.com/p/pbsi7g/testing-ajax-calls-with-jasmine-2
http://jasmine.github.io/2.0/ajax.html