JavaScript Test-Driven Development with Jasmine and Karma

Cedric Takongmo
cedric.takongmo@mni.thm.de

Abstract

Get a concise introduction to Jasmine, the popular testing framework for JavaScript. This document shows you how to write unit tests with Jasmine that automatically check for bugs in your application using the Test-Driven development pattern and the Karma test runner. After you get an overview of test-driven development, learn how to write specifications for individual components, and then use those specs to test the code you write. Write useful specs by determining what you need to test and what you dont. Test the behavior of new and existing code against the specs you create. Apply Jasmine matchers and discover how to build your own. Organize code suites into groups and subgroups as your code becomes more complex. Use a Jasmine spy in place of a function or an object and learn why its valuable.

1 Credits

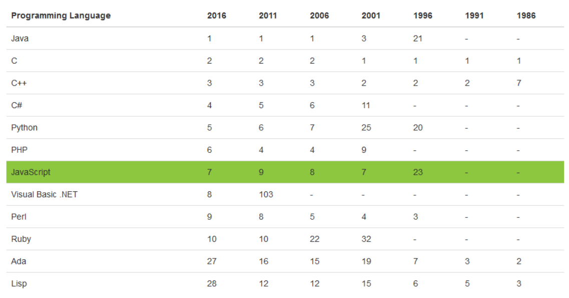
This document has been wrote on the basis of some external Internet links and a presentation of Christopher Bartling about JavaScript tests with Jasmine and Karma. The statistics to illustrate the importance of JavaScript today come from studies of diverse companies like TIOBE software. A detailed presentation of the Jasmine library was inspired by the online documentation on the official Jasmine website.

2 Motivation for Javascript TDD

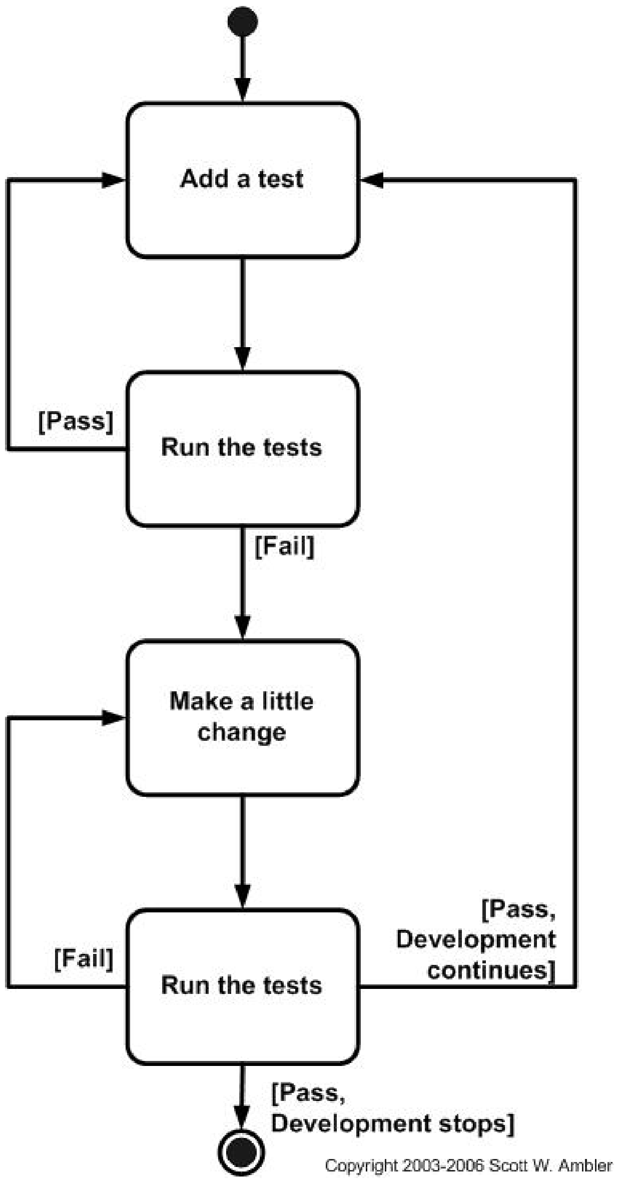
Today, Javascript is growing in popularity and will continue to grow. As show in the Abbildung 1, from 1996 to 2016, JavaScript is from the 23th to the 7th position of the most used programming languages. JavaScript is also used all over the place, and is used even with every backend language. This programming language is also the most commonly used in frontend development to dynamically give more user experience to web applications. JS-libraries are used for the development of web and mobile applications with JQuery Mobile/PhoneGap. Famous libraries like Angular.js, jQuery, Node.js, React and various others tools inherit JavaScript. Some hardware engineer are using JavaScript to do embedded programming so that JavaScript can be used for the direct communication with the outside world by using MicroControllers and other electronics related stuff, although this is definitely less common at the moment. This is why JavaScript is such a good language to learn.

A software product has to be considered like every other commercial product and it quality must be tested. This is also valid for javaScript Software. In the software engineering, we are talking about Code Quality. Code quality is a loose approximation of how long-term useful and long-term maintainable the code is. If the Code is thrown away tomorrow is mean that this Code has a Low quality. High quality Code is being carried over from product to product, developed further, maybe even open sourced after establishing is value. The software quality is defined with a clear and understandable design and implementation and also well defined interfaces. Good Code is ease to build and use and have to be extensible. The minimum extra dependencies and the good documentation are very important criteria for Software Quality. One off he most important criteria are the tests: Unit, integration, and functional/acceptance testing.

In this context of Code Quality and Unit Testing in JavaScript, Test-driven development (TDD) is the new trend. TDD is a new way to software development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation (Martin, Newkirk, and Kess 2003). In other words, its one way to think through your requirements or design before your write your functional code (implying that TDD is both an important agile requirements and agile design technique). Another view is that TDD is a programming technique. As Ron Jeffries likes to say, the goal of TDD is to write clean code that works. I think that there is merit in both arguments, although I lean towards the specification view, but I leave it for you to decide.



Programming Language	2016	2011	2006	2001	1996	1991	1986
Java	1	1	1	3	21	-	-
C	2	2	2	1	1	1	1
C++	3	3	3	2	2	2	7
C#	4	5	6	11	-	-	-
Python	5	6	7	25	20	-	-
PHP	6	4	4	9	-	-	-
JavaScript	7	9	8	7	23	-	-
Visual Basic .NET	8	103	-	-	-	-	-
Perl	9	8	5	4	3	-	-
Ruby	10	10	22	32	-	-	-
Ada	27	16	15	19	7	3	2
Lisp	28	12	12	15	6	5	3


Add a test
[Pass]
Run the tests
[Fail]
Make a little change
[Fail]
Run the tests
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

3 Test-driven development cycle

TDD can be described, as explained in, with this simple formula: TDD = The steps of test first development (TFD) + a Refactoring. TFD are overviewed in the UML activity diagram of Abbildung 2. With this approach a test should be quickly added. Normally just enough code to fail. After that the tests should be run, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. Then, You update your functional code to make it should pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).

4 Benefits of TDD

TDD enables you to take small steps when writing software. This is the most important thing of this concept. This practice is far more productive than attempting to code in large steps. For example, assume some new functional code have been added to code, compiled, and tested. Chances are important that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.

5 Karma

Karma is a test runner for JavaScript that runs on Node.js. It is very well suited to testing any JavaScript projects. Using Karma to run tests using one of many popular JavaScript testing suites (Jasmine, QUnit, Mocha, etc.) and have those tests executed not only in the browsers of your choice, but also on any platform (desktop, phone, tablet.) Karma is highly configurable, integrates with popular continuous integration packages (Jenkins, Travis, and Semaphore) and has excellent plugin support. Karma is an Open Source distribution and actually in the version 1.0.

- Installation - Karma requires Node.js and the Node Package Manager (NPM). So we can install Karma with the simple command:

  $ npm install -g karma

- Configuration - A configuration file have first to be created. Then Karma can do what you want. This configuration file can be a JavaScript or a CoffeeScript file. The configuration file can be created manually or generated step by step via the command line:

  $ karma init

  In this way, the created configuration file is then as follows:

```
module.exports = function(config) {
    config.set({
        //  root path location that will be used to resolve
        //all relative paths in files and exclude sections,
        //should be the root of your project
        basePath: '../',

        // files to include, ordered by dependencies
        files: [
        // include relevant Angular files and libs
        'app/lib/angular/angular.js',
        'test/lib/angular/angular-mocks.js',

        // include js files
        'app/js/app.js',

            // include unit test specs
            'test/unit/*.js'
        ],
        // files to exclude
        exclude: [
         'app/lib/angular/angular-loader.js',
            'app/lib/angular/*.min.js',
            'app/lib/angular/angular-scenario.js'
        ],

        // karma has its own autoWatch feature but
        //Grunt watch can also do this
        autoWatch: false,

        // testing framework, be sure to install
        //the karma plugin
        frameworks: ['jasmine'],

        // browsers to test against, be sure to install
        //the correct karma browser launcher plugin
        browsers: ['Chrome', 'PhantomJS', 'Firefox'],

        // progress is the default reporter
        reporters: ['progress'],

        // map of preprocessors that is used mostly
        //for plugins
        preprocessors: {

        },

        // list of karma plugins
        plugins: [
            'karma-junit-reporter',
            'karma-chrome-launcher',
            'karma-firefox-launcher',
            'karma-jasmine',
            'karma-phantomjs-launcher'
        ]
    })
}
```

Listing 1: Karma configuration file

- Application - Karma is started on the console with the following command:

  $ karma start [path/to/config/file.js]

All tests are now performed and Karma wait for code changes. while tests succeed, Karma will automatically start another test run. In the case of faulty test the result will be show in the console. In this case karma waits for an update or correction of JavaScript code.

6 PhantomJS

PhantomJS (phantomjs.org) is a headless WebKit scriptable with JavaScript. The latest stable release is version 2.1. PhantomJS is an open source, and is distributed under the BSD license. PhantomJS is created and maintained by Ariya Hidayat, with the help of many contributors.

PhantomJS can be used for many purpurses:

- Headless web testing: PhantomJS allows the Lightning-fast testing without the browser!
- Page automation: With the tool it is also possible to Access and manipulate web pages with the standard DOM API, or with any JavaScript library.
- Screen capture. PhantomJS provides a way to Programmatically capture web contents, including CSS, SVG and Canvas.
- Network monitoring. PhantomJS facilitates the Automate performance analysis, tracks page loading and exports it as standard HAR format.

Among those use cases some features are implemented in PhantomJS:

- Multiplatform. This webkit is available on major operating systems like Windows, Mac OS X, Linux, and other Unices.
- It provides a Fast and native implementation of web standards: DOM, CSS, JavaScript, Canvas, and SVG.
- Pure headless (no X11) on Linux is made available, ideal for continuous integration systems. PhantonJS runs also on Amazon EC2, Heroku, and Iron.io.
- PhantomJS is easy to install: Download, unpack, and start having fun in just 5 minutes.

7 Jasmine

Jasmine is an open source Javascript Unit Test Framework. It is behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests. Now we will see how this framework is structured. The main particularities of Jasmine are:

- test Suite has a hierarchical structure,
- tests integrated as specification,
- matchers, both built-in and custom,
- Setup and Teardown,
- Spies, a test double pattern.

7.1 Suites

A test suite represents a group of specifications. The global Jasmine function describe defines a test suite. This function takes two parameters: a string and a closure. The title or name for a spec suite is defined by the string. It is usually what is being tested. A block of code that will be implemented will be wrote in the function.

```
'use strict';

describe('A specification suite
    for Controller: MainCtrl',
    function () {

    ...

});
```

Listing 2: Suites example

7.2 Specs

The global jasmine function it define specs. This function, like describe takes a string and a function. The title of the spec is defined by the string and the function is the spec, or test. One or more expectations that verify the state of the code are contained in a spec. In Jasmine an expectation is an assertion. This assertion can be true or false. A passing spec is a spec with all true expectations. A falling spec is rather a spec with one or more false expectations.

```
describe('Controller: StackCtrl:
    describes an abstract data' +
        'type that serves as a
           collection of
           elements', function
           () {
```

```
        it('specEmpty: if this stack
            is empty; Returns "true"'
             +
           ' only if this stack don t
               contains  items; "
              false" otherwise',
              function () {

            scope.stack = [{
                firstname: "Peter",
                lastName: "Schneider"
                }];
            expect(scope.isEmpty()).
                toBe(false);
            scope.stack = [];
        expect(scope.isEmpty()).toBe(
            true);

    });
});
```

Listing 3: Specs example

Because the describe and it are function, ana executable code necessary to implement the spec can be defined in these blocks. So variables declared in a describe are available to any it block inside the suite.

7.3 Expectations

The function expect is necessary to build expectations. This function takes an argument called the actual and is chained with a matcher function, which takes the expected value. A boolean comparison between the actual and the expected value will be implemented by each matcher. This matcher should report to Jasmine if the expectation is true or false. Then Jasmine decides if the spec pass or fails. With a not before calling the matcher any matcher can evaluate to a negative chaining. Here is a list of most used matchers:

- expect(fn).toThrow(e);
- expect(instance).toBe(instance);
- expect(mixed).toBeDefined();
- expect(mixed).toBeFalsy();
- expect(number).toBeGreaterThan(number);
- expect(number).toBeLessThan(number);
- expect(mixed).toBeNull();
- expect(mixed).toBeTruthy();
- expect(mixed).toBeUndefined();
- expect(array).toContain(member);
- expect(string).toContain(substring);
- expect(mixed).toEqual(mixed);
- expect(mixed).toMatch(pattern);

Negative matchers can be call as followed:

- expect(instance).not.toBe(instance);
- expect(mixed).not.toBeDefined();
- expect(mixed).not.toBeFalsy();

The fail function brings a spec to fail. A failure message or an Error object can be take as parameter.

Custom matching code can be defined using Jasmine. A custom matcher can be defined as a comparison function, which take an actual value and an expected value. Then the custom factory should be passed to Jasmine. The best practice to do that is to pass it in the beforeEach and it will be in scope and available for all of the specs inside a given call to describe. Custom matchers are destroyed between specs. The name of the factory will be the name of the matcher exposed on the return value of the call to expect. This object has a custom matcher named toBeGoofy.

```
function compare(actual, expected) {
    var result = {};
    result.pass = actual <
        expected;
    if (result.pass === false) {
        result.message = function() {
            return "Expected " +
                actual +
                " not to be less than "
                  +
                expected;
        };
    } else {
        result.message =
            function() {
                return "";
            };
    }
    return result;
}
beforeEach(function() {
    jasmine.addMatchers({
        toBeLessThan: function() {
            return {
                compare: compare
            }
        }
    });
});

...

it('Custum matchers', function() {
    expect(2).toBeLessThan(3);
});
```

Listing 4: Custom matchers

7.4 Setup and Teardown

Related specs can be grouped by the describe function. The string parameter is for naming the collection of specs, and will be concatenated with specs to make a specs full name. This is usefull in finding specs in a large suite. specs can be read as full sentences in traditional BDD style, if they are good named in the test blocks. Any duplicated setup and teardown code can be implemented in the global beforeEach, afterEach, beforeAll and afterAll functions. The beforeEach function is called in the describe in which it is called once before each spec. The afterEach function is used once after each spec.

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

    var scope;

    beforeEach(function() {

        scope = new StackCtrl();
        scope.stack = [{
            firstname: "Cedric",
            lastName: "Takongmo"
        }, {
            firstname: "Peter",
            lastName: "Schneider"
        }];

    });

    afterEach(function() {
        scope = null;
    });

    it('specIsEmpty', function() {

        expect(scope.isEmpty()).toBe(
            false);
        scope.stack = [];
        expect(scope.isEmpty()).toBe(
            true);

    });
    it('specPush', function() {
        Scope.push({
            firstname: "Teddy",
            lastName: "Ricardo"
        });
    });

});
```

Listing 5: Setup and Teardown

The Abbildung 4 represents an example for beforeEach and afterEach. In this case a state will be give to the scope object we want to test before each test. Jasmine calls the beforeAll function only once before all the specs in the test suite are run, and the afterAll after all specs finish. These functions can be used to speed up test suites with expensive setup and teardown. Through the this keyword variables between a beforeEach, it, and afterEach function can be shared. Each specs beforeEach/it/ afterEach has the this as the same empty object that is set back to empty for the next specs beforeEach/it/ afterEach.

7.5 Spies

Test double function called spies can be defined by Jasmine. Any function can be stub by a spy ,which can tracks call to it and all arguments. definition area of a spy ist he describe and the it block in which it is defined. The spy will be destroyed after each spec. Some important matcher after a spy action are:

- The toHaveBeenCalled matcher will return true if the spy was called.
- The toHaveBeenCalledTimes matcher will pass if the spy was called the specified number of times.
- The toHaveBeenCalledWith matcher will return true if the argument list matches any of the recorded calls to the spy.

```
describe("A spy", function() {
    var foo, bar = null;

    beforeEach(function() {
        foo = {
            setBar: function(value) {
                bar = value;
            }
        };

        spyOn(foo, 'setBar');

        foo.setBar(123);
        foo.setBar(456, 'another param')
            ;
    });

    it("tracks that the spy was called",
        function() {
        expect(foo.setBar).
            toHaveBeenCalled();
    });

    it("tracks all the arguments of its
        calls", function() {
        expect(foo.setBar).
            toHaveBeenCalledWith(123);
        expect(foo.setBar).
            toHaveBeenCalledWith(456,
            'another param');
    });

    it("stops all execution on a function", function() {
        expect(bar).toBeNull();
    });
});
```

Listing 6: Spy usage

More others functions can be call by chaining the spy with:

- and.callThrough: the spy will still track all calls to it but in addition it will delegate to the actual implementation.
- and.returnValue: all calls to the function will return a specific value.
- and.callFake: all calls to the spy will delegate to the supplied function.
- and.throwError: all calls to the spy will throw the specified value as an error.
- and.stub: the original stubbing behavior can be returned at any time.

8 Karma-coverage

Karma-coverage is a npm peer dependency to generate test report using the Istanbul library. This plugin can be simply installed by the command:

$ npm install karma karma-coverage –save-dev

Then the developper can configurate the plugin using the karma.conf.je file. The Abbildung 7 shows how to configurate a karma-coverage plugin after install it to genarate Test reports.

```
// karma.conf.js
module.exports = function(config) {
    config.set({
        files: [
            'src/**/*.js',
            'test/**/*.js'
        ],
        reporters: ['progress', 'coverage'],
        preprocessors: {
            'src/**/*.js': ['coverage']
        },
        coverageReporter: {
            // specify a common output directory
            dir: 'build/reports/coverage',
            reporters: [
                // reporters not supporting the `file` property
                {
                    type: 'html',
                    subdir: 'report-html'
                }, {
                    type: 'lcov',
                    subdir: 'report-lcov'
                },
                // reporters supporting the `file` property, use `subdir` to directly
                // output them in the `dir` directory
                {
                    type: 'cobertura',
                    subdir: '.',
                    file: 'cobertura.txt'
                }, {
                    type: 'lcovonly',
                    subdir: '.',
                    file: 'report-lcovonly.txt'
                }, {
                    type: 'teamcity',
                    subdir: '.',
                    file: 'teamcity.txt'
                }, {
                    type: 'text',
                    subdir: '.',
                    file: 'text.txt'
                }, {
                    type: 'text-summary',
                    subdir: '.',
                    file: 'text-summary.txt'
                },
            ]
        }
    });
};
```

Listing 7: Karma-coverage configuration's file

9 TDD - Best Practices

In this part some of the best practices to be followed in TDD projects are presented:

- Avoid functional complexity
- Focus on what you need to achieve
- Maintain code austerity: Ensure your code has just enough meat to satisfy your test case
- Test repeatedly
- Maintain code sanctity
- Application knowledge
- Know when to use TDD

10 Summary

References

When do developers use JavaScript and why? - Quora. [Internet]. [cited 2016 Jul 20]. Available from: *https://www.quora.com/When-do-developers-use-JavaScript-and-why*

How to define code quality - Quora. [Internet]. [cited 2016 Jul 20]. Available from: *https://www.quora.com/How-do-you-define-code-quality*

Kent Beck 2003. *Test Driven Development: By Example*, 1st Edition. Addison Wesley, Boston, MA.

David Astels 2003. *Test-Driven Development: A Practical Guide: A Practical Guide*, 1st Edition. Pearson Education, Upper Saddle River, NJ.

Max Guernsey III 2013. *Test-Driven Database Development: Unlocking Agility (Net Objectives Lean-Agile Series)*, 1st Edition. Pearson Education, USA.

Gojko Adzic 2011. *Specification by Example: How Successful Teams Deliver the Right Software*, 1st Edition. Manning Publications Co., Shelter Island, NJ.

Introduction to Test Driven Development (TDD). [Internet]. Ambysoft Inc. 2013 [cited 2016 Jul 20]. Available from: *http://agiledata.org/essays/tdd.html#PartingThoughts*

Karma - a Javascript Test Runner. [Internet]. Michael G Bielski. 2013 [cited 2016 Jul 20]. Available from: *http://www.methodsandtools.com/tools/karma.php*

JavaScript Tests mit Karma schnell an den Start bringen - Mayflower Blog. [Internet]. Norbert Schmidt. 2013 [cited 2016 Jul 20]. Available from: *https://blog.mayflower.de/4333-Karma-Testrunner-Einfuehrung.html*

phantomjs/README.md at master ariya/phantomjs [Internet]. Ariya Hidayat. 01.2016 [cited 2016 Jul 20]. Available from: *https://github.com/ariya/phantomjs*

custom_matcher.js [Internet]. Pivotal Labs. 2016 [cited 2016 Jul 20]. Available from: *http://jasmine.github.io/edge/custom_matcher.html*

introduction.js [Internet]. Pivotal Labs. 2016 [cited 2016 Jul 20]. Available from: *http://jasmine.github.io/2.0/introduction.html#section-Spies*

Karma-runner/karma-coverage: A Karma plugin. Generate code coverage. [Internet]. Friedel Ziegelmayer. 07.2016 [cited 2016 Jul 20]. Available from: *https://github.com/karma-runner/karma-coverage*