How to test your JavaScript

Emil Wall

May 29, 2013

The final version will have title page and endpaper generated from
`http://pdf.teknik.uu.se/pdf/exjobbsframsida.php` and
`http://pdf.teknik.uu.se/pdf/abstract.php`.
Hence, this page and the abstract are temporary, to be replaced in the final version.

Abstract

Abstract goes here... Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam sollicitudin varius libero ac consectetur. Nullam ornare, massa et sagittis consectetur, neque mi scelerisque arcu, in fringilla lectus risus non arcu. Suspendisse vestibulum tellus id mauris lacinia non hendrerit nibh tempor. Proin tempor interdum justo et elementum. Ut ultricies adipiscing ipsum et pharetra. Vestibulum pretium luctus est, quis egestas augue luctus et. Praesent volutpat pharetra lectus vitae elementum.

Integer fringilla ligula eu sem semper tincidunt. Nullam mi lacus, blandit non sollicitudin eget, tempor eu ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi ornare sem et purus consequat ac adipiscing nunc tincidunt. Curabitur nisi ante, ornare vel adipiscing et, scelerisque vitae erat. Etiam blandit egestas magna, quis dapibus nulla euismod quis. Sed interdum interdum malesuada. Suspendisse lacinia imperdiet laoreet. Maecenas ullamcorper laoreet nunc ac egestas. Cras consequat elit eu lacus sollicitudin ut pharetra magna venenatis. Suspendisse scelerisque condimentum pulvinar. Mauris ut tellus sit amet nulla porttitor tristique. Suspendisse eleifend erat sed nisi lacinia eu lacinia metus porta. Nulla pretium, risus eget semper laoreet, dolor odio malesuada eros, at mattis enim turpis gravida felis. Aliquam adipiscing varius nibh, ac auctor eros bibendum non.

Acknowledgment

Thanks goes to my supervisor Jimmy Larsson for providing me with valuable feedback and connections, to my reviewer Roland Bol for guiding me through the process and giving useful and constructive comments on my work, to all my wonderful colleagues at Valtech which never fails to surprise me with their helpfulness and expertise, to my girlfriend Matilda Kant for her endurance and support and to my family and friends (and cats!) for all the little things that ultimately matters the most.

Contents

1 Introduction

The testing community around JavaScript still has some ground to cover. The differences in testing ambitions becomes especially clear when compared to other programming communities such as Ruby and Java. As illustrated by Mark Bates[1]:

> "Around the beginning of 2012, I gave a presentation for the Boston Ruby Group, in which I asked the crowd of 100 people a few questions. I began, 'Who here writes Ruby?' The entire audience raised their hands. Next I asked, 'Who tests their Ruby?' Again, everyone raised their hands. "Who writes JavaScript or CoffeeScript?' Once more, 100 hands rose. My final question: 'Who tests their JavaScript or CoffeeScript?' A hush fell over the crowd as a mere six hands rose. Of 100 people in that room, 94% wrote in those languages, but didn't test their code. That number saddened me, but it didn't surprise me."

JavaScript is a scripting language primarily used in web browsers to perform client-side actions not feasible through plain HTML and CSS. Due to the dynamic nature of the language, there is typically little static analysis performed on JavaScript code compared to code written in a statically typed compiled language. Granted, there are tools available such as JSLint, JavaScript Lint, JSure, the Closure compiler, JSHint and PHP CodeSniffer. JSLint is perhaps the most popular of these and does provide some help to avoid common programming mistakes, but does not perform flow analysis[2] and type checking as a fully featured compiler would do, rendering proper testing routines the appropriate measure against programming mistakes. After all, there are benefits of testing code in general, for reasons that we will come back to, but JavaScript is particularly important to test properly due to its dynamic properties and poor object orientation support. Despite the wide variety of testing frameworks that exists for JavaScript, it is generally considered that few[1] developers use them. The potential risk of economic loss associated with untested code being put into production, due to undetected bugs, shortened product lifetime and increased costs in conjunction with further development and maintenance, constitutes the main motivation for this thesis.

The economic risk of having untested JavaScript is especially high when the code is a prerequisite for, or part of, business critical operations. This is presumably becoming increasingly common since more than 90 % of today's websites use JavaScript[3]. For instance, application failure for a webshop may cause loss of orders and any web site that is perceived as broken can harm trademarks associated with it and change people's attitude for the worse. Moreover, when automatic regression tests are missing, making changes to the code is error prone. Issues related to browser compatibility or subtle dependencies between functions and events are easily overlooked instead of being detected by high quality tests prior to setting the site into production. Manually testing a web page with all the targeted combination of browsers, versions and system platforms is not a viable option[4] so multi-platform automated testing is required.

High quality tests are maintainable and test the right thing. If these conditions are not met, responding to changes is harder, and the tests will tend to cause frustration among the developers instead of detecting bugs and driving the understanding and development

of the software[5]. The criteria for maintainability in this context are that the tests should have low complexity (typically short test methods without any control flow), consist of readable code, use informative variable names, have reasonably low level of repeated code (this can be accomplished through using Test Utility Methods[6, p. 599]), be based on interfaces rather than a specific implementation and have meaningful comments (if any). Structuring the code according to a testing pattern such as the Arrange-Act-Assert[7] pattern and writing the code so that it reads like sentences can help in making the code more readable, in essence by honouring the communicate intent principle[6, p. 41]. Testing the right thing means focusing on the behaviour that provides true business value rather than trying to fulfil some coverage criteria, testing that the specification is fulfilled rather than a specific implementation and to find a balance in the amount of testing performed in relation to the size of the system under test. Typically some parts of the system will be more complex and require more rigorous testing but there should be some level of consistency in the level of ambition regarding testing across the entire application. Specifically, if some part of the code is hard to test it is likely to be beneficial in the long run to refactor the design to provide better testability than to leave the code untested.

Unit testing is particularly powerful when run in combination with integration test in a CI build[1]. Then you are able to harness the power of CI, avoiding errors otherwise easily introduced as changes propagate and affect other parts of the system in an unexpected way. This will make developers changing parts of the system that the JavaScript depends upon aware if they are breaking previous functionality.

Testing JavaScript paves the way for test-driven development, which brings benefits in terms of the design becoming more refined and increased maintainability. Tests can serve as documentation for the code and forcing it to be written in a testable manner, which in itself tends to mean adherence to key principles such as separation of concerns, and single responsibility.

The goal with this thesis is to investigate why JavaScript testing is performed to such a small extent today, and what potential implications an increased amount of testing could provide for development and business value to customers. Providing possible approaches to testing JavaScript under different conditions are also part of the goal.

Writing tests for JavaScript is nothing new, the first known testing framework JsUnit was created in 2001 by Edward Hieatt[8, 9] and since then several other test framework has appeared such as QUnit [10] and JsUnits sequel Jasmine [11], as well as tools for mocking[2] such as Sinon.JS[12]. It seems as if the knowledge of how to smoothly get started, how to avoid making the tests non-deterministic and time consuming, and what to test, is rare. Setting up the structure needed to write tests is a threshold that most JavaScript programmers do not overcome[1] and thus, they lose the benefits, both short and long term, otherwise provided by testing.

In guides on how to use different JavaScript testing frameworks, examples are often decoupled from the typical use of JavaScript - the Web. They tend to merely illustrate

[1] Continuous Integration build servers are used for automatic production launch

[2] mocking and stubbing involves simulation of behavior of real objects in order to isolate the system under test from external dependencies

testing of functions without side effects and dependencies. Under these circumstances, the testing is trivial and most JavaScript programmers would certainly be able to put up a test environment for such simple code. In contrast, the problem domain of this thesis is to focus on how to test the behaviour of JavaScript that manipulates DOM elements (Document Object Model, the elements that html code consists of), interacts with databases and fetches data using asynchronous calls, as well as when and why you should do it.

2 Description of Work

Researching today's limited testing of JavaScript may be done from a multiple different points of view. There are soft aspects such as:

- Differences in attitudes towards testing between different communities and professional groups
- How JavaScript is typically conceived as a language and how it is used
- Knowledge about testing among JavaScript developers
- Economic viability and risk awareness

There are also more technical aspects:

- Testability of JavaScript code written without tests in mind
- Usability of testing tools and frameworks
- Reasons not to include frameworks in a project for the sole purpose of facilitating testing
- Limitations in what can be tested
- Complexity in setting up the test environment; installing frameworks, configuring build server, exposing functions to testing but not to users in production, etc.

2.1 Consequences of JavaScript testing

There are consequences (good and bad) of testing JavaScript both from a short and from a longer perspective. The development process is affected; through time spent thinking about and writing tests, shorter feedback loops, executable documentation and new ways of communicating requirements with customers. The business value of the end result is also likely to be affected, as well as the quality and maintainability of the code. Ideally, the pace of development does not stagnate and making changes becomes easier when the application is supported by a rigorous set of tests. The extra time required to set up the test environment and write the actual tests may or may not turn out to pay off, depending on how the application will be used and maintained.

2.2 Covering common and advanced cases

Accounting for how to conveniently proceed with JavaScript testing should cover not only the simplest cases but also the most common and the hardest ones, preferably while also providing evaluation and introduction to available tools and frameworks. Many introductions and tutorials found for the testing frameworks today tends to focus on the simple cases of testing, possibly because making an impression that the framework is simple to use has been more highly prioritised than covering different edge cases of how it can be used that might not be relevant to that many anyway. To provide valuable guidance in how to set up a testing environment and how to write the tests, attention must be paid to the varying needs of different kinds of applications. It is also important to keep in mind that the tests should be as maintainable as the system under test, to minimise maintenance costs and maximise gain.

3 Methods

The methods used are first and foremost qualitative in nature, in order to prioritise insight into the problem domain above quantitatively verifying hypotheses. The chance of finding out the true reasons to why JavaScript is tested to such a small extent increases with open questions. Specifically, aside from literature studies, the main method of this thesis work has been to perform and analyse interviews of JavaScript programmers (mainly those concerned with user interface). There has also been some hands on evaluation of tools and frameworks, and assessment of testability and impact of adding tests to existing projects. In order to describe methods of writing tests for JavaScript, the practical work involved testing an existing application, performing TDD as described in Test-driven JavaScript Development[13] and doing some small TDD projects during the framework evaluation. Another method used was a workshop field study, where programmers were allowed to work in pairs to solve pre-defined problems using TDD.

The following testing frameworks have been evaluated: Jasmine[11] (+ Jasmine-species[14]), qUnit[10], Karma, Mocha[15], JsTestDriver[16], Buster.JS[17] and Sinon.JS[12]. The code written while evaluating the frameworks is publicly available as git repositories under my github account *emilwall*, together with the LaTeX code for this report.

4 Methods

Semi-structured interviews were used rather than surveys to gather individual views on the subject. This approach allowed for harnessing unique as well as common experiences which would not be picked up in a standardised survey.

4.1 Interview Questions

4.1.1 Formalities

The interviews took place in calm, undisturbed locations, and began with a short recap on the background and purpose of the interviews. The interviewee was informed that the purpose of the interview was to gain a better understanding of different aspects of JavaScript testing. What problems and benefits exists and how it is connected with other software engineering practices and tools.

- Is it ok if I record our conversation?
- Do you want to be anonymous?

4.1.2 The interviewee

- What kind of applications do you typically develop with JavaScript?
- What tools and frameworks have you used? What roles have they played in your development processes?
- Which are your favourites among the frameworks? Why?

4.1.3 JavaScript in general

- How productive do you feel when coding in JavaScript compared to other languages?
- How do you typically perceive JavaScript code written by others?
- What advanced features of JavaScript do you use, such as prototypal inheritance, dynamic typing and closures?
- How do you think the JavaScript syntax and features impacts maintainability?
- How would you assess the probability of making mistakes while coding in JavaScript?

4.1.4 JavaScript testing experience

- What is your experience with unit testing of JavaScript?
- What is your experience with UI testing?
- What is your experience with integration and end-to-end tests?
- Have you practiced test driven development with JavaScript? To what extent? Has this been helpful? (if not, why? what did you do instead?)
- Have you used any mocking and stubbing tools? Which, and what has been your experience with these?

4.1.5 Challenges in testing

- How do you go about determining what to test?
- What principles do you apply when writing the tests? (short test methods, avoiding control flow, code duplication)
- Have you ever set up a testing environment? If so, did you find it hard? If not, do you imagine it to be difficult?

4.1.6 Benefits of testing

- In your opinion, what are the main benefits from testing your JavaScript?
- When do you think testing JavaScript pays off?
- Have you ever had tests that impaired your productivity by being too hard to change or even understand?
- Has tests helped you in debugging and quickly finding the source of a bug?
- Has testing helped you discover bugs in the first place? Has this saved you from trouble further on?
- Has testing helped your design?
- What role has JavaScript testing played in any continuous integration you've had?
- What type of JavaScript coding do you think is best suited for TDD?

4.1.7 Adding tests to existing application

- Have you ever been given the task of adding tests to an existing (JavaScript) application?
- Was this hard?
- What changes in the application were required in order to be able to write the tests?
- Did you feel safe in changing the application or were you afraid that you'd might introduce new bugs?

5 Patterns

Rather than proposing best practices for JavaScript testing, the reader should be made aware that different approaches are useful under different circumstances. This applies both to choice of tools and how to organise the tests.

6 Draft

Considering all the different options in available frameworks, one is easily deceived into believing that the main reason why people don't test their JavaScript is because they are lazy or uninformed. This is not necessarily true, there are respectable obstacles for doing TDD both in the process of fitting the frameworks into your application and in writing the JavaScript code in a testable way.

For instance, when setting up JsTestDriver (JSTD)[16] with the Jasmine adapter there are pitfalls in which version you're using. At the time of writing, the latest version of the Jasmine JSTD adapter (1.1) is not compatible with the latest version of Jasmine (1.3.1), so in order to use it you need to find an older version of Jasmine (such as 1.0.1 or 1.1.0) or figure out how to modify the adapter to make it compatible. Moreover, the latest version of JSTD (1.3.5) does not support relative paths to parent folders when referencing script files in jsTestDriver.conf although a few older versions do (such as 1.3.3d), which is a problem if you want to place the test driver separate from the system under test rather than in a parent folder, or if you want to reference another framework such as Jasmine if it is placed in another directory.

Regardless whether or not the frameworks are effortlessly installed and configured or not, there is still the issue of testability. It is common to argue that TDD forces developers to write testable code which tends to be maintainable. This is true in some respects, but one has to bear in mind that JavaScript is commonly used with many side-effects that may not be easily tested. More importantly, it is common to place all the JavaScript code in a single file and hide the implementation using some variant of the module pattern[18, p. 40], which means that only a small subset of the code is exposed as globally accessible functions, commonly functions that are called to initialize some global state such as event listeners. In order to test the functions, they need to be divided into parts, which will typically have to be more general in order to make sense as stand-alone modules. This conflicts with the eagerness of most developers to just get something that works without making it more complicated than necessary.

The fundamental problem is probably that most developers are used to manually test their JavaScript in a browser. This gives an early feedback loop and although it does not come with the benefits of design, quality and automated testing that TDD does, it tends to give a feeling of not doing any extra work and getting the job done as fast as possible. Developers do not want to spend time on mocking dependencies when they are not sure that the solution they have in mind will even work. Once an implementation idea pops up, it can be tempting to just try it out rather than writing tests. If this approach is taken, it may feel like a superfluous task to add tests afterwards since that will typically require some refactoring in order to make the code testable. If the code seems to work good enough, the developer may not be willing to introduce this extra overhead. There is also a risk involved in refactoring untested code[19, p. 17], since manually checking that the refactoring does not introduce bugs is time consuming and difficult to do well, although there is an exception when the refactoring is required in order to add tests. This is because leaving the code untested means even greater risk of bugs and the refactoring may be necessary in the future anyway, in which case it will be

even harder and more error-prone.

7 AngularJS, Jasmine and Karma

The AngularJS framework uses Jasmine and Karma in the official tutorial.

> "Since testing is such a critical part of software development, we make it easy to create tests in Angular so that developers are encouraged to write them"[20]

This is likely a large contributing factor for increasing the probability of Angular developers testing their JavaScript.

8 Definitions

A mock has pre-programmed expectations and built-in behaviour verification[13, p. 453].

Because JavaScript has no notion of interfaces, it is easy to accidentally use the wrong method name or argument order when stubbing a function[13, p. 471].

9 Real world experiences

9.1 Meetup open space discussion

During a talk at a meetup on python APIs (2013-05-22 at Tictail's office, an e-commerce startup based in Stockholm), the speaker mentioned that their application depended heavily on JavaScript. It turned out that they had done some testing efforts but without any lasting results. During the open space after the talks, testing became a discussion subject in a group consisting of one of the developers of the application, among others. The developer explained that they had been unable to unit test their JavaScript because the functionality was so tightly coupled that the only observable output that they could possibly test was the appearance of the web page, via Selenium tests. He sought reassurance that they had done the right thing when deciding not to base their testing on Selenium due to instability (tests failing for the wrong reasons) and time required to run the tests. He also sought answers to how they should have proceeded.

The participants in the discussion were in agreement that testing appearance is the wrong way to go and that tests need to be fast and reliable. The experience with testing frameworks seemed to vary, some had used Jasmine and appreciated its behaviour driven approach and at least one had used Karma but under its former name Testacular. The idea that general JavaScript frameworks such as AngularJS could help in making code testable and incorporating tests as a natural part of the development was not frowned upon. The consensus seemed to be that in general, testing JavaScript is good if done right, but also difficult.

9.2 Ideas spawned when talking about this thesis

During my work on this thesis, I have explained to numerous people what it is that I'm doing. Typically, I've started out with saying something like "I'm looking at testing of JavaScript´´. Depending on if the person asking knows a lot about JavaScript or not, the conversation then might proceed in different directions, but the most common follow up is that I explain further that I'm looking at why people don't do it, when and how they should do it and what the problems and benefits are. Especially I'm looking at the problems.

One not so uncommon response is that testing of JavaScript probably is so uncommon because people programming in JavaScript often have a background as web graphic designers, without that much experience of automated testing. Another common conception is that JavaScript in practise is usually not testable because it has too much to do with the front-end parts of an application, so tests are inevitably slow, unmaintainable and/or unreliable because of the environment they have to run in.

References

[1] Mark Bates. *Testing Your JavaScript/CoffeeScript*. Last checked: April 9, 2013. URL: `http://www.informit.com/articles/article.aspx?p=1925618`.

[2] Douglas Crockford. *JSLint - The JavaScript Code Quality Tool*. Last checked: May 1, 2013. URL: `http://www.jslint.com/lint.html`.

[3] W3Techs - World Wide Web Technology Surveys. *Usage of JavaScript for websites*. Last checked: April 9, 2013. URL: `http://w3techs.com/technologies/details/cp-javascript/all/all`.

[4] John Resig. *JavaScript testing does not scale*. Last checked: April 9, 2013. URL: `http://ejohn.org/blog/javascript-testing-does-not-scale/`.

[5] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. PRENTICE HALL, 2009. ISBN: 9780132350884.

[6] Gerard Meszaros. *xUnit Test Patterns - refactoring test code*. ADDISON-WESLEY, 2007. ISBN: 9780131495050.

[7] Cunningham & Cunningham Inc. *Arrange Act Assert*. Last checked: April 24, 2013. URL: `http://c2.com/cgi/wiki?ArrangeActAssert`.

[8] Edward Hieatt and Robert Mee. "Going Faster: Testing The Web Application". In: *IEEE Software* (Mar. 2002), p. 63.

[9] Github. *pivotal/jsunit*. Last checked: April 3, 2013. URL: `https://github.com/pivotal/jsunit`.

[10] The jQuery Foundation. *QUnit: A JavaScript Unit Testing framework*. Last checked: April 3, 2013. URL: `http://qunitjs.com/`.

[11] Running documentation. *Jasmine is a behavior-driven development framework for testing JavaScript code*. Last checked: April 3, 2013. URL: `http://pivotal.github.com/jasmine/`.

[12] Christian Johansen. *Sinon.JS: Standalone test spies, stubs and mocks for JavaScript*. Last checked: April 5, 2013. URL: `http://sinonjs.org/`.

[13] Christian Johansen. *Test-Driven JavaScript Development*. ADDISON-WESLEY, 2010. ISBN: 9780321683915.

[14] Rudy Lattae. *Jasmine-species: Extended BDD grammar and reporting for Jasmine*. Last checked: April 5, 2013. URL: `http://rudylattae.github.com/jasmine-species/`.

[15] TJ Holowaychuk. *mocha - simple, flexible, fun javascript test framework for node.js & the browser*. Last checked: April 3, 2013. URL: `http://visionmedia.github.com/mocha/`.

[16] Cory Smith et al. *JsTestDriver*. Last checked: April 5, 2013. URL: `https://code.google.com/p/js-test-driver/`.

[17] August Lilleaas and Christian Johansen. *BusterJS: A powerful suite of automated test tools for JavaScript*. Last checked: April 5, 2013. URL: `http://docs.busterjs.org/`.

[18] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN: 9780596517748.

[19] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. ADDISON-WESLEY, 1999. ISBN: 0201485672.

[20] Igor Minar, Misko Hevery, and Vojta Jina. *Angular Templates*. Last checked: May 2, 2013. URL: `http://docs.angularjs.org/tutorial/step_02`.