Christian Johansen

Test-Driven JavaScript Development

Developer's Library

Test-Driven JavaScript Development

Test-Driven JavaScript Development

Christian Johansen

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Johansen, Christian, 1982-
Test-driven JavaScript development / Christian Johansen.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-68391-5 (pbk. : alk. paper)
ISBN-10: 0-321-68391-9 (pbk. : alk. paper)
1. JavaScript (Computer program language) I. Title.
QA76.73.J39J64 2011
005.13'3–dc22 2010027298

ISBN-13: 978-0-321-68391-5

ISBN-10: 0-321-68391-9

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Second printing, May 2012

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
John Fuller

Project Editor
Madhu Bhardwaj,
Glyph International

Project Coordinator
Elizabeth Ryan

Copy Editor
Mike Read

Indexer
Robert Swanson

Proofreader
David Daniels

Technical Reviewers
Andrea Giammarchi
Joshua Gross
Jacob Seidelin

Cover Designer
Gary Adair

Compositor
Glyph International

To Frøydis and Kristin, my special ladies.

This page intentionally left blank

Contents

Preface

Author's Vision for the Book

Over the recent years, JavaScript has grown up. Long gone are the glory days of "DHTML"; we are now in the age of "Ajax," possibly even "HTML5." Over the past years JavaScript gained some killer applications; it gained robust libraries to aid developers in cross-browser scripting; and it gained a host of tools such as debuggers, profilers, and unit testing frameworks. The community has worked tirelessly to bring in the tools they know and love from other languages to help give JavaScript a "real" development environment in which they can use the workflows and knowledge gained from working in other environments and focus on building quality applications.

Still, the JavaScript community at large is not particularly focused on automated testing, and test-driven development is still rare among JavaScript developers—in spite of working in the language with perhaps the widest range of target platforms. For a long time this may have been a result of lacking tool support, but new unit testing frameworks are popping up all the time, offering a myriad of ways to test your code in a manner that suits you. Even so, most web application developers skimp on testing their JavaScript. I rarely meet a web developer who has the kind of confidence to rip core functionality right out of his application and rearrange it, that a strong test suite gives you. This confidence allows you to worry less about breaking your application, and focus more on implementing new features.

With this book I hope to show you that unit testing and test-driven development in JavaScript have come a long way, and that embracing them will help you write better code and become a more productive programmer.

What This Book is About

This book is about programming JavaScript for the real world, using the techniques and workflow suggested by Test-Driven Development. It is about gaining confidence in your code through test coverage, and gaining the ability to fearlessly refactor and organically evolve your code base. It is about writing modular and testable code. It is about writing JavaScript that works in a wide variety of environments and that doesn't get in your user's way.

How This Book is Organized

This book has four parts. They may be read in any order you're comfortable with. Part II introduces a few utilities that are used throughout the book, but their usage should be clear enough, allowing you to skip that part if you already have a solid understanding of programming JavaScript, including topics such as unobtrusive JavaScript and feature detection.

Part I: Test-Driven Development

In the first part I'll introduce you to the concept of automated tests and test-driven development. We'll start by looking at what a unit test is, what it does, and what it's good for. Then we'll build our workflow around them as I introduce the test-driven development process. To round the topic off I'll show you a few available unit testing frameworks for JavaScript, discuss their pros and cons, and take a closer look at the one we'll be using the most throughout the book.

Part II: JavaScript for Programmers

In Part II we're going to get a deeper look at programming in JavaScript. This part is by no means a complete introduction to the JavaScript language. You should already either have some experience with JavaScript—perhaps by working with libraries like jQuery, Prototype, or the like—or experience from other programming languages. If you're an experienced programmer with no prior experience with JavaScript, this part should help you understand where JavaScript differs from other languages, especially less dynamic ones, and give you the foundation you'll need for the real-world scenarios in Part III.

If you're already well-versed in advanced JavaScript concepts such as closures, prototypal inheritance, the dynamic nature of `this`, and feature detection, you may want to skim this part for a reminder, or you may want to skip directly to Part III.

While working through some of JavaScript's finer points, I'll use unit tests to show you how the language behaves, and we'll take the opportunity to let tests drive us through the implementation of some helper utilities, which we'll use throughout Part III.

Part III: Real-World Test-Driven Development in JavaScript

In this part we'll tackle a series of small projects in varying environments. We'll see how to develop a small general purpose JavaScript API, develop a DOM dependent widget, abstract browser differences, implement a server-side JavaScript application, and more—all using test-driven development. This part focuses on how test-driven development can help in building cleaner API's, better modularized code and more robust software.

Each project introduces new test-related concepts, and shows them in practice by implementing a fully functional, yet limited piece of code. Throughout this part we will, among other things, learn how to test code that depends on browser API's, timers, event handlers, DOM manipulation, and asynchronous server requests (i.e., "Ajax"). We will also get to practice techniques such as stubbing, refactoring, and using design patterns to solve problems in elegant ways.

Throughout each chapter in this part, ideas on how to extend the functionality developed are offered, giving you the ability to practice by improving the code on your own. Extended solutions are available from the book's website.[1]

I've taken great care throughout these projects to produce runnable code that actually does things. The end result of the five chapters in Part III is a fully functional instant messaging chat client and server, written exclusively using test-driven development, in nothing but JavaScript.

Part IV: Testing Patterns

The final part of the book reviews some of the techniques used throughout Part III from a wider angle. Test doubles, such as mocks and stubs, are investigated in closer detail along with different forms of test verification. Finally, we review some guidelines to help you write good unit tests.

Conventions Used in This Book

JavaScript is the name of the language originally designed by Brendan Eich for Netscape in 1995. Since then, a number of alternative implementations have

---

1. http://tddjs.com

surfaced, and the language has been standardized by ECMA International as ECMA-262, also known as ECMAScript. Although the alternative implementations have their own names, such as Microsoft's JScript, they are generally collectively referred to as "JavaScript," and I will use JavaScript in this sense as well.

Throughout the text, monospaced font is used to refer to objects, functions, and small snippets of code.

Who Should Read This Book

This book is for programmers—especially those who write, or are interested in writing JavaScript. Whether you're a Ruby developer focusing primarily on Ruby on Rails; a Java or .Net developer working with web applications; a frontend web developer whose primary tools are JavaScript, CSS, and HTML; or even a backend developer with limited JavaScript experience, I hope and think you will find this book useful.

The book is intended for web application developers who need a firmer grasp of the finer details of the JavaScript language, as well as better understanding on how to boost their productivity and confidence while writing maintainable applications with fewer defects.

Skills Required For This Book

The reader is not required to have any previous knowledge of unit testing or test-driven development. Automated tests are present through the whole book, and reading should provide you with a strong understanding of how to successfully use them.

Equally, the reader is not required to be a JavaScript expert, or even intermediate. My hope is that the book will be useful to programmers with very limited JavaScript experience and savvy JavaScripters alike. You are required, however, to possess some programming skills, meaning that in order to fully enjoy this book you should have experience programming in some language, and be familiar with web application development. This book is not an introductory text in any of the basic programming related topics, web application-specific topics included.

The second part of the book, which focuses on the JavaScript language, focuses solely on the qualities of JavaScript that set it apart from the pack, and as such cannot be expected to be a complete introduction to the language. It is expected that you will be able to pick up syntax and concepts not covered in this part through examples using them.

In particular, Part II focuses on JavaScript's functions and closures; JavaScript's object model, including prototypal inheritance; and models for code-reuse. Additionally, we will go through related programming practices such as unobtrusive JavaScript and feature detection, both required topics to understand for anyone targeting the general web.

About the Book's Website

The book has an accompanying website, http://tddjs.com. At this location you will find all the code listings from the book, both as zip archives and full Git repositories, which allow you to navigate the history and see how the code evolves. The Git repositories are especially useful for the Part III sample projects, where a great deal of refactoring is involved. Navigating the history of the Git repositories allows you to see each step even when they simply change existing code.

You can also find my personal website at http://cjohansen.no in which you will find additional articles, contact information, and so on. If you have any feedback regarding the book, I would love to hear back from you.

This page intentionally left blank

Acknowledgments

Quite a few people have made this book possible. First of all I would like to commend Trina MacDonald, my editor at Addison-Wesley, for being the one who made all of this possible. Without her, there would be no book, and I deeply appreciate her initiative as well as her ongoing help and motivation while I stumblingly worked my way through my first book.

I would also like to extend my gratitude toward the rest of the team working with me on this book; Songlin Qiu for making sure the text is comprehensible and consistent, and for keeping sane while reviewing a constantly changing manuscript. Her insights and suggestions have truly made the book better than I could ever manage on my own. The same can be said for my technical reviewers, Andrea Giammarchi, Jacob Seidelin, and Joshua Gross. Their impressive attention to detail, thoughtful feedback, and will to challenge me have helped clarify code, remove errors, and generally raise the quality of both code samples and surrounding prose, as well as the structure of the book. Last, but not least, Olivia Basego helped me cope with the administrative side of working with a publisher like Addison-Wesley and some challenges related to living in Norway while writing for an American publisher.

Closer to home, my employers and coworkers at Shortcut AS deserve an honorable mention. Their flexibility in allowing me to occasionally take time off to write and their genuine interest in the book at large have been very motivating and key to finishing the manuscript in time. In particular I would like to thank Marius Mårnes Mathiesen and August Lilleaas for frequent discussions of a truly inspiring and insightful nature, as well as feedback on early drafts.

Last, but definitely not least; Frøydis and Kristin, friends and bandmates who have given me space to complete this project and stayed patient while I've been zombie-like tired after long nights of writing, unavailable for various occasions, and generally chained to the kitchen table for months (that's right, I wrote this book in the kitchen)—thank you for your support.

Finally I would like to extend my appreciation for the open source community at large. Without it, this book would not be what it is. Open source is what ultimately got me into writing in the first place. It kept my blog alive; it crossed my path with my editor's; and now it is responsible for the book you're holding in your hands. Most of the code throughout the book would not have been possible were it not for people tirelessly putting out top-notch code for anyone to freely peruse, modify, and use.

All software involved in my part of the production of this book are open source as well. The book was written entirely in Emacs, using the document preparation system LaTeX. A host of minor open source tools have been involved in the workflow, many of which are native citizens in my operating system of choice—GNU Linux.

When the book hits the streets, it will have brought with it at least one new open source project, and I hope I will contribute many more in the years to come.

About the Author

Christian Johansen lives in Oslo, Norway, where he currently works for Shortcut AS, a software company focusing on open source technology, web applications, and mobile applications. Originally a student in informatics, mathematics, and digital signal processing, Christian has spent his professional career specializing in web applications and frontend technologies such as JavaScript, CSS, and HTML, technologies he has been passionate about since around the time the HTML 4.01 spec was finalized.

As a consultant, Christian has worked with many high profile companies in Norway, including leading companies within the finance and telecom sector, where he has worked on small and big web applications ranging from the average CMS-backed corporate website via e-commerce to self service applications.

In later years Christian has been an avid blogger. Derived from the same desire to share and contribute to the community that gave him so much for free, Christian has involved himself in and contributed to quite a few open source projects.

After working on several projects with less than trivial amounts of JavaScript, Christian has felt the pain of developing "the cowboy style." In an attempt at improving code quality, confidence, and the ability to modify and maintain code with greater ease, he has spent a great deal of his time both at work and in his spare time over the last few years investigating unit testing and test-driven development in JavaScript. Being a sworn TDD-er while developing in traditional server-side languages, the cowboy style JavaScript approach wasn't cutting it anymore. The culmination of this passion is the book you now hold in your hands.

This page intentionally left blank

3 Tools of the Trade

In Chapter 1, *Automated Testing,* we developed a very simple `testCase` function, capable of running basic unit tests with test case setup and teardown methods. Although rolling our own test framework is a great exercise, there are many frameworks already available for JavaScript and this chapter explores a few of them.

In this chapter we will take a look at "the tools of the trade"—essential and useful tools to support a test-driven workflow. The most important tool is of course the testing framework, and after an overview of available frameworks, we will spend some time setting up and running JsTestDriver, the testing framework used for most of this book's example code. In addition to a testing framework, this chapter looks at tools such as coverage reports and continuous integration.

3.1 xUnit Test Frameworks

In Chapter 1, *Automated Testing,* we coined *xUnit* as the term used to describe testing frameworks that lean on the design of Java's JUnit and Smalltalk's SUnit, originally designed by Kent Beck. The xUnit family of test frameworks is still the most prevalent way of writing automated tests for code, even though the past few years have seen a rise in usage for so-called *behavior-driven development* (or BDD) testing frameworks.

3.1.1 Behavior-Driven Development

Behavior-driven development, or BDD, is closely related to TDD. As discussed in Chapter 2, *The Test-Driven Development Process,* TDD is *not* about testing, but rather about design and process. However, due to the terminology used to describe the process, a lot of developers never evolve beyond the point where they simply write unit tests to verify their code, and thus never experience many of the advantages associated with using tests as a design tool. BDD seeks to ease this realization by focusing on an improved vocabulary. In fact, vocabulary is perhaps the most important aspect of BDD, because it also tries to normalize the vocabulary used by programmers, business developers, testers, and others involved in the development of a system when discussing problems, requirements, and solutions.

Another "double D" is Acceptance Test-Driven Development. In acceptance TDD, development starts by writing automated tests for high level features, based on acceptance tests defined in conjunction with the client. The goal is to pass the acceptance tests. To get there, we can identify smaller parts and proceed with "regular" TDD. In BDD this process is usually centered around *user stories*, which describe interaction with the system using a vocabulary familiar to everyone involved in the project. BDD frameworks such as Cucumber allow for user stories to be used as executable tests, meaning that acceptance tests can be written together with the client, increasing the chance of delivering the product the client had originally envisioned.

3.1.2 Continuous Integration

Continuous integration is the practice of integrating code from all developers on a regular basis, usually every time a developer pushes code to a remote version control repository. The continuous integration server typically builds all the sources and then runs tests for them. This process ensures that even when developers work on isolated units of features, the integrated whole is considered every time code is committed to the upstream repository. JavaScript does not need compiling, but running the entire test suite for the application on a regular basis can help catch errors early.

Continuous integration for JavaScript can solve tasks that are impractical for developers to perform regularly. Running the entire test suite in a wide array of browser and platform combinations is one such task. Developers working with TDD can focus their attention on a small representative selection of browsers, while the continuous integration server can test much wider, alerting the team of errors by email or RSS.

Additionally, it is common practice for JavaScript to be served minified—i.e., with unneeded white-space and comments stripped out, and optionally local identifiers munged to occupy fewer bytes—to preserve bytes over the wire. Both minifying code too aggressively or merging files incorrectly can introduce bugs. A continuous integration server can help out with these kinds of problems by running all tests on the full source as well as building concatenated and minified release files and re-running the test suite for them.

3.1.3 Asynchronous Tests

Due to the asynchronous nature of many JavaScript programming tasks such as working with `XMLHttpRequest`, animations and other deferred actions (i.e., any code using `setTimeout` or `setInterval`), and the fact that browsers do not offer a `sleep` function (because it would freeze the user interface), many testing frameworks provide a means to execute asynchronous tests. Whether or not asynchronous *unit* tests is a good idea is up for discussion. Chapter 12, *Abstracting Browser Differences: Ajax,* offers a more thorough discussion on the subject as well as an example.

3.1.4 Features of xUnit Test Frameworks

Chapter 1, *Automated Testing,* already introduced us to the basic features of the xUnit test frameworks: Given a set of test methods, the framework provides a test runner that can run them and report back the results. To ease the creation of shared test fixtures, test cases can employ the `setUp` and `tearDown` functions, which are run before and after (respectively) each individual test in a test case. Additionally, the test framework provides a set of assertions that can be used to verify the state of the system being tested. So far we have only used the `assert` method which accepts any value and throws an exception when the value is falsy. Most frameworks provide more assertions that help make tests more expressive. Perhaps the most common assertion is a version of `assertEqual`, used to compare actual results against expected values.

When evaluating test frameworks, we should assess the framework's test runner, its assertions, and its dependencies.

3.1.4.1 The Test Runner

The test runner is the most important part of the testing framework because it basically dictates the workflow. For example, most unit testing frameworks available for JavaScript today use an in-browser test runner. This means that tests must run inside a browser by loading an HTML file (often referred to as an HTML fixture) that itself loads the libraries to test, along with the unit tests and the testing framework. Other types of test runners can run in other environments, e.g., using Mozilla's Rhino implementation to run tests on the command line. What kind of test runner is suitable to test a specific application depends on whether it is a client-side application, server-side, or maybe even a browser plugin (an example of which would be FireUnit, a unit testing framework that uses Firebug and is suitable for developing Firefox plugins).

A related concern is the test report. Clear fail/success status is vital to the test-driven development process, and clear feedback with details when tests fail or have errors is needed to easily handle them as they occur. Ideally, the test runner should produce test results that are easily integrated with continuous integration software.

Additionally, some sort of plugin architecture for the test runner can enable us to gather metrics from testing, or otherwise allow us to extend the runner to improve the workflow. An example of such a plugin is the test coverage report. A coverage report shows how well the test suite covers the system by measuring how many lines in production code are executed by tests. Note that 100% coverage does not imply that every thinkable test is written, but rather that the test suite executes each and every line of production code. Even with 100% coverage, certain sets of input can still break the code—it cannot guarantee the absence of, e.g., missing error handling. Coverage reports are useful to find code that is not being exercised by tests.

3.1.5 Assertions

A rich set of assertions can really boost the expressiveness of tests. Given that a good unit test clearly states its intent, this is a massive boon. It's a lot easier to spot what a test is targeting if it compares two values with `assertEqual(expected, actual)` rather than with `assert(expected == actual)`. Although `assert` is all we really need to get the job done, more specific assertions make test code easier to read, easier to maintain, and easier to debug.

Assertions is one aspect where an exact port of the xUnit framework design from, e.g., Java leaves a little to be desired. To achieve good expressiveness in tests, it's helpful to have assertions tailored to specific language features, for instance, having assertions to handle JavaScripts special values such as `undefined`, `NaN` and `infinity`. Many other assertions can be provided to better support testing JavaScript, not just some arbitrary programming language. Luckily, specific assertions like those mentioned are easy to write piggybacking a general purpose `assert` (or, as is common, a `fail` method that can be called when the assertion does not hold).

3.1.6 Dependencies

Ideally, a testing framework should have as few dependencies as possible. More dependencies increase the chance of the mechanics of the framework not working in some browser (typically older ones). The worst kind of dependency for a testing framework is an obtrusive library that tampers with the global scope. The original version of JsUnitTest, the testing framework built for and used by the Prototype.js library, depended on Prototype.js itself, which not only adds a number of global properties but also augments a host of global constructors and objects. In practice, using it to test code that was not developed with Prototype.js would prove a futile exercise for two reasons:

- Too easy to accidentally rely on Prototype.js through the testing framework (yielding green tests for code that would fail in production, where Prototype.js would not be available)
- Too high a risk for collisions in the global scope (e.g., the MooTools library adds many of the same global properties)

3.2 In-Browser Test Frameworks

The original JavaScript port of the JUnit framework was JsUnit, first released in 2001. Not surprisingly, it has in many ways set the standard for a lot of testing frameworks following it. JsUnit runs tests in a browser: The test runner prompts for the URL to a test file to execute. The test file may be an HTML test suite which links to several test cases to execute. The tests are then run in sandboxed frames, and a green progress bar is displayed while tests are running. Obviously, the bar turns red whenever a test fails. JsUnit still sees the occasional update, but it has not been significantly updated for a long time, and it's starting to lag behind. JsUnit has served many developers well, including myself, but there are more mature and up-to-date alternatives available today.

Common for the in-browser testing frameworks is how they require an HTML fixture file to load the files to test, the testing library (usually a JavaScript and a CSS file), as well as the tests to run. Usually, the fixture can be simply copy-pasted for each new test case. The HTML fixture also serves the purpose of hosting dummy markup needed for the unit tests. If tests don't require such markup, we can lessen the burden of keeping a separate HTML file for each test case by writing a script that scans the URL for parameters naming library and test files to load, and then load them dynamically. This way we can run several test cases from the same HTML fixture simply by modifying the URL query string. The fixture could of course also be generated by a server-side application, but be careful down this route. I advise you to keep things simple—complicated test runners greatly decreases the likelihood of developers running tests.

3.2.1 YUI Test

Most of the major JavaScript libraries available today have their own unit testing framework. YUI from Yahoo! is no exception. YUI Test 3 can be safely used to test arbitrary JavaScript code (i.e., it has no obtrusive dependencies). YUI Test is, in its own words, "not a direct port from any specific xUnit framework," but it "does derive some characteristics from nUnit and JUnit," with nUnit being the .NET interpretation of the xUnit family of frameworks, written in C#. YUI Test is a mature testing framework with a rich feature set. It supports a rich set of assertions, test suites, a *mocking* library (as of YUI 3), and asynchronous tests.

3.2.1.1 Setup

Setup is very easy thanks to YUI's loader utility. To get quickly started, we can link directly to the YUI seed file on the YUI server, and use `YUI.use` to fetch the necessary dependencies. We will revisit the `strftime` example from Chapter 1, *Automated Testing,* in order to compare YUI Test to the `testCase` function introduced in that chapter. Listing 3.1 shows the HTML fixture file, which can be saved in, e.g., `strftime_yui_test.html`.

Listing 3.1 YUI Test HTML fixture file

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Testing Date.prototype.strftime with YUI</title>
    <meta http-equiv="content-type"
          content="text/html; charset=UTF-8">
  </head>
  <body class="yui-skin-sam">
    <div id="yui-main"><div id="testReport"></div></div>
    <script type="text/javascript"
  src="http://yui.yahooapis.com/3.0.0/build/yui/yui-min.js">
    </script>
    <script type="text/javascript" src="strftime.js">
    </script>
    <script type="text/javascript" src="strftime_test.js">
    </script>
  </body>
</html>
```

The `strftime.js` file contains the `Date.prototype.strftime` implementation presented in Listing 1.2 in Chapter 1, *Automated Testing.* Listing 3.2 shows the test script, save it in `strftime_test.js`.

Listing 3.2 `Date.prototype.strftime` YUI test case

```
YUI({
  combine: true,
  timeout: 10000
}).use("node", "console", "test", function (Y) {
  var assert = Y.Assert;

  var strftimeTestCase = new Y.Test.Case({
    // test case name - if not provided, one is generated
    name: "Date.prototype.strftime Tests",

    setUp: function () {
      this.date = new Date(2009, 9, 2, 22, 14, 45);
    },

    tearDown: function () {
      delete this.date;
    },

    "test %Y should return full year": function () {
      var year = Date.formats.Y(this.date);

      assert.isNumber(year);
      assert.areEqual(2009, year);
    },

    "test %m should return month": function () {
      var month = Date.formats.m(this.date);

      assert.isString(month);
      assert.areEqual("10", month);
    },

    "test %d should return date": function () {
      assert.areEqual("02", Date.formats.d(this.date));
    },

    "test %y should return year as two digits": function () {
      assert.areEqual("09", Date.formats.y(this.date));
    },
```

```
    "test %F should act as %Y-%m-%d": function () {
      assert.areEqual("2009-10-02", this.date.strftime("%F"));
    }
  });

  //create the console
  var r = new Y.Console({
    newestOnTop : false,
    style: 'block'
  });

  r.render("#testReport");
  Y.Test.Runner.add(strftimeTestCase);
  Y.Test.Runner.run();
});
```

When using YUI Test for production code, the required sources should be downloaded locally. Although the loader is a convenient way to get started, relying on an internet connection to run tests is bad practice because it means we cannot run tests while offline.

3.2.1.2 Running Tests

Running tests with YUI Test is as simple as loading up the HTML fixture in a browser (preferably several browsers) and watching the output in the console, as seen in Figure 3.1.

3.2.2 Other In-Browser Testing Frameworks

When choosing an in-browser testing framework, options are vast. YUI Test is among the most popular choices along with JsUnit and QUnit. As mentioned, JsUnit is long overdue for an upgrade, and I suggest you not start new projects with it at this point. QUnit is the testing framework developed and used by the jQuery team. Like YUI Test it is an in-browser test framework, but follows the traditional xUnit design less rigidly. The Dojo and Prototype.js libraries both have their test frameworks as well.

One might get the impression that there are almost as many testing frameworks out there as there are developers unit testing their scripts—there is no defacto standard way to test JavaScript. In fact, this is true for most programming tasks that are not directly related to browser scripting, because JavaScript has no general purpose standard library. CommonJS is an initiative to rectify this situation, originally motivated to standardize server-side JavaScript. CommonJS also includes a

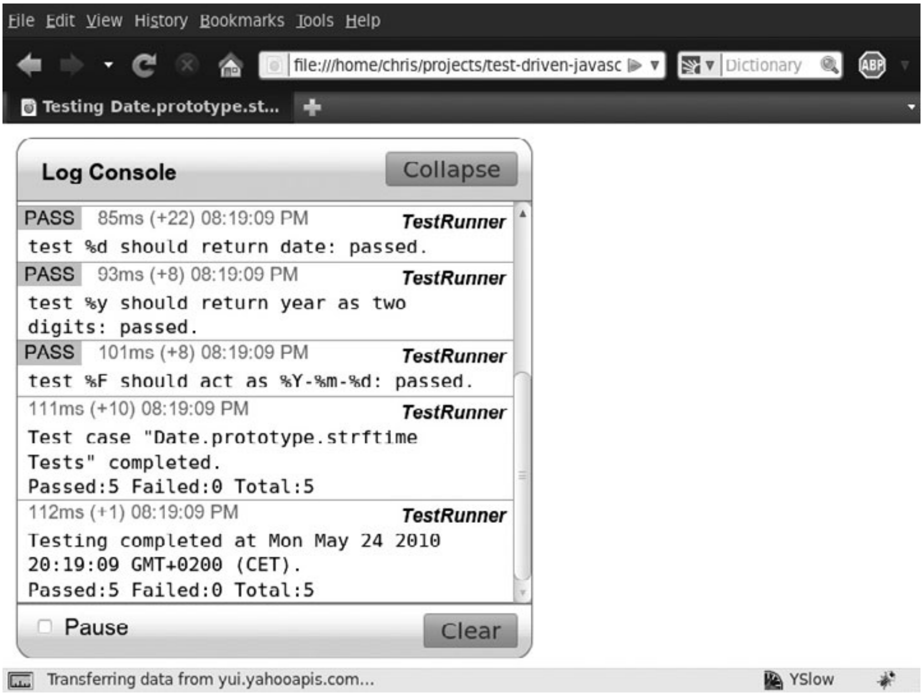

Figure 3.1 Running tests with YUI Test.

unit testing spec, which we will look into when testing a `Node.js` application in Chapter 14, *Server-Side JavaScript with Node.js.*

3.3 Headless Testing Frameworks

In-browser testing frameworks are unfit to support a test-driven development process where we need to run tests frequently and integrated into the workflow. An alternative to these frameworks is headless testing frameworks. These typically run from the command line, and can be interacted with in the same way testing frameworks for any other server-side programming language can.

There are a few solutions available for running headless JavaScript unit tests, most originating from either the Java or Ruby worlds. Both the Java and Ruby communities have strong testing cultures, and testing only half the code base (the server-side part) can only make sense for so long, probably explaining why it is these two communities in particular that have stood out in the area of headless testing solutions for JavaScript.

3.3.1 Crosscheck

Crosscheck is one of the early headless testing frameworks. It provides a Java backed emulation of Internet Explorer 6 and Firefox versions 1.0 and 1.5. Needless to say, Crosscheck is lagging behind, and its choice of browsers are unlikely to help develop applications for 2010. Crosscheck offers JavaScript unit tests much like that of YUI Test, the difference being that they can be run on the command line with the Crosscheck jar file rather than in a browser.

3.3.2 Rhino and env.js

`env.js` is a library originally developed by John Resig, creator of the jQuery JavaScript framework. It offers an implementation of the browser (i.e., BOM) and DOM APIs on top of Rhino, Mozilla's Java implementation of JavaScript. Using the env.js library together with Rhino means we can load and run in-browser tests on the command line.

3.3.3 The Issue with Headless Test Runners

Although the idea of running tests on the command line is exciting, I fail to recognize the power of running tests in an environment where production code will never run. Not only are the browser environment and DOM emulations, but the JavaScript engine (usually Rhino) is an altogether different one as well.

Relying on a testing framework that simply emulates the browser is bad for a few reasons. For one, it means tests can only be run in browsers that are emulated by the testing framework, or, as is the case for solutions using Rhino and env.js, in an alternate browser and DOM implementation altogether. Limiting the available testing targets is not an ideal feature of a testing framework and is unlikely to help write cross-browser JavaScript. Second, an emulation will never match whatever it is emulating perfectly. Microsoft probably proved this best by providing an Internet Explorer 7 emulation mode in IE8, which is in fact not an exact match of IE7. Luckily, we can get the best from both worlds, as we will see next, in Section 3.4, *One Test Runner to Rule Them All.*

3.4 One Test Runner to Rule Them All

The problem with in-browser testing frameworks is that they can be cumbersome to work with, especially in a test-driven development setting where we need to run tests continuously and integrated into the workflow. Additionally, testing on a wide array of platform/browser combinations can entail quite a bit of manual work. Headless frameworks are easier to work with, but fail at testing in the actual environment the code will be running in, reducing their usefulness as testing tools. A fairly new player on the field of xUnit testing frameworks is JsTestDriver, originating from Google. In contrast to the traditional frameworks, JsTestDriver is first and foremost a test runner, and a clever one at that. JsTestDriver solves the aforementioned problems by making it easy both to run tests and to test widely in real browsers.

3.4.1 How JsTestDriver Works

JsTestDriver uses a small server to run tests. Browsers are captured by the test runner and tests are scheduled by issuing a request to the server. As each browser runs the tests, results are sent back to the client and presented to the developer. This means that as browsers are idly awaiting tests, we can schedule runs from either the command line, the IDE, or wherever we may feel most comfortable running them from. This approach has numerous advantages:

- Tests can be run in browsers without requiring manual interaction with the browser.
- Tests can be run in browsers on multiple machines, including mobile devices, allowing for arbitrary complex testing grids.
- Tests run **fast**, due to the fact that results need not be added to the DOM and rendered, they can be run in any number of browsers simultaneously, and the browser doesn't need to reload scripts that haven't changed since the tests were last run.
- Tests can use the full DOM because no portion of the document is reserved for the test runner to display results.
- No need for an HTML fixture, simply provide one or more scripts and test scripts, an empty document is created on the fly by the test runner.

JsTestDriver tests are **fast**. The test runner can run complex test suites of several hundred tests in under a single second. Because tests are run simultaneously, tests will still run in about a second even when testing 15 browsers at the same time. Granted, some time is spent communicating with the server and optionally refreshing the browser cache, but a full run still completes in a matter of a few seconds. Single test case runs usually complete in the blink of an eye.

As if faster tests, simpler setup, and full DOM flexibility weren't enough, JsTestDriver also offers a plugin that calculates test coverage, XML test report output compatible with JUnit's reports, meaning we can immediately use existing continuous integration servers, and it can use alternative assertion frameworks. Through plug-ins, any other JavaScript testing framework can take advantage of the JsTestDriver test runner, and at the time of writing, adapters for QUnit and YUI Test already exist. This means tests can be written using YUI Test's assertions and syntax, but run using JsTestDriver.

3.4.2 JsTestDriver Disadvantages

At the time of writing, JsTestDriver does not support any form of asynchronous testing. As we will see in Chapter 12, *Abstracting Browser Differences: Ajax,* this isn't necessarily a problem from a unit testing perspective, but it may limit the options for integration tests, in which we want to fake as little as possible. It is possible that asynchronous test support will be added to future versions of JsTestDriver.

Another disadvantage of JsTestDriver is that the JavaScript required to run tests is slightly more advanced, and may cause a problem in old browsers. For instance, by design, a browser that is to run JsTestDriver needs to support the `XMLHttpRequest` object or similar (i.e., Internet Explorer's corresponding `ActiveX` object) in order to communicate with the server. This means that browsers that don't support this object (older browsers, Internet Explorer before version 7 with ActiveX disabled) cannot be tested with the JsTestDriver test runner. This problem can be effectively circumvented, however, by using YUI Test to write tests, leaving the option of running them manually with the default test runner in any uncooperative browser.

3.4.3 Setup

Installing and setting up JsTestDriver is slightly more involved than the average in-browser testing framework; still, it will only take a few minutes. Also, the setup is only required once. Any projects started after the fact are dirt simple to get running. JsTestDriver requires Java to run both the server component and start test runs. I won't give instructions on installing Java here, but most systems have Java installed already. You can check if Java is installed by opening a shell and issue the `java -version` command. If you don't have Java installed, you will find instructions on java.com.

3.4.3.1 Download the Jar File

Once Java is set up, download the most recent JsTestDriver jar file from http://code.google.com/p/js-test-driver/downloads/list. All the examples in this book use version 1.2.1, be sure to use that version when following along with the examples. The jar file can be placed anywhere on the system, I suggest `~/bin`. To make it easier to run, set up an environment variable to point to this directory, as shown in Listing 3.3.

Listing 3.3 Setting the $JSTESTDRIVER_HOME environment variable

```
export JSTESTDRIVER_HOME=~/bin
```

Set the environment variable in a login script, such as `.bashrc` or `.zshrc` (depends on the shell—most systems use Bash, i.e., `~/.bashrc`, by default).

3.4.3.2 Windows Users

Windows users can set an environment variable in the `cmd` command line by issuing the `set JSTESTDRIVER_HOME=C:\bin` command. To set it permanently, right-click *My Computer* (*Computer* in Windows 7) and select *Properties.* In the *System window*, select *Advanced system properties,* then the *Advanced tab,* and then click the *Environment Variables ...* button. Decide if you need to set the environment variable for yourself only or for all users. Click *New*, enter the name (JSTESTDRIVER_HOME) in the top box, and then the path where you saved the jar file in the bottom one.

3.4.3.3 Start the Server

To run tests through JsTestDriver, we need a running server to capture browsers with. The server can run anywhere reachable from your machine—locally, on a machine on the local network, or a public facing machine. Beware that running the server on a public machine will make it available to anyone unless the machine restricts access by IP address or similar. To get started, I recommend running the service locally; this way you can test while being offline as well. Open a shell and issue the command in either Listing 3.4 or Listing 3.5 (current directory is not important for this command).

Listing 3.4 Starting the JsTestDriver server on Linux and OSX

```
java -jar $JSTESTDRIVER_HOME/JsTestDriver-1.2.1.jar --port
    4224
```

Listing 3.5 Starting the JsTestDriver server on Windows

```
java -jar %JSTESTDRIVER_HOME%\JsTestDriver-1.2.1.jar --port
    4224
```

Port 4224 is the defacto standard JsTestDriver port, but it is arbitrarily picked and you can run it on any port you want. Once the server is running, the shell running it must stay open for as long as you need it.

3.4.3.4 Capturing Browsers

Open any browser and point it to http://localhost:4224 (make sure you change the port number if you used another port when starting the server). The resulting page will display two links: *Capture browser* and *Capture in strict mode*. JsTestDriver runs tests inside an HTML 4.01 document, and the two links allow us to decide if we want to run tests with a transitional or strict doctype. Click the appropriate link, and leave the browser open. Repeat in as many browsers as desired. You can even try hooking up your phone or browsers on other platforms using virtual instances.

3.4.3.5 Running Tests

Tests can be run from the command line, providing feedback in much the same way a unit testing framework for any server-side language would. As tests are run, a dot will appear for every passing test, an F for a failing test, and an E for a test with errors. An error is any test error that is not a failing assertion, i.e., an unexpected exception. To run the tests, we need a small configuration file that tells JsTestDriver which source and test files to load (and in what order), and which server to run tests against. The configuration file, jsTestDriver.conf by default, uses YAML syntax, and at its simplest, it loads every source file and every test file, and runs tests at http://localhost:4224, as seen in Listing 3.6.

Listing 3.6 A barebone jsTestDriver.conf file

```
server: http://localhost:4224

load:
  - src/*.js
  - test/*.js
```

Load paths are relative to the location of the configuration file. When it's required to load certain files before others, we can specify them first and still use the `*.js` notation, JsTestDriver will only load each file once, even when it is referenced more than once. Listing 3.7 shows an example where `src/mylib.js` always need to load first.

Listing 3.7 Making sure certain files load first

```
server: http://localhost:4224

load:
  - src/mylib.js
  - src/*.js
  - test/*.js
```

In order to test the configuration we need a sample project. We will revisit the `strftime` example once again, so start by copying the strftime.js file into the src directory. Then add the test case from Listing 3.8 in `test/strftime_test.js`.

Listing 3.8 `Date.prototype.strftime` test with JsTestDriver

```
TestCase("strftimeTest", {
  setUp: function () {
    this.date = new Date(2009, 9, 2, 22, 14, 45);
  },

  tearDown: function () {
    delete this.date;
  },

  "test %Y should return full year": function () {
    var year = Date.formats.Y(this.date);

    assertNumber(year);
    assertEquals(2009, year);
  },

  "test %m should return month": function () {
    var month = Date.formats.m(this.date);

    assertString(month);
    assertEquals("10", month);
  },

  "test %d should return date": function () {
    assertEquals("02", Date.formats.d(this.date));
  },

  "test %y should return year as two digits": function () {
    assertEquals("09", Date.formats.y(this.date));
  },
```

```
  "test %F should act as %Y-%m-%d": function () {
    assertEquals("2009-10-02", this.date.strftime("%F"));
  }
});
```

The test methods are almost syntactically identical to the YUI Test example, but note how this test case has less scaffolding code to support the test runner. Now create the configuration file as shown in Listing 3.9.

Listing 3.9 JsTestDriver configuration

```
server: http://localhost:4224

load:
  - src/*.js
  - test/*.js
```

We can now schedule tests to run by issuing the command in Listing 3.10 or Listing 3.11, depending on your operating system.

Listing 3.10 Running tests with JsTestDriver on Linux and OSX

```
java -jar $JSTESTDRIVER_HOME/JsTestDriver-1.2.1.jar --tests
    all
```

Listing 3.11 Running tests with JsTestDriver on Windows

```
java -jar %JSTESTDRIVER_HOME%\JsTestDriver-1.2.1.jar--tests
    all
```

The default configuration file name is `jsTestDriver.conf`, and as long as this is used we don't need to specify it. When using another name, add the `--config path/to/file.conf` option.

When running tests, JsTestDriver forces the browser to refresh the test files. Source files, however, aren't reloaded between test runs, which may cause errors due to stale files. We can tell JsTestDriver to reload everything by adding the `--reset` option.

3.4.3.6 JsTestDriver and TDD

When TDD-ing, tests will fail frequently, and it is vital that we are able to quickly verify that we get the failures we expect in order to avoid buggy tests. A browser such as Internet Explorer is not suitable for this process for a few reasons. First, its error messages are less than helpful; you have probably seen "Object does not support this property or method" more times than you care for. The second reason is that IE, at least in older versions, handles script errors badly. Running a TDD session in IE will cause it to frequently choke, requiring you to manually refresh it. Not to mention the lack of performance in IE, which is quite noticeable compared to, e.g., Google Chrome.

Disregarding Internet Explorer, I would still advise against keeping too many browsers in your primary TDD process, because doing so clutters up the test runner's report, repeating errors and log messages once for every captured browser. My advice is to develop against one server that only captures your browser of choice, and frequently run tests against a second server that captures many browsers. You can run against this second server as often as needed—after each passed test, completed method, or if you are feeling bold, even more. Keep in mind that the more code you add between each run, the harder it will be to spot any bugs that creep up in those secondary browsers.

To ease this sort of development, it's best to remove the `server` line from the configuration file and use the `--server` command line option. Personally I do this kind of development against Firefox, which is reasonably fast, has good error messages, and always runs on my computer anyway. As soon as I pass a test, I issue a run on a remote server that captures a wider variety of browsers, new and old.

3.4.4 Using JsTestDriver From an IDE

JsTestDriver also ships plugins for popular integrated development environments (IDEs), Eclipse and IntelliJ IDEA. In this section I will walk through setting up the Eclipse plugin and using it to support a test-driven development process. If you are not interested in developing in Eclipse (or Aptana), feel free to skip to Section 3.4.5, *Improved Command Line Productivity*.

3.4.4.1 Installing JsTestDriver in Eclipse

To get started you need to have Eclipse (or Aptana Studio, an IDE based on Eclipse aimed at web developers) installed. Eclipse is a free open source IDE and can be downloaded from http://eclipse.org. Once Eclipse is running, go to the *Help* menu and select *Install new software.* In the window that opens, enter the following URL as a new *update site:* http://js-test-driver.googlecode.com/svn/update/

"JS Test Driver Eclipse Plugin" should now be displayed with a checkbox next to it. Check it and click *Next*. The next screen is a confirmation that sums up the plugins to be installed. Click *Next* once again and Eclipse asks you to accept the terms of use. Check the appropriate radio button and click *Next* if you accept. This should finish the installation.

Once the plugin is installed we need to configure it. Find the Preferences pane under the Window menu (Eclipse menu on OS X). There should be a new entry for Js Test Driver; select it. As a bare minimum we need to enter the port where Eclipse should run the server. Use 4224 to follow along with the example. You can also enter the paths to browsers installed locally to ease browser capturing, but it's not really necessary.

3.4.4.2 Running JsTestDriver in Eclipse

Next up, we need a project. Create a new project and enter the directory for the command line example as location. Now start the server. Locate the JsTestDriver panel in Eclipse and click the green play button. Once the server is running, click the browser icons to capture browsers (given that their path was configured during setup). Now right-click a file in the project, and select *Run As* and then *Run Configurations . . .* Select *Js Test Driver Test* and click the sheet of paper icon indicating "new configuration." Give the configuration a name and select the project's configuration file. Now click run and the tests run right inside Eclipse, as seen in Figure 3.2.

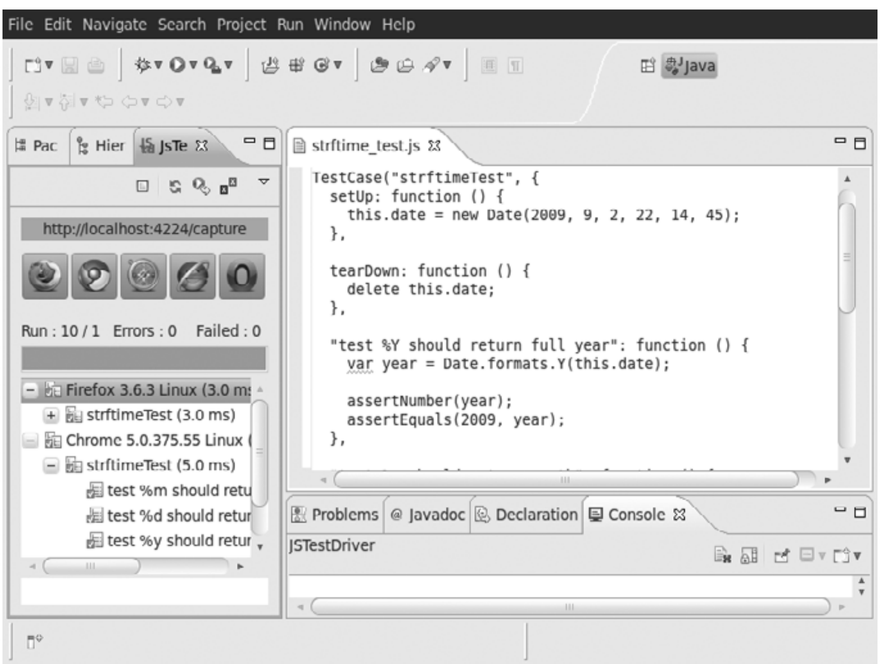

Figure 3.2 Running JsTestDriver tests inside Eclipse.

On subsequent runs, simply select *Run As* and then *Name of configuration.* Even better, check the *Run on every save* checkbox in the configuration prompt. This way, tests are run anytime a file in the project is saved, perfect for the test-driven development process.

3.4.5 Improved Command Line Productivity

If the command line is your environment of choice, the Java command to run tests quickly becomes a bit tiresome to type out. Also, it would be nice to be able to have tests run automatically whenever files in the project change, just like the Eclipse and IDEA plugins do. Jstdutil is a Ruby project that adds a thin command line interface to JsTestDriver. It provides a leaner command to run tests as well as an `jsautotest` command that runs related tests whenever files in the project change.

Jstdutil requires Ruby, which comes pre-installed on Mac OS X. For other systems, installation instructions can be found on ruby-lang.org. With Ruby installed, install Jstdutil by running ``gem install jstdutil`` in a shell. Jstdutil uses the previously mentioned `$JSTESTDRIVER_HOME` environment variable to locate the JsTestDriver jar file. This means that running tests is a simple matter of ``jstestdriver --tests all``, or for autotest, simply ``jsautotest``. If the configuration file is not automatically picked up, specify it using ``jstestdriver --config path/to/file.conf --tests all``. The `jstestdriver` and `jsautotest` commands also add coloring to the test report, giving us that nice red/green visual feedback.

3.4.6 Assertions

JsTestDriver supports a rich set of assertions. These assertions allow for highly expressive tests and detailed feedback on failures, even when a custom assertion message isn't specified. The full list of supported assertions in JsTestDriver is:

- `assert(msg, value)`
- `assertTrue(msg, value)`
- `assertFalse(msg, value)`
- `assertEquals(msg, expected, actual)`
- `assertNotEquals(msg, expected, actual)`
- `assertSame(msg, expected, actual)`
- `assertNotSame(msg, expected, actual)`
- `assertNull(msg, value)`

- `assertNotNull(msg, value)`
- `assertUndefined(msg, value)`
- `assertNotUndefined(msg, value)`
- `assertNaN(msg, number)`
- `assertNotNaN(msg, number)`
- `assertException(msg, callback, type)`
- `assertNoException(msg, callback)`
- `assertArray(msg, arrayLike)`
- `assertTypeOf(msg, type, object)`
- `assertBoolean(msg, value)`
- `assertFunction(msg, value)`
- `assertNumber(msg, value)`
- `assertObject(msg, value)`
- `assertString(msg, value)`
- `assertMatch(msg, pattern, string)`
- `assertNoMatch(msg, pattern, string)`
- `assertTagName(msg, tagName, element)`
- `assertClassName(msg, className, element)`
- `assertElementId(msg, id, element)`
- `assertInstanceOf(msg, constructor, object)`
- `assertNotInstanceOf(msg, constructor, object)`

We will be using JsTestDriver for most examples throughout this book.

3.5 Summary

In this chapter we have taken a look at what tools can be helpful to support the test-driven development process, as well as a few available tools. Getting a good test-driven development rhythm requires adequate tools, and for the remaining examples of this book, JsTestDriver was selected to run tests. It offers both a highly efficient workflow as well as thorough testing on a wide array of platform and browser combinations.

This chapter also touched briefly on BDD and "specs" and how test-driven development, as practiced in this book, shares a lot in common with it.

Although we visited the topics of test coverage reports and continuous integration in this chapter, no setup or examples were given for such tools. On the book's website[1] you will find a guide to running the Coverage plugin for JsTestDriver as well as a guide on how to run JsTestDriver tests in the open source continuous integration server Hudson.

In the next chapter we will have a look at some other ways to utilize unit tests before we move on to Part II, *JavaScript for Programmers.*

1. http://tddjs.com

This page intentionally left blank

Index

A

V

W

X

Y

Z