JASMINE TESTING

A Cloak & Dagger Guide

Joshua Clanton

Jasmine Testing for JavaScript

A Cloak & Dagger Guide

Joshua Clanton

This book is for sale at http://leanpub.com/jasmine-testing

This version was published on 2015-01-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Joshua Clanton by spreading the word about this book on Twitter!

The suggested hashtag for this book is #jasminetesting.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#jasminetesting

Also By Joshua Clanton

A Drip of JavaScript

Contents

A Note About The Book

This book is a work in progress and as such you may run across mistakes, misspellings and grammatical errors. If you see something like this please let me know.

I can be reached on Twitter as @joshuacc[1].

Thanks!

[1] http://twitter.com/joshuacc

Preface

The Universal Language

Are you a developer? Then chances are that at some point or another you've worked with JavaScript. For a long time JavaScript was seen as something of a "toy language" used only to add trivial things like animation and form validation to web sites, but in recent years, JavaScript has expanded far beyond the browser.

JavaScript Object Notation (JSON) is one of the most widely used data formats in the world. Projects like Node.js have brought JavaScript development server-side. Non-relational databases like MongoDB and CouchDB use JavaScript as a querying language. And Windows 8 supports writing desktop applications in JavaScript.

Now that JavaScript is finally being taken seriously, many developers are looking for tools to help design and maintain larger JavaScript codebases than in the past.

Testing in JavaScript

That, of course, leads into testing. Unit testing has long been used to ensure the reliability and maintainability of software. And more recently, test-first methods like Test-Driven Development and Behavior Driven Development have come to be seen as best practices as well.

Because it was long regarded (wrongly) as a second-class language, the culture and tools for effective JavaScript testing are relatively young. But they are maturing at a rapid pace.

There are many JavaScript testing frameworks, but according to an informal survey by Addy Osmani[2] the most popular appear to be, in order:

1. Jasmine[3]
2. QUnit[4]
3. Mocha[5] with the Chai assertion library[6]
4. BusterJS[7]

[2] http://bit.ly/JSTestingSurvey
[3] http://pivotal.github.com/jasmine/
[4] http://qunitjs.com/
[5] http://visionmedia.github.com/mocha/
[6] http://chaijs.com/
[7] http://busterjs.org

5. JS Test Driver[8]
6. CasperJS[9]

Jasmine

Each of these frameworks has advantages and disadvantages, but Jasmine has a unique set of features that makes it particularly easy to get started with.

Only Minimal Setup is Required

To use Jasmine you don't need to have Node.js installed, have any experience with the command line, or choose any configuration options. All you need is basic familiarity with running JavaScript in an HTML page.

An Elegant and Expressive Syntax for Writing Tests

A basic Jasmine suite with one spec (test) looks something like this:

```
describe("My hello world function", function() {
    it("should return 'Hello, World!'", function() {
      expect(helloWorld()).toBe("Hello, World!");
    });
});
```

As you can see, this way of describing how you expect your code to respond is quite readable. It is also easy to write.

> This way of writing tests in a human readable format is a form of Behavior Driven Development[a].
>
> In order to emphasize the fact that the tests should be written before the functional code, they are typically referred to as "specs." This language will be used through the rest of the book.
>
> [a] https://en.wikipedia.org/wiki/Behavior_Driven_Development

[8] http://code.google.com/p/js-test-driver/
[9] http://casperjs.org/

Solid Documentation

Pivotal Labs, the creators of Jasmine, have done a solid job of documenting it's features both on the Jasmine home page[10] and the Jasmine wiki[11].

For something a bit more interactive, take a look at the Try Jasmine[12] site by Justin Searls.

An Active Community

As mentioned above, the Jasmine community is probably the largest of the JavaScript testing frameworks'. There are a plethora of blog posts about different aspects of Jasmine. And if you are looking for help figuring out a specific problem, posting on Stack Overflow[13] using the tag "jasmine" will probably get you the help that you need.

How to Use This Book

Most of this book is dedicated to a walk-through tutorial where you will learn about Jasmine by using it to specify, create and test working software.

I recommend that you type in each example by hand. This will help establish muscle memory early on, and eventually give you the ability to think in terms of suites and specs rather than worrying about the syntax of implementing them.

Toward the end of this book you will find appendices on various aspects of Jasmine. These are suitable for general reference.

Technical Requirements

Jasmine 2.1.2: This is the latest release at the time of this writing.

Version Control: While not required, I recommend keeping your exercise code under version control with Git. If you are not familiar with git, the excellent Try Git[14] website will get you started. If you prefer books, Pragmatic Git[15] is a good introduction and Pro Git[16] delves into the subject with more depth.

[10] http://jasmine.github.io/
[11] https://github.com/pivotal/jasmine/wiki
[12] http://tryjasmine.com/
[13] http://stackoverflow.com/
[14] http://try.github.com
[15] http://amzn.to/U8HEGk
[16] http://amzn.to/U8HJd0

Getting Started

Welcome to MI7.5

"Welcome to Technology Division," says the man in the white lab coat. My name is **REDACTED**, but you can call me T.

"Thank you. It's an honor to meet you, Mr. T," you reply.

"Oh, good heavens. Just T, not 'Mr. T.' Do you see any gold chains around my neck? We're MI7.5, not one of those American television programs you watched as a child."

Embarrassed, you nod in reply.

"Right then. Just follow me and we'll get you to your first assignment."

As you follow T through a maze of hallways, you pass several rooms full of antique clutter. The one thing that makes you stop in your tracks, though, is the mechanical shark.

"Oh, do come along," says T, gesturing towards a door. "We haven't got all day."

He leads you into a small room that looks like a converted broom closet. "This will be your office for the duration of your internship. You'll find two folders on your desk. One details your first assignment. The other explains the requirements for all code developed at MI7.5. Read that one first."

Also on the desk is what looks like a PC from 1994, complete with a CRT monitor.

"You can call me if anything goes wrong, but if you do, it had better be *really* wrong."

As T leaves, you pick up the thicker of the two folders and begin reading about the development process you'll follow.

The Development Process

After leafing around the process folder, you have a good idea of what practices MI7.5 requires. Foremost among those practices are:

- Writing executable tests (specs) to demonstrate that your code functions correctly.
- You must use the Test Driven Development process at all times. This means taking the following steps:
    1. **Red**: Write your test first, and see it fail.
    2. **Green**: Write just enough code to make your test pass.

3. **Refactor**: Using your tests to keep you safe, refactor your code to be more elegant, understandable, and maintainable.

For JavaScript, Jasmine is the required testing framework. And since your assignment is to be written in JavaScript, you'll need to set Jasmine up to work on your project.

Getting Started with Jasmine

Jasmine's Default Examples

The first thing you'll need to do is download Jasmine "standalone" 2.1.3 from GitHub[17]. Extract the contents of the zip file into the folder you'd like to use for the project. Afterwards you should see the following:

```
lib // This is where Jasmine and other libraries go
    * jasmine-2.1.3 // Contains the actual Jasmine library
        * boot.js
        * console.js
        * jasmine.css
        * jasmine.js
        * jasmine_favicon.png
        * jasmine-html.js
spec // This is where your specs and custom helpers go
    * PlayerSpec.js // An example spec file which tests the "player"
    * SpecHelper.js // An example helper function
src // This is where your source files go
    * Player.js // An example source file
    * Song.js // An example source file
MIT.LICENSE // The open source software license
SpecRunner.html // The HTML file which runs the specs over the source files
```

Open `SpecRunner.html` in a web browser and you will see something like this:

[17] https://github.com/jasmine/jasmine/releases

The Spec Runner's HTML reporter

What's happening behind the scenes is that `SpecRunner.html` executes both the source and spec files, then reports the results of the specs. Taking a look inside the spec runner's `<head>` is instructive.

First the spec runner includes references to the Jasmine library resources.

Spec Runner's Jasmine resources

```
1  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.1.3/jasmine_favic\
2  on.png">
3  <link rel="stylesheet" href="lib/jasmine-2.1.3/jasmine.css">
4
5  <script src="lib/jasmine-2.1.3/jasmine.js"></script>
6  <script src="lib/jasmine-2.1.3/jasmine-html.js"></script>
7  <script src="lib/jasmine-2.1.3/boot.js"></script>
```

Then the spec runner includes references to the source files which are under test.

Spec Runner's source files

```
1  <!-- include source files here... -->
2  <script type="text/javascript" src="src/Player.js"></script>
3  <script type="text/javascript" src="src/Song.js"></script>
```

Then the spec runner includes references to the spec files which are to be executed.

Spec Runner's spec files

```
<!-- include spec files here... -->
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

You may also wish to take a look at the spec and source files, but that's not necessary.

Setting Up Your Project Structure

For now, delete the files in the `spec` and `src` directories. Once you've done that, edit `SpecRunner.html` to remove the `script` tags which reference them. Leave the comments intact.

Now if you refresh the spec runner in your browser you should get a blank page.That's a little boring. Let's make it more interesting by adding a new spec file and a new source file.

In the `src` directory, create a new file named `authCodeValidator.js`. And in the `spec` directory, create a new file named `authCodeValidatorSpec.js`.

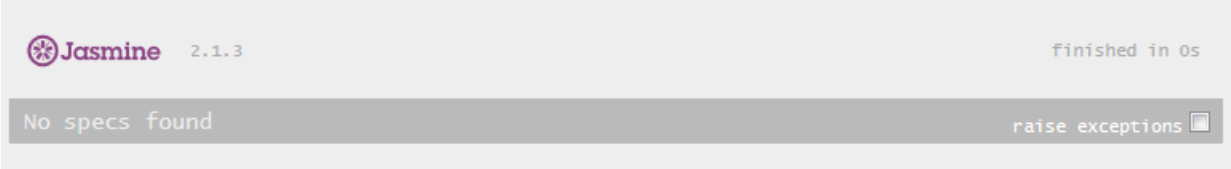
Beneath the "source files here" comment, add the following line:

```
<!-- include source files here... -->
<script src="src/authCodeValidator.js"></script>
```

Beneath the "spec files here" comment, add this line:

```
<!-- include spec files here... -->
<script src="spec/authCodeValidatorSpec.js"></script>
```

If you refresh your browser, you'll see a page that looks like this:



The Spec Runner With No Specs

That's because there aren't any specs in our spec file yet, so there's nothing to report.

Checking Our Setup

To make sure we've set the spec file up correctly, let's write a quick test spec:

authCodeValidatorSpec.js

```
describe('spec runner', function() {
  it('runs successfully', function() {
    expect(myVar).toBe(true);
  });
});
```

If you've typed everything in correctly, you should see a spec failure report when you refresh the spec runner in your browser.

Our first failing spec

Why do we see a failure? Well, the reporter itself tells us that. The variable `myVar` is not defined yet. Let's fix that.

authCodeValidator.js

```
var myVar = true;
```

Now refresh your browser and you should see a report indicating one passing spec.

Our first passing spec

This means that we've gotten our file structure set up correctly. But since these specs aren't particularly useful for what we're trying to build, it's time to delete them. Keep the files, but delete the text in both `authCodeValidator.js` and `authCodeValidatorSpec.js`.

Refresh one more time to see a blank report from the spec runner, and it's time to move on to coding our actual project.

Appendix A: Matchers Index

toBe

Uses strict equality (===) to compare the actual and expected values. This avoids JavaScript's nasty tendency to coerce values to the same type in order to compare them.

API

```
expect(actual).toBe(expected);
```

Example

```
expect(12).toBe(12);   // pass
expect("12").toBe(12); // fail
```

toBeCloseTo

Checks that numeric actual and expected values are equal up to a given level of decimal precision. If no precision is specified, it defaults to `2`.

API

```
expect(actualNumber).toBeCloseTo(expectedNumber, precision);
```

Example

```
expect(2.00).toBeCloseTo(2.01, 1);  // pass
expect(2.00).toBeCloseTo(2.01, 2);  // fail

expect(3.00).toBeCloseTo(3.01);     // fail
expect(3.000).toBeCloseTo(3.001);   // pass
```

toBeDefined

Checks that the actual value is defined (`!== undefined`).

API

```
expect(actual).toBeDefined();
```

Example

```
var a = 1;
expect(a).toBeDefined(); // pass

var b;
expect(b).toBeDefined(); // fail

var c = null;
expect(c).toBeDefined(); // pass
```

toBeFalsy

Checks whether the actual value is falsy. A falsy value is one that can evaluate to false.

API

```
expect(actual).toBeFalsy();
```

Example

```
expect(false).toBeFalsy();      // pass
expect(null).toBeFalsy();       // pass
expect(undefined).toBeFalsy();  // pass
expect("").toBeFalsy();         // pass
expect(0).toBeFalsy();          // pass

expect(true).toBeFalsy();     // fail
expect("BMW Z3").toBeFalsy(); // fail
expect(1).toBeFalsy();        // fail
expect({}).toBeFalsy();       // fail
```

toBeGreaterThan

Checks whether the actual value is greater than the comparison value.

API

```
expect(actual).toBeGreaterThan(comparison);
```

Example

```
expect(2).toBeGreaterThan(1); // pass
expect(1).toBeGreaterThan(2); // fail
```

toBeLessThan

Checks whether the actual value is less than the comparison value.

API

```
expect(actual).toBeLessThan(comparison);
```

Example

```
expect(1).toBeLessThan(2); // pass
expect(2).toBeLessThan(1); // fail
```

toBeNull

Checks whether the actual value is `null`.

API

```
expect(actual).toBeNull();
```

Example

```
expect(null).toBeNull();      // pass
expect(undefined).toBeNull(); // fail
expect("").toBeNull();        //false
```

toBeTruthy

Checks whether the actual value is truthy. A truthy value is one that can evaluate to true.

API

```
expect(actual).toBeTruthy();
```

Example

```
expect(true).toBeTruthy();           // pass
expect("Aston Martin").toBeTruthy(); // pass
expect(1).toBeTruthy();              // pass
expect({}).toBeTruthy();             // pass

expect(false).toBeTruthy();      // fail
expect(null).toBeTruthy();       // fail
expect(undefined).toBeTruthy();  // fail
expect("").toBeTruthy();         // fail
expect(0).toBeTruthy();          // fail
```

toBeUndefined

Checks that the actual value is `undefined`.

API

```
expect(actual).toBeUndefined();
```

Example

```
var a;
expect(a).toBeUndefined(); // pass

var b = 1;
expect(b).toBeUndefined(); // fail

var c = null;
expect(c).toBeUndefined(); // fail
```

toContain

Checks an array to determine whether it contains the expected value.

API

```
expect(actualArray).toContain(expectedItem);
```

Example

```
var weapons = ["Walther PPK", "Walther P99"];
expect(weapons).toContain("Walther PPK"); // pass
expect(weapons).toContain("Colt 45");     // fail
```

toEqual

Checks whether the actual and expected values are equal. For primitives like strings and numbers, strict equality (===) is used. For objects and arrays, their members are compared with strict equality.

API

```
expect(actual).toEqual(expected);
```

Example

```
expect(1).toEqual(1); // pass
expect("1").toEqual(1); // fail

var villains = ["Professor Negative", "Silvernose"];
expect(villains).toEqual(["Professor Negative", "Silvernose"]); // pass
expect(villains).toEqual(["Professor Negative", "Number 2"]);   // fail

var agent = { designation: "Septimus" };
expect(agent).toEqual({ designation: "Septimus" }); // pass
expect(agent).toEqual({ designation: "Octavius" }); // fail
```

toHaveBeenCalled

Checks to see if a spy function has been called.

API

```
expect(spy).toHaveBeenCalled();
```

Example

```
var kgb = {
    agent: function(intel) {
        return intel;
    }
}

spyOn(kgb, "agent");

expect(kgb.agent).toHaveBeenCalled(); // fail

kgb.agent();

expect(kgb.agent).toHaveBeenCalled(); // pass
```

toHaveBeenCalledWith

Checks to see if a spy function has been called with a given set of arguments.

API

```
expect(spy).toHaveBeenCalledWith(arg1...);
```

Example

```
var kgb = {
    agent: function(intel) {
        return intel;
    }
}

spyOn(kgb, "agent");
kgb.agent("Secret Document");

expect(kgb.agent).toHaveBeenCalledWith("Weapon Plans");    // fail
expect(kgb.agent).toHaveBeenCalledWith("Secret Document"); // pass
```

toMatch

Checks to see if the actual value matches the expected regular expression.

API

```
expect(actual).toMatch(expectedRegex);
```

Example

```
var transmission = "coordinates: 125, 200";

expect(transmission).toMatch(/coordinates/); // pass
expect(transmission).toMatch(/plan/);        // fail
```

toThrow

Checks to see if a function throws an exception.

API

```
expect(theFunction).toThrow();
```

Example

```
function planA() {
  throw new Error("Kitten acquisition failed");
}

function planB() {
  return "Kitten acquired";
}

expect(planA).toThrow(); // pass
expect(planB).toThrow(); // fail
```

not

Chaining `not` between the `expect` function and the matcher function will reverse the meaning of the matcher.

API

```
expect(actual).not.toBe(unexpected);
```

Example

```
expect("7").not.toBe(7);   // pass
expect("7").not.toBe("7"); // fail
```