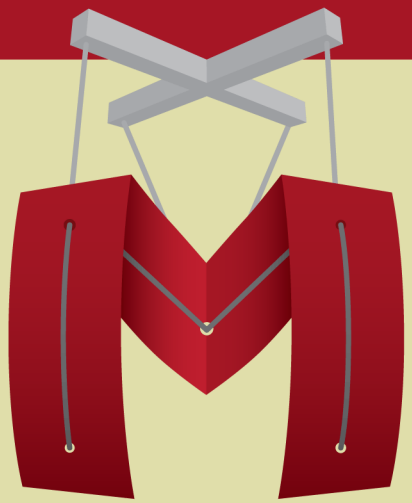
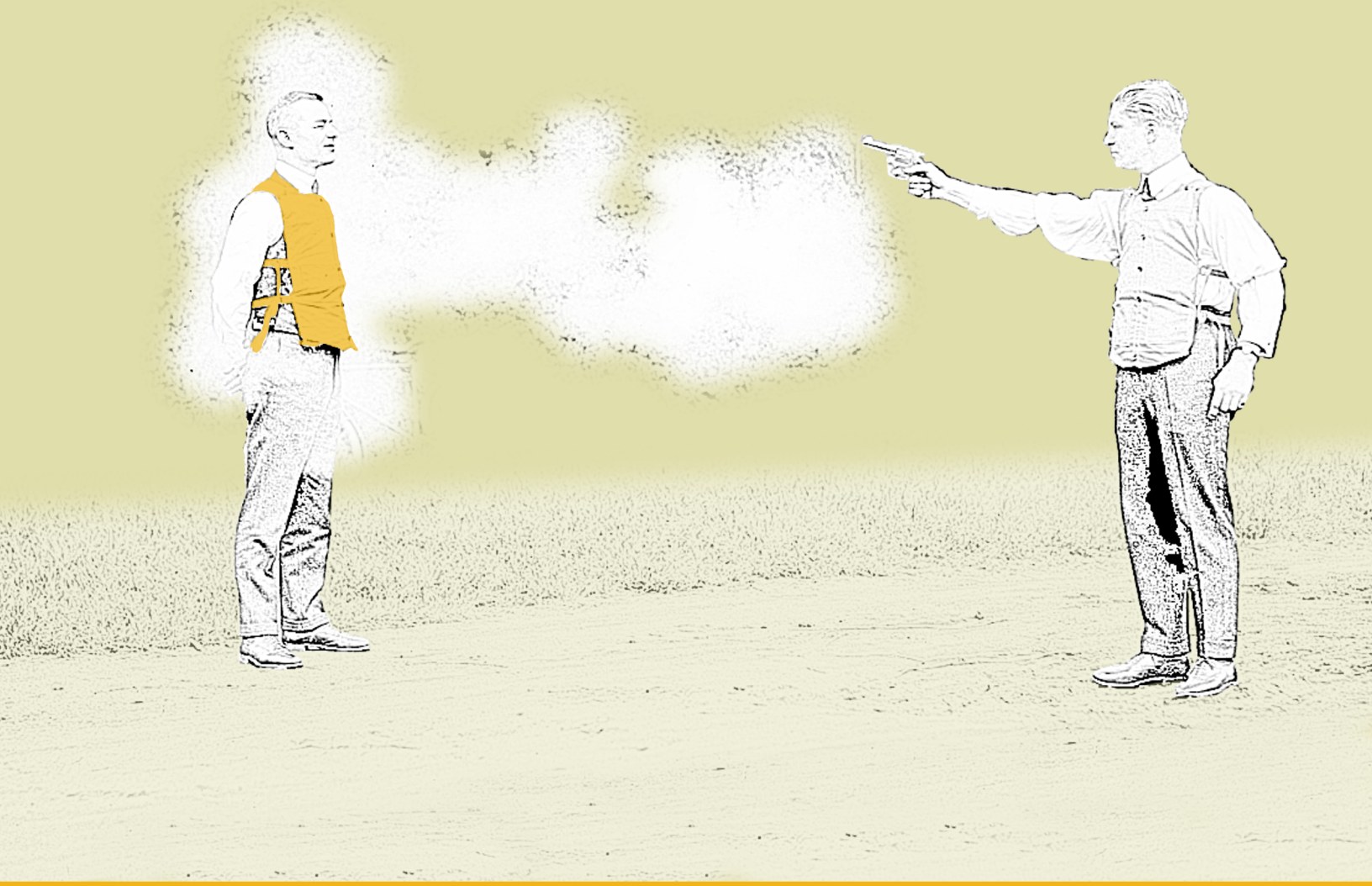
Improving Resilience and Maintainability



Backbone Marionette.js



Testing and Refactoring

by David Sulc

Marionette.js: Testing and Refactoring

Improving Resilience and Maintainability

David Sulc

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

This version was published on 2015-06-01

Also By David Sulc

Backbone.Marionette.js: A Gentle Introduction

Structuring Backbone Code with RequireJS and Marionette Modules

Backbone.Marionette.js: A Serious Progression

Contents

Cover Credits

The cover image depicts two men testing a bullet-proof vest. It is based on the public domain image available from Wikipedia[1].

[1] http://en.wikipedia.org/wiki/Bulletproof_vest#/media/File:Testing_bulletproof_vest_1923.jpg

Work in Progress

This book is currently being written. As such, content might shift around and code could be refactored, making following along slightly more "active". I always try to keep reader irritation (and therefore major changes) to a minimum, but the end objective is to produce a high-quality book. Put another way, you might have to deal with temporary annoyances in order to end up with a a book you can keep refering to in the future, and for this I apologize.

Who This Book is For

This book is for web developers who have built their first Marionette application (or are about to) and want to ensure their work will age well. This means new features sould be quick and painless to add, and existing functionality shouldn't break.

This book assumes familiarity with the "Contact Manager" application developed in the Backbone.Marionette.js: A Gentle Introduction[2] book (although you could pick it up as we go along), as well as passable javascript knowledge (no need to be a wizard).

[2] https://leanpub.com/marionette-gentle-introduction

Following Along with Git

This book covers writing tests for a Marionette.js application. As such, it's accompanied by source code in a Git repository hosted at https://github.com/davidsulc/marionette-testing[3].

The book and referenced Git commits all use Marionette 2.3.2.

Throughout the book, as we code our app, we'll refer to commit references within the git repository like this:

Git commit with our scaffold code:

bb6f4b54f853196fb597719286ea083718a21047[4]

This will allow you to follow along and see exactly how the codebase has changed: you can either look at that particular commit in your local copy of the git repository, or click on the link to see an online display of the code differences.

Any change in the code will affect all the following commit references, so the links in your version of the book might become desynchronized. If that's the case, make sure you update your copy of the book to get the new links. At any time, you can also see the full list of commits here[5], which should enable you to locate the commit you're looking for (the commit names match their descriptions in the book).

Even if you haven't used Git yet, you should be able to get up and running quite easily using online resources such as the Git Book[6]. This chapter is by no means a comprehensive introduction to Git, but the following should get you started:

- Set up Git with Github's instructions[7]
- To get a copy of the source code repository on your computer, open a command line and run

[3] https://github.com/davidsulc/marionette-testing
[4] https://github.com/davidsulc/marionette-testing/commit/bb6f4b54f853196fb597719286ea083718a21047
[5] https://github.com/davidsulc/marionette-testing/commits/master
[6] http://git-scm.com/book
[7] https://help.github.com/articles/set-up-git

```
git clone git://github.com/davidsulc/marionette-testing.git
```

- From the command line move into the `marionette-testing` folder that Git created in the step above, and execute

```
git show bb6f4b54f853196fb597719286ea083718a21047
```

to show the code differences implemented by that commit:

- '-' lines were removed
- '+' lines were added

You can also use Git to view the code at different stages as it evolves within the book:

- To extract the code as it was during a given commit, execute

```
git checkout bb6f4b54f853196fb597719286ea083718a21047
```

- Look around in the files, they'll be in the exact state they were in at that point in time within the book
- Once you're done looking around and wish to go back to the current state of the codebase, run

```
git checkout master
```

What if I don't want to use Git, and only want the latest version of the code?

You can download a zipped copy of the repository[8]. This will contain the full Git commit history, in case you change your mind about following along.

Jumping in for Advanced Readers

Although you'll learn the most by following along with the code, you can simply skim the content and checkout the Git commit corresponding to the point in the book where you wish to join in.

[8] https://github.com/davidsulc/marionette-testing/archive/master.zip

Why Test?

Well, you've cracked open a book on testing, so hopefully you're at least open to the possibility that code testing has some benefits. Whether you're the one needing convincing or you're looking for amunition to use in your arguments with peers and bosses, let's attempt to answer the question everybody asks when they first learn about the concept of testing code: why bother? Instead of writing code to test existing features we *know* are working (we've even clicked through the interface not just once, but FIVE times to make sure everything is fine!), shouldn't we invest that time cranking out new features our customers are waiting for? If we fix a bug as soon as it's discovered, everything will be fine, right?

How expensive a bug is to fix depends in part in *when* it is found, which is intuitive to any programmer who has wondered which idiot wrote the code he's looking at, only to find out he wrote it himself 6 months ago. But it turns out there's even real science to back this up: in the early 1980s, Barry Boehm found that the cost of making a change increases as you move from the stages of requirements analysis to architecture, design, coding, testing and deployment. A requirements mistake found and corrected while you are still defining the requirements costs almost nothing. But if you wait until after you've finished designing, coding and testing the system and delivering it to the customer, it costs a lot more.

In addition, writing a new feature (or changing an existing one) can have unintended consequences causing regressions. If you broke something, wouldn't it be best if you found out and fixed it before shipping to the customer?

Enjoyable Code

Nobody likes to fix a bug only to discover another dozen pop up, and everybody hates having to completely rewrite tightly coupled code to add a simple feature. What can you do to prevent that? Invest in your own happiness and maintain a test suite to:

- *leverage developer laziness* to produce decoupled code: if it's less work to test, it's less work to code for;
- *stop living in fear*: rely on regression tests to never commit broken code again;
- *work as a team* to ship complex software: your new features might have unintended consequences on your colleague's code (or be based on different assumptions) but the tests she wrote will bring them to light while your new code is still fresh in your mind;
- *increase code quality*: as corner cases are identified and added to the test suite, robustness automatically increases;

- *decrease code complexity*: complex code gets broken down into smaller chunks to keep tests small and independent;
- *contribute to documentation*: the way code is expected to behave (including edge cases) is clearly exposed within the test suite.

What Dogfights Can Teach Us

During the Korean War[9], North Korean and Chinese fighter pilots in Soviet-made MiG-15[10] planes were being dominated by American pilots flying F-86 Sabre[11] jets. By all accounts, the MiG had superior capabilities in turning, climbing, and acceleration; but American fighter pilots maintained an average kill ratio of 10:1 during the war[12]. How did this happen?

Colonel John Boyd[13] decided to investigate and determined that the F-86 had two points in its favor. First, it had better side visibility. While the MiG-15 pilot could see farther in front, the F-86 pilot could see slightly more on the sides. Second, while the F-86 had a hydraulic flight control, the MiG-15 had a manual flight control.

Without hydraulics, it took slightly more physical energy to move the MiG-15 flight stick than it did the F-85 flight stick. Even though the MiG-15 would turn faster (or climb higher) once the stick was moved, the amount of energy it took to move the stick was greater for the MiG-15 pilot. With each iteration, the MiG-15 pilot grew a little more fatigued than the F-86 pilot. And as he got more fatigued, it took just a little bit longer to *observe, orient, plan, and act* (which Boyd christened the OOPA loop). The MiG-15 pilot didn't lose because he got outfought. He lost because he got out-OOPAed.

This lead Boyd to discover that the primary determinant to winning dogfights was not observing, orienting, planning, or acting *better*. The primary determinant to winning dogfights was observing, orienting, planning, and acting *faster*. In other words, **speed of iteration beats quality of iteration** [14].

What insight can we glean from this to apply to the problem at hand? Maintain a suite of tests for your code, because it will enable your own OOPA loop to be faster: you'll know immediately if your code has unintended consequences, instead of waiting weeks or months and trying to remember why certain architectural choices had been made. Of course, you'll only benefit from this quicker OOPA loop if you run your tests after every change, so you need to keep them small and fast to execute.

[9] http://en.wikipedia.org/wiki/Korean_War
[10] http://en.wikipedia.org/wiki/Mikoyan-Gurevich_MiG-15
[11] http://en.wikipedia.org/wiki/North_American_F-86_Sabre
[12] http://www.sci.fi/~fta/JohnBoyd.htm
[13] http://en.wikipedia.org/wiki/John_Boyd_%28military_strategist%29
[14] http://blog.codinghorror.com/boyds-law-of-iteration/

What Types of Tests?

There are 3 main types of software testing levels:

- unit testing[15]: the focus of this book. Unit testing isolates each part of the program to show each one is correct. A unit test provides a strict, written contract that the piece of code must satisfy;
- integration testing[16]: occurs after unit testing, and combines individual software modules to test them as a group;
- system testing[17]: occurs after integration testing and is conducted on a complete, integrated system. As it falls in the scope of black box testing, it should require no knowledge of the inner design of the code or logic. System testing usually aims to exercise the software as it would be used in the wild (e.g. simulating a user clicking within a web interface and entering data).

Each level has its uses, and ideally all should be part of a comprehensive test suite. Back in reality, time and resources are limited, and you often have to make choices as to what should be tested. My experience is that the most "profitable" tests are the ones on each end of the spectrum: unit tests ensure each module works as expected (especially with edge cases), while system testing verifies the whole system works as the user expects. Since system tests tend to be much slower (both to write and run), their scope and quantity is sometimes reduced. That said, I would at least make a conscious effort to have system tests covering the user's happy path[18].

Naturally, integration tests also have their use (especially when different teams/developers have issues integrating their code), but my experience is that there isn't usually a need to test every possible integration: use them judiciously to ensure you're spending your coding time wisely.

[15] http://en.wikipedia.org/wiki/Unit_testing
[16] http://en.wikipedia.org/wiki/Integration_testing
[17] http://en.wikipedia.org/wiki/System_testing
[18] http://en.wikipedia.org/wiki/Happy_path

Setting Up

First, let's grab a copy of the source code for our basic Contact Manager application (as it was developed in my "Backbone.Marionette.js: A Gentle Introduction[19]" book) from here[20] and put it in a *contact_manager* folder. We're doing this for convenience, since we'll now have all libraries and assets (images, icons, etc.) available and won't have to deal with fetching them in the next chapters. Downloading the entire app also means we have all of our modules ready to go, simply waiting to be tested.

We've now got a copy of our original application ready to go, but we'll still need somewhere to run our tests and check their results. For this purpose, we'll create a *test* folder at the same level as the *contact_manager* folder. This *test* folder will contain all code relating to testing our application. In particular, let's add a *test.html* file to start our tests and display their results. Since our *test.html* file will need to include all of our app files to be able to test them, we'll simply copy over our original *index.html* in *test/test.html* to save ourselves some work. Here's what our file structure looks like right now:

- contact_manager
  - index.html
  - assets folder
- test
  - test.html (copy of index.html)

If you open *test.html* right now, it won't find any of the application files because their paths are incorrect. Don't worry about that right now, we'll get to it in a few pages.

Getting Mocha

Mocha[21] is the javascript test framework we'll use to write and execute our tests. Grab the javascript file from the book's code repo here[22] (Mocha's latest version is here[23]) and save it in *test/assets/js/vendor/mocha.js*. Also get the CSS file[24] and store it in *assets/css/mocha.css*.

[19] https://leanpub.com/marionette-gentle-introduction

[20] https://github.com/davidsulc/marionette-testing/archive/bb6f4b54f853196fb597719286ea083718a21047.zip

[21] http://mochajs.org/

[22] https://raw.githubusercontent.com/davidsulc/marionette-testing/master/test/assets/js/vendor/mocha.js

[23] https://raw.githubusercontent.com/mochajs/mocha/master/mocha.js

[24] https://raw.githubusercontent.com/mochajs/mocha/master/mocha.css

Now that the relevant Mocha files have been downloaded, we'll need to include them in *test.html*. While we're at it, we can also remove the references to the CSS files, as we won't be needing them in our tests (none of our app views will be displayed):

test/test.html

```
<!DOCTYPE html>
  <html lang="en">
    <head>
      <meta charset="utf-8">
      <title>Marionette Contact Manager Tests</title>
      <link href="./assets/css/bootstrap.css" rel="stylesheet">
      <link href="./assets/css/application.css" rel="stylesheet">
      <link href="./assets/css/jquery-ui-1.10.3.custom.css" rel="stylesheet">
      <link href="assets/css/mocha.css" rel="stylesheet">
      <script src="assets/js/vendor/mocha.js"></script>
    </head>
```

Mocha will also need us to provide a `div` for it to display the test results, so let's add that:

test/test.html

```
<body>
  <div id="mocha"></div>
```

Finally, we need to configure Mocha and have it run all tests when the *test.html* file gets loaded:

test/test.html

```
    <script src="./assets/js/apps/header/list/list_controller.js"></script>

    <script type="text/javascript">
      ContactManager.start();
      mocha.setup("bdd");
      window.onload = function () {
        mocha.run();
      };
    </script>
  </body>
</html>
```

As you can tell on line 5, we're going to use the "BDD" style simply because I find it more natural to write and read. Other test interfaces (e. g. TDD) can be used with Mocha (see documentation[25]). Then, on lines 6-8, we simply get Mocha to run our tests when the `window` is loaded.

If you open *test.html* in a browser, you'll see the static message from our app ("Here is static content in the web page. You'll notice that it gets replaced by our app as soon as we start it.") which didn't get removed because we never started our app. In addition to that, you'll see that Mocha inserted some test stats in the upper right hand corner.

Mocha test statistics

Those statistics look a little sad, don't they? We'll soon create our first test, but before that let's remove our placeholder text so it no longer distracts us:

test/test.html

```
1  <div id="main-region" class="container">
2    <p>Here is static content in the web page. You'll notice that it gets
3       replaced by our app as soon as we start it.</p>
4  </div>
```

We're going to put all of our tests inside a *test/assets/js/spec* folder (using a `.spec.js` suffix), so let's start by creating our "hello, world" test in there:

test/assets/js/spec/hello_world.spec.js

```
1  describe("Mocha", function(){
2    it("should work as expected");
3  });
```

Of course, Mocha won't run our new test if we don't add it to *test.html*, so let's do that right now (line 8):

[25] http://mochajs.org/#interfaces

test/test.html

```
<script type="text/javascript">
  mocha.setup("bdd");
  window.onload = function () {
    mocha.run();
  };
</script>

<script src="assets/js/spec/hello_world.spec.js"></script>
```

Before anything else, let's give it a whirl in the browser and see what happens:

Our first test

As you can see, 100% of our test suite ran in 0.03 seconds, with no tests passed and no tests failed. Why is that, when we just added a test? Because our test is currently empty, so it is considered in a "pending" state and therefore ignored: it is simply displayed with the results so you don't forget about it. Actually, if you hover over the test's name, Mocha will tell you it's pending:

Our first test: pending

So what did we put in our first test? Let's take another look at it:

test/assets/js/spec/hello_world.spec.js

```
describe("Mocha", function(){
  it("should work as expected");
});
```

First, we name a certain funcationality (in this case Mocha), then we provide a function to execute. This anonymous function will typically contain one or more call to the `it` function: these calls will test the functionality we're describing and make sure it works as expected. So if we read the above test in English, we could say "We're going to describe the "Mocha" functionality in here, and it should work as expected."

So far, however, we have yet to test anything. Let's fix that by providing a function as the second argument to the `it` call:

test/assets/js/spec/hello_world.spec.js

```
describe("Mocha", function(){
  it("should work as expected", function(){
    return true;
  });
});
```

We've added a function for Mocha to run and test. Depending on what happens when this function is run, Mocha will mark the test as passing or failing. Let's try it out:

Passing a test

This time around, we can see the test is passing: Mocha adds a green checkmark next to it, and the count of passed tests in the upper right-hand corner is 1. Let's now add a test that fails:

test/assets/js/spec/hello_world.spec.js

```
describe("Mocha", function(){
  it("should work as expected", function(){
    return true;
  });

  it("shouldn't throw an error", function(){
    throw new Error("Something broke!");
  });
});
```

When we run our test suite, we can see we now have 1 test passing and another failing:

Failing a test

We can take a look a the code for the failing test by clicking on the test:

Expanding a failed test

We now know how Mocha works on a simplistic level: as long as the function provided to `it` doesn't raise an exception when it is executed, the test is considered successful. Of course, these two tests still don't actually test anything, so we'll create a function to test as well as a helper function that will raise an exception if our code doesn't behave as expected. Delete everything in *hello_world.spec.js* and replace it with:

test/assets/js/spec/hello_world.spec.js

```
var assertEqual = function(a, b){
  if(a !== b){
    throw new Error("Expected " + a + " to be equal to " + b);
  }
};

var addTwo = function(a){
  return a + 2;
};

describe("addTwo", function(){
  it("should add 2 to an integer", function(){
    assertEqual(addTwo(3), 5);
  });

```

```
16    it("should cast strings to integers", function(){
17      assertEqual(addTwo("3"), 5);
18    });
19  });
```

What have we here? First, we define `assertEqual` which will compare both its arguments and raise an exception if they aren't equal. Then, we define the `addTwo` function we're going to test. Finally, we describe `addTwo`'s expected behavior:

- it should add 2 to an integer
- it should cast strings to integers

Let's run the tests and see what we get:

Running our tests

So our first test passed, but the second one failed because javascript will append to strings when using the "+" operator... Let's fix our code and run the test again:

test/assets/js/spec/hello_world.spec.js

```
// assertEqual edited for brevity

var addTwo = function(a){
  return parseInt(a, 10) + 2;
};

describe("addTwo", function(){
  // edited for brevity

  it("should cast strings to integers", function(){
    assertEqual(addTwo("3"), 5);
  });
});
```

And now, our tests can prove our function works as expected:

Running our corrected tests

Of course, we're going to do a lot more than checking function results are equal to some result: we'll want to check they are not equal, greater than some value, etc. Instead of writing all of these helper functions ourselves, we'll be using Chai to care of that for us.

Getting Chai

Download Chai from the book's code repo here[26] (Chai's latest version is here[27]), save it in *test/assets/js/vendor/chai.js*, and include it in *test.html*:

[26] https://raw.githubusercontent.com/davidsulc/marionette-testing/master/test/assets/js/vendor/chai.js
[27] http://chaijs.com/chai.js

test/test.html

```
<title>Marionette Contact Manager Tests</title>
<link rel="stylesheet" href="assets/css/mocha.css" />
<script src="assets/js/vendor/mocha.js"></script>
<script src="assets/js/vendor/chai.js"></script>
```

We'll now be able to use Chai's functions to assert the tested code behaves as it should. Let's modify our tests to reflect that:

test/assets/js/spec/hello_world.spec.js

```
var assertEqual = function(a, b){
  if(a !== b){
    throw new Error("Expected " + a + " to be equal to " + b);
  }
};

var addTwo = function(a){
  return parseInt(a, 10) + 2;
};

describe("addTwo", function(){
      it("should add 2 to an integer", function(){
        chai.expect(addTwo(3)).to.equal(5);
      });

      it("should cast strings to integers", function(){
        chai.expect(addTwo("3")).to.equal(5);
      });
});
```

This is a good start, but constantly typing `chai.expect` is going to be annoying, and also reduces readability. So let's fix that by attaching the `expect` method to `window` so it becomes globally accessible:

test/test.html

```
<script type="text/javascript">
  mocha.setup("bdd");
  window.expect = chai.expect;
```

We can now tweak our test code:

test/assets/js/spec/hello_world.spec.js

```
describe("addTwo", function(){
  it("should add 2 to an integer", function(){
    expect(addTwo(3)).to.equal(5);
  });

  it("should cast strings to integers", function(){
    expect(addTwo("3")).to.equal(5);
  });
});
```

Now that we've written our first tests, let's get cracking!

The "hello, world" test code won't be of any use to us in the future, so go ahead and delete file *test/assets/js/spec/hello_world.spec.js*, and remove it from *test.html*:

```
      </script>

    ~~<script src="assets/js/spec/hello_world.spec.js"></script>~~
  </body>
```

Although we'll be using the `expect` interface to verify our assertions throughout the book, other styles[28] are available. Where in the book we would use

```
expect(foo).to.equal("bar");
```

you could achieve the same results using

- `foo.should.equal("bar");`
- `assert.equal(foo, "bar");`

[28] http://chaijs.com/guide/styles/

Header Entity

Header Model

Let's start by testing our Header models. We'll organize our test files to mirror the app file organization:

- *contact_manager/assets/js/entities/header.js* -> the original application file we're going to test
- *test/assets/js/spec/entities/header.spec.js* -> the location of the corresponding test file

All we're going to check in our header model is that it is selectable. Here's what we'll start with in our test file:

test/assets/js/spec/entities/header.spec.js

```
describe("Header entity", function(){
  describe("Model", function(){
    it("defines (de)selection functions");
    it("is selectable");
    it("is not 'selected' by default");
  });
});
```

We're nesting `describe` blocks to organize our tests: each `describe` block will group tests that belong together logically. So far, we've got a group of tests for the model, and later we'll add a second group for the collection. Then, within the test group for the model, we add a list of things we want to check. Great!

Before we forget, let's add this new file to *test.html*:

test/test.html

```
    </script>

    <script src="assets/js/spec/entities/header.spec.js"></script>
  </body>
</html>
```

In our first test, we want to check that header models define `select` and `deselect` functions. Here we go:

test/assets/js/spec/entities/header.spec.js

```
  it("defines (de)selection functions", function(){
    var header = new ContactManager.Entities.Header();
    expect(typeof(header.select)).to.equal("function");
    expect(typeof(header.deselect)).to.equal("function");
  });
```

We're defining a header model instance to use, then testing the existence of the functions using javascript's `typeof`[29] operator.

If we open our test page in a browser, however, we're told that `ContactManager` doesn't exist... Why is that when we're including our entire application in *test.html*? Well, because the paths aren't correct. Let's fix that (you'll need to change the URLs for all included application files, not just the ones shown):

test/test.html

```
<!-- edited for brevity -->

~~<script src="./assets/js/app.js"></script>~~
<script src="../contact_manager/assets/js/app.js"></script>

<!-- edited for brevity -->

~~<script src="./assets/js/entities/header.js"></script>~~
<script src="../contact_manager/assets/js/entities/header.js"></script>

<!-- edited for brevity -->
```

Our first test now works properly!

Git commit with our first header test:

f48d3b8b66f8bf4442ccc3323fa746f7996c0fa9[30]

Let's now write another test:

[29] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof

[30] https://github.com/davidsulc/marionette-testing/commit/f48d3b8b66f8bf4442ccc3323fa746f7996c0fa9

test/assets/js/spec/entities/header.spec.js

```
1  describe("Header entity", function(){
2    describe("Model", function(){
3      it("defines (de)selection functions", function(){
4        var header = new ContactManager.Entities.Header();
5        expect(typeof(header.select)).to.equal("function");
6        expect(typeof(header.deselect)).to.equal("function");
7      });
8
9      it("is selectable", function(){
10       var header = new ContactManager.Entities.Header();
11       header.select();
12       expect(header.selected).to.be.true;
13       header.deselect();
14       expect(header.selected).to.be.false;
15     });
16
17     it("is not 'selected' by default");
18   });
19 });
```

Another test in the bag! You'll notice that Chai provides us with "readable assertions" to check whether a value is true or false. You can also invert the assertion by using the `not` operator in the chain, for example:

```
expect(header.selected).to.not.be.true;
```

Sadly, we've started to introduce some duplication in our tests: lines 4 and 10 both declare a `header` model instance to perform tests on. Since we're going to need this model in each of our tests, it would be great if we could have some sort of function to prepare our tests to spare us from writing boring boilerplate code. Luckily, we aren't the first to think of this, and it's nicely provided to us via Mocha's `beforeEach` and `afterEach` hooks[31] which, as their names imply, run before and after each test in the `describe` block they belong to. Let's use them:

[31] http://mochajs.org/#hooks

test/assets/js/spec/entities/header.spec.js

```
1  describe("Header entity", function(){
2    describe("Model", function(){
3      beforeEach(function(){
4        this.header = new ContactManager.Entities.Header();
5      });
6
7      afterEach(function(){
8        delete this.header;
9      });
10
11     it("defines (de)selection functions", function(){
12       expect(typeof(this.header.select)).to.equal("function");
13       expect(typeof(this.header.deselect)).to.equal("function");
14     });
15
16     it("is selectable", function(){
17       this.header.select();
18       expect(this.header.selected).to.be.true;
19       this.header.deselect();
20       expect(this.header.selected).to.be.false;
21     });
22
23     it("is not 'selected' by default");
24   });
25 });
```

Mocha's hooks allow us to add attributes to `this` (line 4), which is the context within which all tests are run. We then simply refer to the attribute on `this` within a test to obtain the desired value (line 12). Naturally, once our test block has run, we need to clean everything up so these tests don't interfere with any other tests we might run in another block: our `afterEach` function will remove the attribute we've added.

Note that although we don't really reduce the number of lines in our test file, we're vastly improving the maintainability of our tests: each test contains only the actual code we're exercising, which means failed tests will be quicker to debug (since no setup code will interfere). In addition, centralizing setup code will allow us to easily update our tests if the application code gets refactored (e.g. the `Header` module gets moved to another sub-module).

And here's our last test:

test/assets/js/spec/entities/header.spec.js

```
1  describe("Header entity", function(){
2    describe("Model", function(){
3      // edited for brevity
4
5      it("is not 'selected' by default", function(){
6        expect(this.header.selected).to.not.be.ok;
7      });
8    });
9  });
```

Line 6 has an interesting assertion: `ok`. It means that the argument will evaluate to something truthy. Since we're negating it in our use, we want to make sure that the `selected` attribute is falsy: `false`, `null`, `undefined` would all be acceptable values for us.

Not Testing Functionality

There needs to be a limited scope to our tests, or we'll just keep adding to our test suite with continually diminishing returns. Notice that in our tests above for example, we don't test `get` to ensure it works as expected. Since the model's `get` method is provided by Backbone, we can assume it's working correctly (as testing it should be Backbone's responsibility, not ours).

Testing is great, but don't get carried away!

The rest of this chapter is not part of the sample. I hope you've enjoyed reading so far, and hope you'll go further in your testing adventures with the complete book available here[32].

[32] https://leanpub.com/marionette-testing

Chapters not in Sample

This is a sample of the book, several chapters are absent.

You can get the complete book at https://leanpub.com/marionette-testing/[33].

[33] https://leanpub.com/marionette-testing/

Contacts New View

Implement the tests for the contact app's new view yourself. This is the code it contains:

contact_manager/assets/js/apps/contacts/new/new_view.js

```
ContactManager.module("ContactsApp.New", function(New, ContactManager,
                                                   Backbone, Marionette, $, _){
  New.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
    title: "New Contact",

    onRender: function(){
      this.$(".js-submit").text("Create contact");
    }
  });
});
```

Here are some ideas of what to test:

test/assets/js/spec/apps/contacts/new/new_view.spec.js

```
describe("ContactsApp.New.Contact", function(){
  it("inherits from ContactsApp.Common.Views.Form");
  it("sets the 'title' attribute to 'New Contact'");
  it("sets the submit button text to 'Create contact'");
});
```

A few hints:

- javascript's `instanceof`[34] operator will tell you whether an object is an instance of another, and can also be used to check "inheritance";
- don't forget to provide a contact model instance when you create the view;
- you can use jQuery's `find`[35] and `text`[36] functions to determine a DOM element's text value;

When you're ready, move on to the next page. See you there!

[34] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof
[35] http://api.jquery.com/find/
[36] http://api.jquery.com/text/

Before we'll be able to do any testing at all, we need some setup code:

test/assets/js/spec/apps/contacts/new/new_view.spec.js

```
describe("ContactsApp.New.Contact", function(){
  before(function(){
    this.$fixture = $("<div>", { id: "fixture" });
    this.$container = $("#view-test-container");
  });

  after(function(){
    delete this.$fixture;
    this.$container.empty();
    delete this.$container;
  });

  beforeEach(function(){
    this.$fixture.empty().appendTo(this.$container);

    this.view = new ContactManager.ContactsApp.New.Contact({
      el: this.$fixture,
      model: new ContactManager.Entities.Contact()
    });
  });

  afterEach(function(){
    delete this.view;
  });
});
```

And then, on to the tests:

test/assets/js/spec/apps/contacts/new/new_view.spec.js

```
// edited for brevity

afterEach(function(){
  delete this.view;
});

it("inherits from ContactsApp.Common.Views.Form", function(){
  var CommonForm = ContactManager.ContactsApp.Common.Views.Form;
  expect(this.view instanceof CommonForm).to.be.true;
```

```
10  });
11
12  it("sets the 'title' attribute to 'New Contact'", function(){
13    expect(this.view.title).to.equal("New Contact");
14  });
15
16  it("sets the submit button text to 'Create contact'", function(){
17    this.view.once("render", function(){
18      expect(this.$el.find(".js-submit").text()).to.equal("Create contact");
19    });
20    this.view.render();
21  });
```

Let's take a few moments to discuss that last test. We attach our event handler on `this.view`, which means the callback's context will be the value of `this.view` as determined on line 17. Therefore, on line 18, `this` will evaluate to the view instance we are currently testing, which in turn means we can access its `$el` property to perform our tests.

As you may remember, `view.$el` is a convenient shortcut to `$(view.el)`.

If we tried to write our callback like this

```
this.view.once("render", function(){
  expect(this.view.$el.find(".js-submit").text()).to.equal("Create contact");
});
```

our tests would raise an error saying "`this.view` is undefined". That's because our callback's context is the tested view, and `this.view` would therefore be equivalent to tested view's `view` property, which is undefined.

Note that there are other ways to achieve correct test code that might be more readable to you or other members on your team. Here are two:

```
var self = this;
this.view.once("render", function(){
  expect(self.view.$el.find(".js-submit").text()).to.equal("Create contact");
});
```

and

```
var testedView = this.view;
this.view.once("render", function(){
  expect(testedView.$el.find(".js-submit").text()).to.equal("Create contact");
});
```

Both of these take into account that javascript will create a new scope within a function, so they declare variables before the function. Since those variables exist in the scope when the function is created, they will exist within the function's closure. Problem solved!

Git commit with our tests for the contact app's 'new' view:

344d0b1deea61ee4acdff34cbfd042c392bf8df3[37]

[37] https://github.com/davidsulc/marionette-testing/commit/344d0b1deea61ee4acdff34cbfd042c392bf8df3

Chapters not in Sample

This is a sample of the book, several chapters are absent.

You can get the complete book at https://leanpub.com/marionette-testing/[38].

[38] https://leanpub.com/marionette-testing/

About this Sample

You're currently viewing a sample of the book's content. Please note that the table of contents included here is **NOT** the full contents of the book. To see the full (intended) contents, please refer to the book's landing page at https://leanpub.com/marionette-testing[39].

Thanks for reading!

[39] https://leanpub.com/marionette-testing