



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Jasmine JavaScript Testing

Leverage the power of unit testing to create bigger and better JavaScript applications

Paulo Ragonha

[PACKT] open source*
PUBLISHING community experience distilled

Jasmine JavaScript Testing

Leverage the power of unit testing to create bigger and better JavaScript applications

Paulo Ragonha



BIRMINGHAM - MUMBAI

Jasmine JavaScript Testing

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1190813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-720-4

www.packtpub.com

Cover Image by Alexandre Santiago (xanps71@gmail.com)

Credits

Author

Paulo Ragonha

Project Coordinator

Michelle Quadros

Reviewers

Diego Castorina

Prateek Dayal

Zeno Rocha

Sergey Simonchik

Proofreader

Christopher Smith

Indexer

Mariammal Chettiyar

Acquisition Editor

Grant Mizen

Graphics

Sheetal Aute

Commissioning Editor

Harsha Bharwani

Production Coordinator

Shantanu Zagade

Technical Editors

Pooja Arondekar

Aparna K

Larissa Pinto

Cover Work

Shantanu Zagade

About the Author

Paulo Ragonha is a software engineer. He loves web development for the opportunities that it carries; "to be able to craft a piece of software that can be instantly accessible by anyone" (with internet connection, of course).

In his early days of software development, he was mostly involved in game development and Java. But since his discovery of Ruby and JavaScript, he has worked uniquely on web applications.

His last three projects were big JavaScript applications, developed entirely driven by tests and with amazing tooling support.

He has an amazing wife that he loves very much, lives in the beautiful Florianópolis, a coast city in the south of Brazil. He is a casual speaker, a biker, a runner, and a hobbyist photographer (he has earned an actual award taking pictures).

I would like to thank Juliana, my wife, for supporting me during the writing of this book. It was not easy for her to have her husband almost unavailable for over two months. I would also like to thank my family for cheering me when I needed the most, and my friends, who were there for me, helping me boot up this book. I would specially like to thank Felipe Munhoz who was always available to hear me whinging about my texts, and Fabiano Soriani who replied to every single e-mail I've sent to him about the queries regarding the book. Thank you all!

About the Reviewers

Diego Castorina is a Software Engineer with more than 7 years of experience building Web Applications and SOA Systems.

After graduating in Computer Science at the University of Catania, Sicily, he moved to work in the Netherlands first and in the Czech Republic afterwards.

He has extensive knowledge and experience with Java, Ruby, Scala, and JavaScript, of course.

During the past couple of years he has focused his attention on JavaScript because of the fundamental role of the language in the future of the Web.

Prateek Dayal is the co-founder of SupportBee. In SupportBee, he works with a great technology team to build the world's best collaborative customer support software. He is a Ruby on Rails and JavaScript programmer by training, but he is now trying to learn the ropes of business in his new CEO role. He is also writing a book on 'Building Single Page Web Applications' that will be available free of cost at `singlepagebook.supportbee.com`.

Prateek enjoys traveling and working from new cities and countries. He is currently in Vietnam exploring Vietnamese food and culture. He is happy to be your tour guide if you are visiting Vietnam. Tweet to him @prateekdayal.

Zeno Rocha used to work as a software developer at Petrobras, (the largest company in Latin America) and Globoesporte.com (the most accessed sports website in Brazil), now he's a front-end engineer at Liferay focused on the AlloyUI project. Besides, he's a writer at *Smashing Magazine*, host at Zone Of Front-Enders podcast, and co-founder of BrazilJS Foundation. At only 22 years of age, he's one of the top 20 most active users on GitHub and a reference in Brazil, when the subject is frontend development.

Sergey Simonchik is a software developer living and working in Saint-Petersburg, Russia. He is lucky because he has a wonderful wife and a kind cat. Sergey develops WebStorm IDE at JetBrains. He is working on improving JavaScript unit testing support and other IDE features.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Getting Started with Jasmine	7
JavaScript – the bad parts	7
Jasmine and Behavior-Driven Development	8
Downloading and first run	10
Summary	12
Chapter 2: Your First Spec	13
The Investment Tracker application	13
Jasmine basics and thinking in BDD	14
Setup and teardown	19
Nested describes	24
Setup and teardown	24
Coding a spec with shared behavior	25
Understanding matchers	26
Custom matchers	26
Built-in matchers	30
The toEqual built-in matcher	31
The toBe built-in matcher	31
The toBeTruthy and toBeFalsy matchers	33
The toBeUndefined, toBeNull, and toBeNaN built-in matchers	34
The toBeDefined built-in matcher	35
The toContain built-in matcher	35
The toMatch built-in matcher	36
The toBeLessThan and toBeGreaterThan built-in matchers	36
The toThrow built-in matcher	36
Summary	37

Chapter 3: Testing Frontend Code	39
Thinking in terms of components (Views)	40
The module pattern	41
Using HTML fixtures	42
Basic View coding rules	46
The View should encapsulate a DOM element	46
Integrate Views with observers	49
Testing Views with jQuery matchers	53
The toBe jQuery matcher	54
The toContainHtml jQuery matcher	55
The toContain jQuery matcher	55
The toHaveValue jQuery matcher	55
The toHaveAttr jQuery matcher	55
The toBeFocused jQuery matcher	56
The toBeDisabled jQuery matcher	56
More matchers	56
Summary	56
Chapter 4: Asynchronous Testing – AJAX	57
Setting up the scenario	58
Installing Node.js	58
Coding the server	59
Running the server	60
Writing the spec	60
The waitsFor() function	61
The runs() function	63
Summary	65
Chapter 5: Jasmine Spies	67
The "bare" Spy	67
Spy an object's functions	68
Testing DOM events	69
Summary	69
Chapter 6: Light Speed Unit Testing	71
Jasmine Stubs	71
Sinon.JS	73
Installing Sinon.JS	73
A Fake XMLHttpRequest	73
A Fake server	75
Summary	77

Chapter 7: Testing Backbone.js Applications	79
The Backbone model	79
Declaring a new model	80
The model attributes	80
Default attribute values	82
Events	83
Sync and AJAX requests	84
Testing Backbone collections	87
Declaring a new collection	88
Sync and AJAX requests	89
Testing Backbone Views	91
Declaring a new View	92
The el property	93
Rendering	95
Updating the View on model changes	96
Binding DOM events	98
Testing Backbone routers	101
Defining a new router	101
Defining routes	102
Using routers	103
Routers should only route	103
Summary	104
Chapter 8: Build Automation	105
RequireJS	106
Module definition	106
Project setup	107
The new SpecRunner.JS file	108
The RequireJS configuration	109
Testing a module	111
Optimizing for production	114
Headless testing with PhantomJS	115
Downloading and installing PhantomJS	116
Running your tests without opening a browser	116
Grunt	116
Installation	117
Project setup	117
A RequireJS optimizer task	117
A Jasmine task	119

Table of Contents

Watch for changes and run the specs	120
Managing NPM dependencies	120
Summary	122
Chapter 9: Conclusion	123
Index	125

Preface

This book is about being a better JavaScript developer. So throughout the chapters, the reader will not only learn about writing tests in the Jasmine "idiom", but also about the best practices on writing a software in JavaScript language. It is about acknowledging JavaScript as a real platform for application development, and leveraging all its potential. It is about tooling and automation, and how to make your life easier and more productive.

Most importantly, this book is about craftsmanship in producing not only working software, but also well-crafted software.

The *Jasmine JavaScript Testing* book is a practical guide to writing and automating JavaScript testing for web applications. It uses technologies such as Jasmine, Sinon.JS, RequireJS, and Grunt.

Over the course of the chapters, the concept of test-driven development is explained through the development of a simple Stock Market Investment Tracker Application. It starts with the basics of testing through the development of the base domain classes (such as Stock and Investment), passing through the concepts of maintainable browser code, and concluding with a full refactoring to a Backbone.js application with RequireJS dependency management, and automated build.

What this book covers

Chapter 1, Getting Started with Jasmine, covers the motivations behind testing a JavaScript application. It presents the concept of BDD and how it helps you to write better tests. It also demonstrates how easy it is to download Jasmine and start coding your first test.

Chapter 2, Your First Spec, helps you learn the thought process behind thinking in test-driven development. You will code your very first JavaScript functionality driven by tests. You will also learn the basic functions of Jasmine, and how to structure your tests. It also demonstrates how Jasmine matchers work, and how you can create one of your own to improve your tests' code readability.

Chapter 3, Testing Frontend Code, covers some patterns in writing maintainable browser code. You will learn about thinking in components, and how to use the Module Pattern to better organize your source files. You will also be presented with the concept of HTML fixtures, and how you can use it to test your JavaScript code without requiring your servers to render a HTML. You will also learn about a Jasmine plugin called Jasmine jQuery, and how it can help you write better tests with jQuery.

Chapter 4, Asynchronous Testing – AJAX, talks about the challenges in testing AJAX requests, and how you can use Jasmine to test any asynchronous code. You will learn about NodeJS, and how to create a very simple HTTP server to use as a fixture to your tests.

Chapter 5, Jasmine Spies, presents the concept of test doubles and how to use Spies to perform behavior checking.

Chapter 6, Light Speed Unit Testing, helps you learn about the issues with AJAX testing, and how you can make your tests run faster by using Stubs or Fakes (such as SinonJS Fake Server).

Chapter 7, Testing Backbone.js Applications, presents the four base abstractions of Backbone.js, and how you can write tests to a Backbone.js application. You will learn about what to test, and not to test, when using Backbone.js.

Chapter 8, Build Automation, presents you the power of automation. You will learn about RequireJS, and how it helps you organize the dependencies of large JavaScript applications. You will start to think in modules and their dependencies, and you will learn how to code your tests as modules. You will also learn about packing and minifying the code to production, and how to automate this process. Then you are going to use PhantomJS to run all your tests without a browser window, and even how to make them run automatically on any file change.

Chapter 9, Conclusion, talks about the future of testing and tooling in the JavaScript ecosystem.

Chapter 10, Testing jQuery Plugins, presents the concept of jQuery plugins and how you can use Jasmine to drive the development of a plugin of your own. This bonus chapter is available at the following link: http://www.packtpub.com/sites/default/files/downloads/72040S_Testing_jQuery_Plugins.pdf.

Chapter 11, Continuous Integration with Travis CI, teaches you how to set up a continuous integration environment for a JavaScript application using Travis CI. This bonus chapter is available at the following link: http://www.packtpub.com/sites/default/files/downloads/72040S_Continuous_Integration_with_Travis_CI.pdf.

Chapter 12, The Future: Jasmine 2.0, gives you a glimpse on what are the major changes and how to roll an incremental migration to the upcoming Jasmine 2.0 release. This bonus chapter is available at the following link: http://www.packtpub.com/sites/default/files/downloads/72040S_The_Future_Jasmine_2_0.pdf.

Who this book is for

This book is a must-have material for web developers new to the concept of unit testing. It's assumed basic knowledge of JavaScript and HTML.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Jasmine provides another global function called `beforeEach`."

A block of code is set as follows:

```
describe("Investment", function() {  
    it("should be of a stock", function() {  
        expect(investment.stock).toBe(stock);  
    });  
});
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in **bold**:


```
describe("Investment", function() {  
    it("should be of a stock", function() {  
        var investment = new Investment();  
        expect(investment.stock).toBe(stock);  
    });  
});
```


Any command-line input or output is written as follows:

```
# npm install -g requirejs
```

New **terms** and **important words** are shown in bold.

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started with Jasmine

It is an exciting time to be a JavaScript developer; technologies have matured, web browsers are more standardized and there are new things to play with every day. JavaScript has become an established language, and the Web is the true open platform of today. We've seen the rise of single-page web applications, the proliferation of MVC frameworks such as Backbone.js and AngularJS, the use of JavaScript on the server with Node.js, and even mobile applications created entirely with HTML, JavaScript, and CSS using technologies such as PhoneGap.

From its humble beginning with handling HTML forms, to the massive applications of today, the JavaScript language has come very far, and with it, a number of tools have matured to ensure that you can have the same level of quality with it that you have with any other language.

This book is about the tools that keep you in control of your JavaScript development.

JavaScript – the bad parts

There are many complications when dealing with client JavaScript code; the obvious one is that you cannot control the client's runtime. While on the server you can run a specific version of your Node.js Server, you can't oblige your clients to run the latest version of Chrome or Firefox.

The JavaScript language is defined by the ECMAScript specification; therefore each browser can have its own implementation of a runtime, meaning there could be small differences or bugs between them.

Besides that, you have issues with the language itself. JavaScript was developed by Brendan Eich in just 10 days, under a lot of management pressure on Netscape. Although it got itself right in its simplicity, first-class functions, and object prototypes, it also introduced some problems with the attempt to make the language malleable and allow it to evolve.

Every JavaScript Object is mutable; this means that there is nothing you can do to prevent a module from overwriting pieces of other modules. The following code illustrates how simple it is to overwrite the global `console.log` function.

```
console.log('test');
>> 'test'
console.log = 'break';
console.log('test');
>> TypeError: Property 'log' of object #<Console> is not a
    function
```

This was a conscious decision on the language design; it allows developers to tinker and add missing functionality to the language. But given such power, it is relatively easy to make a mistake.

A newer version of the language introduced the `Object.seal` function, which prevents further changes on any object once called. But its current support is not widespread; it was only introduced on Internet Explorer 9 and is currently missed on Opera.

Another problem is in how JavaScript deals with type. In other languages an expression like `'1' + 1` would probably raise an error, in JavaScript, due to some non-intuitive type coercion rules, the above code results in `'11'`. But the main problem is in its inconsistency; on a multiplication the string is converted to a number, so `'3' * 4`, is actually 12.

This can lead to some hard-to-find problems on big expressions. Suppose you have some data coming from a server, and although you are expecting numbers, one value came as a string:

```
var a = 1, b = '2', c = 3, d = 4;
var result = a + b + c * d;
```

The above result value is `'1212'`, a string.

These are just two common problems faced by developers. Throughout the book, you are going to be applying best practices and writing tests to guarantee that you don't fall on these and other pitfalls.

Jasmine and Behavior-Driven Development

Jasmine is a little Behavior-Driven Development framework, created by the guys at Pivotal Labs to allow you to write automated JavaScript unit tests.

But before we can go any further, first we need to get some fundamentals, starting with what is a test unit.

A test unit is a piece of code that tests a functionality unit of the application code. But sometimes it can be tricky to understand what a functionality unit can be, so for that reason, Dan North came up with a solution in the form of **Behavior-Driven Development (BDD)**, which is a rethink of **Test-Driven Development (TDD)**.

In traditional unit testing practice, the developer is left with no guidelines on how to start in the process of testing, what to test, how big a test should be or even how to call a test.

To fix these problems, Dan took the concept of **user stories** from the standard agile construct, as a model on how to write tests.

For example, a music player application could have an acceptance criterion like:

Given a player, **when** the song has been paused **then** it should indicate that the song is currently paused.

As you can see, this acceptance criterion is written following an underlying pattern:

- **given** [an initial context]
- **when** [an event occurs]
- **then** [ensure some outcome]

In Jasmine this translates into a very expressive language that allows tests to be written in a way that reflects actual business values. The above acceptance criterion written as a Jasmine test unit would be:

```
describe("Player", function() {  
  describe("when song has been paused", function() {  
    it("should indicate that the song is paused", function() {  
  
    });  
  });  
});
```

You can see how the criterion translates well into the Jasmine syntax. In the next chapter we will get into the details of how these functions work.

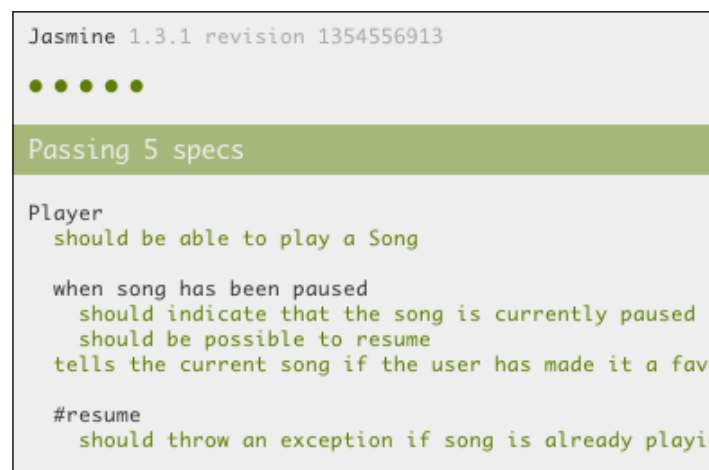
With Jasmine, as with other BDD frameworks, each acceptance criterion directly translates to a test unit. For that reason, each test unit is usually called a **spec**, short for specification. During the course of this book, we will be using this terminology.

Downloading and first run

To get started with Jasmine is actually pretty simple. Open the Jasmine website at <http://pivotal.github.io/jasmine/> and download the Standalone Release.

While at the Jasmine website, you might notice that it is actually a live page, executing the specs contained on it. This is made possible by the simplicity of the Jasmine framework, allowing it to be executed in the most diverse environments.

After you've downloaded the distribution and uncompressed it, you can open the `SpecRunner.html` file on your browser. It will show the results of a sample test suite (including the acceptance criterion we showed you earlier):



SpecRunner.html opened on the browser

This `SpecRunner.html` file is a Jasmine browser spec runner. It is a simple HTML file that references the Jasmine code, the source files, and the test files. For convention purposes we are going to refer to this file simply as **runner**.

You can see how simple it is by opening it on a text editor. It is a small HTML file that references the Jasmine source:

```
<script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.js"></script>
<script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-html.js"></script>
```

References the source files:

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

References the spec files:

```
<script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

And contains a small Jasmine boot code:

```
var jasmineEnv = jasmine.getEnv();
jasmineEnv.updateInterval = 1000;

var htmlReporter = new jasmine.HtmlReporter();

jasmineEnv.addReporter(htmlReporter);

jasmineEnv.specFilter = function(spec) {
  return htmlReporter.specFilter(spec);
};

var currentWindowOnload = window.onload;

window.onload = function() {
  if (currentWindowOnload) {
    currentWindowOnload();
  }
  execJasmine();
};

function execJasmine() {
  jasmineEnv.execute();
}
```

In *Chapter 8, Build Automation*, we are going to get a deep understanding on how to customize this runner, but for now you can see that:

- It configures a `jasmine.HtmlReporter`, to report the test results on HTML
- It adds a callback on `window.onload`
- And it executes the Jasmine test suite through `jasmineEnv.execute()` once the window has been loaded

It is important that you take a glimpse at this code to understand the flexibility that Jasmine provides; you can see that it has no dependencies, and even though the results are being rendered on an HTML file, this can be easily changed by switching the reporter:

```
jasmineEnv.addReporter(new jasmine.TerminalReporter());
```

And then running the same code on a **headless browser** such as PhantomJS, and having the results written on the console.

A headless browser is a browser environment without its graphical user interface. It can either be an actual browser environment, such as PhantomJS, which uses the WebKit rendering engine, or a simulated browser environment, such as Envjs.

But Jasmine can also be used to test server code, using the same reporter on a Node.js runtime.

This Jasmine flexibility is amazing, because you can use the same tool to test all sorts of JavaScript code.

Summary

In this chapter, you have seen some of the motivations behind testing a JavaScript application. I showed you some common pitfalls of the JavaScript language and how BDD and Jasmine both help you write better tests.

You have also seen how easy it is to download and get started with Jasmine. In the next chapter, you are going to learn how to think in BDD and code your very first spec.

2

Your First Spec

This chapter is about the basics. We are going to guide you through how to write your first spec, thinking in test first, and also show all the available global Jasmine functions. By the end of the chapter, you should know how Jasmine works and be ready to start doing your first tests by yourself.

The Investment Tracker application

To get you started, we need an example scenario: consider you are developing an application to track investments in the stock market.

The following form illustrates better how a user might create a new investment on this application:

Symbol:	Shares:	Share price:	
PETO	100	35	Add

Form to add investments

This form will allow the input of three values that define an investment:

- A **Symbol** which represents what company (stock) the user is investing in
- How many **Shares** the user has bought (or invested in)
- How much the user has paid for each share (the **Share price**)



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

If you are unfamiliar with how the stock market works, imagine you are shopping for groceries. To make a purchase you must specify what you are buying, how many items you are buying, and how much you are going to pay. These concepts translate to an investment as:

- A stock which is defined by a symbol, such as PETO, can be understood to be a grocery type
- The number of shares is the quantity of items you have purchased
- The share price is the unit price of each item

Once the user has added an investment, it must be listed along with his or her other investments.

The image shows a user interface for managing stock investments. At the top, there is a form with three input fields: 'Symbol:', 'Shares:', and 'Share price:'. The 'Shares:' field contains the number '0' and has a small icon to its right. The 'Share price:' field contains the number '0' and also has a small icon to its right. To the right of these fields is a brown 'Add' button. Below the form, there is a list of two investments. The first investment is 'AOUE', shown in a green box, with a profit of '101.80%' and a 'remove' button below it. The second investment is 'PETO', shown in a red box, with a loss of '-42.34%' and a 'remove' button below it.

Form and list of investments

The idea is to display how well his or her investments are going. Since the prices of the stocks fluctuate over time, the difference between the price the user has paid and the current price, indicate whether it is a good (profit) or a bad (loss) investment.

In the preceding figure, we can see that the user has two investments:

- One in the **AOUE** stock, which is scoring a profit of **101.80%**
- And another in the **PETO** stock, which is scoring a loss of **-42.34%**

This is a very simple application and we will get a deeper understanding of its functionality as we go on with its development.

Jasmine basics and thinking in BDD

Based on the application presented previously, we can start writing **acceptance criteria** that define **investment**:

- Given an investment, it should be of a stock
- Given an investment, it should have the invested shares quantity

- Given an investment, it should have the share paid price
- Given an investment, it should have a cost

And to start coding these as a Jasmine spec, the first thing we need to do, is to create a new spec file. This file can be created anywhere, but it is a good idea to stick to a convention, and Jasmine already has a good one: specs should be in the `/spec` folder. Create a `spec/InvestmentSpec.js` file and add the following lines:

```
describe("Investment", function() {  
  
});
```

`describe` is a global Jasmine function used to define test contexts. When used as the first call in a spec, it creates a new test suite. It accepts two parameters:

- The name of the test suite, in this case, `Investment`
- A functions which will contain all its specs

Then to translate the first acceptance criterion (given an investment, it should be of a stock) into a Jasmine spec, we are going to use another global Jasmine function called `it`:

```
describe("Investment", function() {  
  it("should be of a stock", function() {  
  
  });  
});
```

It also accepts two parameters:

- The title of the spec, in this case, `should be of a stock`
- A function, which will contain the spec code

To run this spec, add it to the runner:

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

And execute it by opening the runner on the browser:



First spec's passing result on the browser

This might sound strange to have an empty spec passing, but in Jasmine, as with other test frameworks, a failed assertion is required to make the spec fail.

An **assertion** is a comparison between two values that must result in a Boolean value. The assertion is only considered a success if the result of the comparison is true.

In Jasmine, assertions are written by using the global Jasmine function `expect`, along with a **matcher** that indicates what comparison must be made with the values.

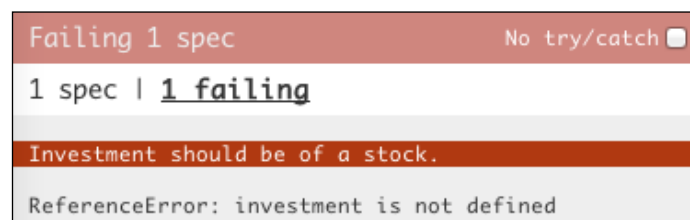
Regarding the current spec (it is expected that the investment is of a stock), in Jasmine this translates into:

```
describe("Investment", function() {  
  it("should be of a stock", function() {  
    expect(investment.stock).toBe(stock);  
  });  
});
```

The `expect` function takes only one parameter, which defines the **actual value**, or in other words, what is going to be tested: `investment.stock` and expects the chaining call to a matcher function: in this case `toBe`. That defines the **expected value**: `stock`, and the comparison method to be performed (to be the same).

Behind the scenes, Jasmine makes a comparison to check if the actual value (`investment.stock`) and expected value (`stock`) are the same, and if they are not, the test fails.

With the assertion written, the previously passing test is now failing:



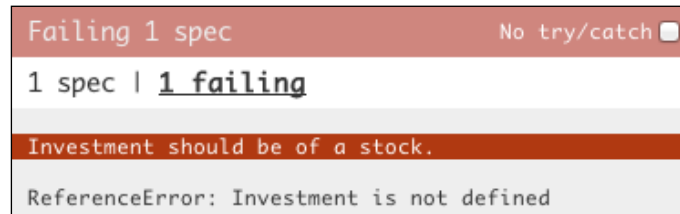
First spec's failing results

This spec is failing because, as the error message states: **investment is not defined**.

The idea here is to do only what the error is indicating us to do, so although you might feel the urge to write something else, for now let's just create this **investment** variable with an `Investment` instance.

```
describe("Investment", function() {
  it("should be of a stock", function() {
    var investment = new Investment();
    expect(investment.stock).toBe(stock);
  });
});
```

Don't worry that the `Investment()` function doesn't exist yet, the spec is about to ask for it on the next run:



Spec asks for an `Investment` class

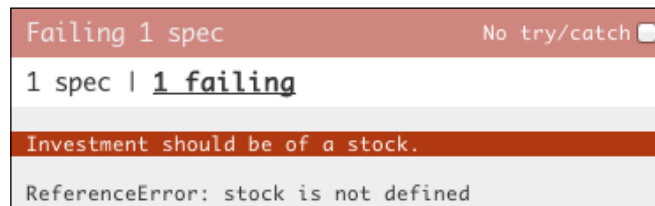
You can see that the error has changed to **Investment is not defined**. It now asks for the `Investment` function. So create a new file `Investment.js` in the `src` folder and add it to the runner:

```
<!-- include source files here... -->
<script type="text/javascript" src="src/Investment.js"></script>
```

To define the `Investment`, write a constructor function inside the `src/Investment.js` file:

```
function Investment () {};
```

This makes the error change. It now complains about the missing `stock` variable:

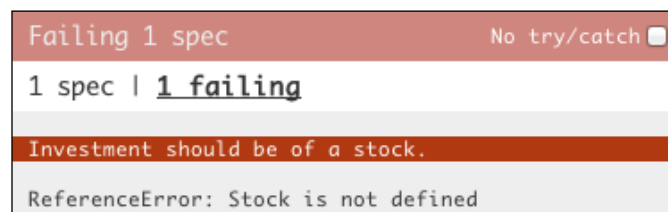


Missing `stock` error

One more time, we feed the code for which it is asking:

```
describe("Investment", function() {  
  it("should be of a stock", function() {  
    var stock = new Stock();  
    var investment = new Investment();  
    expect(investment.stock).toBe(stock);  
  });  
});
```

And the error changes again, this time is the missing Stock function:



Spec asks for a Stock class

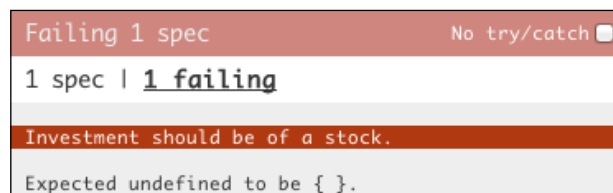
Create a new file called `src/Stock.js`, and add it to the runner. Since the `Stock` function is going to be a dependency of `Investment`, we should add it just before `Investment`:

```
<!-- include source files here... -->  
<script type="text/javascript" src="src/Stock.js"></script>  
<script type="text/javascript" src="src/Investment.js"></script>
```

And write the `Stock` constructor function:

```
function Stock () {};
```

Finally the error is on the expectation:



Expected undefined to be Stock

To fix this and complete this exercise, open the `src/Investment.js` file and add the reference to the `stock` parameter:

```
function Investment (stock) {  
  this.stock = stock;  
};
```

And on the spec, pass `stock` as a parameter to `Investment`:

```
describe("Investment", function() {  
  it("should be of a stock", function() {  
    var stock = new Stock();  
    var investment = new Investment(stock);  
    expect(investment.stock).toBe(stock);  
  });  
});
```

Finally you shall have a passing test:



Passing Investment spec

This exercise was meticulously conducted, to show how a developer works by feeding the spec with what it wants, when doing test-first development.



The drive to write code must come from a failing spec. You must not write code, unless its purpose is to fix a failing spec.

Setup and teardown

There are still three more acceptance criteria to be implemented. The next in the list is:

"Given an investment, it should have the invested shares quantity."

Writing it should be as simple as the previous spec was. In the `spec/InvestmentSpec.js` file, you can translate this new criterion into a new spec called `should have the invested shares quantity`:

```
describe("Investment", function() {
  it("should be of a stock", function() {
    var stock = new Stock();
    var investment = new Investment({
      stock: stock,
      shares: 100
    });
    expect(investment.stock).toBe(stock);
  });

  it("should have the invested shares quantity", function() {
    var stock = new Stock();
    var investment = new Investment({
      stock: stock,
      shares: 100
    });
    expect(investment.shares).toEqual(100);
  });
});
```

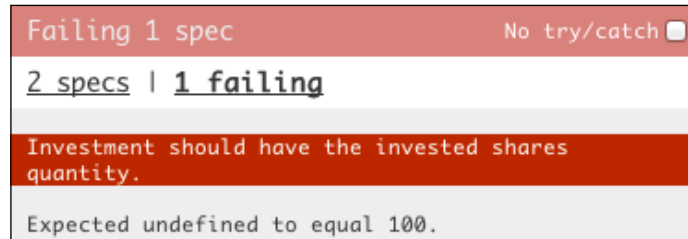
You can see that apart from having written the new spec, we have also refactored the call to the `Investment` constructor, to support the new `shares` parameter.

To do so, we used an object as a single parameter in the constructor, to simulate **named parameters**, a feature JavaScript doesn't have natively.

To implement this in the `Investment` function is pretty simple: instead of having multiple parameters on the function declaration, it instead has only one, which is expected to be an object. Then, the function probes each of its expected parameters from this object, making the proper assignments:

```
function Investment (params) {
  var params = params || {};
  this.stock = params.stock;
};
```

The code is now refactored. We can run the tests to see that only the new spec should be failing:



Failing shares spec

To fix it, change the `Investment` constructor to make the assignment to the `shares` property:

```
function Investment (params) {  
  var params = params || {};  
  this.stock = params.stock;  
  this.shares = params.shares;  
};
```

And finally everything is green:



Passing shares spec

But as you can see, the code that instantiates the `Stock` and the `Investment` is duplicated on both specs:

```
var stock = new Stock();  
var investment = new Investment({  
  stock: stock,  
  shares: 100  
});
```

To eliminate this duplication, Jasmine provides another global function called `beforeEach` that, as the name states, is executed once before each spec. So for these two specs, it will run twice—once before each spec.

Refactor the previous specs, by extracting the setup code with the use of `beforeEach`:

```
describe("Investment", function() {  
    var stock, investment;  
  
    beforeEach(function() {  
        stock = new Stock();  
        investment = new Investment({  
            stock: stock,  
            shares: 100  
        });  
    });  
  
    it("should be of a stock", function() {  
        expect(investment.stock).toBe(stock);  
    });  
  
    it("should have the invested shares quantity", function() {  
        expect(investment.shares).toEqual(100);  
    });  
});
```

Much cleaner, we not only removed the code duplication, but also simplified the specs. They became much easier to read and maintain, since their only responsibility now is to perform the expectation.

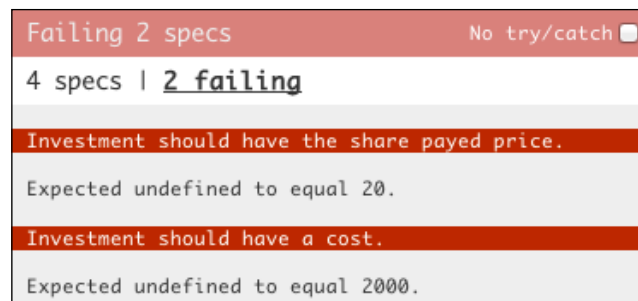
There is also a teardown function (`afterEach`) that sets code to be executed after each spec. It is very useful in situations where a cleanup is required after each spec. We will see an example of its application in *Chapter 6, Light Speed Unit Testing*.

To finish the specification of `Investment`, add the remaining two specs to the `spec/InvestmentSpec.js` file:

```
describe("Investment", function() {  
    var stock;  
    var investment;  
  
    beforeEach(function() {  
        stock = new Stock();  
        investment = new Investment({  
            stock: stock,  
            shares: 100,  
            sharePrice: 20  
        });  
    });  
});
```

```
it("should have the share payed price", function() {  
  expect(investment.sharePrice).toEqual(20);  
});  
  
it("should have a cost", function() {  
  expect(investment.cost).toEqual(2000);  
});  
});
```

Run the specs to see them failing:

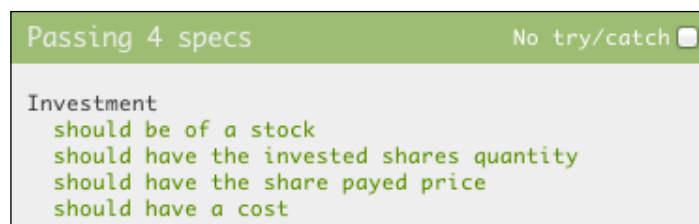


Failing cost and price specs

Add the code to fix them in the `src/Investment.js` file:

```
function Investment (params) {  
  var params = params || {};  
  this.stock = params.stock;  
  this.shares = params.shares;  
  this.sharePrice = params.sharePrice;  
  this.cost = this.shares * this.sharePrice;  
};
```

Run the specs for the last time to see them green:



Passing all four Investment specs



It is important to always see a spec failing before writing the code to fix it, otherwise how would you know that you really fix it? Imagine this as a way to test the test.

Nested describes

Nested describes are useful when you want to describe similar behavior between specs. Suppose we want the following two new acceptance criteria:

- Given an investment, when its stock share price valorizes, it should have a positive return of investment (ROI);
- Given an investment, when its stock share price valorizes, it should be a good investment.

They both share the same behavior **when its stock share prices valorizes**.

To translate this into Jasmine you can nest a call to the `describe` function inside the existing one in the `spec/InvestmentSpec.js` file: (I removed the rest of the code for the purpose of demonstration; it is still there.)

```
describe("Investment", function()
  describe("when its stock share price valorizes", function() {

  });
});
```

It should behave just like the outer one, so you can add specs (`it`) and use setup and teardown functions (`beforeEach`, `afterEach`).

Setup and teardown

When using setup and teardown functions, Jasmine respects the outer setup and teardown functions as well, so they are run as expected. For each spec (`it`):

- Jasmine runs all setup functions (`beforeEach`) from the outside in
- Runs a spec code (`it`)
- Runs all the teardown functions (`afterEach`) from the inside out

So we can add a setup function to this new `describe` that changes the share price of the stock, so it's greater than the share price of the investment:

```
describe("Investment", function() {
  var stock;
  var investment;

  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });
  });

  describe("when its stock share price valorizes", function() {
    beforeEach(function() {
      stock.sharePrice = 40;
    });
  });
});
```

Coding a spec with shared behavior

Now that we have the shared behavior implemented, we can start coding the acceptance criteria described earlier. Each is, just as before, a call to the global Jasmine function it:

```
describe("Investment", function() {
  describe("when its stock share price valorizes", function() {
    beforeEach(function() {
      stock.sharePrice = 40;
    });

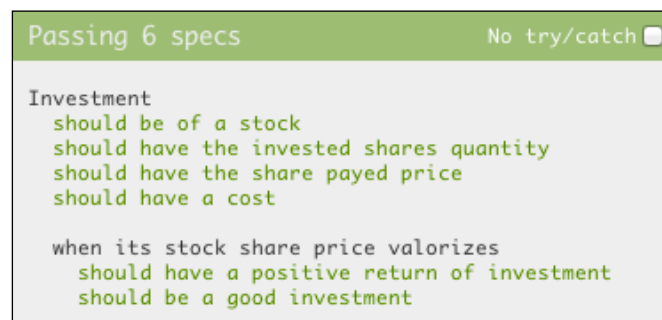
    it("should have a positive return of investment", function() {
      expect(investment.roi()).toEqual(1);
    });

    it("should be a good investment", function() {
      expect(investment.isGood()).toBeTruthy();
    });
  });
});
```

And after adding the missing functions to `Investment` in `src/Investment.js`:

```
Investment.prototype.roi = function() {  
  return (this.stock.sharePrice - this.sharePrice) / this.sharePrice;  
};  
  
Investment.prototype.isGood = function() {  
  return this.roi() > 0;  
};
```

You can run the specs, and see that they are passing:



Passing nested describe specs

Understanding matchers

By now, you've already seen plenty of usage examples for matchers, and probably can feel how they work.

You have seen how to use the `toBe`, the `toEqual`, and `toBeTruthy` matchers. These are a few of the built-in matchers available in Jasmine, but we can extend Jasmine by writing matchers of our own.

So, to really understand how Jasmine matchers work, we need to create one ourselves.

Custom matchers

Consider this expectation from the previous section:

```
expect(investment.isGood()).toBeTruthy();
```

Although it works, it is not very expressive. Imagine if we could rewrite it instead, as:

```
expect(investment).toBeAGoodInvestment();
```

This makes a much better relation to the acceptance criterion:

"should be a good investment"→expect investment to be a good investment

And to implement it is quite simple. You do so by calling the `this.addMatcher` Jasmine function inside a `setup` (`beforeEach`) or a `spec` (`it`).

Although you can put this new matcher definition inside the `spec/InvestmentSpec.js` file, Jasmine already has a default place to add custom matchers, the file `spec/SpecHelper.js`. If you are using the Standalone Distribution, it already comes with a sample custom matcher; delete it and let's start from scratch.

The `addMatcher` function accepts a single parameter—an object where each attribute corresponds to a new matcher. So to add this new matcher, change the contents of the `spec/SpecHelper.js` file to:

```
beforeEach(function() {  
  this.addMatchers({  
    toBeAGoodInvestment: function() {}  
  });  
});
```

A Jasmine matcher is simply a function that returns a Boolean value: `true`, to indicate that the expectation is passing and `false` if otherwise.

But to implement this matcher, we need access to the `investment` object, available via the `this.actual` property:

```
toBeAGoodInvestment: function() {  
  var investment = this.actual;  
  return investment.isGood();  
};
```

After getting access to the `investment` object, implementing the matcher was a simple return of the `isGood()` value.

By now, this matcher is ready to be used by the specs:

```
it("should be a good investment", function() {  
  expect(investment).toBeAGoodInvestment();  
});
```

After the change, the specs should still be passing. But what happens if a spec fails? What is the error message that Jasmine reports?

We can see it by deliberately breaking the `investment.isGood` implementation in `src/Investment.js`, to always return `false`:

```
Investment.prototype.isGood = function() {  
  return false;  
};
```

When running the specs again, this is the error message that Jasmine generates:

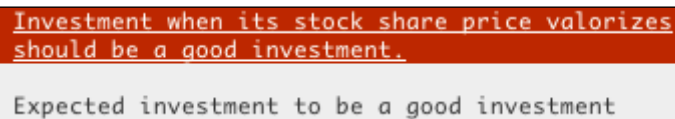


Custom matcher's message

It is not so bad, but sure can be made better. Jasmine allows the customization of this message via the `this.message` property inside the matcher declaration. Jasmine expects this property to be a function that returns the error message:

```
toBeAGoodInvestment: function() {  
  var investment = this.actual;  
  
  this.message = function() {  
    return 'Expected investment to be a good investment';  
  };  
  
  return investment.isGood();  
}
```

Run the specs again and the error message should change:



Custom matcher's custom message

Now, let's consider another acceptance criterion:

"Given an investment, when its stock share price devalorizes, it should be a bad investment".

Although it is possible to create a new custom matcher (`toBeABadInvestment`), Jasmine allows the negation of any matcher by chaining `not` before the matcher call. So we can write that a "bad investment" is "not a good investment":

```
expect(investment).not.toBeAGoodInvestment();
```

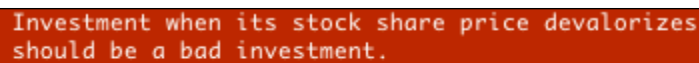
Implement this new acceptance criterion in `spec/InvestmentSpec.js` by adding a new nested describe and spec:

```
describe("when its stock share price devalorizes", function() {  
  beforeEach(function() {  
    stock.sharePrice = 0;  
  });  
  
  it("should be a bad investment", function() {  
    expect(investment).not.toBeAGoodInvestment();  
  });  
});
```


But there is a catch! Let's break the Investment code so that it is always a good investment:

```
Investment.prototype.isGood = function() {  
  return true;  
};
```

After running the specs again, you can see that this new spec is failing, but the error message is wrong: **Expected investment to be a good investment**.



Investment when its stock share price devalorizes
should be a bad investment.



Expected investment to be a good investment

Custom matcher's wrong custom negated message

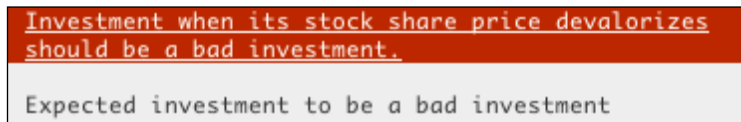
That is the message that was hard coded inside the matcher. To fix this, you need to make the message dynamic, based on how the matcher was called.

Luckily there is a property available inside the matcher declaration that tells if the matcher was called with a chained `not`, `this.isNot`.

Following is the fixed matcher with the dynamic message:

```
toBeAGoodInvestment: function() {  
  var investment = this.actual;  
  var what = this.isNot ? 'bad' : 'good';  
  
  this.message = function() {  
    return 'Expected investment to be a '+' what +' investment';  
  };  
  
  return investment.isGood();  
}
```

This fixes the message:



Custom matcher's custom dynamic message

And now this matcher can be used anywhere.

What lacked in this example was a way to show how to pass an expected value to a matcher like this:

```
expect(investment.cost).toEqual(2000)
```

It turns out that a matcher can receive any number of expected values as parameters. So for instance, the preceding matcher could be implemented as:

```
toEqual: function(expectedValue) {  
  return this.actual == expectedValue;  
};
```

Always before implementing any matcher, check first if there is one available that already does what you want.

Built-in matchers

Jasmine comes with a bunch of default matchers, covering the basis of value checking in the JavaScript language. To understand how they work and where to use them properly is a journey on how JavaScript handles type.

The toEqual built-in matcher

This is probably the most commonly used matcher, and you should use it whenever you want to check equality between two values.

It works for all primitive values (number, string, and Boolean) as well as any object (including arrays).

```
describe("toEqual", function() {
  it("should pass equal numbers", function() {
    expect(1).toEqual(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toEqual("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toEqual(true);
  });

  it("should pass equal objects", function() {
    expect({a: "testing"}).toEqual({a: "testing"});
  });

  it("should pass equal arrays", function() {
    expect([1, 2, 3]).toEqual([1, 2, 3]);
  });
});
```

The toBe built-in matcher

The `toBe` matcher has a very similar behavior to the `toEqual` matcher, in fact it gives the same result while comparing primitive values, but the similarities stop there.

While the `toEqual` matcher has a complex implementation (you should take a look at the Jasmine source code) that checks if all attributes of an object and all elements of an array are the same, here it is a simple use of the **strict equals operator** (`===`).

If you are unfamiliar with the strict equals operator, its main difference from the **equals operator** (`==`) is that the latter performs type coercion if the compared values aren't of the same type.



The strict equals operator always considers false any comparison between values of distinct types.

Here are some examples of how this matcher (and the strict equals operator) works:

```
describe("toBe", function() {
  it("should pass equal numbers", function() {
    expect(1).toBe(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toBe("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toBe(true);
  });

  it("should pass same objects", function() {
    var object = {a: "testing"};
    expect(object).toBe(object);
  });

  it("should pass same arrays", function() {
    var array = [1, 2, 3];
    expect(array).toBe(array);
  });

  it("should not pass equal objects", function() {
    expect({a: "testing"}).not.toBe({a: "testing"});
  });

  it("should not pass equal arrays", function() {
    expect([1, 2, 3]).not.toBe([1, 2, 3]);
  });
});
```

It is advised to use the `toEqual` operator in most cases, and resort to the `toBe` matcher only when you want to check if two variables reference the same object.

The toBeTruthy and toBeFalsy matchers

Besides its primitive Boolean type, everything else in the JavaScript language also has an inherent Boolean value, which is generally known as being either **truthy** or **falsy**.

Luckily in JavaScript, there are only a few values that are identified as falsy, as shown in the following examples for the `toBeFalsy` matcher.

```
describe("toBeFalsy", function () {
  it("should pass undefined", function() {
    expect(undefined).toBeFalsy();
  });

  it("should pass null", function() {
    expect(null).toBeFalsy();
  });

  it("should pass NaN", function() {
    expect(NaN).toBeFalsy();
  });

  it("should pass the false boolean value", function() {
    expect(false).toBeFalsy();
  });

  it("should pass the number 0", function() {
    expect(0).toBeFalsy();
  });

  it("should pass an empty string", function() {
    expect("").toBeFalsy();
  });
});
```

Everything else is considered truthy, as demonstrated by these examples of the `toBeTruthy` matcher:

```
describe("toBeTruthy", function() {
  it("should pass the true boolean value", function() {
    expect(true).toBeTruthy();
  });

  it("should pass any number different than 0", function() {
    expect(1).toBeTruthy();
  });
});
```

```
it("should pass any non empty string", function() {
  expect("a").toBeTruthy();
});

it("should pass any object (including an array)", function() {
  expect([]).toBeTruthy();
  expect({}).toBeTruthy();
});
});
```

But if you want to check if something is equal to an actual Boolean value, it might be a better idea to use the `toEqual` matcher.

The `toBeUndefined`, `toBeNull`, and `toBeNaN` built-in matchers

These matchers are pretty straightforward, and should be used to check for undefined, null, and NaN values.

```
describe("toBeNull", function() {
  it("should pass null", function() {
    expect(null).toBeNull();
  });
});

describe("toBeUndefined", function() {
  it("should pass undefined", function() {
    expect(undefined).toBeUndefined();
  });
});

describe("toBeNaN", function() {
  it("should pass NaN", function() {
    expect(NaN).toBeNaN();
  });
});
```

Both `toBeNull` and `toBeUndefined` can be written as `toBe(null)` and `toBe(undefined)` respectively, but that is not the case with `toBeNaN`.

In JavaScript, the `NaN` value is not equal to any value, not even `NaN`. So trying to compare it to itself is always `false`.

```
NaN === NaN // false
```

The Jasmine `toBeNaN` matcher is actually implemented taking into account that `NaN` is the only value that is not equal to itself. Here is how it is implemented:

```
jasmine.Matchers.prototype.toBeNaN = function() {  
  return (this.actual !== this.actual);  
};
```

See how it checks if the actual is different from itself? So as a good practice, try to use these matchers whenever possible instead of their `toBe` counterparts.

The `toBeDefined` built-in matcher

This matcher is useful if you want to check whether a variable is defined and you don't care about its value.

```
describe("toBeDefined", function() {  
  it("should pass any value other than undefined", function() {  
    expect(null).toBeDefined();  
  });  
});
```

Anything except `undefined` will pass under this matcher, even `null`.

The `toContain` built-in matcher

Sometimes it is desirable to check if an array contains an element or if a string can be found inside another string. For these use cases, you can use the `toContain` matcher.

```
describe("toContain", function() {  
  it("should pass if a string contains another string", function()  
  {  
    expect("My big string").toContain("big");  
  });  
  
  it("should pass if an array contains an element", function() {  
    expect([1, 2, 3]).toContain(2);  
  });  
});
```


The toMatch built-in matcher

Although the `toContain` and the `toEqual` matchers can be used in most string comparisons, sometimes the only way to assert if a string value is correct, is through a regular expression. For these cases, you can use the `toMatch` matcher along with a regular expression.

```
describe("toMatch", function() {
  it("should pass a matching string", function() {
    expect("My big matched string").toMatch(/My(.+)string/);
  });
});
```

The matcher works by testing the actual value ("My big matched string") against the expected regular expression (`/My(.+)string/`).

The toBeLessThan and toBeGreaterThan built-in matchers

The `toBeLessThan` and `toBeGreaterThan` are two simple matchers to perform numeric comparisons, as best described by the following examples:

```
describe("toBeLessThan", function() {
  it("should pass when the actual is less than expected",
    function() {
      expect(1).toBeLessThan(2);
    });
});

describe("toBeGreaterThan", function() {
  it("should pass when the actual is greater than expected",
    function() {
      expect(2).toBeGreaterThan(1);
    });
});
```

The toThrow built-in matcher

Exceptions are a language's way to demonstrate when something goes wrong.

So for example while coding an API, you might decide to throw an exception when a parameter is passed incorrectly. So how do you test this code?

Jasmine has the built-in `toThrow` matcher that can be used to verify that an exception has been thrown.

The way it works is a little bit different from the other matchers. Since the matcher has to run a piece of code and check if it throws an exception, the matcher's **actual** value must be a function.

Here is an example of how it works:

```
describe("toThrow", function() {
  it("should pass when the exception is thrown", function() {
    expect(function () {
      throw("Some exception");
    }).toThrow("Some exception");
  });
});
```

When the test is run, the anonymous function gets executed, and if it throws the `Some exception` exception, the test passes.

Summary

In this chapter, you have learned how to think in Behavior Driven Development (BDD) and drive your code from your specs. You have also become acquainted with the basic Jasmine global functions (`describe`, `it`, `beforeEach`, and `afterEach`) and got a good understanding of what is required to create a spec in Jasmine.

You have got familiar with Jasmine matchers, and know how they are powerful in describing a spec intent. You have even learned to create a matcher of your own.

By now you should be familiar with creating new specs and driving the development of your new application.

In the next chapter, we are going to take a look on how we can use the concepts learned in this chapter to start testing web applications: most commonly jQuery and HTML forms.

3

Testing Frontend Code

Testing JavaScript browser code has been notoriously considered hard, and although there are many complications while dealing with cross-browser testing, the most common problem is not with the testing process, but rather that the application code itself is not testable.

Since every element in the browser's document is accessible globally, it is easy to write a monolithic piece of JavaScript code that deals with the whole page. This leads to a number of problems, and the biggest one is that it is pretty hard to test.

In this chapter, you are going to get the best practices on how to write maintainable and testable browser code.

To implement the user interface, you are going to use jQuery, a well-known JavaScript library that abstracts the browser's DOM in a clean and simple API that works across different browsers.

To make the writing of the specs easier, we're going to use Jasmine jQuery, a Jasmine extension that adds new matchers to perform assertions on jQuery objects. To install it and its jQuery dependency, download the following files:

- <https://raw.githubusercontent.com/velesin/jasmine-jquery/master/lib/jasmine-jquery.js>
- <https://raw.githubusercontent.com/velesin/jasmine-jquery/master/vendor/jquery/jquery.js>

Place them inside the `lib/` folder, and add them to `SpecRunner.html`:

```
<script type="text/javascript" src="lib/jquery.js"></script>
<script type="text/javascript" src="lib/jasmine-
  jquery.js"></script>
```

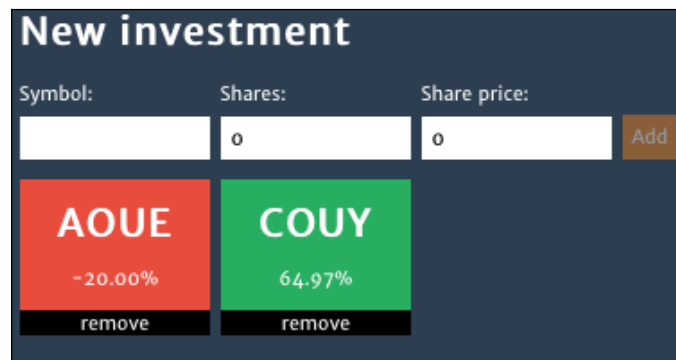
As seen so far, we have already created separate abstractions to handle both an investment and its associated stock. Now it is time to develop this application's user interface, and to achieve a good result, which is all a matter of organization and good practices.

The same principles of software engineering that you apply on your server-based code must not be left behind when writing your frontend JavaScript code. It is still important to think about components and proper separation of concerns.

Thinking in terms of components (Views)

We've talked about the monolithic JavaScript codebases that plague most of the Web, codebases that are impossible to test. And the best way not to fall into this trap, is by coding your application driven by tests.

Consider the mockup interface of our Investment Tracker application:



Investment Tracker mockup interface

How would you go on implementing it? It is easy to see that this application has two different responsibilities:

- One to add an investment
- And another to list the added investments

So we could start by breaking this interface into two different components. And to better describe them, we are going to borrow a concept from **MVC frameworks**, such as `Backbone.js`, and call them **Views**.

So here it is, at the top level of the interface, with two base components:

- `NewInvestmentView`: This will be responsible for creating new investments
- `InvestmentListView`: This is going to be a list of all added investments

The module pattern

So we understand how we must breakup the code, but how do we organize it? So far we have created a file for each new function. This is a good practice, and we are going to see how we can improve on that.

Let's start by thinking about our `NewInvestmentView` component. We can follow the pattern we've used so far and create a new file for it at `src/NewInvestmentView.js`:

```
(function ($, Investment, Stock) {  
  function NewInvestmentView (params) {  
  
  };  
  
  this.NewInvestmentView = NewInvestmentView;  
})(jQuery, Investment, Stock);
```

You can see that this JavaScript file is more robust than the examples shown so far. We have wrapped all the `NewInvestmentView` code inside an **immediately invoked function expression (IIFE)**.

It is called an IIFE because it declares a function and immediately invokes it, effectively creating a new scope to create local variables in.

A good practice is to use only local variables inside the IIFE. If it needs to use a global dependency, pass it through a parameter. In this example it is already passing three dependencies to the `NewInvestmentView` code: `jQuery`, `Investment`, and `Stock`.

As you can see by the function declaration:

```
function ($, Investment, Stock)
```

And immediate invocation:

```
})(jQuery, Investment, Stock);
```

The biggest advantage of this practice is that we no longer need to worry about polluting the global namespace, since everything we declare inside the IIFE will be local. This makes it much harder to mess with the global scope.

And if we need to make anything global, we do that explicitly by attaching it with the global object:

```
this.NewInvestmentView = NewInvestmentView;
```

Another advantage is the explicit dependency declaration. We know all about a file's external dependencies by glancing at its first line.

And although this practice has not much advantage right now, (since all of the components are being exposed globally), we are going to see how to benefit from it in *Chapter 8, Build Automation*.

This pattern is also known as the **module pattern**, and we will be using it throughout the rest of book (even though sometimes it is omitted for simplification purposes).

Using HTML fixtures

Continuing with the development of the `NewInvestmentView` component, we can write some basic acceptance criteria:

- `NewInvestmentView` should allow the input of the stock symbol
- `NewInvestmentView` should allow the input of shares
- `NewInvestmentView` should allow the input of the share price

There are many more, but these are a good start.

Create a new spec file for this component, in the new file `spec/NewInvestmentViewSpec.js` and we can start to translate those specs:

```
describe("NewInvestmentView", function() {
  it("should allow the input of the stock symbol", function() {
  });

  it("should allow the input of shares", function() {
  });

  it("should allow the input of the share price", function() {
  });
});
```

But before we can start to implement these, we must first understand the concept of **HTML fixtures**.

Test fixtures provide the base state in which the tests run. It could be a class instantiation, the definition of an object, or a piece of HTML. In other words, to test JavaScript code that handles a form submission, you need to have the form available when running the tests. The HTML code containing the form is an HTML fixture.

One way to handle this requirement is by manually appending the required DOM element inside a setup function:

```
beforeEach(function() {
  $('body').append('<form id="my-form"></form>');
});
```

And removing it during teardown:

```
afterEach(function() {
  $('#my-form').remove();
});
```

Otherwise the spec would be appending a lot of garbage inside the document, and it could interfere with the results of other specs.



It is important to know that specs should be independent and that they can be run in any particular order. So as a rule, treat your specs completely in isolation from each other.

A better approach is to have a container in the document where we always put the HTML fixtures:

```
<div id="html-fixtures">
</div>
```

And change the code to:

```
beforeEach(function() {
  $('#html-fixtures').html('<form id="my-form"></form>');
});
```

That way, the next time a spec runs, it automatically overwrites the previous fixture with its own.

But this can soon escalate into an incomprehensible mess, as the fixtures get more complex:

```
beforeEach(function() {
  $('#html-fixtures').html('<form id="new-investment"><h1>New
investment</h1><label>Symbol:<input type="text" class="new-
investment-stock-symbol" name="stockSymbol"
value=""></label><input type="submit" name="add"
value="Add"></form>');
});
```


Wouldn't it be great if this fixture could be loaded from an external file? That is exactly what the Jasmine jQuery extension does, with its **HTML Fixture** module.

We can place that HTML code in an external file and load it in the document with a simple call to `loadFixtures`, passing the fixture file path:

```
beforeEach(function() {  
    loadFixtures('MyFixture.html');  
});
```

The extension looks for a file `spec/javascripts/fixtures/MyFixture.html`, and loads its content inside a container:

```
<div id="jasmine-fixtures">  
  <form id="new-investment">  
    <h1>New investment</h1>  
    <label>  
      Symbol:  
      <input type="text" class="new-investment-stock-symbol"  
        name="stockSymbol" value="">  
    </label>  
    <input type="submit" name="add" value="Add">  
  </form>  
</div>
```

We can also use another of the extension's global functions, to recreate the first example. The `setFixtures(html)` accepts a parameter with the content to be placed in the container:

```
beforeEach(function() {  
    setFixtures('<form id="my-form"></form>');  
});
```

Other available functions are:

- `appendLoadFixtures(fixtureUrl[, fixtureUrl, ...])`: Instead of overwriting the content of the fixture container, this appends it
- `readFixtures(fixtureUrl[, fixtureUrl, ...])`: This reads a fixture content, but instead of appending it to the document, it returns a string with its contents
- `appendSetFixtures(html)`: This is the same as the `appendLoadFixtures`, but with an HTML string instead of a file

The Jasmine jQuery fixture module caches each file, so we can load the same fixture multiple times without penalty in the test suite speed.

It loads the fixtures using AJAX, and sometimes a test might want to modify the inner workings of JavaScript or jQuery AJAX as we will see in *Chapter 6, Light Speed Unit Testing*, which would break the loading of a fixture. A workaround for this issue, is to preload the required fixtures on the cache using the `preloadFixtures()` function.

The `preloadFixtures(fixtureUrl[, fixtureUrl, ...])` loads one or more files in the cache without appending them to the document.

There is an issue though, while using HTML fixtures with Chrome. Jasmine jQuery loads the HTML fixtures using AJAX, but because of the **Same Origin Policy**, Chrome blocks all AJAX requests when opening the `SpecRunner.html` with a `file://` protocol.

A solution to this problem is to serve the spec runner through an HTTP server, as described in *Chapter 4, Asynchronous Testing – AJAX*.

For now, the easiest solution might be to use another browser, such as Firefox.

More details on this Chrome issue can be seen at this GitHub ticket:

<https://github.com/velesin/jasmine-jquery/issues/4>

Back to the `NewInvestmentView` component, we can start the development of the spec with the help of this HTML fixture plugin.

Based on the mockup interface we can create a new HTML fixture called `spec/fixtures/NewInvestmentView.html`

```
<form id="new-investment">
  <h1>New investment</h1>
  <label>
    Symbol:
    <input type="text" class="new-investment-stock-symbol"
      name="stockSymbol" value="">
  </label>
  <label>
    Shares:
    <input type="number" class="new-investment-shares"
      name="shares" value="0">
  </label>
  <label>
    Share price:
    <input type="number" class="new-investment-share-price"
      name="sharePrice" value="0">
  </label>
  <input type="submit" name="add" value="Add">
</form>
```

And because we are saving this fixture not at the plugin's default path, we need to add a new configuration at the end of the `SpecHelper.js` file:

```
jasmine.getFixtures().fixturesPath = 'spec/fixtures';
```

In the spec, add the call to load the fixture:

```
describe("NewInvestmentView", function() {  
  beforeEach(function() {  
    loadFixtures('NewInvestmentView.html');  
  });  
});
```

And finally, add both the spec and the source to the runner:

```
<script type="text/javascript"  
  src="src/NewInvestmentView.js"></script>  
<script type="text/javascript"  
  src="spec/NewInvestmentViewSpec.js"></script>
```

Basic View coding rules

Now it is time to start coding the first View component. To help you through the process we are going to lay two basic rules for View coding happiness:

- The View should encapsulate a DOM element
- Integrate Views with observers

So let's see how they work individually.

The View should encapsulate a DOM element

As mentioned earlier, a View is the behavior associated with a DOM element, so it makes sense to have this element related to the View. A good pattern is to pass a CSS selector in the View instantiation that indicates the element to which it should refer. Here is the spec for the `NewInvestmentView` component:

```
describe("NewInvestmentView", function() {  
  var view;  
  beforeEach(function() {  
    loadFixtures('NewInvestmentView.html');  
    view = new NewInvestmentView({selector: '#new-investment'});  
  });  
});
```

In the constructor function, it uses jQuery to get the element for this selector, and store it in an instance variable `$el` (source):

```
function NewInvestmentView (params) {  
  this.$el = $(params.selector);  
}
```

To make sure this code works, we should write a test for it:

```
it("should expose a property with its DOM element", function() {  
  expect(view.$el).toExist();  
});
```

The `toExist` is a custom matcher provided by the Jasmine jQuery extension to check if an element exists in the document. It not only validates the existence of the property on the JavaScript object, but also that the association with the DOM element worked.

Passing the selector to the View allows it to be instantiated multiple times to different elements on the document.

Another advantage of having the explicit association is knowing that this View is not changing anything else on the document, as we will see next.

A View is the behavior associated with a DOM element, so it shouldn't be messing around everywhere on the page. It should only change or access the element associated with it.

To demonstrate this concept, let's implement another acceptance criterion regarding the default state of the View.

```
it("should have an empty stock symbol", function() {  
  expect(view.getSymbolInput()).toHaveValue('');  
});
```

A naive implementation of the `getSymbol` method, might use a global jQuery lookup to find the input and return its value:

```
NewInvestmentView.prototype = {  
  getSymbolInput: function () {  
    return $('.new-investment-stock-symbol')  
  }  
};
```

But that could lead to a problem; if there is another input with that class name somewhere else in the document, it might get the wrong result.

A better approach is to use the View's associated element to perform a scoped lookup:

```
getSymbolInput: function () {  
    return $this.$el.find('.new-investment-stock-symbol')  
}
```

The `find` function will only look for elements that are children of `this.$el`. It is as if `this.$el` represents the entire document for the View.

Since we will be using this pattern everywhere inside the View code, we can create a function and use it instead:

```
NewInvestmentView.prototype = {  
    $: function () {  
        return this.$el.find.apply(this.$el, arguments);  
    },  
    getSymbolInput: function () {  
        return this.$('.new-investment-stock-symbol')  
    }  
};
```

Now suppose that from somewhere else in the application, we want to change the value of a `NewInvestmentView` form input. We know its class name, so it could be as simple as:

```
$('.new-investment-stock-symbol').val('from outside the view');
```

But that simplicity hides a serious problem of encapsulation. This one line of code is creating a coupling with what should be an implementation detail of the `NewInvestmentView`.

If tomorrow, another developer changes the `NewInvestmentView`, renaming the input class name from `.new-investment-stock-symbol` to `.new-investment-symbol`, that one line would be broken.

To fix it, the developer would need to look at the entire codebase for references to that class name.

A much safer approach is to respect the View and use its APIs:

```
newInvestmentView.setSymbol('from outside the view');
```

Which when implemented would look like:

```
NewInvestmentView.prototype.setSymbol = function(value) {  
    this.$('.new-investment-stock-symbol').val(value);  
};
```

That way, when the code gets refactored, there is only one point to perform the change—inside the `NewInvestmentView` implementation.

And since there is no sandboxing in the browser's document, which means that from anywhere in the JavaScript code you can make a change anywhere in the document, there is not much you can do, besides good practice, to prevent these mistakes.

Integrate Views with observers

Following the development of the Investment Tracker application, we would eventually need to implement the list of investments. But how would you go on integrating the `NewInvestmentView` and the `InvestmentListView`?

You could write an acceptance criterion for the `NewInvestmentView` as: Given the new investment View, when its add button is clicked, then it should add an investment to the list of investments.

This is a very straightforward thinking, and you can see by the writing that we are creating a direct relationship between the two Views. Translating this into a spec clarifies this perception:

```
describe("NewInvestmentView", function() {
  beforeEach(function() {
    loadFixtures('NewInvestmentView.html');
    appendLoadFixtures('InvestmentListView.html');

    listView = new InvestmentListView({
      id: 'investment-list'
    });

    view = new NewInvestmentView({
      id: 'new-investment',
      listView: listView
    });
  });

  describe("when its add button is clicked", function() {
    beforeEach(function() {
      // fill form inputs
      // simulate the clicking of the button
    });

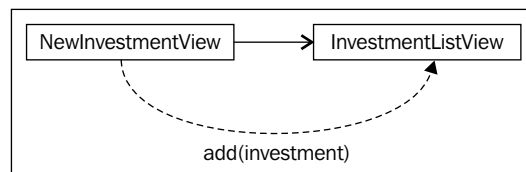
    it("should add the investment to the list", function() {
      expect(listView.count()).toEqual(1);
    });
  });
});
```

This solution creates a dependency between the two Views. The `NewInvestmentView` constructor now receives an instance of the `InvestmentListView` as its `listView` parameter.

And on its implementation, the `NewInvestmentView` calls the `addInvestment` method of the `listView` object when its form is submitted:

```
function NewInvestmentView (params) {  
  this.listView = params.listView;  
  
  this.$el.on('submit', function () {  
    this.listView.addInvestment(/* new investment */);  
  }).bind(this);  
}
```

To better clarify how this code works, here is a diagram of how the integration is done:



Direct relationship between the two Views

Although very simple, this solution introduces a number of architectural problems. The first, and most obvious, is the increased complexity of the `NewInvestmentView` specs.

And secondly, it makes evolving these components even more difficult, due to the tight coupling.

To better clarify this last problem, imagine that in the future you want to list investments also in a table. This would impose a change in the `NewInvestmentView` to support both the list and table Views:

```
function NewInvestmentView (params) {  
  this.listView = params.listView;  
  this.tableView = params.tableView;  
  
  this.$el.on('submit', function () {  
    this.listView.addInvestment(/* new investment */);  
    this.tableView.addInvestment(/* new investment */);  
  }).bind(this);  
}
```

Rethinking on the acceptance criterion, we can get into a much better and future proof solution. Let's rewrite it as: Given the Investment Tracker, when a new investment is created, then it should add the investment to the list of investments.

You can see by the acceptance criterion that it has introduced a new subject to be tested: the Investment Tracker. This implies a new source and spec file. After creating both the files accordingly and adding them to the runner, we can write this acceptance criterion as a spec:

```
describe("InvestmentTracker", function() {
  beforeEach(function() {
    loadFixtures('NewInvestmentView.html');
    appendLoadFixtures('InvestmentListView.html');

    listView = new InvestmentListView({
      id: 'investment-list'
    });

    newView = new NewInvestmentView({
      id: 'new-investment'
    });

    application = new InvestmentTracker({
      listView: listView,
      newView: newView
    });
  });

  describe("when a new investment is created", function() {
    beforeEach(function() {
      // fill form inputs
      newView.create();
    });

    it("should add the investment to the list", function() {
      expect(listView.count()).toEqual(1);
    });
  });
});
```

You can see the same setup code that once was inside the `NewInvestmentView` spec. It loads the fixtures required by both Views, instantiates both an `InvestmentListView` and `NewInvestmentView`, and creates a new instance of `InvestmentTracker` passing both Views as parameters.

Later on, while describing the behavior when a new investment is created, you can see the function call to the `newView.create` function to create a new investment.

And later it checks that a new item was added to the `listView` object by checking that the `listView.count()` is equal to 1.

But how does the integration happen? We can see it by looking at the `InvestmentTracker` implementation:

```
function InvestmentTracker (params) {
  this.listView = params.listView;
  this.newView = params.newView;

  this.newView.onCreate(function (investment) {
    this.listView.addInvestment(investment);
  }).bind(this);
}
```

It uses the `onCreate` function to register an observer function as a callback at the `newView`. This observer function will be invoked later when a new investment is created.

The implementation inside the `NewInvestmentView` is quite simple. The `onCreate` method stores the callback parameter as an attribute of the object:

```
NewInvestmentView.prototype.onCreate = function(callback) {
  this._callback = callback;
};
```

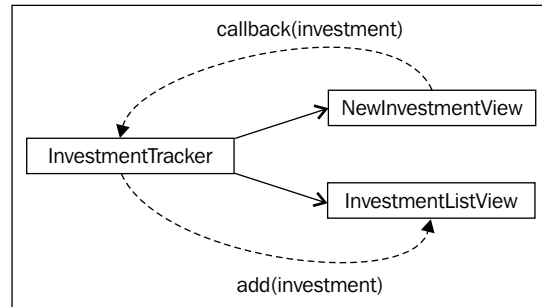
The naming convention of the `_callback` attribute might sound strange, but it is a good convention to indicate it as a private member.

Although the prepended underline character won't actually change the visibility of the attribute, it at least informs a user of this object that the `_callback` attribute might change or be even removed in the future.

And later when the `create` method is invoked, it invokes `_callback` passing the new investment as a parameter.

```
NewInvestmentView.prototype.create = function() {
  this._callback(/* new investment */);
};
```

Here is the solution illustrated for better understanding:



Using callbacks to integrate the two Views

Later in *Chapter 7, Testing Backbone.js Applications*, we will see how the implementation of this `NewInvestmentView` spec turned out to be.

Testing Views with jQuery matchers

Besides its HTML fixture module, the Jasmine jQuery extension comes with a set of custom matchers, which help in writing expectations with elements in the DOM.

The biggest advantage of using these custom matchers, as demonstrated, is the better error messages they generate. So although you can write all your specs without using any of these matchers, doing so would get you much more useful information when an error happens.

To better understand this advantage, we can revisit the example of the spec `should expose a property with its DOM element`. There, it uses the `toExist` matcher:

```
it("should expose a property with its DOM element", function() {  
  expect(view.$el).toExist();  
});
```

If this spec fails, we get a nice error message:

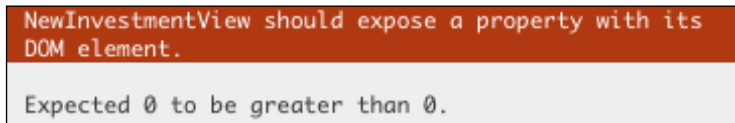
```
NewInvestmentView should expose a property with its  
DOM element.  
  
Expected { selector : '#new-investment', context :  
HTMLNode, length : 0 } to exist.
```

Nice custom matcher error message

But if we rewrite this spec without the custom matcher (still making the same validation):

```
it("should expose a property with its DOM element", function() {  
  expect($(document).find(view.$el).length).toBeGreaterThan(0);  
});
```

The error message gets less informative:



By reading the error you can't understand what it is truly testing

So use these matchers whenever you can to get better error messages. Let's go over some of the available custom matchers, demonstrated by example, with these acceptance criteria of the `NewInvestmentView` class:

- `NewInvestmentView` should allow the input of the stock symbol
- `NewInvestmentView` should allow the input of shares
- `NewInvestmentView` should allow the input of the share price
- `NewInvestmentView` should have an empty stock symbol
- `NewInvestmentView` should have its shares value at zero
- `NewInvestmentView` should have its share price value at zero
- `NewInvestmentView` should have its stock symbol input on focus
- `NewInvestmentView` should not allow to add

It is important that you understand that these next examples, although useful to demonstrate how the Jasmine jQuery matchers work, are not really testing any JavaScript code, but only the HTML elements that were loaded by the HTML Fixture.

The toBe jQuery matcher

This checks if the element matches the passed CSS selector.

```
it("should allow the input of the stock symbol", function() {  
  expect(view.$el.find('.new-investment-stock-symbol')).toBe('input[type=text]');  
});
```

The toContainHtml jQuery matcher

This checks if the content of the element matches the passed HTML.

```
it("should allow the input of shares", function() {  
  expect(view.$el).toContainHtml('<input type="number" class="new-  
    investment-shares" name="shares" value="0">');  
});
```

The toContain jQuery matcher

This checks if the element contains any child element matching the passed CSS selector.

```
it("should allow the input of the share price", function() {  
  expect(view.$el).toContain('input[type=number].new-investment-  
    share-price');  
});
```

The toHaveValue jQuery matcher

Only valid for inputs, this validates the expected value against the element's value attribute.

```
it("should have an empty stock symbol", function() {  
  expect(view.$el.find('.new-investment-stock-  
    symbol')).toHaveValue('');  
});  
  
it("should have its shares value to zero", function() {  
  expect(view.$el.find('.new-investment-  
    shares')).toHaveValue('0');  
});
```

The toHaveAttr jQuery matcher

This tests if the element has any attribute with the name and value specified. The following example shows how to use this matcher to test an input for its value attribute, an expectation that could have been written with the toHaveValue matcher.

```
it("should have its share price value to zero", function() {  
  expect(view.$el.find('.new-investment-share-  
    price')).toHaveAttr('value', '0');  
});
```

The toBeFocused jQuery matcher

This checks if the input element is focused.

```
it("should have its stock symbol input on focus", function() {  
  expect(view.$el.find('.new-investment-stock-symbol')).toBeFocused();  
});
```

The toBeDisabled jQuery matcher

This checks if the element is disabled.

```
it("should not allow to add", function() {  
  expect(view.$el.find('input[type=submit]')).toBeDisabled();  
});
```

More matchers

The extension has many more available matchers; be sure to check the documentation of the project at <https://github.com/velesin/jasmine-jquery#jquery-matchers>.

Summary

In this chapter, you learned how testing can become so much easier once you drive the application development by tests. You have seen how to use the Module pattern to better organize the project code, and how the View pattern can help create a more maintainable browser code.

You learned how to use HTML fixtures, making your specs much more readable and understandable. I have also showed you how to test code that interacts with the browser's DOM by the use of custom jQuery matchers.

To learn a bit more about frontend testing, be sure to check the bonus chapter on *Testing jQuery plugins*.

In the next chapter, we will go a step further, and start testing server integration and asynchronous code.

4

Asynchronous Testing – AJAX

Inevitably there comes a time in every JavaScript application when asynchronous code needs to be tested.

Asynchronous means that you cannot deal with it in a linear fashion – a function might return immediately after its execution, but the result will come later, usually through a callback.

This is a very common pattern while dealing with AJAX requests, for example through jQuery:

```
$.ajax('http://0.0.0.0/data.json', {  
  success: function (data) {  
    // handle the result  
  }  
});
```

To demonstrate Jasmine support on asynchronous testing, we are going to implement the following acceptance criterion:

"Stock when fetched, should update its share price."

By using the techniques we showed you so far, you could write this acceptance criterion in the `spec/StockSpec.js` file as follows:

```
describe("when fetched", function() {  
  beforeEach(function() {  
    stock.fetch();  
  });  
  
  it("should update its share price", function() {
```

```
        expect(stock.sharePrice).toEqual(23.67);
    });
});
```

That would lead to the implementation of the `fetch` function at the `src/Stock.js` file:

```
Stock.prototype.fetch = function() {
    var that = this;
    var url = 'http://0.0.0.0:8000/stocks/'+that.symbol;

    $.getJSON(url, function (data) {
        that.sharePrice = data.sharePrice;
    });
};
```

The important part is the `$.getJSON` call, an AJAX request expecting a JSON response containing the updated share price:

```
{
  "sharePrice": "23.67"
}
```

By now, you can see that we are stuck; in order to run this spec, we will need a server running.

Setting up the Scenario

Since this book is all about JavaScript, we are going to create a very simple Node.js server to be used by the specs. Node.js is a platform that allows the development of network applications, such as web servers, using JavaScript.

In *Chapter 6, Light Speed Unit Testing*, we are going to see alternative solutions to test AJAX requests without the need of a server.

Installing Node.js

If you already have Node.js installed, you can skip to the next section.

There are installers available for Windows and Mac OS X:

- Go to the Node.js website <http://nodejs.org/>
- Click on the **Install** button
- Once the download is completed, execute the installer and follow the steps

To check other installation methods and instructions on how to install on Linux distributions, check the official documentation at <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>.

Once you are done, you should have a `node` command available on your command line.

Coding the server

For the purpose of learning how to write asynchronous specs, we are going to create a server that returns some fake data. Create a new file in the project's root folder called `server.js`, and add its content:

```
var express = require('express');
var app = express();

app.get('/stocks/:symbol', function (req, res) {
  res.setHeader('Content-Type', 'application/json');
  res.send({ sharePrice: 20.18 });
});

app.use(express.static(__dirname));

app.listen(8000);
```

To handle the HTTP requests, we are using Express, a Node.js web application framework. By reading the code, you can see that it defines a route to `/stocks/:symbol`, so it accepts requests such as `http://0.0.0.0:8000/stocks/AOUE`, and respond with JSON data.

We also use the `express.static` module to serve the spec runner at `http://0.0.0.0:8000/SpecRunner.html`.

There is a requirement to circumvent the **Same Origin Policy**. This is a policy that dictates, for security reasons, that AJAX requests aren't allowed to be performed on domains different from the application.

This issue was first demonstrated while using HTML fixtures with the Chrome browser in *Chapter 3, Testing Frontend Code*, as there are some browsers with more rigid requirements than others while dealing with `file://` URLs.

Using the Chrome browser inspector you can see errors in the console while opening the `SpecRunner.html` file with a `file://` protocol (basically the way you've been doing until now):

```
XMLHttpRequest cannot load
http://0.0.0.0:8000/stocks/YH00. Origin null is
not allowed by Access-Control-Allow-Origin.
```

Same origin policy error

By serving the runner, and all of the application and test code under the same base URL, we prevent this problem from happening, and are able to run the specs on any browser.

Running the server

To run the server, first you need to install its dependencies (Express) using Node's package manager. Inside the application root folder, run the `npm` command:

```
$ npm install express
```

This command will download Express and place it inside a new folder called `node_modules` inside the project folder.

Now you should be able to run the server, by invoking the `node` command:

```
$ node server.js
```

To check if it is working, hit `http://0.0.0.0:8000/stocks/AQVE` on your browser, and you should receive the JSON response:

```
{
  "sharePrice": "23.66"
}
```

Now that we have our server dependency working, we can get back to writing the spec.

Writing the spec

With the server running, open your browser at `http://0.0.0.0:8000/SpecRunner.html`, to see the results of our specs.

You can see that even though the server is running, and the spec appears to be correct, it is failing. It's due to the fact that `stock.fetch()` is asynchronous. A call to `stock.fetch()` returns immediately, allowing Jasmine to run the expectations before the AJAX request is completed:

```
it("should update its share price", function() {  
    expect(stock.sharePrice).toEqual(23.67);  
});
```

To fix this, we need to embrace the asynchronicity of the `stock.fetch()` function and instruct Jasmine to wait for its execution before running the expectations.

The `waitsFor()` function

To tell Jasmine to wait for an asynchronous call, we need to use another of its global functions, `waitsFor()`.

Before we can dig into how it works, let's jump ahead and adapt the previous test code to use this new function:

```
describe("when fetched", function() {  
    var fetched = false;  
  
    beforeEach(function() {  
        stock.fetch({  
            success: function () {  
                fetched = true;  
            }  
        });  
  
        waitsFor(function (argument) {  
            return fetched;  
        }, 'Timeout fetching stock data', 2000);  
    });  
  
    it("should update its share price", function() {  
        expect(stock.sharePrice).toEqual(23.67);  
    });  
});
```

The first thing you will notice is that we have added a `success` callback to the `stock.fetch()` function, to set the `fetch` variable to `true` after the fetch is complete:

```
stock.fetch({
  success: function () {
    fetched = true;
  }
});
```

Its implementation is as follows:

```
Stock.prototype.fetch = function(parameters) {
  var that = this;
  var params = parameters || {};
  var success = params.success || function () {};
  var url = 'http://0.0.0.0:8000/stocks/'+that.symbol;

  $.getJSON(url, function (data) {
    that.sharePrice = data.sharePrice;
    success(that);
  });
};
```

Then we use the `waitsFor()` function to hold the execution of the `it` block, until the `fetch` variable is `true`:

```
waitsFor(function (argument) {
  return fetched;
}, 'Timeout fetching stock data', 2000);
```

And if the stock isn't fetched in 2000 milliseconds, it throws an error, making the spec fail.

Let's recap, the `waitsFor()` function accepts three parameters:

- A function that Jasmine will poll, until it gets a truth result:
function (argument) { return fetched; }
- An error message to show if the waiting times out:
"Timeout fetching stock data"
- The amount of time (in milliseconds) it waits before timing out: 2000.

So whenever you have any expectations that depend on the result of an asynchronous call, you can hold its execution by using the `waitsFor()` function inside a `beforeEach` block.

Next, we will see how to use the `waitsFor()` function directly inside the `it` block.

The `runs()` function

We have seen that we can use the `waitsFor()` function inside `beforeEach`, but what if we need to write a test code that has an asynchronous call inside an `it` block?

As an exercise, let's rewrite the previous spec without nesting it in a `describe` block, but rather as single `it` block:

```
it("should be able to update its share price", function() {
    var fetched = false;

    stock.fetch({
        success: function() {
            fetched = true;
        }
    });

    waitsFor(function (argument) {
        return fetched;
    }, 'Timeout fetching stock data', 2000);

    expect(stock.sharePrice).toEqual(23.67);
});
```

By running this example, you will see that the problem of synchronism has come back. That is because the `waitsFor()` function is not blocking the execution.

It worked previously, because Jasmine waits to run the `it` block until `waitsFor()` has been completed.

So we need a way to schedule this expectation code to be run after the `waitsFor()` completes. As you might have guessed, it is going to be through another Jasmine global function, the `runs` function.

All you have to do is move the code that you want to respect the asynchronous behavior inside a `runs` block:

```
it("should be able to update its share price", function() {
  var fetched = false;

  stock.fetch({
    success: function() {
      fetched = true;
    }
  });

  waitsFor(function (argument) {
    return fetched;
  }, 'Timeout fetching stock data', 2000);

  runs(function() {
    expect(stock.sharePrice).toEqual(23.67);
  });
});
```

That way, Jasmine runs that code only after `waitsFor()` completes.

And you can even put multiple `runs` blocks, and they will run in the order they were declared:

```
it("should be able to update its share price", function() {
  var fetched = false;

  runs(function() {
    stock.fetch({
      success: function() {
        fetched = true;
      }
    });
  });

  waitsFor(function (argument) {
    return fetched;
  }, 'Timeout fetching stock data', 2000);

  runs(function() {
    expect(stock.sharePrice).toEqual(23.67);
  });
});
```

```
runs(function() {  
    expect(stock.sharePrice).not.toBeUndefined();  
});  
  
runs(function() {  
    expect(stock.sharePrice).toBeGreaterThan(0);  
});  
});
```

Summary

In this chapter, you have seen how to test asynchronous code, a scenario common when testing server interactions (AJAX). I have showed you how to use the different Jasmine global functions such as `waitFor()` and `runs` to test asynchronous code.

I have also presented you the Node.js platform, and used it to code a simple server to be used as a test fixture.

In *Chapter 6, Light Speed Unit Testing*, we are going to see different solutions to AJAX testing, solutions that don't require a server running.

5

Jasmine Spies

Test double is a pattern on unit testing. It replaces a test dependent component with an equivalent implementation that is specific to the test scenario.

These implementations are called doubles because although their behavior might be specific to the test, they act like and have the same API as the object they impersonate.

Spies are Jasmine's solution to test doubles.

At its core, a Jasmine Spy is a special type of function that records all interactions that happen with it. Therefore, they are very useful when a returned value or change in an object's state can't be used to determine if a test expectation was a success.

Or in other words, Jasmine Spies are perfect when a test success can only be determined by **behavior checking**.

The "bare" Spy

To understand the concept of behavior checking, let's revisit an example presented in *Chapter 3, Testing Frontend Code*, and test the observable behavior of `NewInvestmentView`:

```
describe("and when an investment is created", function() {
  var callbackSpy;
  var investment;

  beforeEach(function() {
    callbackSpy = jasmine.createSpy('callback');
    view.onCreate(callbackSpy);

    investment = view.create();
  });
});
```



```
it("should invoke the 'onCreate' callback with the investment",
function() {
  expect(callbackSpy).toHaveBeenCalled();
  expect(callbackSpy).toHaveBeenCalledWith(investment);
});
});
```

During the spec setup, it creates a new Jasmine Spy using the `jasmine.createSpy` function while passing a name for it (`callback`), then it sets this spy as an observer of the View's create event using the `onCreate` function, and finally it invokes the `create` function to create a new investment.

Later on at the expectations, the spec uses the matcher's `toHaveBeenCalled` and `toHaveBeenCalledWith` to check if the `callbackSpy` was called and with the right parameters (`investment`), thereby making a behavior check.

Spy an object's functions

A Spy by itself is very useful, but its true power comes by changing an object's original implementation by using a counterpart Spy.

Consider the following example, which wants to validate that when the form is submitted, the `create` function of `view` has to be called.

```
describe("and when the form is submitted", function() {
  beforeEach(function() {
    spyOn(view, 'create');
    view.$el.submit();
  });

  it("should create an investment", function() {
    expect(view.create).toHaveBeenCalled();
  });
});
```

Here we make use of the global Jasmine function `spyOn` to change the `create` function of `view` by a spy.

Then later at the spec, we use the `toHaveBeenCalled` Jasmine matcher, to validate that the `view.create` function was called.

After the spec is done, Jasmine restores the object's original behavior.

Testing DOM events

DOM events are used all the time while coding frontend applications, and sometimes we want to write a spec that checks whether an event is being triggered.

An event could be something like a form submission or an input having changed, so how can we use Spies to do that?

We can write a new acceptance criterion to `NewInvestmentView` to check that its form is being submitted when the add button is clicked:

```
describe("and when its add button is clicked", function() {
  beforeEach(function() {
    spyOnEvent(view.$el, 'submit');
    view.$el.find('input[type=submit]').click();
  });

  it("should submit the form", function() {
    expect('submit').toHaveBeenTriggeredOn(view.$el);
  });
});
```

To write this spec, we use the `spyOnEvent` global function provided by the Jasmine jQuery plugin.

It works by accepting a DOM element (`view.$el`) and an event we want to spy on (`submit`). Then later on, we can check with `toHaveBeenTriggeredOn` whether the event was triggered on the element.

Summary

In this chapter, you were presented with the concept of test doubles, and how you can use Spies to perform behavior checking on your specs.

In the next chapter, we will take a look on how we can use Fakes and Stubs to replace real dependencies of our specs and speed up their executions.

6

Light Speed Unit Testing

In the *Chapter 4, Asynchronous Testing – AJAX*, we have seen how including AJAX testing in the application can increase the complexity of the tests. In the example, we created a server where the results were predictable. It was basically a complex test fixture. And even though we could have used a real server implementation, it would increase the complexity of the test even more, try changing the state of a server with a database or third-party services from the browser, it is not an easy or scalable solution.

There is also the impact on productivity; these requests take time to process and to transmit, which hurts the quick feedback loop that unit testing usually provides.

You can also say that these specs are testing both the client and the server code, and therefore could not be considered as unit tests, but rather as integration tests.

A solution to all these problems is to use either **Stubs** or **Fakes** in place of the real dependencies of the code. So instead of making a request to the server, we use a test double of the server, inside the browser.

We are going to use the same example from *Chapter 4, Asynchronous Testing – AJAX*, and rewrite it using different techniques.

Jasmine Stubs

We have already seen some use cases for Jasmine spies. Remember that a spy is a special function that records how it was called. You can think of a Stub as a Spy with behavior.

We use Stubs whenever we want to force a specific path in our specs, or replace a real implementation for a simpler one.

Let's revisit the example of the acceptance criteria, "Stock when fetched, should update its share price", by rewriting it using Jasmine Stubs.

Since we know that the stock's `fetch` function is implemented using the `$.getJSON` function:

```
Stock.prototype.fetch = function(parameters) {
  $.getJSON(url, function (data) {
    that.sharePrice = data.sharePrice;
    success(that);
  });
};
```

We could use the `spyOn` function to set up a spy on the `getJSON` function:

```
describe("when fetched", function() {
  beforeEach(function() {
    spyOn($, 'getJSON').andCallFake(function(url, callback) {
      callback({ sharePrice: 20.13 });
    });
    stock.fetch();
  });

  it("should update its share price", function() {
    expect(stock.sharePrice).toEqual(20.13);
  });
});
```

But this time we will use the `andCallFake` function to set a behavior to our spy (by default a spy does nothing and returns undefined). We make the spy invoke its callback parameter with an object response (`{ sharePrice: 20.13 }`).

Later, at the expectation, we use the `toEqual` assertion to verify that the stock's `sharePrice` has changed.

To run this spec, you no longer need a server to make the requests to, which is a good thing, but there is one issue with this approach. If the `fetch` function gets refactored to use `$.ajax` instead of `$.getJSON`, then the test will fail. A better solution, provided by the `Sinon.JS` library, is to Stub the browser's AJAX infrastructure instead, so the implementation of the AJAX request is free to be done in different manners.

Sinon.JS

Sinon.JS is a great library created by Christian Johansen, author of the great book, *Test-Driven JavaScript Development*, to make easy dealing with Stubs, Spies, and Mocks.

Although Jasmine already has support for Stubs and Spies, we are going to use a specific functionality of Sinon.JS to test AJAX requests, its `FakeXMLHttpRequest` and `FakeServer` functions.

The main difference between a Stub and a Fake, as you will see with the `FakeXMLHttpRequest` object, is that a Fake is like a simpler but still complete implementation of a real component, and it is usually set at a system level.

Installing Sinon.JS

Before we dig into the spec implementation, first we need to add Sinon.JS to the project. Go to <http://sinonjs.org/> and download the current release, placing it inside the `lib` folder.

We also need to add it to the `SpecRunner.html` file, so go ahead and add another script:

```
<script type="text/javascript" src="lib/sinon.js"></script>
```

A Fake XMLHttpRequest

Whenever you are making AJAX requests with jQuery, under the hood it is using the `XMLHttpRequest` to actually perform the request.

`XMLHttpRequest` is the standard JavaScript HTTP API. Even though its name suggests that it uses XML, it supports other types of content such as JSON, the name has remained the same for compatibility reasons.

So instead of Stubbing jQuery, we could instead Fake, the global `XMLHttpRequest` object. That is exactly what Sinon.JS does with its `FakeXMLHttpRequest` implementation.

Let's rewrite the previous spec to use this Fake implementation:

```
describe("when fetched", function() {  
    var xhr;  
  
    beforeEach(function() {  
        var fetchRequest;
```

```
xhr = sinon.useFakeXMLHttpRequest();

xhr.onCreate = function (request) {
  fetchRequest = request;
};

stock.fetch();

fetchRequest.respond(
  200,
  { "Content-Type": "application/json" },
  '{ "sharePrice": 20.13 }'
);
});

afterEach(function() {
  xhr.restore();
});

it("should update its share price", function() {
  expect(stock.sharePrice).toEqual(20.13);
});
});
```

First, we tell Sinon.JS to replace the original implementation by its Fake using the `sinon.useFakeXMLHttpRequest` function.

Then, we add an observer to get the newly created requests by setting a function as a value of the `xhr.onCreate` attribute, storing them on a variable named `fetchRequest`.

We then invoke the `stock.fetch` function, which will invoke `$.getJSON`, creating a new `XMLHttpRequest` under the hood.

And finally, we use the `fetchRequest` variable (which contains the `FakeXMLHttpRequest` object caught by the observer), to respond with a fake content.

We use the `respond` function, which accepts three parameters:

- An integer defining the HTTP status code
- An object containing the HTTP headers
- A string with the response body

Then, it's all a matter of running the expectations:

```
it("should update its share price", function() {  
    expect(stock.sharePrice).toEqual(20.13);  
});
```

Since Sinon.JS changes the global `XMLHttpRequest` object, you must remember to tell Sinon.JS to restore it to its original implementation after the test runs, otherwise you could interfere with the code (such as the Jasmine jQuery fixtures module) from other specs:

```
afterEach(function() {  
    xhr.restore();  
});
```

A Fake server

Sinon.JS's `FakeXMLHttpRequest` is a very good solution to stub AJAX requests, but things can start to get complicated if you need to deal with more than one request, or need to have different responses for different requests.

To help manage `FakeXMLHttpRequest` instances, Sinon.JS comes with another solution, the Fake server.

Sinon.JS Fake server abstracts the manipulation of the individual `FakeXMLHttpRequest` instances into a high-level API, that lets you focus on what response you want for a particular request type.

Again, let's rewrite the same example, but now using the Fake server functionality:

```
describe("when fetched", function() {  
    var xhr;  
  
    beforeEach(function() {  
        xhr = sinon.fakeServer.create();  
        xhr.respondWith([  
            200,  
            { "Content-Type": "application/json" },  
            '{ "sharePrice": 20.13 }'  
        ]);  
  
        stock.fetch();  
  
        xhr.respond();  
    });
```



```
    afterEach(function() {
      xhr.restore();
    });

    it("should update its share price", function() {
      expect(stock.sharePrice).toEqual(20.13);
    });
  });
});
```

Now, instead of dealing with `XMLHttpRequest`, we create a new instance of the Fake server using the `sinon.fakeServer.create` function.

Then, we call the `respondWith` function to configure the Fake server to always respond to requests with a Fake response.

After the `stock.fetch()` call, we tell the Fake server to respond to all made requests.

After each spec runs, it is also important to restore the original `XMLHttpRequest` behavior.

The coolest thing about Fake server is its ability to create different responses, based on different URLs. For instance, we could have written the previous server response as:

```
xhr.respondWith(
  '/stocks/AOUE',
  [
    200,
    { "Content-Type": "application/json" },
    '{ "sharePrice": 20.13 }'
  ]
);
```

Notice the extra parameter `'/stocks/AOUE'` telling the Fake server to only respond to requests made with that URL. It is even possible to specify the HTTP method (GET, POST, and so on) and to use regular expressions to match the URL:

```
xhr.respondWith(
  'GET',
  /\sstocks\/(.+)\s/,
  [
    200,
    { "Content-Type": "application/json" },
    '{ "sharePrice": 20.13 }'
  ]
);
```

You can also pass a function to the body parameter, and have dynamic responses:

```
xhr.respondWith(  
  'GET',  
  /\//stocks\/(.+)\//,  
  function (request, stockSymbol) {  
    request.respond(  
      200,  
      { "Content-Type": "application/json" },  
      '{ "sharePrice": 20.13 }'  
    );  
  }  
);
```

Notice the `stockSymbol` parameter, it contains the matched value extracted from the request URL based on the `/\//stocks\/(.+)\//` regular expression. Whenever using regular expressions and function bodies to handle requests on a Fake server, the matched strings passed to the function in the order they are found.

Summary

In this chapter, you have learned how asynchronous tests can hurt the quick feedback loop you can get with unit testing. I showed how you can use either Stubs or Fakes to make your specs run quicker and with fewer dependencies.

We have seen three different ways you could test AJAX request, with a simple Jasmine Stub and with the more advanced, Fake XMLHttpRequest, and the Fake server from the library Sinon.JS.

You also got more familiar with Spies and Stubs, and should be more comfortable in using them in different scenarios.

In the next chapter, we are going further in the complexity of our application, and we will do an overall refactoring to transform it into a fully featured single page application with Backbone.js.

7

Testing Backbone.js Applications

Testing Backbone applications is no different than testing any other application; you are still going to drive your code from your specs, except that Backbone is already leveraging a lot of functionality for you, for free. So expect to write less code, and consequently less specs.

Backbone is a micro framework designed to give web applications just enough structure to allow them to grow. It provides four base abstractions:

- **Model:** It provides a key-value store for the application data along with custom events
- **Collection:** It provides a rich enumerable API
- **View:** It creates the interface building blocks
- **Router:** It provides methods for client-side routing

We will see some common testing scenarios when dealing with each type of abstraction, and some common mistakes people make while creating these specs.

The Backbone model

They are the real backbone of Backbone, they are the abstractions on which we build the business logic, they hold the data and are responsible for performing validations and synchronization with a remote server; they are the Backbone models.

Although we are not using Backbone on our sample application, we already have some well-defined models, both the Stock and Investment objects. But before we dig in how we could refactor them to become Backbone models, we need first to get a little bit of understanding on how they work.

Declaring a new model

To create a new model object we need first to extend it from the base Backbone model. To make matters more interesting, we are going to rewrite the entire Stock spec, by expecting the Stock to be a Backbone model.

On the Stock spec we could write the following acceptance criterion, that although not business related, guarantees that this model will inherit all of the model's functionalities:

```
describe("Stock", function() {
  var stock;

  beforeEach(function() {
    stock = new Stock();
  });

  it("should be a Backbone.Model", function() {
    expect(stock).toEqual(jasmine.any(Backbone.Model));
  });
});
```

To implement this new model we need to use the `Backbone.Model.extend` function in the Stock source file:

```
(function (Backbone) {
  var Stock = Backbone.Model.extend();
  this.Stock = Stock;
})(Backbone);
```

And we are done! The Stock is now a fully featured Backbone model with all of its niceness.

But now that we are done, what can we do with it?

The model attributes

At the core of a Backbone model is a key-value store that should be used to hold all of the model's data. And you can access it by a very simple API of getter and setter functions. Given here is a code snippet of how it works for the previously declared model:

```
var stock = new Stock();
stock.set('sharePrice', 10)
stock.get('sharePrice') // 10
```

You can also pass objects to the `set` function, in order to set multiple parameters:

```
stock.set({
  sharePrice: 20,
  symbol: 'AOUE'
});
```

And still be able to get each attribute individually:

```
stock.get('sharePrice') // 20
stock.get('symbol') // AOUE
```

You can even pass all of the Model's attributes on its instantiation, and the getter method would still work for each attribute individually:

```
var stock = new Stock({
  sharePrice: 20,
  symbol: 'AOUE'
});
```

Now that you understand how the model attributes work, let's revisit a simple test case of the Stock spec (already ported to Backbone):

```
beforeEach(function() {
  stock = new Stock({
    sharePrice: 10
  });
});

it("should have a share price", function() {
  expect(stock.get('sharePrice')).toEqual(10);
});
```

Although this scenario was useful to test a piece of code previously, it has now become obsolete. We are basically testing a Backbone functionality, something that is already well tested by the Backbone library itself, so you can safely remove that spec and still be confident that the model attributes are working (as long as you have the spec to test that it is a Backbone model).

Good! So far, Backbone has helped us write both less code and fewer specs.



Although you can write a spec code to everything, try not to repeat yourself (by rewriting other people's specs) and write specs that cover a piece of code that you have actually written.

Default attribute values

Back to the model attributes, we are going to explore a Backbone functionality that does require some testing from our part, but that is because we are also going to write some code to make it work. It is a very simple functionality to set default model values on instantiation.

Still at the Stock example, let's add another acceptance criterion that it should have a default share price value of zero:

```
describe("Stock", function() {
  var stock;

  beforeEach(function() {
    stock = new Stock({ symbol: 'AOUE' });
  });

  it("should have a default share price of zero", function() {
    expect(stock.get('sharePrice')).toEqual(0);
  });
});
```

We have seen standard Backbone attribute handling previously. Notice that we are not setting the share price value at any time on the spec, so it is up to the Stock code to make it happen.

And here comes another Backbone functionality, normally we would set this on a constructor function, but Backbone deals with that for us, all we have to do is declare a `defaults` object in the Stock definition, providing the default value for each of the model's attributes. Let's see how it works:

```
var Stock = Backbone.Model.extend({
  defaults: {
    'sharePrice': 0
  }
});
```

Simple and clean.

Events

As we have seen in *Chapter 3, Testing Frontend Code*, the Observable pattern was a great solution to perform the integration between two Views, while reducing coupling. In Backbone, every one of its four abstractions is built on a shared event infrastructure, making all of them Observable objects by default. In models, it is possible to observe changes in the whole model or individual attributes, be notified when it syncs with a backend (as we will see later) and even possible to create custom events. If you are familiar with how to listen for events on jQuery, using Backbone functions will feel right at home.

Again, let's see some code to understand how it works:

```
var stock = new Stock();
stock.on('change:sharePrice', function () {
  alert('it has changed!');
});
stock.set({ 'sharePrice': 30 })
```

We are using the same Stock model defined earlier. First, we add an observer through the `on` function, to listen for changes of the share price value on the stock instance. You can see that we are passing the event name (`change:sharePrice`) and a function to be called back once the event happens.

Then later, we change the value of the share price attribute. Behind the scenes, Backbone notices the change and notifies all listening observers, showing the alert message (`it has changed`) on your browser.

This is great to keep interfaces and models in sync as we will see later on. But how can we harness this functionality to any good purpose inside the model itself?

Remember the Investment object? After converting it to a Backbone model, we can rewrite one of its specs to demonstrate a test case involving model events.

Here is a good candidate involving the `roi` attribute:

```
describe("when its stock share price valorizes", function() {
  beforeEach(function() {
    stock.set('sharePrice', 40);
  });

  it("should have a positive return of investment", function() {
    expect(investment.get('roi')).toEqual(1);
  });
});
```


The `roi` attribute is a so called virtual attribute, as it is a result of a calculation between two other attributes. Previously, we had chosen to implement it like a function, and we still could, but doing so we would lose the benefit of having a homogeneous interface between all of the model attributes, and besides, `roi` being a regular attribute, others can also observe for changes on it as well.

Recapitulating, an Investment return of investment is the ratio between the paid share price and its current value. We could create a function to calculate it like:

```
function calculateROI () {  
  var sharePrice = this.get('sharePrice');  
  var stockSharePrice = this.get('stock').get('sharePrice');  
  this.set('roi', (stockSharePrice - sharePrice) / sharePrice);  
}
```

By now you probably already know how to make the spec pass. We want to calculate the return of investment every time an investment stock changes its share price.

To do that inside the Investment object, we need to add an observer to the `stock` attribute once an investment is created:

```
var Investment = Backbone.Model.extend({  
  initialize: function () {  
    this.get('stock').on('change:sharePrice', calculateROI, this);  
  }  
});
```

Done! We had to pass another configuration during the definition of Investment to specify an `initialize` function. It acts like a constructor, being called once a model has been instantiated.

Obviously this implementation is not complete, since the `stock` attribute itself might change to a different stock, requiring the rebinding of the change observer. But I'll leave that as an exercise for you.

We will dig more into Backbone events in this chapter, but for a complete reference, be sure to check the official documentation available at <http://backbonejs.org/#Events>.

Sync and AJAX requests

A model wouldn't be a model without some sort of mechanism to allow it to be read or saved to a server. And as you have guessed, this piece of implementation is about to get much simpler for you.

By default, Backbone comes with support for backend servers that implement the REST standard, and all that is left for you to do is to configure an end point for it to make requests and it automatically fetches, updates, creates, and delete models.

For now, the only piece of application that depends on a backend server is the Stock model with its `fetch` function. Let's see what Backbone has to offer on this matter.

Here is the original spec implementation with a few tweaks to make it simpler and compatible with Backbone:

```
describe("Stock", function() {
  var stock;

  beforeEach(function() {
    stock = new Stock({ symbol: 'AOUE' });
  });

  describe("when fetched", function() {
    var fakeServer;

    beforeEach(function() {
      fakeServer = sinon.fakeServer.create();
      fakeServer.respondWith('/stocks/AOUE',
                            '{ "sharePrice": 20.13 }'
                            );

      stock.fetch();

      fakeServer.respond();
    });

    afterEach(function() {
      fakeServer.restore();
    });

    it("should update its share price", function() {
      expect(stock.get('sharePrice')).toEqual(20.13);
    });
  });
});
```

This is the implementation using Sinon.JS Fake server seen in *Chapter 6, Light Speed Unit Testing*. Before we had to implement this `fetch` function by ourselves, but with Backbone, all we have to do is set up two configurations on the Stock definition:

```
var Stock = Backbone.Model.extend({
  idAttribute: 'symbol',
  urlRoot: '/stocks'
});
```

These attributes are:

- `urlRoot`: This indicates the root URL that Backbone needs to perform AJAX requests into
- `idAttribute`: This indicates which of the model's attribute it must use as an ID while making the AJAX request

Once again we are done! The spec should be passing and everyone in the room should be happy!

But that spec we have written is appearing to show its age, it is testing so much more than the code we have written. Although it is not a bad thing, we can make things simpler by trusting that Backbone is doing a correct implementation of its `fetch` function (given that we supply the right parameters) and rewrite the spec, now that we are familiar with the Backbone API:

```
describe("Stock", function() {
  var stock;

  beforeEach(function() {
    stock = new Stock({ symbol: 'AOUE' });
  });

  it("should allow fetching its information", function() {
    expect(stock.idAttribute).toEqual('symbol');
    expect(stock.urlRoot).toEqual('/stocks');
  });
});
```

We have replaced the whole `when fetched` describe function by a single spec with two assertions, while still assuring that the `fetch` function works.

Once again, less code and fewer specs. But take notice that we could have left the old spec implementation. It does provide a little more confidence, since you could (in theory) write both the implementation and the spec wrongly, such as a mistype of `urlRoot` as `urlroot` on both files.

It is a tradeoff between simplicity and a little more security. It is your call when to use each approach.

There is much more to Backbone sync, such as support for Local Storage or XML; be sure to check the full documentation available at <http://backbonejs.org/#Sync>.

Testing Backbone collections

Backbone collections are basically Arrays with superpowers:

- They come with a bunch of enumerable functions built in, such as map, sort, and select
- They have support for all sorts of events such as addition, removal, and even changes made to individual models it contains
- They support reading data from a remote server

Expect to use it a lot in conjunction with models in your Backbone application.

Let's see a small code snippet to get a feeling on how it works. Here we instantiate a new collection passing an array with an initial data item for it to start:

```
var collection = new Backbone.Collection([
  { id: 1, name: 'first' }
]);
```

It will by default, create a new `Backbone.Model` for each object in that array, but it is possible to specify your own custom Model objects.

Next, we show how to add a new Model to the collection:

```
collection.add(new Backbone.Model({ id: 2, name: 'second' }));
```

And finally, we can retrieve the added model by its `id` attribute:

```
var model = collection.get(1)
model.get('name') // first
```

They also come with event support, so it is possible to listen for changes on the collection:

```
collection.on('add', function (newModel) { })
```

And a lot more features.

Declaring a new collection

Before we can dig some of the features and how we can test Backbone collections, we must first learn how to declare one, and for that we are going to need an example.

We are going to create a new collection of Stocks, it will need a new source and spec file, written at `src/StockCollection.js` and `spec/StockCollectionSpec.js` respectively.

In the spec file, we can start by expecting that this new `StockCollection` to be a Backbone collection and also that it is a collection of Stocks:

```
describe("StockCollection", function() {
  var collection;

  beforeEach(function() {
    collection = new StockCollection();
  });

  it("should be a Backbone Collection", function() {
    expect(collection).toEqual(jasmine.any(Backbone.Collection));
  });

  it("should be of Stocks", function() {
    expect(collection.model).toBe(Stock);
  });
});
```

To define which model a collection contains is not a requirement, but by specifying, the collection knows how to create new instances of the model, for example, while doing a fetch.

Here is the `StockCollection` implementation:

```
(function (Backbone, Stock) {
  var StockCollection = Backbone.Collection.extend({
    model: Stock
  });

  this.StockCollection = StockCollection;
})(Backbone, Stock);
```

Make the `StockCollection` an extension of the base `Backbone.Collection`, and set its model as the `Stock`.

Sync and AJAX requests

As with the model, it is also possible to fetch data of a collection, except that we are retrieving one or more models in a single request.

In `StockCollection`, we want it to have a `fetch` function that updates its models. To make it more interesting, we are going to add a requirement that we must send to the server about which stock data we want, based on the stocks available in the collection.

Our server will be expecting a URL containing a query string with the stock IDs, something like (for our development server): `http://0.0.0.0:8000/stocks/?ids[]=AOUE&ids[]=COUY`.

This spec is going to be a little more complicated, so we are going to start with just its skeleton, so you can have a feeling of how we are going to write it:

```
describe("StockCollection", function() {
  describe("given a populated collection", function() {
    describe("when fetch", function() {
      it("should request by the Stocks it contains", function(){});
      it("should update its models share price", function(){});
    });
  });
});
```

And then, we are going to show you how each piece is implemented, starting with given a populated collection:

```
describe("given a populated collection", function() {
  beforeEach(function() {
    model1 = new Stock({ symbol: 'AOUE' });
    model2 = new Stock({ symbol: 'COUY' });

    collection = new StockCollection([
      model1,
      model2
    ]);
  });
});
```

Where it creates a collection with a set of two models.

Next, we need to perform the `fetch` function. For this to work, we are going to use the Sinon.JS Fake server:

```
describe("when fetch", function() {
  beforeEach(function() {
    fakeServer = sinon.fakeServer.create();
    fakeServer.respondWith(JSON.stringify([
      {
        symbol: 'AOUE',
        sharePrice: 20.13
      },
      {
        symbol: 'COUY',
        sharePrice: 14
      }
    ]));

    collection.fetch();
    fakeServer.respond();
  });

  afterEach(function() {
    fakeServer.restore();
  });
});
```

You can see that here we are configuring the Fake server with a Fake response, performing the `fetch` function, and telling the Fake server to respond to all the requests made.

Finally, we can check if our collection is making the correct request:

```
it("should have request by the Stocks it contains", function() {
  // encoded '/stocks?ids[]=AOUE&ids[]=COUY'
  var url = '/stocks?' + $.param({ ids: ['AOUE', 'COUY'] });

  expect(fakeServer.requests[0].url).toEqual(url);
});
```

And then check if the models were updated with the Fake response data:

```
it("should update its models share price", function() {
  expect(model1.get('sharePrice')).toEqual(20.13);
  expect(model2.get('sharePrice')).toEqual(14);
});
```

And that is it. Although extensive, it is pretty simple to understand. But what about the actual implementation? For switching back to the source file we need to define a URL property that is used by Backbone while making requests:

```
(function (Backbone, Stock) {
  var StockCollection = Backbone.Collection.extend({
    model: Stock,
    url: function () {
      return "/stocks" + idsQueryString.call(this);
    }
  });

  function modelIds () {
    return this.map(function (model) { return model.id; });
  }

  function idsQueryString () {
    var ids = modelIds.call(this);

    if (ids.length === 0) { return ''; }
    return '?' + $.param({ ids: ids });
  }

  this.StockCollection = StockCollection;
})(Backbone, Stock);
```

There it is. You can see by the highlighted part that the URL is dynamic, and we are constructing it based on the IDs of the models stored on the collection, much in the way that we wanted.

Back to the explanation on how to test the Sync of models, we told you how the spec was outdated. This could also be the case in here. See how you could simplify this spec while still guaranteeing that it works.

There are many more features to Backbone collections, for more information, be sure to check the official documentation available at <http://backbonejs.org/#Collection>.

Testing Backbone Views

We already have seen some of the advantages of using the View pattern in *Chapter 3, Testing Frontend Code*, and are already creating our interface components in such a manner. So how can a Backbone View be different from what we have done so far?

It retains a lot of the patterns that we have discussed as best practices for creating maintainable browser code, but with some syntax sugar and automation to make our life easier.

They are the glue code between the HTML and the model, and the Backbone View's main responsibility is to add behavior to the interface, while keeping it in sync with a model or collection.

As we will see, Backbone's biggest triumph is how it makes an easy-to-handle DOM event delegation, a task usually done with jQuery.

Declaring a new View

Very similar to what we have seen so far, declaring a new View is going to be a matter of extending the base `Backbone.View` object.

To demonstrate how it works we need an example. We are going to create a new View and its responsibility is going to be to render a single investment on the screen.

We are going to create it in such a way that allows its use by the `InvestmentListView` component discussed briefly in *Chapter 3, Testing Frontend Code*.

This is a new component and spec, written in `src/InvestmentView.js` and `spec/InvestmentViewSpec.js` respectively.

In the spec file, we can write something similar to what we have seen previously:

```
describe("InvestmentView", function() {
  var view;

  beforeEach(function() {
    view = new InvestmentView();
  });

  it("should be a Backbone View", function() {
    expect(view).toEqual(jasmine.any(Backbone.View));
  });
});
```

Which translates into an implementation that extends the base `Backbone.View` component:

```
(function (Backbone) {
  var InvestmentView = Backbone.View.extend()
  this.InvestmentView = InvestmentView;
})(Backbone);
```

And now we are ready to explore some of the new functionalities provided by Backbone.

The `el` property

Like the View pattern described in *Chapter 3, Testing Frontend Code*, a Backbone View also has an attribute containing the reference to its DOM element.

The difference here is that Backbone comes with it by default, providing:

- `view.el`: The DOM element
- `view.$el`: The jQuery object for that element
- `view.$`: A scoped jQuery lookup function (the same way we have implemented)

And if you don't provide an element on the constructor, it creates an element for you automatically. Of course the element it creates is not attached to the document, and is up to the View's user code to attach it.

Here is a common pattern you see while using Views:

1. Instantiate it:

```
var view = new InvestmentView();
```
2. Call the `render` function to draw the View's components (as we will see in the next section):

```
view.render();
```
3. Append its element to the page document:

```
$('body').append(view.el);
```

Given our clean implementation of the `InvestmentView`, if you would go ahead and execute the preceding code on a clean page, you would get the following result:

```
<body>
  <div></div>
</body>
```

An empty `div` element; that is the default element created by Backbone. But we can change that with a few configuration parameters on the `InvestmentView` declaration.

Let's say we want the DOM element of `InvestmentView` to be a list item (`li`) with an `investment` CSS class. We could write this spec using the familiar Jasmine jQuery matchers:

```
describe("InvestmentView", function() {
  var view;

  beforeEach(function() {
    view = new InvestmentView();
  });

  it("should be a list item with 'investment' class", function() {
    expect(view.$el).toBe('li.investment');
  });
});
```

You can see that we didn't use the `setFixtures` function, since we can run this test against the element instance available on the View.

Now to the implementation; all we have to do, is define two simple attributes in the View definition, and Backbone will use them to create the View's element:

```
var InvestmentView = Backbone.View.extend({
  className: 'investment',
  tagName: 'li'
});
```

By looking at the implementation you might be wondering if we shouldn't test it like we did with in the *Backbone Model: Sync and AJAX requests* section. Here I would recommend against it, since you wouldn't get any benefit from that approach, as this spec is much more solid.

That is great, but how do we add content to that DOM element? That is up to the render function we are going to see next.

Remember that we could have passed an element while constructing the View, in the same way we were doing in *Chapter 3, Testing Frontend Code*:

```
var view = new InvestmentView({ el: $('body') });
```

But by letting the View handle its rendering, we get better componentization and we can also gain on performance.

Rendering

Now that we understand that it is a good idea to have an empty element available on the View, we must get into the details of how to draw on this empty canvas.

Backbone Views already come with an available `render` function, but it is a dummy implementation, so it is up to you to define how it works.

Going back to the `InvestmentView` example, let's add a new acceptance criterion to describe how it should be rendered. We are going to start by expecting that it renders the return of investment as a percentage value. Here is the spec implementation:

```
describe("InvestmentView", function() {
  var view, investment;

  beforeEach(function() {
    investment = new Investment();
    view = new InvestmentView({ model: investment });
  });

  describe("when rendering", function() {
    beforeEach(function() {
      investment.set('roi', 0.1);
      view.render();
    });

    it("should render the return of investment", function() {
      expect(view.$el).toContainHtml('10%');
    });
  });
});
```

That is a very standard spec with concepts that we have seen before and the implementation is just a matter of defining the `render` function on the `InvestmentView` declaration:

```
var InvestmentView = Backbone.View.extend({
  className: 'investment',
  tagName: 'li',

  render: function () {
    this.$el.html('<p>' + formattedRoi.call(this) + '<p>');
    return this;
  }
});

function formattedRoi () {
  return (this.model.get('roi') * 100) + '%';
}
```

It is using the `this.$el` property to add some HTML content to the View's element. There are some details that are important for you to notice regarding the `render` function implementation:

- We are using the `jQuery.html` function, so that we can invoke the `render` function multiple times without duplicating the View's content.
- The `render` function returns the View instance once it has completed rendering. This is a common pattern to allow chained calls, such as:

```
$('#body').append(new InvestmentView().render().el);
```

Now back to the test. You can see that we weren't testing for the specific HTML snippet, but rather, that just 10 percent text was rendered. You could have done a more thoroughly written spec by checking the exact same HTML at the expectation, but that ends up adding test complexity with little benefit.

Updating the View on model changes

We understand how Views are rendered and how model events work, wouldn't it be great if we could tie these things together and make the View render itself every time a model changes? That is exactly what we can do!

Back at the `InvestmentView` spec we can add a new spec to check if the View renders itself once the model gets updated:

```
describe("when the investment changes", function() {
  beforeEach(function() {
    spyOn(view, 'render');
    investment.trigger('change', investment);
  });

  it("should update the interface", function() {
    expect(view.render).toHaveBeenCalled();
  });
});
```

The spec works by triggering a change event, the same way the model does when we set an attribute, and then expect that the `render` function was called.

That looks ok, so the next step is to add the implementation in the `InvestmentView` constructor:

```
var InvestmentView = Backbone.View.extend({
  className: 'investment',
  tagName: 'li'
```

```

    initialize: function () {
      this.model.on('change', this.render, this);
    },

    render: function () {
      this.$el.html('<p>' + formattedRoi.call(this) + '<p>');
      return this;
    }
  });

```

Just like other Backbone abstractions, the constructor can be implemented by adding an `initialize` function to the View's definition.

Next, we use the model's event infrastructure to bind the `change` event to the View's `render` function:

```

    this.model.on('change', this.render, this);

```

This implementation is correct, but if you try to run the spec, it should be failing because of a detail on how Spies work.

You see, by spying on the View's `render` function, we are actually replacing its original implementation by a `spy` function. Since the `InvestmentView` binds the event on its constructor, it is still binding on the original `render` implementation, not our `spy`.

To make this test work, we need to set up the `spy` on the `render` function before instantiating the View. We can do that by setting the `spy` on the `render` function in the `InvestmentView`'s prototype, before instantiating the View:

```

beforeEach(function() {
  spyOn(InvestmentView.prototype, 'render');

  investment = new Investment();
  view = new InvestmentView({
    model: investment
  });
});

describe("when the investment changes", function() {
  beforeEach(function() {
    investment.trigger('change', investment);
  });

  it("should update the interface", function() {
    expect(view.render).toHaveBeenCalled();
  });
});

```

And now the specs should be passing!

Before we can go to the next section, there is one more thing that is important to know, and it concerns memory leaking. By adding that event listener, the View and model instances are going to be bound forever, so even after destroying the View instance, it is never going to be freed from the memory until the model instance has been freed.

To fix that, Backbone provides another function to all of its components to allow listening to other component events; the `listenTo` function.

So instead of adding the event handler to the model:

```
this.model.on('change', this.render, this);
```

We ask the View to listen for that event on the model:

```
this.listenTo(this.model, 'change', this.render, this);
```

By using `listenTo`, the View knows of all the event handlers created by it, and once it is destroyed, it can remove all of them.

But to make it work, you must remember that whenever you finish using a View, you explicitly remove it by invoking the View's `remove` function:

```
view.remove()
```

Binding DOM events

Up until now we have the View rendering and being updated on every model change, but what about updating the model once the View changes?

It is all a matter of adding event handlers to the View DOM elements and once they are triggered, we update the model.

To demonstrate this concept, let's add another acceptance criteria to our `InvestmentView`. We want to add a new button, that once clicked, triggers the destruction of the investment model.

To be able to click on the destroy button, first it needs to be rendered. So let's write this spec as follows:

```
describe("InvestmentView", function() {
  var view, investment;

  beforeEach(function() {
    investment = new Investment();
```

```
    view = new InvestmentView({
      model: investment
    });
  });

  describe("when rendering", function() {
    beforeEach(function() {
      view.render();
    });

    describe("when the destroy button is clicked", function() {
      beforeEach(function() {
        spyOn(investment, 'destroy');
        view.$('.destroy-investment').click();
      });

      it("should destroy the model", function() {
        expect(investment.destroy).toHaveBeenCalled();
      });
    });
  });
});
```

We add a spy on the `investment.destroy` function, then we simulate the click, and finally expect the `destroy` function to have been called.

Now comes the nice part. Normally you would have to add that click event on jQuery by hand, but with Backbone, all you have to do is add an `events` object to the `InvestmentView` definition.

This `events` object must contain the DOM events that you want to bind, next to a function definition or function name, and Backbone does the binding for you.

Here is the code for the `InvestmentView`:

```
var InvestmentView = Backbone.View.extend({
  className: 'investment',
  tagName: 'li',
  events: {
    'click .destroy-investment': function () {
      this.model.destroy();
    }
  },
  initialize: function () {
```



```
    this.listenTo(this.model, 'change', this.render, this);
  },

  render: function () {
    this.$el.html('<p>'+ formattedRoi.call(this) +'<p>');
    return this;
  }
});
```

Here you can see that we are listening for clicks on the **destroy-investment** button and passing a function handler that calls the `destroy` function of the model.

To define the event, we need to pass which event we want to listen to and on which element. Remember that this is all scoped to the View's DOM element:

```
'click .destroy-investment'
```

But this definition can also receive just an event type, in case you want to add events to the View DOM element itself. Suppose we want to get all clicks to View:

```
events: {
  'click': function () {}
}
```

It also supports passing a function name, letting Backbone look for the function definition on the View instance, and call it for you:

```
var InvestmentView = Backbone.View.extend({
  events: {
    'click .destroy-investment': 'destroyTheModel'
  },

  destroyTheModel: function () {
    this.model.destroy();
  },
});
```

Another thing you might have noticed is that Backbone is taking care of calling your event handlers while binding the `this` value to the View instance, a little detail you don't have to worry anymore.

Testing Backbone routers

With the shift of application code to the client, we are continually seeing more single page web applications. But they come with a drawback. Since we are keeping the users on the same page, we are losing the natural ability of tracking steps through the URL changes.

Routers in Backbone are a solution to this problem. They allow the definition of routes inside your JavaScript code, giving back the feel of a web page to your rich web application.

Defining a new router

Suppose that we want our application to support the URL routes to display different types of investment:

- `/investments/good`: To show only a list of the good investments
- `/investments/bad`: To show only a list of the bad investments
- `/investments/all`: To show a list of all the investments

Normally this would be handled on the server, but with Backbone, we can define how to handle these routes in JavaScript on the browser. There is one catch though, for this to work it needs to use hash the URL fragments (`#/investments`). So all the above URLs would actually be (on our development server) like this `http://0.0.0.0:8000#/investments/good`.

It is possible to make Backbone use the new History API, and support actual URLs, but this topic is out of the scope of this book.

Back to the implementation of our router, we can see that all these routes deal with investments. So we can start by creating a new `InvestmentRouter`, with its source and spec file.

At the spec file we can start by expecting it to be a Backbone router:

```
describe("InvestmentsRouter", function() {
  var router;

  beforeEach(function() {
    router = new InvestmentsRouter();
  });

  it("should be a Backbone Router", function() {
    expect(router).toEqual(jasmine.any(Backbone.Router));
  });
});
```

Which in the code translates to a familiar Backbone component declaration:

```
(function (Backbone) {  
  var InvestmentsRouter = Backbone.Router.extend();  
  this.InvestmentsRouter = InvestmentsRouter;  
})(Backbone);
```

Defining routes

So how do we go on implementing these routes?

One approach we could take is to keep the router files as simple as possible, letting them handle the URLs and triggering events so that the application can handle the user requests.

To better understand what it means, let's see a spec regarding the `'/investments/good'` route:

```
describe("InvestmentsRouter", function() {  
  var router, observer;  
  
  beforeEach(function() {  
    router = new InvestmentsRouter();  
    observer = jasmine.createSpy();  
  });  
  
  it("should route '/investments/good'", function() {  
    router.on('route:goodInvestments', observer);  
  
    Backbone.history.loadUrl('/investments/good')  
  
    expect(observer).toHaveBeenCalled();  
  });  
});
```

Let's see what this spec does:

- First, it uses the familiar events infrastructure to add an observer to a `route:goodInvestment` event:
`router.on('route:goodInvestments', observer);`
- Then, we use Backbone to simulate a URL request:
`Backbone.history.loadUrl('/investments/good');`
- And finally, we expect that our observer was called:
`expect(observer).toHaveBeenCalled();`

To implement this on the router is a matter of adding a new entry to a routes object:

```
var InvestmentsRouter = Backbone.Router.extend({
  routes: {
    'investments/good': 'goodInvestments'
  }
});
```

We define a route by a matching pattern and a name, in this case `investments/good` and `goodInvestments` respectively.

This is how you test and write routers.

Using routers

I bet you are curious about how we could integrate this into the application. Here is a snippet showing how that could be done:

```
this.router = new InvestmentsRouter();
this.router.on('route:goodInvestments', function () {
  this.applicationView.showGoodInvestments();
}, this);

Backbone.history.start();
```

The last line is important; it tells Backbone to start listening for URL changes, effectively enabling the Routers.

Routers should only route

There is a common mistake that Backbone beginners do trying to drive the application development from routes. This is a common pattern in web applications that require a new request to an URL for each action the user does. But that is no longer the case. You are developing a rich web application, something that is much more like a native application.

So, drive your code from your views, you can always add routes later.

Summary

In this chapter, you have seen how to use Backbone to do some heavy lifting, allowing you to focus more on your application code. You have learned about the four Backbone abstractions and some common scenarios while testing applications developed with them.

I showed you the power of events, and how they make integration between different components much easier, allowing you to keep your models and Views in sync.

You have also seen how to take advantage of these powerful abstractions, and to write less code and fewer specs. You were also faced with the decision of writing simpler specs and to trust that Backbone implementations were correct.

By now you should feel comfortable in using Backbone to create a new single-page application, and I hope to have showed you how to think about testing applications that are built on top of frameworks. In the next chapter, we are going to see how to pack and minify the application code.

8

Build Automation

We have seen how to create an application from the ground up using tests with Jasmine. But as the application grows and the numbers of files starts to increase, managing the dependencies between them can become a little difficult.

For instance, we have a dependency between the Investment and the Stock models, and they must be loaded in a proper order to work. So we do what we can; we order the loading of the scripts so that the Stock is available once the Investment is loaded:

```
<script type="text/javascript" src="src/Stock.js"></script>
<script type="text/javascript" src="src/Investment.js"></script>
```

But that soon can become cumbersome and unmanageable.

Another problem is the number of requests the application uses to load all of its files. Up until now it is 13 different files, so 13 requests.

So we have here a paradox; although it is good for code maintainability to break it in small modules, it is bad for the client performance, where a single file is much more desirable.

A perfect world would be to match the following two requirements at the same time:

- In development we have a bunch of small files containing different modules
- In production we have a single file with the content of all those files

Clearly what we need is some sort of build process. There are many different ways to achieve these goals with JavaScript, but we are going to focus on RequireJS.

RequireJS

RequireJS is an AMD implementation. **AMD (Asynchronous Module Definition)** is a standard on how to write modules in JavaScript.

There are other specifications such as CommonsJS that is used by `Node.js`, and they all work. But AMD is different from the others, in that it works in browsers seamlessly between development and production.

It was created based on the specifics of the browser environment, where things cannot always be synchronous, and loading a different module might require a request that will be completed at a later time.

Before we can go any further and start the project setup on RequireJS, we first need to understand the structure of an AMD module.

Module definition

In *Chapter 3, Testing Frontend Code*, we have seen how to use the module pattern with IIFE to organize our code. An AMD module is built on the same principles: a file and a function. However, instead of using an IIFE, we invoke the AMD `define` function passing a callback function as an argument. At a later time, RequireJS will invoke this function argument once it is needed by another module.

Here is a simple module definition without any dependency:

```
define(function () {  
    function MyModule() {};  
    return MyModule;  
});
```

That is very similar to what we have done so far. The following example shows how that code would be if written using the conventions presented in *Chapter 3, Testing Frontend Code*:

```
(function () {  
    function MyModule() {};  
    return MyModule;  
})();
```

And what about the dependencies? Up until now everything was globally available, so we passed the dependencies to the module as parameters to the IIFE as follows:

```
(function ($) {  
    function MyModule() {};  
    return MyModule;  
})(jQuery);
```

But as you start using RequireJS on the project, there will be no more global variables. So how do we get those dependencies into our module?

If you looked closer at our simple module definition; it was returning the module value as its last statement:

```
define(function () {  
    function MyModule() {};  
    return MyModule;  
});
```

So if RequireJS knows a module value, all we have to do is ask RequireJS. Let's refer again to the dependency example, but this time as an AMD module:

```
define(['jquery'], function ($) {  
    function MyModule() {};  
    return MyModule;  
});
```

We select what module we need, so RequireJS loads it for us, and once the loading is complete, it calls the module definition function with the jQuery value. Pretty cool huh?

You can pass as many dependencies as you need to the dependencies array, and their values will be passed as arguments, in the same order, to the function, once they became available.

Project setup

Setting up a RequireJS is very simple. The library was created with ease of use in mind; it doesn't require an HTTP server or a compilation step to work (as with other solutions). All you need to get started is download a single JavaScript file and perform some small configuration.

For this example we are using version 2.1.6, so go ahead and download it from <http://requirejs.org/docs/release/2.1.6/minified/require.js>, and place it under the project's `lib` folder.

Next, we need to change our `SpecRunner.html` file to start using RequireJS. The first thing you are going to notice is that we no longer have any JavaScript dependency on the HTML file. Instead we refer to the RequireJS source and specify a special HTML attribute that tells RequireJS which is our main JavaScript file. From there all dependencies are declared on each module:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Jasmine Spec Runner</title>
  <link rel="shortcut icon" type="image/png" href="lib/jasmine-1.3.1/
jasmine_favicon.png">
  <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/
jasmine.css">
  <script src="src/RequireConfig.js"></script>
  <script data-main="spec/SpecRunner" src="lib/require.js"></script>
</head>
<body>
</body>
</html>
```

It is kinda cool to see this very clean HTML file. All that is left are the CSS and the RequireJS references.

The new SpecRunner.JS file

Since we have removed all the JavaScript code from within the `SpecRunner.html` file, it needs to be somewhere.

As you can see by the RequireJS script tag; it sets its main file as a `spec/SpecRunner.js` file. This is where it all begins:

```
require([
  'jquery',
  'jasmine',
  'jasmine-html'
],
function($, jasmine) {
  var jasmineEnv = jasmine.getEnv();
  jasmineEnv.updateInterval = 1000;

  var htmlReporter = new jasmine.HtmlReporter();
  jasmineEnv.addReporter(htmlReporter);
  jasmineEnv.specFilter = function(spec) {
```

```

    return htmlReporter.specFilter(spec);
  };

  $(function () { jasmine.getEnv().execute(); });
});

```

We use the `require` function, since we don't need this file to be available as a module, and pass all of its dependencies (`jquery`, `jasmine`, and `jasmine-html`).

After RequireJS has finished loading them, it will call the function passing all of its dependencies as arguments. This is where we set up the Jasmine reporter and execute the specs; a familiar code that was once inside the `SpecRunner.html` file.

The RequireJS configuration

For this new runner to work, we need to tell RequireJS where to look for modules. So we create a new JavaScript source file called `src/RequireConfig.js`. Here we declare a global object named `require`:

```

var require = {
  baseUrl: 'src',

  paths: {
    'spec': '../spec',

    'jquery': '../lib/jquery',
    'backbone': '../lib/backbone',
    'underscore': '../lib/underscore',

    'sinon': '../lib/sinon',
    'jasmine': '../lib/jasmine-1.3.1/jasmine',
    'jasmine-html': '../lib/jasmine-1.3.1/jasmine-html',
    'jasmine-jquery': '../lib/jasmine-jquery'
  }
};

```

We load this file before RequireJS:

```

<script src="src/RequireConfig.js"></script>
<script data-main="spec/SpecRunner" src="lib/require.js"></script>

```

By the time RequireJS loads, it can read this configuration, and set up itself.

So far, we are setting up two parameters:

- The folder (`baseUrl`) where RequireJS will look for modules by default, which we set to the `src` folder
- A few path translations (`paths`), to allow us to refer to different library dependencies and specs, without using relative paths

This allows us to use the jQuery module like:

```
define(['jquery'], function ($) {} );
```

Instead of:

```
define(['../lib/jquery'], function ($) {} );
```

But there is still one piece of configuration left; to use any module as a dependency, it needs to be a valid AMD module. And this is not the case for all of our external dependencies. Luckily, RequireJS comes with a solution to support non AMD modules, as we will see next.

Using non AMD dependencies with Shim

The only external dependency that we have which supports AMD natively is jQuery, so we don't need any extra parameters for it to work. But for all of the remainder (`Backbone.js`, `Underscore.js`, `Jasmine`, and `Sinon.js`), we need to know:

- What global variable it creates that needs to be exported to any module that requires it as a dependency.
- Any dependencies that this module has. A known example would be the `Underscore.js` dependency for `Backbone.js`.

To fix these requirements, we need to add other configuration parameters to the `RequireConfig.js` file:

```
var require = {  
  // other parameters...  
  
  shim: {  
    'backbone': {  
      deps: ['underscore', 'jquery'],  
      exports: 'Backbone'  
    },  
    'underscore': {  
      exports: '_'  
    },  
    'jasmine': {
```

```

    exports: 'jasmine'
  },
  'sinon': {
    exports: 'sinon'
  },
  'jasmine-html': ['jasmine'],
  'jasmine-jquery': ['jasmine']
}
};

```

We have to add the `shim` property; let's take the `backbone` example to understand how it works:

- First, it tells that it depends on both `underscore` and `jquery` modules
- And later, that it should export the global `Backbone` variable

To any module requesting for `backbone`, `RequireJS` will make sure that both `underscore` and `jquery` modules were already loaded, and it will also pass the correct value for the `Backbone.js` dependency on any `define/require` function.

Testing a module

Now that we have finished setting up `RequireJS` and our test runner, it is time to adapt our specs and code to `RequireJS`. And, as you will see, it is going to be pretty easy.

Let's take the `Investment` model as an example. First we need to wrap all the spec code inside the module definition:

```

define(function () {
  describe("Investment", function() {
    var stock;
    var investment;

    beforeEach(function() {
      stock = new Stock();
      investment = new Investment({ stock: stock });
    });

    it("should be a Backbone.Model", function() {
      expect(investment).toEqual(jasmine.any(Backbone.Model));
    });
  });
});

```

Then, we need to specify what are the spec dependencies by adding an array with the names of the depending modules:

```
define([
  'spec/SpecHelper',
  'backbone',
  'models/Investment',
  'models/Stock'
],
function () {
  describe("Investment", function() {
  });
});
```

Finally, we add the callback function arguments, to receive these dependencies into the module:

```
define([
  'spec/SpecHelper',
  'backbone',
  'models/Investment',
  'models/Stock'
],
function (jasmine, Backbone, Investment, Stock) {
  describe("Investment", function() {
  });
});
```

Since all specs require Jasmine, its custom matcher and plugins to be properly configured, they will all have the SpecHelper as a dependency.

And as with everything else, we need to make this SpecHelper an AMD module:

```
define([
  'jasmine',
  'jasmine-jquery'
],
function (jasmine) {
  jasmine.getFixtures().fixturesPath = 'spec/fixtures';

  beforeEach(function() {
    this.addMatchers({
      toBeAGoodInvestment: function() {
        var investment = this.actual;
        var what = this.isNot ? 'bad' : 'good';
        this.message = function() {
          return 'Expected investment to be a ' + what + ' investment';
        };
      }
    });

    return investment.get('isGood');
  });
});
```

```
    }  
  });  
});  
  
  return jasmine;  
});
```

As you can see, it has dependencies to Jasmine and all of its plugins (in our case, just `jasmine-jquery`).

Since it is actually setting up Jasmine, we return `jasmine` as the module value.

And since all dependencies are in place, all we have to do to make this spec run, is add it as a dependency to the `SpecRunner` module at `spec/SpecRunner.js`:

```
require([  
  'jquery',  
  'jasmine',  
  'jasmine-html',  
  'spec/models/InvestmentSpec'  
],  
function($, jasmine) {  
  // Spec Runner code...  
});
```

By now, you should be able to run this spec by opening the `SpecRunner.html` file on your browser. But as you might have expected, it should be failing.

So let's move to the `Investment` implementation, to see how we can fix this. Since we were using an IIFE, the process of conversion is much simpler than of the spec.

We already have all dependencies in place, all we have to do is add the `define` function and the array of dependency names. And instead of assigning the `Investment` to the global namespace, we return it as the module value:

```
define([  
  'backbone',  
  'models/stock'  
],  
function (Backbone, Stock) {  
  var Investment = Backbone.Model.extend();  
  
  return Investment;  
});
```

This was very simple to accomplish because we were already using good practices to organize our code.

Optimizing for production

After we have finished porting the entire codebase to RequireJS, we are ready to use the optimizer to pack and minify our code. Therefore, for achieving the second goal we have a single file deployed on production.

To use the optimizer you are going to need `Node.js` and its package manager. The installation process was explained in *Chapter 4, Asynchronous Testing – AJAX*.

To simplify its use, we are going to install it globally, so it is always available on your path. Using NPM from any folder on your computer, install the package:

```
$ npm install -g requirejs
```

After the installation is complete, let's create a build configuration file with our build parameters. Add a new file at the project root directory called `Build.js`:

```
(( {
  mainConfigFile: 'src/RequireConfig.js',
  baseUrl: "src",
  out: "build/boot.js",
  name: "Boot"
}))
```

As you can see it imports the parameters from the former `RequireConfig.js` file:

```
mainConfigFile: 'src/RequireConfig.js',
```

It also sets the `baseUrl` parameter again (Leaving it blank was causing a problem in the build process while using version 2.1.6 of the optimizer.).

```
baseUrl: "src",
```

It sets the destination for the build artifact; a file that will contain all our packed and minified source and dependencies:

```
out: "build/Boot.js",
```

Finally, it specifies the main file. For our `SpecRunner.html` it was the `spec/SpecRunner.js` file. And here, it is the `src/Boot.js` file:

```
name: "Boot"
```

As for the `Boot` file, it requires the `Application` to start:

```
require([
  'Application'
],
function (Application) {
  Application.start();
});
```

We haven't covered this Boot file in the book, so be sure to check the attached source files to understand better how it works.

Everything set; we are ready to run the optimizer. On the console, in the `projects` folder, type the following command:

```
$ r.js -o build.js
```

You should see something like this:

```
Tracing dependencies for: Boot
Uglifying file: code/build/Boot.js

code/build/Boot.js
-----
code/lib/underscore.js
code/lib/jquery.js
code/lib/backbone.js
code/src/models/stock.js
code/src/models/Investment.js
code/src/models/Stock.js
code/src/plugins/jquery-disable-input.js
code/src/views/NewInvestmentView.js
code/src/views/InvestmentView.js
code/src/views/InvestmentListView.js
code/src/views/ApplicationView.js
code/src/routers/InvestmentsRouter.js
code/src/Application.js
code/src/Boot.js
```

This means that `build/Boot.js` was created.

You can take a look at it, a packed and minified version of the application code and its dependencies, ready for deployment!

Headless testing with PhantomJS

Remember we said back in the introduction, that we could execute Jasmine without the need of a browser window? To do so, we are going to use PhantomJS, a scriptable headless WebKit browser (the same rendering engine of Safari).

Downloading and installing PhantomJS

PhantomJS ships binary distributions to Windows, Mac, and Linux. So to get started is as easy as downloading a ZIP file and executing the `phantomjs` command.

Go to the PhantomJS download page available at <http://phantomjs.org/download.html> and download the appropriate distribution for your platform.

Once downloaded, you can place the executable at the root of the project folder.

Running your tests without opening a browser

With the executable available, we are going to download a small script to allow running the Jasmine specs with PhantomJS.

You can download it from <https://github.com/ariya/phantomjs/blob/master/examples/run-jasmine.js>, and place it at the root of the project folder.

Done! Now you can run yours specs in the command line like:

```
$ phantomjs run-jasmine.js SpecRunner.html
```

And you will see the result on the console, with no browser window open:

```
'waitFor()' finished in 493ms.
```

```
Passing 60 specs
```

So now we can run and get the results of the specs right from the console. This will allow some awesome automation as we will see next.

Grunt

Grunt is a JavaScript tool to create and automate project tasks. It solves the same problems as Ant for Java, Rake for Ruby, and so on, but in the JavaScript language.

It is widely used by the Node.js community, and it is gaining a lot of traction on all sorts of JavaScript projects.

We are going to use it to make our lives simpler and more automated, so that we can focus more on the development time!

Installation

Grunt is also a `Node.js` package, so its installation is pretty simple once you have `Node.js` installed (as explained in *Chapter 4, Asynchronous Testing – AJAX*).

Open up a terminal and invoke `Node.js`'s package manager to install grunt's command-line interface and make it available globally:

```
$ npm install -g grunt-cli
```

At the project's root folder, install the grunt library:

```
$ npm install grunt
```

Once completed, you will have a `grunt` command available.

Project setup

To make our project grunt compatible, we need to create a `Gruntfile`. It is much like a `Makefile`, or a `Rakefile`, containing task definitions, but in JavaScript.

The `Gruntfile` should be named `Gruntfile.js` and should be placed in the project's root folder, containing the following skeleton:

```
module.exports = function(grunt) {  
  grunt.initConfig({});  
};
```

And this is it; we are ready to start adding helpful tasks to our project.

A RequireJS optimizer task

We are going to start by creating a task for our optimizer process. To make it simpler, we are going to use a grunt plugin, the `grunt-contrib-requirejs`.

Since it is also a `Node.js` package, we can install it with an NPM command. At the project root folder, execute the following command:

```
$ npm install grunt-contrib-requirejs
```

This will install the package and make it available for use inside our `Gruntfile`.

The first thing we need to do is load this plugin's tasks to our Gruntfile by adding a new line:

```
module.exports = function(grunt) {  
  grunt.initConfig({});  
  grunt.loadNpmTasks('grunt-contrib-requirejs');  
};
```

This will load a new grunt task called `requirejs`. But before we can start using it, we need to make a few configurations such as passing the same parameters to our `Build.js` file.

Add a new entry to `grunt.initConfig`:

```
grunt.initConfig({  
  requirejs: {  
    compile: {  
      options: {  
        mainConfigFile: 'src/RequireConfig.js',  
        baseUrl: "src",  
        name: "Boot",  
        out: "build/Boot.js"  
      }  
    }  
  }  
});
```

You can notice that it is the same configuration parameters we had in the `Build.js` file. And since we are going to start using grunt for now on, you can even delete the old file.

All set, we can run our first grunt task and build our project:

```
$ grunt requirejs
```

This will perform the build, and create the resulting artifact, the same way as directly using the optimization script.

But we can still do better. We can make this `requirejs` task default, by adding another configuration parameter after we load the `requirejs` task inside our Gruntfile:

```
grunt.loadNpmTasks('grunt-contrib-requirejs');  
grunt.registerTask('default', ['requirejs']);
```

Now, all we have to do, to perform a build, is to call `grunt`.

```
$ grunt
```

A Jasmine task

We don't want to generate a build artifact without making sure that the specs are passing, after all we don't want to ship bugs to our production environment.

So let's create a new grunt task to run our specs and see how we can make it run before a build.

There are specific grunt plugins to run Jasmine specs, but we are going to use a different approach here. We want to use the same runner we specified earlier.

To execute the PhantomJS command from inside grunt, we are going to use the `grunt-exec` plugin;

Install it by invoking the familiar NPM command from the project folder:

```
$ npm install grunt-exec
```

Also, add its task in the Gruntfile, like we did with the `requirejs` plugin:

```
grunt.loadNpmTasks('grunt-exec');
```

Next, we need to set up a new task to run PhantomJS. So add a new entry to the grunt configuration object:

```
grunt.initConfig({
  exec: {
    jasmine: {
      command: 'phantomjs run-jasmine.js SpecRunner.html'
    }
  }
});
```

Now, we are ready to start running our specs with a grunt command:

```
$ grunt exec:jasmine
```

This will invoke PhantomJS and display our spec results.

So how can we ensure that we run the specs before the build? We can create a new grunt task called `build`, make it default, and set it to run both the Jasmine and RequireJS tasks:

```
grunt.registerTask('build', ['exec:jasmine', 'requirejs']);
grunt.registerTask('default', ['build']);
```

By calling `grunt`, it will run all of these tasks for us automatically:

```
$ grunt
```

Watch for changes and run the specs

If you thought that was cool, here is a thought, "Wouldn't it be great if my specs run every time I change a file?"

Can we do that? Sure we can, and with another grunt plugin.

We are going to use `grunt-contrib-watch`, a plugin that watches for file changes, and runs specified tasks as a result.

First, install the plugin (Always at the project root directory.):

```
$ npm install grunt-contrib-watch
```

Next, load it in the Gruntfile:

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

And finally, set up the plugin to watch for changes on our `source`, `spec`, and `lib` folders, and as a result run the Jasmine task:

```
grunt.initConfig({
  watch: {
    scripts: {
      files: ['src/**/*.js', 'spec/**/*.js', 'lib/**/*.js'],
      tasks: ['exec:jasmine']
    }
  }
});
```

Now you can invoke grunt's watch task and see yours specs run each time you change any file on those directories:

```
$ grunt watch
```

Now this is development nirvana!

Managing NPM dependencies

There are a lot of NPM dependencies in the project, and we have always installed them by hand. But we want to have this package installation process automated, especially if we want others to contribute on the same project.

Every `Node.js` project can have a `package.json` file with the definition of the project. We are going to create this file at the root of our project.

At its basics, it is just a simple description of the project, with a name and version:

```
{
  "name": "investment-tracker",
  "description": "Jasmine Testing Example Application",
  "version": "0.0.1"
}
```

But it can also be used to define its dependencies:

```
{
  "name": "investment-tracker",
  "description": "Jasmine Testing Example Application",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.x"
  },
  "devDependencies": {
    "grunt": "0.4.x",
    "grunt-cli": "0.1.x",
    "grunt-exec": "~0.4.0",
    "grunt-contrib-requirejs": "~0.4.0",
    "grunt-contrib-watch": "~0.4.3",
    "grunt-contrib-jshint": "~0.4.3",
    "grunt-contrib-connect": "~0.3.0",
    "requirejs"
  }
}
```

As a production dependency, we have expressed the framework we used to run our web server in *Chapter 4, Asynchronous Testing – AJAX*.

As development dependencies, we have everything we need on the development machine, such as grunt and all of its plugins.

So if you come on a new machine and need to install all of these dependencies, just type on the project root folder:

```
$ npm install
```

And all those dependencies will be installed for you.

Summary

In this chapter, I hope to have showed you the power of automation, and how we can use scripts to make our life easier. You have learned about RequireJS, and how it can be used to manage the dependencies between your modules, and help you generate the production code (packed and minified).

You have also seen how to run your specs headless, and also how to run them before the build guaranteeing that no break spec should go to production.

We have also seen how to have the specs running automatically, letting us focus on the code editor all the time.

Be sure to check the bonus chapter on *Continuous Integration with Travis-CI*, to learn how to run yours specs in the cloud on every push to the project's source repository.

9

Conclusion

The increasing demand for more complex JavaScript projects has created a rich and thriving ecosystem of tools and standards. Despite this book's focus on Jasmine, I hope to have showed you that it is possible to get amazing productivity and code quality, while developing JavaScript applications.

We have seen how amazingly simple Jasmine is, and how we can start very quickly from a simple application, based around a single HTML, to a project with automated build and headless unit testing. All with very little effort.

And although BDD does offer a good guideline on how to write the specs, we were also faced with complex decisions on how to write tests that involves integration. Should you stub or use real components when testing a code? We have seen how sometimes the decision can be blurred, and choosing each approach can be based on the level of confidence you have on the other component's tests, or how simpler a spec can become. There isn't a single simple answer here.

But this is just the beginning. The JavaScript ecosystem is evolving at a rapid pace, and new frameworks get released every day. So keep up with your studying, and happy coding!

Index

Symbols

`$.getJSON` function 72

A

abstractions, Backbone 79

acceptance criteria

about 14

implementing 19-23

implementing, for asynchronous testing 57

writing 14-19

actual value 16

addInvestment method 50

addMatcher function 27

afterEach function 22

AJAX

used, for loading HTML fixtures 45

AJAX request, Backbone collections 89-91

AJAX request, Backbone model 84-86

AMD 106

andCallFake function 72

AngularJS 7

**appendLoadFixtures(fixtureUrl[,
 fixtureUrl,])** function 44

appendSetFixtures(html) function 44

assertion 16

asynchronous 57

Asynchronous Module Definition. *See*
 AMD

asynchronous testing

acceptance criterion, implementing 57

attribute, Backbone model 80, 81

B

Backbone

about 79

abstractions 79

applications, testing 79

Backbone collections

about 79, 87

AJAX request 89-91

declaring 88

features 87

Sync request 89-91

testing 87

URL, for documentation 91

Backbone.js 7

Backbone model

about 79

AJAX request 84-86

attribute 80, 81

declaring 80

default attribute value 82

events 83, 84

Sync request 84-86

Backbone router

about 79, 101

defining 101

routes, defining 102, 103

routing 103

testing 101

using 103

Backbone variable 111

Backbone View

declaring 92

- DOM events, binding 98-100
- el property 93, 94
- rendering 95
- testing 91
- updating 96, 98
- baseUrl parameter 114**
- beforeEach function 21**
- behavior checking**
 - about 67
 - performing, Jasmine Spy used 67, 68
- Behavior-Driven Development (BDD) 8, 9**
- Build.js file 118**
- built-in matchers**
 - about 30
 - toBe built-in matcher 31
 - toBeDefined built-in matcher 35
 - toBeFalsy built-in matcher 33
 - toBeGreaterThan built-in matcher 36
 - toBeLessThan built-in matcher 36
 - toBeNaN built-in matcher 34
 - toBeNull built-in matcher 34
 - toBeTruthy built-in matcher 33
 - toBeUndefined built-in matcher 34
 - toContain built-in matcher 35
 - toEqual built-in matcher 31
 - toMatch built-in matcher 36
 - toThrow built-in matcher 36
- C**
- callback parameter 72**
- Chrome**
 - HTML fixtures, using with 45
 - reference link 45
- Chrome browser 60**
- collection. See Backbone collections**
- console.log function 8**
- create function 68**
- create method 52**
- custom matchers 27-30, 53**
- D**
- default attribute value 82**
- define/require function 111**
- describe function 24**
- destroy function 99, 100**

- DOM element**
 - about 46
 - encapsulating 46-49
- DOM events**
 - about 69
 - binding 98-100
 - testing 69

E

- ECMAScript 7**
- el property 93, 94**
- Envjs 12**
- equals operator 31**
- events, Backbone model**
 - about 83, 84
 - URL 84
- events object 99**
- expected value 16**
- expect function 16**

F

- Fake server 75-77**
- FakeServer function 73**
- FakeXMLHttpRequest 73, 74**
- FakeXMLHttpRequest function 73**
- FakeXMLHttpRequest instance 75**
- FakeXMLHttpRequest object 73, 74**
- fetch function 58, 72, 85, 89**
- fetchRequest variable 74**
- find function 48**

G

- getJSON function 72**
- getSymbol method 47**
- global Jasmine function**
 - about 15
 - afterEach function 22
 - beforeEach function 21
 - describe function 15
 - expect function 16
 - it function 15
- Grunt**
 - about 116
 - installing 117

- Jasmine task 119
- RequireJS optimizer task 117, 118
- setup 117

grunt-exec plugin 119

H

headless browser 12

HTML fixtures

- about 42
- loading, AJAX used 45
- using 42-46
- using, with Chrome 45

I

idAttribute attribute 86

immediately invoked function expression (IIFE) 41

installation, Node.js 58

investment 14

Investment() function 17

Investment Tracker application

- developing 13, 14

it function 15

J

Jasmine

- about 8, 9
- downloading 10-12
- jQuery matchers 53
- matchers 26
- URL 116
- URL, for downloading 10

Jasmine jQuery

- about 39
- built-in matchers 30
- custom matchers 26-30
- URL 56
- URL, for downloading 39

Jasmine Spy

- about 67
- used, for performing behavior checking 67, 68
- used, for restoring the object original behavior 68

Jasmine Stubs 71, 72

Jasmine task 119

JavaScript

- about 7
- pitfalls 7, 8

JavaScript browser code

- testing 39

JavaScript language 7

JavaScript Object 8

jQuery 39

jQuery matchers

- about 53
- custom matchers 53
- toBeDisabled jQuery matcher 56
- toBeFocused jQuery matcher 56
- toBe jQuery matcher 54
- toContainHtml jQuery matcher 55
- toContain jQuery matcher 55
- toHaveAttr jQuery matcher 55
- toHaveValue jQuery matcher 55
- View, testing with 53, 54

L

listenTo function 98

listView object 52

M

matcher 16

matchers, Jasmine

- about 26
- built-in matchers 30
- custom matchers 26-30

model. *See* **Backbone model**

module

- testing 111-113

module definition 106, 107

module pattern 41, 42

MVC framework 40

N

named parameters 20

nested describes

- about 24
- setup function 24

- teardown function 24
- NewInvestmentView class** 54
- Node.js**
 - about 58
 - installing 58
 - URL 58
- Node.js package** 117
- non AMD dependencies**
 - used, with Shim 110, 111
- NPM dependencies**
 - managing 120, 121

O

- object original behavior**
 - restoring, Jasmine Spy used 68
- Object.seal function** 8
- observers**
 - View, integrating with 49-52
- onCreate function** 52
- optimizer task** 117, 118

P

- package.json file** 120
- PhantomJS**
 - about 12
 - installing 116
 - testing with 115
 - tests, executing 116
 - URL 116
- phantomjs command** 116
- PhoneGap** 7
- preloadFixtures() function** 45
- production**
 - optimizing for 114, 115

R

- readFixtures(fixtureUrl[, fixtureUrl,])**
 - function 44
- remove function** 98
- render function**
 - about 95, 96
 - implementing 96
- RequireConfig.js file** 110
- RequireJS**
 - about 106

- configuring 109, 110
- module definition 106, 107
- module, testing 111-113
- production optimization 114, 115
- setting up 107, 108
- SpecRunner.JS file 108, 109

- RequireJS configuration**
 - non AMD dependencies, using with Shim 110, 111
- requirejs plugin** 119
- respond function** 74
- respondWith function** 76
- return of investment attribute** 84
- router.** *See* **Backbone router**
- routes**
 - defining 102, 103
- runner** 10
- runs() function** 63

S

- Same Origin Policy** 45, 59
- server**
 - creating 59, 60
 - running 60
- set function** 81
- setup function** 24, 43
- shared behavior**
 - spec, coding with 25, 26
- Shim**
 - non AMD dependencies, using 110, 111
- sinon.fakeServer.create function** 76
- Sinon.JS**
 - about 73
 - Fake server 75-77
 - Fake XMLHttpRequest 73, 74
 - installing 73
 - URL 73
- Sinon.JS library** 72
- sinon.useFakeXMLHttpRequest function** 74
- spec**
 - about 9
 - coding 15-19
 - coding, with shared behavior 25, 26
 - executing 120
 - file, creating 15
 - writing 60

- SpecRunner.html file 10, 73, 108, 109
- SpecRunner.JS file 108, 109
- spec/SpecRunner.js file 114
- Spies 67
- spyOn function 72
- spyOn global Jasmine function 68
- stock.fetch() 76
- stock.fetch function 74
- Stock function 18
- stockSymbol parameter 77
- strict equals operator 31
- Sync request, Backbone collections 89-91
- Sync request, Backbone model
 - about 84-86
 - URL, for documentation 87

T

- teardown function 24
- test double 67
- Test-Driven Development (TDD) 9
- test fixture 42
- tests
 - executing 116
- test unit 9
- toBe built-in matcher 31
- toBeDefined built-in matcher 35
- toBeDisabled jQuery matcher 56
- toBeFalsy built-in matcher 33
- toBeFocused jQuery matcher 56
- toBeGreaterThan built-in matcher 36
- toBe jQuery matcher 54
- toBeLessThan built-in matcher 36
- toBe matchers 26
- toBeNaN built-in matcher 34
- toBeNull built-in matcher 34
- toBeTruthy built-in matcher 33
- toBeTruthy matchers 26

- toBeUndefined built-in matcher 34
- toContain built-in matcher 35
- toContainHtml jQuery matcher 55
- toContain jQuery matcher 55
- toEqual built-in matcher 31
- toEqual matchers 26
- toExist custom matcher 47
- toHaveAttr jQuery matcher 55
- toHaveValue jQuery matcher 55
- toMatch built-in matcher 36
- toThrow built-in matcher 36

U

- urlRoot attribute 86

V

- View.** *See also* **Backbone View**

View

- about 40, 79
- coding 46
- DOM element, encapsulating 46-49
- integrating, with observers 49-52
- testing, with jQuery matchers 53, 54

View, coding

- basic rules 46

- virtual attribute 84

W

- waitFor() function**

- about 61, 62
- parameters 62

X

- XMLHttpRequest object 73, 75



Thank you for buying
Jasmine JavaScript Testing

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



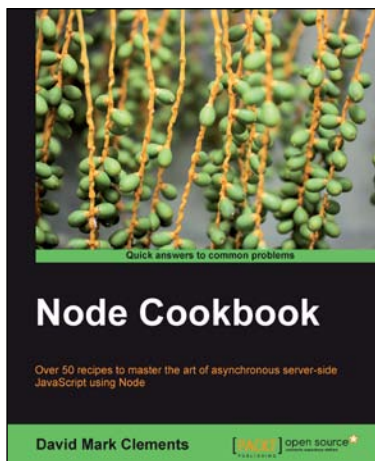
Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0

Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips



Node Cookbook

ISBN: 978-1-84951-718-8

Paperback: 342 pages

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node

1. Packed with practical recipes taking you from the basics to extending Node with your own modules
2. Create your own web server to see Node's features in action
3. Work with JSON, XML, web sockets, and make the most of asynchronous programming

Please check www.PacktPub.com for information on our titles



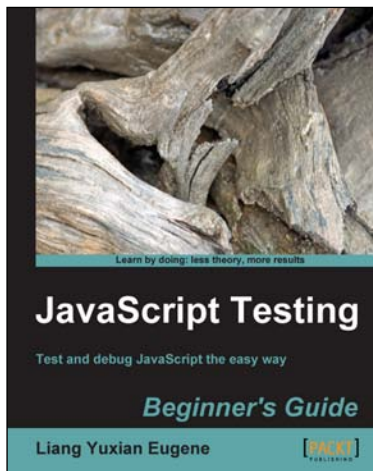
JavaScript Unit Testing

ISBN: 978-1-78216-062-5

Paperback: 190 pages

Your comprehensive and practical guide to efficiently performing and automating JavaScript unit testing

1. Learn and understand, using practical examples, synchronous and asynchronous JavaScript unit testing
2. Cover the most popular JavaScript Unit Testing Frameworks including Jasmine, YUITest, QUnit, and JsTestDriver
3. Automate and integrate your JavaScript Unit Testing for ease and efficiency



JavaScript Testing Beginner's Guide

ISBN: 978-1-84951-000-4

Paperback: 272 pages

Test and debug JavaScript the easy way

1. Learn different techniques to test JavaScript, no matter how long or short your code might be
2. Discover the most important and free tools to help make your debugging task less painful
3. Discover how to test user interfaces that are controlled by JavaScript

Please check www.PacktPub.com for information on our titles